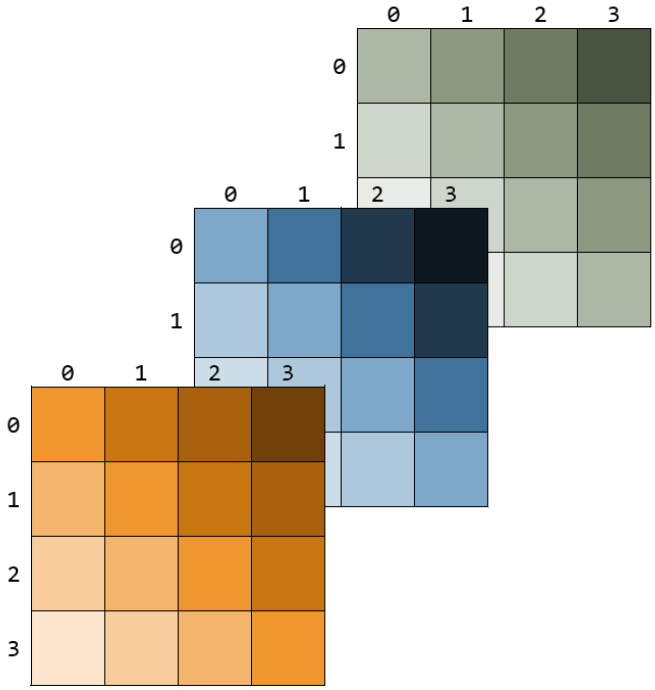
Марија Михова

Бојан Илијоски

Дизајн на алгоритми за динамичко програмирање



Уредник за издавачка дејност на УКИМ:

проф. д-р Никола Јанкуловски, ректор

Уредник на публикацијата:
Д-р Марија Михова
М-р Бојан Илијоски
Факултет за информатички науки и компјутерско инженерство - Скопје

Рецензенти

1. д-р Ѓорѓи Јованчевски
2. д-р Слободан Калајџиски

Техничка обработка

Марија Михова, Бојан Илијоски

Лектура на македонски јазик:
Даница Гавриловска - Атанасовска

CIP - Каталогизација во публикација
Национална и универзитетска библиотека "Св. Климент Охридски", Скопје

519.857:004.421.2(075.8)

МИХОВА, Марија
Дизајн на алгоритми за динамичко програмирање [Електронски извор] / Марија Михова, Бојан Илијоски. - Скопје : Универзитет "Св. Кирил и Методиј" - Скопје, 2020

Начин на пристап (URL):
http://www.ukim.edu.mk/mk_content.php?meni=53
&glavno=41. - Текст во PDF формат, содржи 496 стр., илустр. - Наслов преземен од екранот. - Опис на изворот на ден 20.02.2020. - Библиографија: стр. 502-504

ISBN 978-9989-43-443-3
1. Гл. ств. насл. 2. Илијоски, Бојан [автор]
а) Динамичко програмирање - Математички алгоритми - Примена - Компјутерско програмирање - Високошколски учебници COBISS.MK-ID 112165130

Предговор

Нашето повеќегодишно искуство во предавање на теми од дизајн на алгоритми на студентите по информатика на Факултетот за информатички науки и компјутерско инженерство, како и на талентираните основци и средношколци кои учествуваат на натпреварите по информатика, покажува дека за нив најголем проблем претставува справувањето со проблеми од динамичко програмирање. Блоговите за алгоритми и системите за решавање на алгоритамски проблеми изобилуваат со задачи од динамичко програмирање, но сепак студентите многу тешко реализираат успешно решение на проблем од овој тип, ако истиот за нив е непознат од претходно. Најголемата пречка во тоа е несистематскиот пристап во размислувањето и недоволно разработените техники за анализа на проблемот и развој на решение. Од друга страна повеќето од овие ресурси се базираат на едноставно листање на примери на кои како решение е дадена само рекурзивната равенка и програмскиот код, без да се навлегува во мисловниот процес кој студентот треба да го доведе до тоа решение и без разгледување на аналогијата на сличностите и разликите меѓу различните проблеми. Во оваа книга сакаме да се фокусираме точно на овој недостаток и да го покажеме нашиот процес на размислување кој го користиме за да стигнеме до решение. Дополнително се обидуваме да ја прикажеме поврзаноста помеѓу различни проблеми, да покажеме дека на некои проблеми може различно да им пристапиме и да добиеме навидум различни решенија кои ги прават истите пресметки и да демонстрираме стратегии со кои може да се намали сложеноста на решението.

Нашата идеја не е да навлегуваме многу длабоко во математичките аспекти на алгоритмите, но сепак да имаме доволно математичка поткрепа за при докажување на нивната точност. Затоа книгата не разработува голем број на задачи, но

секоја од нив е избрана со цел да илустрира некој специфичен аспект и идеја. Од друга страна, на објаснувањето на секоја од нив е посветено големо внимание, што би требало да го оспособи читателот успешно да се справи со многу други проблеми кои се базираат на слична идеја.

Книгата пред се е наменета за студентите по информатика за предметите поврзани со дизајн и анализа на алгоритми и за амбициозните средношколци и основци кои учествуваат на натпреварите по програмирање. Покрај нив, истата ја препорачуваме на љубителите на решавање на алгоритамски проблеми и се надеваме дека на сите нив ќе им помогне да ги подобрат своите вештини во справување со вакви задачи.

Содржина

Вовед

Проблемите од динамичко програмирање се едни од најпопуларните проблеми меѓу љубителите на алгоритамски проблеми, затоа што секој проблем е оригинален во некоја смисла и побарува длабоко да се размисли за да се стигне до решението. Основното решение честопати понатаму може да се модифицира и да се подобрува временската и мемориската сложеност аналитички или со одредени програмерски техники. Динамичкото програмирање не претставува алгоритам или техника, туку збир на принципи и методи со кои комплицираниот проблем се разделува на помали потпроблеми во рекурзивна смисла.

Под поимот динамичко програмирање се подразбираат два вида методи, методи за математичка оптимизација и методи од компјутерско програмирање. Развиен е од Ричард Белман во педесеттите години од 20-тиот век и истиот има примена во голем ранг на реални проблеми, од економија до инженерство. Во термини на алгоритми, разликуваме два вида проблеми кои се решаваат со динамичко програмирање: проблеми од комбинаторика и проблеми за оптимизација. Во проблемите од оптимизацијата е потребно да се изведе едно можно решение за кое вредноста на дадена функција е минимална или максимална, додека во комбинаторните проблеми треба да се пресмета на колку начини може да се направи нешто. Бидејќи рангот на идеи и пристапи е разнообразен, концептот на оваа книга е да се разработат повеќе различни примери.

Првата глава во книгава е посветена на математичките концепти кои се неопходни за развој на понатамошните алгоритми или пак за пресметка на нивната сложеност. Во наредните три глави проблемите се наредени во однос на нивната временска комплексност, од проблеми кои се решаваат во линеарно време, па проблеми кои се решаваат во квадратно време, до проблеми за кои основната идеја има поголема комплексност од квадратна. Илустрирани се и техники со кои во некои ситуации основната поедноставна идеја може да се разработи во насока на намалување на комплексноста на алгоритамот. Во петтата глава разгледуваме специфичен тип на комбинаторни задачи во кои техниката на динамичко програмирање се користи за генерирање или броење на одредени пермутации. Шестата глава е посветена на оптимизациони проблеми за кои иницијално се дизајнира решение со динамичко програмирање, кое со дополнителна анализа може да се сведе на алчен алгоритам. Во седмата глава ги разгледуваме најпознатите алгоритми од динамичко програмирање и најпознатите алчни алгоритми кои работат над графови, додека во последната, осма глава, даваме примери на НП комплексни проблеми и проблеми за кои не е можно решение со полиномна сложеност.

Глава 1. Математички концепти

При анализата на алгоритми често се користат различни математички алатки кои се употребуваат во повеќе аспекти во текот на процесот на дизајн на самиот алгоритам. Од една страна тие се почетна точка за идејата за решението на проблемот или пак самиот проблем се базира на математичка идеја. Од друга страна пак анализата на комплексноста на алгоритмите секогаш побарува некакви математички пресметки, кои понекогаш се едноставни техники, но понекогаш можат да бидат и понапредни техники непознати за студентите кои сакаат да навлезат во тајните на решавање на алгоритми. Во секој случај процесот на докажување на точноста на било кој алгоритам најчесто побарува солидно математичко знаење. За алгоритмите од димнамичко програмирање ова математичко знаење е од посебно значење, бидејќи тие во својата основа се изведување на рекурентна релација која го дефинира решението на проблемот.

Во оваа глава ќе се осврнеме на некои посложени математички области кои се неопходни за да се разберат алгоритмите кои ќе се анализираат подоцна, а на некои области, како множества, функции и релации кои исто така се од круцијално значење при дизајн на алгоритми нема да се фокусираме, но читателите кои не се осеќаат комотно со овие области од математиката се поттикнуваат да се консултираат со некоја книга од дискретна математика или множества [1], [2]. Прво ќе се фокусираме на основните принципи на броење: пермутации, комбинации и слично, бидејќи значаен дел од проблемите кои се решаваат со динамичко програмирање се комбинаторни проблеми, но исто така и заради тоа што оваа област се користи и при анализа на комплексноста на алгоритмите. Во вториот дел од оваа глава се зборува за рекурзија и принципите и

техниките за изведување на рекурзивни формули со кои се опишуваат реални проблеми. Во овој дел ќе се разгледаат и техники со кои се решаваат некои рекурзивни релации кои се јавуваат во алгоритмите од динамичко програмирање. Во третиот дел објаснуваме што е асимптотска нотација и како се манипулира со неа, затоа што ова е основа за споредба на ефикасноста на алгоритмите. Во четвртиот дел ја разгледуваме познатата мастер теорема, која го дава редот на големина на функции кои се зададени со рекурзивни врски кои проблемот го делат на неколку скоро еднакви делови. Овој тип на врски најчесто се јавуваат во така наречените раздели па владеј алгоритми, кои не се од наш интерес во оваа книга, но самата техника се јавува и на некои други места и затоа ќе ја разгледаме. Една класа стандардни проблеми кои се решаваат со техники од динамичко програмирање се алгоритми кои работат над графови, па во петтиот и шестиот дел ги даваме основните дефиниции и својства кај графови и дрва.

1.1 Комбинаторика

Комбинаториката е математичка дисциплина која ги изучува техниките со кои може да се определи на колку начини може да се направи избор на елементи од некоја колекција, за која важат одредени правила, или бројот на начини на кои можат да се подредат предмети од одредено множество.

Бидејќи голем број на алгоритамски проблеми, посебно проблеми од динамичко програмирање побаруваат солидно познавање од комбинаторика, во овој дел ќе се дадеме осврт на најважните техники и пристапи од оваа област. Ќе ги дефинираме правилата за збир и производ, ќе го дадеме принципот на вклучување и исклучување, ќе дефинираме што е пермутација, комбинација и број на пермутации и комбинации.

Основни комбинаторни правила се правилото на производ и правилото на збир:

Правило на производ:

> *Ако за извршување на една работа е потребно да се завршат две различни задачи, од кои првата задача може да се заврши на n начини, и за секој од овие начини постојат m начини за да се заврши втората задача. Тогаш постојат $n \cdot m$ начини за завршување на целата работа.*

Во термин на множества ова правило гласи вака:

> *Ако постојат n начини да се избере еден елемент од A и m начини да се избере еден елемент од B, тогаш ќе има $n \cdot m$ начини да се изберат два елементи, еден од A и еден од B.*

Пример 1. 1. Колку двоцифрени парни броеви можат да се запишат со цифрите од 1 до 5?

Решение: За да ја извршиме работата запишување на двоцифрен број, треба да ја извршиме задачата избор на првата цифра, која може да е било кој број од 1 до 5, и задачата избор на втора цифра, која може да биде или 2 или 4. Оттука првата цифра можеме да ја избереме на 5 начини, а втората на 2, што значи имаме вкупно $5 \cdot 2 = 10$ вакви двоцифрени броеви.

Правило на збир:

> *Нека претпоставиме дека има 2 различни пристапи за завршување на една работа. Ако постојат m -начини за да се заврши работата со користење на првиот пристап и n -начини да се заврши со вториот пристап, тогаш вкупниот број на начини на кои може да се заврши работата е $m + n$. Различни пристапи значи дека не може еден начин да се цврсти и во првиот и во вториот пристап.*

Во термин на множества ова правило гласи вака:

> *Ако постојат m начини да се избере еден елемент од A и n начини за избор на еден елемент од B, при што A и B немаа исти елементи, тогаш ќе има $m + n$ начини да се избере еден од A или од B.*

Пример 1. 2. Колкав е бројот на стрингови од единици или нули со должина 8, во кои последниот бит е 1 или последните два бита се 00?

Решение: Да забележиме дека не може истовремено да се случи и последниот бит да е 1 и последните два да се 00. Затоа, од правилото на збир, треба бројот на стрингови на кои последниот бит е 1 да го собереме со бројот на стрингови кои завршуваат на 00. Од правилото на производ, за да формираме стринг од единици или нули со должина 8 кои завршуваат на 1, треба на секоја позиција освен последната да избереме една од цифрите 0 или 1. Значи, за првата позиција имаме 2 начини, за втората исто така два, така до седмата позиција, па бројот на такви стрингови е $2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 = 2^7$. Слично, бројот на стрингови кои завршуваат на 00 е е $2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 = 2^6$. Сега, вкупниот број на вакви стрингови е е $2^7 + 2^6 = 3 \cdot 2^6$.

Да разгледаме друг пример: наместо бројот на стрингови од единици или нули со должина 8, во кои последниот бит е 1 или последните два бита се 00, да го побараме бројот на стрингови од единици или нули со должина 8, во кои првиот бит е 1 и последните два бита се 00. Тука имаме стрингови кои го задоволуваат или едниот и другиот услов, па поради тоа не можеме да направиме прост збир на бројот на стрингови во кои првиот бит е 1 и бројот на стрингови на кои последните два бита се 00. За да го решиме овој проблем воведуваме нова техника позната како принцип на вклучување и исклучување.

Принцип на вклучување и исклучување

Да претпоставиме дека има 2 пристапи за завршување на една работа, при што m начини за првиот пристап и n начини за вториот пристап, но некои од начините можат да се сместат и во првиот и во вториот пристап. Нека се тоа k начини. Тогаш бројот на начини за завршување на таа работа е $m + n - k$.

Во термин на множества ова правило гласи вака:

Ако постојат m начини да се избере еден елемент од A и n начини за избор на еден елемент од B, при што на k начини може да се избере елемент кој припаѓа и во A и во B, тогаш ќе има $m + n - k$. начини да се избере еден од A или B. Накратко,

$$|A \cup B| = |A| + |B| - |A \cap B| \qquad (1.1)$$

Пример 1. 3. Колкав е бројот на стрингови од единици или нули со должина 8, во кои првиот бит е 1 или последните два бита се 00?

Решение: За да го пресметаме бројот на вакви стрингови, прво треба да го пресметаме бројот на стрингови кои почнуваат со 1, кој беше 2^7, а потоа бројот на стрингови кои завршуваат на 00, кој беше 2^6. На крај од збирот на овие два броја ќе го одземе бројот на стрингови кои на почеток имаат 1, а на крај 00. На овој стринг недефинирани се 5

позиции, па вакви стрингови се 2^5. Оттука, бројот кој го бараме е $2^7 + 2^6 - 2^5$.

Пермутации:

> *Подредено разместување на r различни елементи од множество S се вика r -пермутација. Разликуваме пермутации со повторување и пермутации без повторување. Ако е дозволено некој елемент да се појавува повеќе пати, станува збор за пермутација со повторување, а ако не е дозволено станува збор за пермутација без повторување.*

На пример ако *S* = {1, 2, 3}. Тогаш

- 1 2 3, 2 1 3, 3 1 2 се 3-пермутации без повторување на S.
- 1 2, 1 3, 3 1 се 2-пермутации без повторување на S.
- 1 1 3, 2 1 1, 3 1 1 се 3-пермутации со повторување на S, но и 1 2 3 може исто така да се гледа како пермутација со повторување.
- 1 2 3 1 и 1 1 1 1 се 4-пермутации со повторување.

Да забележиме дека ако множеството S има n елементи, тогаш не можеда има r - пермутација без повторување за $r > n$, но може да има r - пермутација со повторување.

Број на пермутации:

> *Бројот на r-пермутации без повторување на множеството S со $|S|$= n елементи е*
>
> $$P(n,r) = n(n-1)\cdot\cdots\cdot(n-r+1) = \frac{n!}{(n-r)!}. \qquad (1.2)$$
>
> *Бројот на r-пермутации со повторување на множеството S со $|S| = n$ елементи е*

$$\bar{P}(n,r) = n^r.$$ Error! Bookmark not defined. Error! Bookmark n

(1. 3)

Пример 1. 4: На колку начини може да се избере прва, втора и трета награда за 100 различни учесници на натпреварот?

Решение: Бидејќи е битно кој учесник која награда ќе ја добие, станува збор за 3-пермутација и нејзиниот број е *P*(100, 3) = 100 · 99 · 98 = 970200.

Пример 1. 5: во едно училиште има од 100 различни ученици. На колку начини може да се избере по еден ученик кој ќе го претставува училиштето на натпреварите по информатика, математика и физика?

Решение: Бидејќи е битно кој ученик на кој натпревар ќе го претставува училиштето, но исто така ист ученик може да се натпреварува на повеќе од еден натпревар, станува збор за 3-пермутација со повторување и бројот е $\bar{P}(100,3) = 100^3$.

Комбинации:

> *Избор на r елементи од множество S, при што распоредот не е битен, се нарекува r - комбинација. Многу често ова го викаме комбинација без повторување.*

Во термини на множества ова може да се искаже како:

> *Комбинација (без повторување) од r елементи од множество S е r - елементно подмножество од S.*

На пример, ако *S* = {1, 2, 3}, тогаш {1, 2}, {1, 3} и {2, 3} се сите комбинации од два елементи од *S*. Да забележиме дека комбинацијата {1, 2} е иста со комбинацијата {2, 1}.

Број на комбинации:

Бројот на r - комбинации $C(n,r)$ или почесто $\binom{n}{r}$ на множество со $n = |S|$ елементи е еднаков на

$$\binom{n}{r} = \frac{n!}{r!(n-r)!} \tag{1. 4}$$

Во суштина комбинација е неподредена пермутација, па секоја комбинација можеме да ја гледаме и како пермутација. Имено, повеќе пермутации сочинуваат една комбинација, или поточно, секое разместување на една комбинација е различна пермутација. Да речеме, пермутациите: 1 2 и 2 1 се всушност една иста комбинација {1, 2}. Бидејќи една r - комбинација може да се размести на $r!$ Начини, ја имаме следнава формула:

$$P(n,r) = \binom{n}{r} r! \tag{1. 5}$$

Од самата формула лесно може да се забележи дека:

$$\binom{n}{r} = \binom{n}{n-r}.$$

Пример 1. 6: На колку начини може да се изберат 7 различни карти од шпил од 52 карти?

Решение: Распоредот на картите не е значаен, па решението е

$$\binom{52}{7} = \frac{52!}{7!\,45!}.$$

Пример 1. 7: На колку начини може 7 играчи да си поделат по една карта од шпил од 52 карти?

Решение: Овде распоредот на картите е значаен, па решението е

$$P(52,7) = \frac{52!}{45!}.$$

Наредниот пример е илустрација на проблемот познат како број на комбинации со повторување на елементите, односно број на начини на избор на n елементи, кога имаме m типови на различни елементи, такви што елементите од еден тип не можеме да ги

разликуваме меѓу себе. Формулата може да се изведе од претходното знаење, па нема да го разгледуваме како посебен начин на избор на елементи.

Пример 1. 8: На колку начини може 10 топчиња да се обојат во некоја од боите плава, црвена и зелена?

Решение: Откога ќе ги обоиме топчињата, јасно е дека црвените топчиња не можат да се разликуваат меѓусебно, исто така и плавите и зелените. Според тоа различен начин на боење би бил различен број на плави, црвени или зелени топчиња, на пример изборот 2 плави, 2 црвени и 6 зелени топчиња ќе се разликува од изборот на 3 плави 3 црвени и 4 зелени или од изборот на 5 плави и 5 зелени топчиња. Топчињата пред да ги обоиме можеме да ги наредиме во редица и да ставиме две цртички со кои ќе означиме дека сите топчиња до првата цртичка ќе бидат обоени во плаво, до втората со црвена и до крај со зелена боја. На пример изборот 2 плави, 2 црвени и 6 зелени топчиња може да се означи со: **|**|******, каде * означува топче, додека изборот 5 плави и 5 зелени топчиња со *****||*****. Така на 12 места, треба да ставиме 10 топчиња и 2 цртички (за една помалку цртички отколку што имаме бои за боење. Оттука решението е да се изберат местата на цртичките, а тоа може да се направи на $C(12,2) = \frac{12!}{10!2!}$ начини.

Една важна работа која треба да се нагласи кога станува збор за алгоритми со кои се решаваат комбинаторни проблеми е тоа дека честопати бројот кој се добива е многу голем. Заради тоа, најчесто се бара да се пресмета по некој модул, најчесто 100 000 007. Оттука, мора да се води сметка за формулите кои вклучуваат делење на факториели, затоа што модулот нема да остане ист. На пример да пресметаме $\binom{5}{3}$ по модул 7. Точната вредност е: $\binom{5}{3} = \frac{5!}{3!2!} = 10$, што е еднакво на 3 по модул 7. Од друга страна 5! = 120 е 1 по модул 7, па ако $\binom{5}{3}$ го пресметуваме како

$$\frac{5!\, mod\ 7}{((3!\, mod 7)(2!\, mod 7) mod 7)} = \frac{1}{(6 \cdot 2 mod 7)} = \frac{1}{5},$$

ќе добиеме сосема погрешен резултат.

За да го разрешиме овој проблем ќе се повикаме на познатото Паскалово равенство, или попознато како паскалов триаголник:

Паскалово равенство:

Нека n и i се природни броеви. Тогаш важи:

$$\binom{n}{i} = \binom{n-1}{i-1} + \binom{n-1}{i}, 0 < i \leq n;$$

$$\binom{n}{0} = \binom{n}{n} = 1, \qquad n \in \mathbb{N}. \tag{1. 6}$$

Пример 1. 9: Да се пресмета $\binom{5}{3} mod\ 7$!

Решение: Ќе го користиме паскаловото равенство се додека не дојдеме до основниот случај $\binom{n}{0}$:

$$\binom{5}{3} mod\ 7 = \left(\left(\binom{4}{2} mod\ 7\right) + \left(\binom{4}{3} mod\ 7\right)\right) mod\ 7;$$

$$\binom{4}{2} mod\ 7 = \left(\left(\binom{3}{1} mod\ 7\right) + \left(\binom{3}{2} mod\ 7\right)\right) mod\ 7;$$

$$\binom{3}{1} mod\ 7 = \left(\left(\binom{2}{0} mod\ 7\right) + \left(\binom{2}{1} mod\ 7\right)\right) mod\ 7$$

$$= 1 + \left(\binom{2}{1} mod\ 7\right);$$

$$\binom{2}{1} mod\ 7 = \left(\binom{1}{0} mod\ 7\right) + \left(\binom{1}{1} mod\ 7\right) = 1 + 1 = 2.$$

Оттука,

$$\binom{3}{1} mod\ 7 = \binom{3}{2} mod\ 7 = 1 + 2 = 3,$$

па $\binom{4}{2} mod\ 7 = 3 + 3 = 6$.

Останува да се пресмета:

$$\binom{4}{3} mod\ 7 = \left(\left(\binom{3}{2} mod\ 7\right) + \left(\binom{3}{3} mod\ 7\right)\right) mod\ 7 = 3 + 1 = 4.$$

На крај ја добиваме точната вредност:

$$\binom{5}{3} mod\ 7 = (6 + 4) mod\ 7 = 3.$$

Иако на прв поглед изгледа дека директната пресметка по формулата е подобра за користење, сепак за големи вредности кои треба да се испечатат по некој модул, формулите во кои се јавува делење не можат да се искористат и потребно е да се побара некој алтернативен начин, најчесто рекурзивна релација како во последниот пример, што е во суштина пристап со динамичко програмирање.

Прашања и задачи

1. Во базенот од прашања има 20 прашања од комбинаторика и по 10 прашања од графови и рекурзија. На колку начини може да се одбере по едно прашање од секоја тема за тестот?

2. Бојан живее во градот G и сака да оди во градот A. Тој може да одбере некој од трите меѓуградски автобуси или двата воза кои би го однеле од дома во градот C. Од таму, тој може да одбере некој од двата меѓуградски автобуси или трите воза кои би го однеле од градот C во посакуваниот град A. На колку различни начини тој може да стигне во градот A?

3. Дадено е множеството од природни броеви $A = \{1, 2, 3, \dots, 120\}$. Колку од броевите кои се елементи на множеството A не се деливи ниту со 2 ниту со 3.

4. Дадено е множеството од природни броеви $A = \{1, 2, 3, \dots, 120\}$. Колку од броевите кои се елементи на множеството *A* не се деливи со никој од броевите 2, 3 и 5.

5. Колкав е бројот на пермутации на буквите ABCDEFGH во кој што се јавува стрингот ABC?

6. Од група од 5 ученици треба да се направи тим од тројца. Колку такви групи постојат?

7. Петре купил 10 различни книги. На колку начини може да одбере 4 од нив за да чита на плажа додека е на одмор? Откога ги избрал книгите, на колку начини може да одбере по кој редослед ќе ги чита?

8. Дадени се две паралелни прави *n* точки на првата, а *m* точки на втората. Колку триаголници можат да се формираат на кои овие точки од правите им се темиња?

9. На колку начини може 10 деца да облечат маички во црвена , зелена и сина боја, така да 4 се во црвена, а по 3 зелена и сина маичка?

10. Имаме 2 црвени, 3 сини и 5 зелени топчиња кои треба да се обележат со броевите од 1 до 10. На колку начини може да се направи тоа?

11. На колку начини може броевите 1, 3, 5, 7, 9, 11, 2, 4, 6 да се распределат во 3 множества со по 3 елементи, така да збирот на броевите во секоја група биде парен?

12. На колку начини може броевите 1, 3, 5, 7, 9, 11, 2, 4, 6 да се распределат во 3 множества со по 3 елементи, така да збирот на броевите во точно една од групите биде парен, а во другите две биде непарен?

13. Дадени се бели, црни и сиви плочки. На колку начини може да се изберат 15 плочки?

14. Дадени се бели, црни и сиви плочки. На колку начини можат да се изберат 15 плочки ако сиви треба да бидат барем 3?

15. На колку начини можат да се распределат 30 исти ранци во 3 различни автомобили?

16. На колку начини можат да се распределат 10 црвени, 10 сини и 10 зелени ранци во 3 различи автомобили?

17. Глувче оди по x-оската $2n$ чекори, при што секој во секој чекор се помрднува за една единица во лево или десно. На колку начини може на крајот од прошетката повторно ќе се најде во почетната позиција?

18. Глувче се движи во рамнина, почнувајќи од координатниот почеток до точката $(m, n), m, n \geq 0$, така што во секој чекор се поместува за една единица во лево или една единица нагоре.

    a. На колку начини може да го направи тоа?
    b. На колку начини може да го направи тоа ако сака да помине во точката $(r, s), 0 \leq r \leq m, 0 \leq s \leq n$?

19. Да се пресмета $\binom{10}{4} mod\ 11$!

1.2 Рекурзија

Под комбинаторен проблем во дизајн на алгоритми генерално, се подразбира да се определи бројот на начини да се распределат одредени објекти или број на начини да се заврши некоја работа, односно, да се изврши одредена задача. Комбинаторните проблеми од динамичко програмирање се проблеми кои ова броење го решаваат со изведување на рекурентна врска која проблемот од поголема инстанца го претставува преку решенијата на истиот тој проблем, но со помали инстанци, што е познато како рекурентна релација. Една таква рекурентна релација е дадена со паскаловото равенство.

Рекурентната релација може да биде изразена преку една, но и преку повеќе равенки, при што се добива систем од линеарни рекурзивни равенки.

Во општ случај рекурентна релација претставува начин на дефинирање на низа од елементи, преку самата себеси, односно секој елемент од низата се дефинира преку претходните елементи од таа низа.

Рекурентна релација:

Рекурентна релација за низата $\{A[n]\}$ *е равенство со кое* n *-тиот член* $A[n]$*се изразува преку еден или повеќе претходни членови од низата, односно, со некои од* $A[0]$, $A[1], \dots, A[n-1]$, *за сите* $n \geq n_0$, n_0 *е позитивен цел број. Така,*

$$A[n] = F(A[n_1], \dots, A[n_k]), k \geq 1, n_i < n. \qquad (1.7)$$

Да напоменеме дека во изразот за $A[n]$ може да се јавуваат и константи и променливата n.

Да погледнеме како ќе ја пресметаме вредноста за $n = 4$ за функцијата зададена со рекурентната релација:

$$A[n] = 2A[n-1] + 1.$$

Пресметката е следнава:

$$A[4] = 2A[3] + 1 = 2(2A[2] + 1) + 1 = 4A[2] + 3$$
$$= 4(2A[1] + 1) + 3 = 8A[1] + 7.$$

Всушност $A[4]$ го изразуваме преку $A[1]$, но не можеме да го пресметаме се додека $A[1]$, или некоја друга вредност не ја знаеме. Затоа во формулите зададени со рекурентни релации, мора да ги дадеме решенијата за почетните инстанци, најчесто почнувајќи од $n = 1$ или $n = 0$. Според тоа, рекурентната врска сама по себе не е доволна за дефинирање на низата и потребно е да се дадат неколку така наречени **почетни услови**. Така, ако во овој случај ние знаеме дека на пример $A[1] = 1$, тогогаш ќе можеме да ја пресметаме и вредноста за $A[4]$.

Рекурентната релација и почетните услови единствено ја определуваат низата. Секој друг член од низата може да се пресмета преку рекурентната релација и почетните членови. За да се добие одреден член од низа зададена со рекурентна релација обично треба да се направат многу пресметки, но за некои типови на рекурентни релации членовите на низата може да се претстават и експлицитно, без рекурзија. Тоа не секогаш би го намалило бројот на пресметки, за што ќе дискутираме подоцна, но е многу корисно при пресметување на сложеноста на алгоритмите.

Хомогени линеарни рекурентни релации со константни коефициенти:

Хомогена линеарна рекурентна равенка со константни коефициенти од ред k е равенка од облик:

$$A[n] = c_1 \cdot A[n-1] + c_2 \cdot A[n-2] + \cdots + c_k \cdot A[n-k], \quad (1.8)$$

каде $c_1, c_2, \ldots, c_k$ се реални броеви и $c_k \neq 0$.

За да равенката биде единствено дефинирана треба да бидат дадени првите k членови на низата: $A[0] = C_0, \cdots, A[k-1] = C_{k-1}$.

Хомогени линеарна рекурентни релации со константни коефициенти од прв ред:

Обликот на оваа релација е:

$$A[n] = c \cdot A[n-1], A[0] \text{ е познато.} \qquad (1.\ 9)$$

а нејзиното решение е $A[n] = c^n \cdot A[0]$.

Хомогени линеарна рекурентни релации со константни коефициенти од втор ред:

Обликот на оваа релација е:

$$A[n] = c_1 \cdot A[n-1] + c_2 \cdot A[n-2],$$

$$A[0] \text{ и } A[1] \text{ се познати.} \qquad (1.\ 10)$$

Решението за ваква равенка може да се изведе и експлицитно, преку карактеристичниот полином на равенката. Овој карактеристичен полином се добива на следниов начин: Ја запишуваме рекурзивната равенка за првиот член од низата кој може да се изрази преку останатите членови (не прави разлика ако се земе било кој член што се изразува преку останатите). Потоа, од вака добиената формула добиваме полином, така што, секогаш кога во равенката ќе сретнете $A[i]$ треба да се замени со x^i.

Во нашиот случај, првиот член е $A[2] = c_1 A[1] + c_2 A[0]$, па карактеристичниот полином е

$$x^2 = c_1 x^1 + c_2 x^0 = c_1 x + c_2.$$

Решенијата на овој полином може да се еднакви или различни.

- Ако решенијата се различни, α_1 и α_2, може да се докаже, иако ова нема да го докажуваме овде, дека

$$A[n] = c{\alpha_1}^n + d{\alpha_2}^n,$$

за некои константи c и d, кои ги добиваме од почетните услови, односно $A[1]$ и $A[0]$, преку решавање на системот:

$$\begin{cases} A[1] = c{\alpha_1}^1 + d{\alpha_2}^1 = c\alpha_1 + d\alpha_2 \\ A[0] = c{\alpha_1}^0 + d{\alpha_2}^0 = c + d \end{cases}.$$

- Ако решенијата се исти, $\alpha_1 = \alpha_2 = \alpha$, може да се докаже, иако ова нема да го докажуваме овде, дека

$$A[n] = c\alpha^n + dn\alpha^n$$

за некои константи c и d, кои ги добиваме од почетните услови, односно $A[1]$ и $A[0]$, преку решавање на системот:

$$\begin{cases} A[1] = c\alpha + dn\alpha \\ A[0] = c + 0 = c \end{cases}.$$

Пример 1.10: Да се реши ја рекурентната релација зададена со:

$$A[n] = A[n-1] + A[n-2]$$

и почетни услови $A[0] = A[1] = 1$.

Решение: Карактеристичната равенка за оваа рекурзивна равенка е:

$$x^2 = x + 1,$$

и нејзини решенија се $\frac{1+\sqrt{5}}{2}$ и $\frac{1-\sqrt{5}}{2}$. Оттука општото решение е од облик

$$A[n] = c\left(\frac{1+\sqrt{5}}{2}\right)^n + d\left(\frac{1-\sqrt{5}}{2}\right)^n,$$

за некои константи c и d, кои ги добиваме од почетните услови, односно $A[1]$ и $A[0]$, преку решавање на системот:

$$\begin{cases} A[1] = 1 = c\left(\frac{1+\sqrt{5}}{2}\right)^1 + d\left(\frac{1-\sqrt{5}}{2}\right)^1 = c\left(\frac{1+\sqrt{5}}{2}\right) + d\left(\frac{1-\sqrt{5}}{2}\right) \\ A[0] = 1 = c\left(\frac{1+\sqrt{5}}{2}\right)^0 + d\left(\frac{1-\sqrt{5}}{2}\right)^0 = c + d \end{cases},$$

од каде се добива $c = \frac{5+\sqrt{5}}{10}$, $d = \frac{5-\sqrt{5}}{10}$.

Пример 1.11: Да се реши ја рекурентната равенка зададена со

$$A[n] = 4A[n-1] - 4A[n-2]$$

со почетни услови $A[0] = 1, A[1] = 4$.

Решение: Равенката $x^2 = 4x - 4$ има едно дупло решение, 2, па општото решение е од облик:

$$A[n] = c2^n + dn2^n,$$

за некои константи c и d, кои ги добиваме од почетните услови, односно $A[1]$ и $A[0]$, преку решавање на системот:

$$\begin{cases} A[1] = 4 = 2c + 2d \\ A[0] = 1 = c2^0 + d \cdot 0 \cdot 2^0 = c \end{cases},$$

од каде се добива $c = d = 1$, па решението е $A[n] = (1+n)2^n$.

Хомогени линеарна рекурентни релации со константни коефициенти од повисок ред:

Обликот на оваа релација е дадена со (1.8). Решението повторно може да се изведе и експлицитно, преку карактеристичниот полином на равенката. Овој полином се добива на сличен начин како за

равенки од втор степен, запишувајќи ја рекурзивната равенка за првиот член од низата кој може да се изрази преку останатите членови како

$$A[k] = c_1 \cdot A[k-1] + c_2 \cdot A[k-2] + \cdots + c_k \cdot A[0]$$

Потоа, од вака добиената формула добиваме полином, така што, секогаш кога во равенката ќе сретнете $A[i]$ треба да се замени со x^i, И потоа тој полином да се скрати со најнискиот степен на x кој се јавува во полиномот. Ќе се добие некој полином од облик:

$$x^s = c_1 x^{s-1} + \cdots + c_s x^0.$$

Обликот на општото решение е производ од степенска функција и полином. Ако кратноста на некое решение α е p, тогаш за него во општото решение се добива член

$$\left(d_{p-1} n^{p-1} + d_{p-2} n^{p-2} + \cdots + d_0\right)\alpha^n.$$

Повторно непознатите константи се наоѓаат од почетните услови.

Пример 1.12: Да се реши рекурентната релација

$$A[n] = -3A[n-1] - 3A[n-2] - A[n-3]$$

со почетни услови $A[0] = 1,\ A[1] = -2$ и $A[2] = -1$.

Решение: Равенката $x^3 + 3x^2 + 3x + 1 = 0$ има едно трикратно решение, –1 , па општото решение е од облик:

$$A[n] = (d_2 n^2 + d_1 n + d_0)(-1)^n.$$

Непознатите константи ги добиваме од почетните услови, односно:

$$\begin{cases} A[0] = 1 = d_0 \\ A[1] = -1 = (d_2 + d_1 + d_0)(-1) \\ A[2] = -1 = (4d_2 + 2d_1 + d_0)(-1)^2 \end{cases}$$

од каде се добива $d_2 = -2, d_1 = 3, d_0 = 1$.

Ако во хомогената рекурентна равенка се додаде член кој е некоја функција од n, добиваме нехомогена рекурентна равенка со константни коефициенти. За да се реши оваа равенка, треба прво да се реши соодветната необоена равенка, а потоа да се побара едно партикуларно решение на нехомогената равенка. Општиот облик на решението е збир од овие две решенија.

Нехомогени линеарни рекурентни релации со константни коефициенти:

Нехомогена линеарна рекурентна равенка со константни коефициенти од ред k е равенка од облик:

$$A[n] = c_1 \cdot A[n-1] + c_2 \cdot A[n-2] + \cdots + c_k \cdot A[n-k] + F(n), \tag{1. 11}$$

каде $c_1, c_2, \dots, c_k$ се реални броеви и $c_k \neq 0$.

За да равенката биде единствено дефинирана треба да бидат дадени првите k членови на низата, $A[0] = C_0, \cdots, A[k-1] = C_{k-1}$.

Хомогената равенка за 1.11 е дадена со 1.8, па општиот облик на нејзиното решение може да се најде со постапката за решавање на хомогени равенки со константни коефициенти, опишана погоре. Со тоа проблемот се сведува на тоа да се најде едно решение на нехомогената равенка кое го нарекуваме *партикуларно решение* на равенката. Тогаш за општиот облик на решението на нехомогената равенка 1.11 ја имаме следнава теорема:

ТЕОРЕМА 1.1*:*

Ако $a_n^{(p)}$ е партикуларно решение на нехомогената линеарна рекурентна равенка со константни коефициенти од ред k дадена со равенката (1.11)*, тогаш нејзино решение е секое*

решение во форма $\left\{a_n^{(p)} + a_n^{(h)}\right\}$, *каде* $\left\{a_n^{(h)}\right\}$ *е решение на соодветната хомогена линеарна равенка.*

Доказ*:* Нека b_n е друго решение на нехомогената линеарна равенка. Тогаш важи

$$b_n = c_1 \cdot b_{n-1} + c_2 \cdot b_{n-2} + \cdots + c_k \cdot b_{n-k} + F(n).$$

Оттука важи и:

$$b_n - a_n^{(p)} = c_1 \cdot (b_{n-1} - a_{n-1}^{(p)}) + \cdots + c_k \cdot (b_{n-k} - a_n^{(p)}) + F(n).$$

Па $b_n - a_n^{(p)}$ е едно решение на соодветната хомогена равенка, пример $a_n^{(h)}$. Следи,

$$b_n = a_n^{(p)} + a_n^{(h)}.$$ □

Сега проблемот се сведува на тоа да се најде едно партикуларно решение и прашањето е како може да најдеме едно такво решение, односно дали постои рецепт за наоѓање на барем едно партикуларно решение? Во општ случај не, но за некои типови на функции имаме начин како да најдеме едно решение. Еден од тие типови функции е полиномна функција со константни коефициенти помножена со константа на n-та степен. Со следнава теорема го даваме обликот на тоа партикуларно решение и ќе објасниме како се добива решението за специфична дадена равенка.

ТЕОРЕМА 1. 2:

Нека $\{a_n\}$ *ја задоволува нехомогената линеарна рекурентна равенка со константни коефициенти од ред* t 1.*8 и форма*

$$F(n) = (b_t n^t + b_{t-1} n^{t-1} + \cdots + b_1 n + b_0) s^n$$

каде b_i *и* s *се реални броеви.*

Кога s не е корен на карактеристична равенка на соодветната хомогена равенка, тогаш постои партикуларно решение во форма

$$p_t n^t + p_{t-1} n^{t-1} + \cdots + p_1 n + p_0$$

Кога s е карактеристичен корен на соодветната хомогена равенка, тогаш постои партикуларно решение во форма

$$n^m (p_t n^t + p_{t-1} n^{t-1} + \cdots + p_1 n + p_0) s^n.$$

Пример 1.13: Да се реши рекурентната релација

$$A[n] = 3A[n-1] + 2n,$$

кога $A[1] = 3$.

Решение: Решение на соодветната хомогена равенка е $a_n^{(h)} = a_1 3^n$.

Бараме едно решение на нехомогената равенка од облик $cn + d$, па овој израз го заменуваме во равенкта. Се добива:

$$cn + d = 3(c(n-1) + d) + 2n = (3c + 2)n + 3(d - c).$$

За да важи ова равенство треба да се најдат константите така што пред соодветните степени на n да имаме еднакви константи , т.е. да се реши системот:

$$\begin{cases} c = 3c + 2 \\ d = 3d - 3c \end{cases},$$

на која решенија се $c = -1$ и $d = -\frac{3}{2}$, па

$$a_n^{(p)} = -n - \frac{3}{2}.$$

Оттука, општото решение е

$$A[n] = -n - \frac{3}{2} + a_1 3^n.$$

Од почетните услови може да се добие константата a_1:

$$A[1] = 3 = -1 - \frac{3}{2} + a_1 3^1.$$

Од каде $a_1 = \frac{11}{6}$, па решението е

$$A[n] = -n - \frac{3}{2} + \frac{11}{6} 3^n.$$

Прашања и задачи

1. Реши ја рекурентната равенка $A[n] = 2A[n-1]$, со почетен услов $A[0] = 1$.
2. Реши ја рекурентната равенка $A[n] = 3A[n-1] - 2A[n-2]$ со почетни услови $A[0] = 1, A[1] = 3$.
3. Реши ја рекурентната равенка $A[n] = 4A[n-2]$ со почетни услови $A[0] = 1, A[1] = 6$.
4. Реши ја рекурентната равенка $A[n] = 4A[n-1] - 4A[n-2]$ со почетни услови $A[0] = 1, A[1] = 4$.
5. Реши ја рекурентната равенка

$$A[n] = -2A[n-1] + A[n-2] + 2A[n-3]$$

со почетни услови $A[0] = 0, A[1] = -1$ и $A[2] = -3$.

6. Реши ја рекурентната равенка

$$A[n] = -2A[n-1] + 3A[n-2] + 4A[n-3] - 4A[n-4]$$

со почетни услови $A[0] = 0, A[1] = 3, A[2] = -3$ и $A[3] = 9$.

7. Реши ја рекурентната равенка $A[n] = A[n-1] + 3$, со почетен услов $A[0] = 1$.
8. Реши ја рекурентната равенка $A[n] = 2A[n-1] + n \cdot 2^n$, со почетен услов $A[0] = 1$.
9. Реши ја рекурентната равенка $A[n] = A[n-1] + n \cdot 2^n$ со почетен услов $A[0] = 1$.

1.3 Ред на големина

Една од карактеристиките на алгоритмите според Кнут [3] е нивната брзина. Еден алгоритам е толку поефикасен колку што побргу ќе го реши проблемот за исти влезни податоци. Секако, кога некој проблем го решаваме со алгоритам обично се смета дека имаме голем влез, иако ова не мора да биде правило. Брзината на алгоритмите е всушност бројот на операции кои треба да се извршат за да се реши проблемот за даден влез со некој обем. Влезните податоци при секое извршување на алгоритамот најчесто се различни, и не секогаш алгоритамот би работел исто време, дури и за податоци од ист обем. Затоа се разгледува или колкаво време е потребно да се изврши алгоритамот кога податоците се „најнесреќни“, односно такви да тој би работел најдолго, што го нарекуваме најлош случај, или во просек од сите можни влезови, што го нарекуваме просечен случај.

За означување на брзината на растење на функциите во компјутерскиот јазик се користат следните симболи: “*o*” (е мало о од), “*O*” (е големо о од), *Θ* (е тета од), ~ (асимптотски тежи кон) и Ω (е омега од).

Мало о од:

Ќе речеме дека $f(x) = o(g(x))$ *, акко*

$$\lim_{x\to\infty} \frac{f(x)}{g(x)} = 0.$$

Мало о означува дека кога x *е доволно големо, програма која има сложеност* $f(x)$ *е супериорна, во однос на програма со сложеност* $g(x)$*.*

Еве неколку примери: $sinx = o(x)$, $\frac{1}{x} = O(1)$, а $\log x = o(0.01x)$.

Во третиот случај може да се забележи дека супериорноста на $logx$ во однос на $0.01x$ е за многу големи вредности на x, додека за помали вредности на x, $\log x$ е поголемо од $0.01x$. $\log x$ станува помало од $0.01x$, отприлика, за вредност на x поголема од 10290. Тоа е прилично голем број, па за помал влез, сепак побргу ќе работи програма која има сложеност $0.01x$.

Големо О од:

Ќе речеме дека $f(x) = O(g(x))$ *акко*

$$\exists C, x_0 \text{ такви што } (\forall x > x_0)|f(x)| < Cg(x).$$

Кога велиме дека $f(x) = O\big(g(x)\big), x \to \infty$, *сметаме дека функцијата* $f(x)$ *не расте побргу од* $g(x)$. *Со оваа нотација се дозволува овие две функции да имаат иста брзина на растење, но се дозволува и* $g(x)$ *да расте многу побргу од* $f(x)$.

Која е разликата меѓу мало и големо О? Мало о всушност ни дава попрецизна информација отколку големо О, затоа што тоа не само што ни кажува дека $|\mathrm{f(x)}| < \mathrm{Cg(x)}$ кога x тежи кон бескрај, туку и дека нивниот однос тежи кон 0. Всушност, секогаш кога важи мало о, важи и големо О. Според тоа, големо О се употребува кога знаеме дека некој алгоритам работи побргу отколку дадената функција, но не сме сигурни дали работи многу побргу или точно со брзината која е претставена со функцијата.

Еве два примери: $-3x^3 = O(x^4)$, $\frac{5x^6+x^3+1}{3x^3+x} = O(x^3)$.

Уште попрецизна информација за видот на растење на функцијата, кога x станува голем број дава функцијата Θ. Оваа функција ни кажува дека двете функции имаат иста брзина на растење, односно дека нивната брзина на растење се разликува само за константа.

Θ од:

Ќе речеме дека $f(x) = \Theta(g(x))$ *ако постојат константи* $c_1 > 0$, $c_2 > 0$ *и* x_0 *такви што* $(\forall x > x_0)$ *важи дека* $c_1 g(x) < f(x) < c_2 g(x)$.

На пример, x^2 е $\Theta(x^2 + x)$, $\frac{5x^6+x^3+1}{x^3+x}$ е $\Theta(5x^3)$ и $2x + \log x$ е $\Theta(2x)$.

Заради наједноставно претставување на растот на $f(x)$, за Θ најчесто ја земаме наједноставната функција, односно ги изоставаме константаните со кои би се множела функцијата и изоставаме збир од функции . На пример, точно е и $3x^2$ е $\Theta(10x^2)$, но константата 10 не ни дава никаква информација, па, најчесто ја изоставаме. Исто, $3x^2$ е $\Theta(x + x^2)$, но термот x исто така најчесто го гледаме како вишок и го изоставаме.

Ω од:

Ќе речеме дека $f(x) = \Omega(g(x))$ *акко постои* $\varepsilon > 0$ *и низа* $x_1, x_2, x_3 \to \infty$, *такви што*

$$\forall j \; |f(x_j)| > \varepsilon \, g(x_j).$$

На пример, $x^2 = \Omega(x)$, а $2x^2 sinx = \Omega(x^2)$.

Често оваа нотација се дефинира како обратна функција од O, односно, се дефинира дека $f(x) = \Omega(g(x))$, акко $g(x) = O(f(x))$. Но оваа дефиниција не е сосема адекватна за самото значење на нотацијата Ω. На пример, нека за некој проблем постојат два алгоритми, такви што за влез со големина n за едниот се потребни $f(n)$ пресметки, а за другиот се потребни $g(n)$. Нека за голем број на влезови со големина n, важи дека $f(n) < g(n)$, но секогаш, може да најдеме доволно големо n такво што $f(n) > g(n)$ f. Дали тогаш би рекле дека првиот алгоритам е подобар од вториот? Секако не, затоа што за да биде подобар треба да е подобар за сите можни влезови.

И оваа нотација многу често се користи при анализата на алгоритмите, посебно кога сакаме да потенцираме дека не е можно да се најде алгоритам кој би имал помала сложеност од некоја

конкретна функција. Да речеме најпознатиот алгоритам за множење на матрици од ред $n \times n$, има сложеност n^3, односно за да го пресметаме производот треба да се извршат n^3 операции множење на реални броеви, но, постојат алгоритми кои истото можат да го направат и за побрзо време. Сепак јасно е дека не може да се смисли алгоритам кој би имал сложеност помала од n^2, бидејќи само за да ги прочитаме елементите од матрите ни требаат $2n^2$ чекори. Дури и ова да го занемариме, јасно е дека на крај ќе мора да се испечати матрица со n^2 елементи. Па, оттука можеме да кажеме дека сложеноста на било кој алгоритам за множење на две матрици од ред n^2 е $\Omega(n^2)$.

За да имаме претстава колку е брз одреден алгоритам, од голема важност е да се направи подредување на функциите во однос на нивната сложеност. Јасно е дека $\log x$ послабо расте од било кој степен на x, колку и тој да е мал, што лесно може да се докаже со Лопиталово правило, техника која е наједноставно да се користи во било која ситуација кога сакаме да провериме и докажеме дека некоја функција е супериорна во однос на друга. Слично, $\log(\log x)$ =o($\log x$) и со секое додавање на логаритам добиваме помала сложеност. Но, постојат и функции кои растат побргу од $\log x$ а побавно од x, да речеме фукциите $(\log x)^a$, за кои исто така со Лопиталово правило може да се докаже дека

$$\lim_{x\to\infty} \frac{(\log x)^a}{x} \leq \lim_{x\to\infty} \frac{(\log x)^{\lceil a \rceil}}{x} \leq \lim_{x\to\infty} \frac{\lceil a \rceil !}{x} = 0.$$

Степенските функции можеме лесно да ги подредиме, бидејќи е јасно дека за $a < b, x^a = o(x^b)$. Јасно е и дека експоненцијалната функција е доминантна во однос на било која степенска фукција. При дизајнот на алгоритми, експоненцијалната сложеност се смета за многу голема сложеност, и голем труд се вложува да се најде алгоритам кој има сложеност која е супериорна во однос на експоненцијална функција, но, за голем број на проблеми ова не е можно де направи. Оваа сложеност е некаква граница до која може да се оди при дизајнирањето на алгоритам, и се смета дека ако некој алгоритам има поголема сложеност од

експоненцијална, веќе не е ефикасен за решавање на проблеми, освен за некои многу мали проблеми.

Експоненцијален раст на функција.

> *За една функција се вели дека расте експоненцијално ако постои константа* $c > 1$ *таква што* $f(x) = \Omega(c^x)$ *и константа* $d > 1$ *таква што* $f(x) = O(d^x)$.

Бидејќи алгоритмите со експоненцијална сложеност ги сметаме за бавни алгоритми, кои можат да се користат само за влезови со релативно мала големина, да се добие алгоритам со сложеност помала од експоненцијална за проблем за кој не бил познат алгоритам со ваква сложеност е големо откритие.

Ако функцијата е дадена со рекурзивна равенка, тогаш не мора да го најдеме егзактното решение за да ја пресметаме големината на раст на таа функција. На пример, ако имаме хомогена диференцијална равенка од прв ред, тогаш нејзиниот раст зависи од решенијата на соодветната карактеристична равенка. Имено, не мора да ги пресметуваме непознатите константи од почетните услови, односно тие самите воопшто не делуваат на растот на функцијата. Всушност, растот на функцијата е $\Theta(c^n)$, каде c е најголемото решение на карактеристичната равенка.

Пример 1.14. Да се најде растот на функцијата зададена со рекурзивната функција

$$A[n] = A[n-1] + A[n-2],$$

кога $A[0] = A[1] = 1$.

Решение: Ова е равенката од Пример 1.10, чија карактеристичната равенка беше $x^2 = x + 1$, и на која решенија се $\frac{1+\sqrt{5}}{2}$ и $\frac{1-\sqrt{5}}{2}$. Поголемото решение е $\frac{1+\sqrt{5}}{2}$, од каде следува дека растот на функцијата е $\Theta\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$.

Нехомогените рекурзивни равенки често се јавуваат како израз за сложеноста на алгоритмите со рекурзија, без да се користи техниката на динамичко програмирање. Овде растот на функцијата може да зависи или од константата, или од функцијата во нехомогениот дел на равенката, $F(n)$, односно партикуларното решение на равенка, но повторно не зависи од почетните услови. Имено, $A[n] = \Theta\left(\max\left(c^n, a_n^p\right)\right)$, каде c е најголемото решение на карактеристичната равенка а a_n^p е партикуларното решение на равенката. Во пракса поприродно е $F(n)$ да е помало од a_n^p затоа што тој член го брои бројот на чекори кои се јавуваат во рекурзивните алгоритми и најчесто е некој полином, па понатаму многу често ќе дискутираме дека рекурзивното решение има сложеност од облик $A[n] = \Theta(c^n)$. Но во секој случај важи дека $A[n] = \Omega(c^n)$

Пример 1.15. Да се најде растот на функцијата зададена со рекурзијата $A[n] = 3A[n-1] + 2n,\ A[1] = 3.$

Решение: Општото решение на равенката е

$$A[n] = -n - \frac{3}{2} + a_1 3^n$$

од каде следува дека растот на функцијата е $\Theta(3^n)$.

Пример 1.16. Да се најде $\Theta(A[n])$, каде $A[0] = 1$и

$$A[n] = 2A[n-1] + n \cdot 2^n.$$

Решение: Општото решение на равенката е

$$A[n] = 2^{n-1}(n^2 + n + 2),$$

од каде следува дека растот на функцијата е $\Theta(n^2 2^n)$.

Прашања и задачи

1. Кои од следниве тврдења се точни за функцијата дефинирана како $f(n) = n^2 + n + 2$?

   a) $f(n)$ е $\Theta(n^2)$

   b) $f(n)$ е $o(n^2)$

   c) $f(n)$ е $O(n^2)$

   d) $f(n)$ е $\Omega(n^2)$

   e) $f(n)$ е $\Theta(n^3)$

   f) $f(n)$ е $o(n^3)$

   g) $f(n)$ е $O(n^3)$

   h) $f(n)$ е $\Omega(n^3)$

   i) $f(n)$ е $\Theta(n)$

   j) $f(n)$ е $o(n)$

   k) $f(n)$ е $O(n)$

   l) $f(n)$ е $\Omega(n)$

   m) $f(n)$ е $\Theta(n^2 \ln n)$

   n) $f(n)$ е $o(n^2 \ln n)$

   o) $f(n)$ е $O(n^2 \ln n)$

2. Кои од следниве тврдења се точни за функцијата дефинирана како $f(n) = n^2(n + 2)$?

   a) $f(n)$ е $\Theta(n^2)$

   b) $f(n)$ е $\mathrm{o}(n^2)$

   c) $f(n)$ е $\mathrm{O}(n^2)$

   d) $f(n)$ е $\Omega(n^2)$

   e) $f(n)$ е $\Theta(n^3)$

f) $f(n)$ е $o(n^3)$

g) $f(n)$ е $O(n^3)$

h) $f(n)$ е $\Omega(n^3)$

3. Кои од следниве тврдења се точни за функцијата дефинирана како $f(n) = n^2 + n + 2^n$?

a) $f(n)$ е $\Theta(n^2)$

b) $f(n)$ е $O(n^2)$

c) $f(n)$ е $\Theta(2^n)$

d) $f(n)$ е $O(2^n)$

4. Кои од следниве тврдења се точни $f(n) = n \log n$?

a) $f(n)$ е $\Theta(n^2)$

b) $f(n)$ е $O(n^2)$

c) $f(n)$ е $\Theta(n)$

d) $f(n)$ е $O(n)$

5. Кои од следниве тврдења се точни $f(n) = (\log n)^2$?

a) $f(n)$ е $o(n)$

b) $f(n)$ е $\Omega(n)$

c) $f(n)$ е $\Theta(n)$

d) $f(n)$ е $O(n)$

6. Кои од следниве тврдења се точни $f(n) = \log_3 n$?

a) $f(n)$ е $\Theta(\log_2 n)$

b) $f(n)$ е $\Omega(\log_2 n)$

c) $f(n)$ е $\Theta(\log_4 n)$

d) $f(n)$ е $\Omega(\log_4 n)$

7. Кои од следниве тврдења се точни $f(n) = \log_3 n^2$?

a) $f(n)$ е $\Theta(\log_3 n)$

b) $f(n)$ е $\Omega(\log_3 n)$

c) $f(n)$ е $O(\log_3 n)$

d) $f(n)$ е $o(\log_3 n)$

8. Кои од следниве тврдења се точни?

   a) $(\log n)^2$ е $O(\log n^2)$

   b) $\log n^2$е $O(\log n)^2$

9. Најди го растот на функцијата $A[n]$ зададена со рекурентната релација $A[n] = 2A[n-1]$.

   a) Со почетни услови $A[0] = 1$.

   b) Со почетни услови $A[0] = 0$.

   c) За било кои почетни услови.

10. Најди го најголемиот можен раст на функцијата $A[n]$ зададена со рекурентната релација $A[n] = 3A[n-1] - 2A[n-2]$.

11. Најди го растот на функцијата $A[n]$ зададена со рекурентната релација $A[n] = 4A[n-1] - 4A[n-2]$ со почетни услови $A[0] = 1, A[1] = 4$.

12. Најди го најголемиот можен раст функцијата $A[n]$ зададена со рекурентната релација $A[n] = 4A[n-2]$.

13. Најди го растот на функцијата $A[n]$ зададена со рекурентната релација $A[n] = -2A[n-1] + A[n-2] + 2A[n-3]$, со почетни услови $A[0] = 0, A[1] = -1$ и $A[2] = -3$.

14. Најди го растот на функцијата $A[n]$ зададена со рекурентната релација $A[n] = A[n-1] + 3$.

15. Најди го растот на функцијата зададена со рекурентната релација $A[n] = -2A[n-1] + 3A[n-2] + 4A[n-3] - 4A[n-4]$.

a) За било какви почетни услови.

b) Што треба да важи за почетните услови за да растот на функцијата биде помал отколку во најлошиот случај?

16. Најди го растот на функцијата $A[n]$ зададена со рекурентната релација $A[n] = 2A[n-1] + n \cdot 2^n$.

a) Со било какви почетни услови

b) Дали растот зависи од почетните услови?

17. Најди го растот на функцијата $A[n]$ зададена со рекурентната релација $A[n] = A[n-1] + n \cdot 2^n$.

a) Со било какви почетни услови

b) Дали растот зависи од почетните услови?

18. Најди го растот на функцијата $A[n]$ зададена со рекурентната релација $A[n] = 2A[n-1] + 1$.

a) Со било какви почетни услови

b) Дали растот зависи од почетните услови?

1.4 Мастер теорема

Во Глава 1.2 разгледавме линеарни рекурзивни равенки, а во овој дел ќе разгледаме друг тип на рекурзивни равенки, кои најчесто се јавуваат во така наречените раздели па владеј алгоритми, како што се брзите алгоритми за сортирање или брзо степенување. Ваквите алгоритми не се од интерес за оваа книга сами по себе, но во некои од нив се користат структури на податоци кои се во суштина бинарни пребарувачки дрва, кои ќе ги разгледуваме во делот 1.6. од оваа глава, а значително ја зголемуваат ефикасноста на соодветните алгоритми. Дел од проблемите кои ќе ги разгледаме пак се базираат на конструкција на најоптимални бинарни пребарувачки дрва, па разбирањето на рекурзивната врска која ќе се разгледува во овој дел е од круцијално значање за нив.

Нека треба да се реши проблем со големина n и да претпоставиме дека тој може да се реши така што ќе се раздели на k потпроблеми, а i -тиот потпроблем е со големина n_i. Дополнително, нека за разделување на проблемот и спојување на решенијата за помалите инстанци се потребни уште $f(n)$ операции. Тогаш бројот на операции $T[n]$ потребни да се реши целиот проблем е:

$$T[n] = f(n) + \sum_{i=1}^{k} T[n_i].$$

Ако потпроблемите се со еднаква или скоро еднаква големина, $\frac{n}{b}$, рекурентната врска ќе биде од облик:

$$T[n] = f(n) + aT\left[\frac{n}{b}\right]. \qquad (1.12)$$

Ова не можеме да го решиме експлицитно за секоја вредност за n, но во дизајн на алгоритми тоа и не е многу битно, поточно повеќе не интересира кој е растот на оваа функција. Тоа го дава следнава теорема

ТЕОРЕМА 1. 3 *(Мастер теорема за полиноми)*

Нека $T[n]$ е монотоно растечка функција која за целите броеви $a \geq 1$ и $b > 1$ и реален број $c > 0$ 0, ја задоволува релацијата 1. 12 *и $T[1] = c$. Нека $f(n) = \Theta(n^d)$ за $d \geq 0$, тогаш:*

$$T[n] = \begin{cases} \Theta(n^d), & a < b^d \\ \Theta(n^d \log n) & a = b^d. \\ \Theta(n^{\log_b a}) & a > b^d \end{cases} \quad (1.\ 13)$$

Пример 1. 17: Да се најде растот на функцијата $T[n]$ зададена со

$$T[n] = \frac{n^2}{2} + n + T\left[\frac{n}{2}\right]$$

Решение: Имаме $a = 1, b = 2, d = 2,$ и бидејќи 1 < 2^2 , важи првиот услов и се применува првиот случај. Добиваме дека $T[n] = \Theta(n^2)$.

Пример 1. 18: Да се најде растот на функцијата $T[n]$ зададена со

$$T[n] = \sqrt{n} + 2T\left[\frac{n}{4}\right].$$

Решение: За оваа равенка $a = 2, b = 4, d = \frac{1}{2}$. Од $a = 2 = \sqrt{4}$, важи вториот услов и се применува вториот случај, па добиваме дека $T[n] = \Theta\left(\sqrt{n}\log n\right)$.

Пример 1. 19: Да се најде растот на функцијата $T[n]$ зададена со

$$T[n] = n + 4T\left[\frac{n}{2}\right].$$

Решение: За оваа равенка $a = 4, b = 2, d = 1$. Од $2 = \sqrt{4}$, важи третиот услов и се применува третиот случај, па добиваме дека $T[n] = \Theta\left(n^{\log_2 4}\right) = \Theta(n^2)$.

ТЕОРЕМА 1. 4 *(Мастер теорема генерален случај)*

Нека $T[n]$ *е монотоно растечка функција која за целите броеви a ≥ 1 и b > 1 и реален број c > 0, ја задоволува релацијата* 1. 12 *и* $T[1] = c$, *тогаш:*

$$T[n] = \begin{cases} \Theta(f(n)), & f(n) = \Omega(n^{log_b a}) \\ \Theta(f(n)\, log\, n) & f(n) = \Theta(n^{log_b a}). \\ \Theta(n^{log_b a}) & f(n) = o(n^{log_b a}) \end{cases} \qquad (1.\ 14)$$

Пример 1. 19: Да се најде растот на функцијата $T[n]$ зададена со

$$T[n] = \log\log n + T\left[\frac{n}{2}\right].$$

Решение: Бидејќи $\log\log n = \Omega\left(n^{\log_2 1}\right) = \Omega(n^0) = \Omega(1)$ добиваме дека $T[n] = \Theta(\log n)$.

Пример 1. 20: Да се најде растот на функцијата $T[n]$ зададена со

$$T[n] = \log n + 2T\left[\frac{n}{2}\right].$$

Решение: Бидејќи $\log n = \mathrm{o}\left(n^{\log_2 2}\right) = \mathrm{o}(n)$, добиваме дека растот на функцијата е $T[n] = \Theta(\log n)$.

Прашања и задачи

1. Најди го растот на функцијата $T[n]$ зададена со рекурентната релација

   a) $T[n] = n\sqrt{n} + n + 2T\left[\frac{n}{2}\right]$.

   b) $T[n] = n\sqrt{n} + n + 3T\left[\frac{n}{2}\right]$.

   c) $T[n] = n\sqrt{n} + n + 4T\left[\frac{n}{2}\right]$.

2. Најди го растот на функцијата $T[n]$ зададена со рекурентната релација

   a) $T[n] = n\log n + 3T\left[\frac{n}{3}\right]$.

   b) $T[n] = n\log n + 4T\left[\frac{n}{3}\right]$.

3. Во купче од n парички има една која е полесна од останатите. Дади рекурзивна равенка за најголемиот број на чекори и изведи го растот на добиената функција, ако полесната паричката се бара со помош на вага со два таса со следниов алгоритам:

   a. Паричките се делат на два еднакви делови, ако не може една паричка се остава надвор и ако вагата е во рамнотежа се заклучува дека полесната паричка не е на вагата, инаку истата постапка се повторува со купот парички кој е полесен.

   b. Паричките се делат на три скоро еднакви делови,при што два од деловите се еднакви. Двата еднакви делови се мерат на вагата, и ако вагата е во рамнотежа се мери купчето кое не е на вагата, инаку истата постапка се повторува со купот парички од вагата кој е полесен.

4. Што можеш да заклучиш ако го споредиш растот на двете функции од задача 3?

1.5 Графови

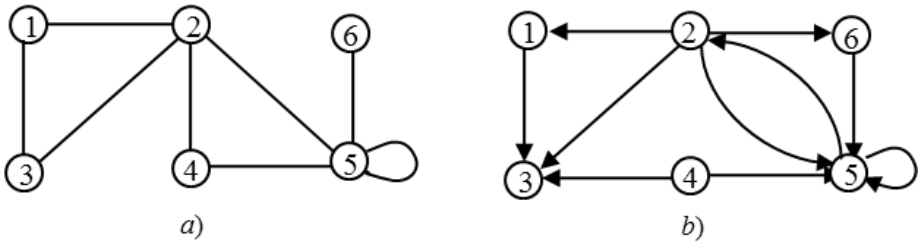
Во многу реални ситуации потребно е да се претстави релација помеѓу два елементи и еден едноставен начин да се направи тоа е со помош на графови. Графовите го визуелизираат проблемот и така помагаат за полесно анализирање и решавање на истиот. Дел од таквите проблеми се решаваат со техника на динамичко програмирање, па ќе бидат анализиран во седмата глава од оваа книга. Заради тоа, во оваа глава ќе се осврнеме на основната теорија за графови, ќе дефинираме што е граф, ги дадеме основните поими кај графови и ќе ги разгледаме типовите на графови постојат.

Неориентиран граф, или само граф, $G = (V, E)$ е подреден пар од множества: множество V, непразно множество од ***темиња*** (или јазли) и множество E, множество од ***ребра***. Со секое ребро во графот може да бидат поврзани или едно или две темиња, па секое ребро се прикажува како множество од две темиња. Овие темиња се нарекуваат крајни точки за реброто. За едно ребро се вели дека ги поврзува неговите крајни точки. Ако реброто поврзува две исти темиња тоа се нарекува ***алка***. Ако u и v се крајни точки на некое ребро во G, тогаш се нарекукуваат ***соседни*** темиња или соседи во G. Ако множеството V на графот G е конечно, грфот се нарекува конечен граф, а ако е бесконечно графот се нарекува бесконечен граф. Во проблемите кои се решаваат со алгоритми, графовите колку и да имаат голем број на темиња, секогаш се конечни. Во општ случај еден граф може да има и повеќе од едно ребро кое поврзува две исти темиња, и тие ребра ги нарекуваме ***паралелни ребра***. Граф кој нема алки и нема паралелни ребра се вика ***едноставен граф***, додека граф кај кој се дозволени паралелни ребра, се нарекува ***мултиграф***. Граф кај кој се дозволени и лупи и паралелни ребра се нарекува ***псевдограф***.

Ако графот е едноставен, тогаш секое ребро e кое ги поврзува темињата u и v може да се запише како $e = \{u, v\}$. Но исто така и граф со алки, кај кој нема паралелни алки, може лесно да се запише со помош на темињата. Ако реброто e го поврзува темето u со самото

него, тогаш ќе пишуваме $e = \{u, u\}$. На Слика 1.1 а) е даден неориентиран граф. Ребрата на овој граф се: $\{1, 2\}$, $\{1, 3\}$, $\{2, 3\}$, $\{2, 4\}$, $\{2, 5\}$, $\{5, 6\}$, $\{5, 5\}$, па графот се претставува како: $G = (\{1, 2, 3, 4, 5, 6\}, \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{2, 4\}, \{2, 5\}, \{5, 6\}, \{5, 5\}\})$.

Ориентиран граф или дограф е повторно подреден пар $G = (V, E)$, каде V е непразно множество од темиња (или јазли), а E е множество од ориентирани или насочени ребра. И овде со секое ребро во графот може да бидат поврзани или едно или две темиња, но разликата е во тоа што во овој случај е битно кое е прво, а кое второ теме на реброто. Ваквите ребра графички можат да се прикажат како стрелки. Секоја стрелка може да се запише како подреден пар од две темиња, со тоа што првиот член е темето од кое тргнува стрелката, а второто темето кон кое покажува стрелката. Првото теме се нарекува почетно теме, а второто крајно теме на реброто. Ако (u, v) е ребро во ориентираниот граф G, ќе речеме дека *u* е *сосед кон* v, а *v* е *сосед од u*. И ориентираниот граф може да содржи лупи или дупли ребра, па аналогно како и кај неориентирани графови, граф кој нема лупи и нема дупли ребра се вика ***едноставен граф*** или *едноставен ориентиран граф* кај кој се дозволени дупли ребра, се нарекува ***ориентиран мултиграф***, додека оној кај кој се дозволени и лупи и дупли ребра се нарекува ***ориентиран псевдограф***. Ако во графот не се дозволени повеќе од едно ребро кои поврзуваат исти темиња, тогаш реброто *e* кое го поврзува темето *u* со темето *v* можеме да го запишеме како $e = (u, v)$. Ако реброто *e* е лупа, односно поврзува темето *u* со самото него, тогаш пишуваме $e = (u, u)$. На Слика 1.1 б) е даден еден ориентиран граф. Ребра на овој граф се: (1, 3), (2, 1), (2, 3), (2, 5), (2, 6), (4, 3), (4, 5), (5, 5) и (5, 5), па графот е G = ({1, 2, 3, 4, 5, 6}, {(1, 3), (2, 1), (2, 3), (2, 5), (2, 6), (4, 3), (4, 5), (5, 5), (5, 5)}). Вообичаено, еден граф има или само ориентирани или само неориентирани ребра, но може да се случи да имаме граф и со едниот и со другиот тип на ребра. Таквите графови се нарекуваат ***мешани графови***.



Слика 1.1. а) Неориентиран граф (граф). b) Ориентиран граф (дограф)

За да можеме да направиме анализа која структура за репрезентираме влезните графови е подобра во даден момент, ќе дефинираме уште неколку поими поврзани со графови. ***Степен на теме*** во неориентиран граф е бројот на ребра кои излегуваат од него, т.е. бројот на ребра на кој тоа теме е крајна точка. Степенот на теме v се бележи со $\deg(v)$. Алките се бројат по два пати, еднаш за почетното и еднаш за крајното теме. Кај едноставни графови, темињата со степен 0 се нарекуваат изолирани темиња, затоа што не се сврзани со остатокот од графот. На пример за графот од Слика 1.1. *а*) темето 4 има степен 2, додека темето 5 има степен 5. Јасно е дека сумата на степените на секое теме е двапати по бројот на ребра, затоа што едно ребро го зголемува степенот на двете темиња кои ги поврзува.

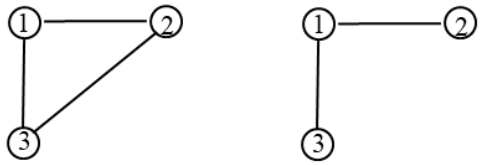
Во ориентиран граф не можеме да дефинираме само степен на теме, затоа што овде се прави разлика дали реброто влегува во темето или излегува од темето. Затоа дефинираме два степени, *влезен степен* на теме, како бројот на ребра за кои тоа теме е крајно теме и *излезен степен* на теме како број на ребра за кои тоа теме е почетно теме. Излезниот степен на теме v се бележи со $\deg^-(v)$, влезниот со $\deg^+(v)$. На пример за графот од Слика 1.1 *b*) темето 4 има влезен степен 1 и излезен степен 1, а темето 2 има влезен степен 1, а излезен степен 4. За ваквите графови е јасно дека сумата на влезните степени и сумата на излезните степени на сите темиња е еднаква на бројот на ребра, затоа што едно ребро го зголемува излезниот степен на почетното теме и влезниот степен на крајното теме.

Подграф на даден граф $G(V,E)$ е граф $H(W,F)$, каде $W \subseteq V$ и $F \subseteq E$. Подграфот H е ***правилен подграф*** ако е различен од G.

Графовите на Слика 1.5.2 се подграфови на графот од Слика 1.5.1 *a*). За дадено множество од темиња $W \subseteq V$ на даден граф $G(V, E)$ ќе речеме дека $H(W, F)$ е индуциран од графот G акко

$$F = \{(u, v) | u, v \in E\}.$$

Всушност индуциран граф значи да се земат сите ребра од графот G кои претходно постоеле помеѓу темињата од W, а подграфот не мора да ги содржи сите тие ребра. Така графот на Слика 1. 2 *a*) е индуциран граф на графот од Слика 1. 2, а графот под *b*) не е индуциран граф.



a) *b)*

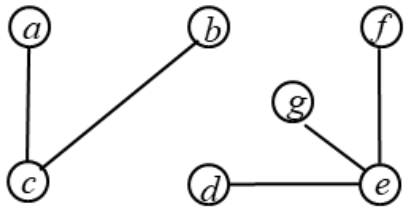
Слика 1. 2. Неориентирани графови подграфови на графот од Слика 1.1 a). Графот под a) е само индуциран граф, а графот под b) е само подграф.

Пат во граф е низа од ребра која започнува во некое теме во графот и патува од теме до теме преку ребрата. ***Должината на патот*** е бројот на ребра низ кои поминува. Поформално, пат со должина n од темето u до темето v во неориентираниот граф G е низа од n ребра $e_1, e_2, \dots, e_n$ од G, така да $e_1 = \{v_0, v_1\}, e_2 = \{v_1, v_2\}, \dots, e_n = \{v_{n-1}, v_n\}$, каде $v_0 = u, v_1 = v$. Патот кај едноставен граф може да се обележи и како $\langle v_0, v_1, \dots, v_n \rangle$ и велиме минува низ темињата $v_0, v_1, \dots, v_n$ и преминува преку $e_1, e_2, \dots, e_n$. Од друга страна, во ориентиран граф G, пат со должина n од темето u до темето v во е низа од n ребра $e_1, e_2, \dots, e_n$ од G, така да $e_1 = (v_0, v_1), e_2 = (v_1, v_2), \dots, e_n = (v_{n-1}, v_n)$, каде повторно $v_0 = u, v_1 = v$. И кај ориентиран едноставен граф патот може да се обележи и како $\langle v_0, v_1, \dots, v_n \rangle$ и велиме минува низ темињата $v_0, v_1, \dots, v_n$ и преминува преку $e_1, e_2, \dots, e_n$.

Ако темето u е поврзано со пат со темето v, велиме дека u е ***дофатливо*** од v или v е дофатливо од u и пишуваме $u \sim v$. Ако сакаме да напоменеме дека дофатливоста е како резултат на патот p, пишуваме $u \sim^p v$.

Потпат на патот е низа од n ребра $e_1, e_2, \dots, e_n$ или $v_0, v_1, \dots, v_n$ е низа од ребра $e_i, e_2, \dots, e_j, 1 \leq i \leq j \leq n$, или низа од темиња $\langle v_i, v_1, \dots, v_j \rangle, 0 \leq i \leq j \leq n$.

Еден пат е ***прост или едноставен***, ако не содржи едно теме повеќе од еднаш, односно не поминува по едно ребро двапати. На пример, во графот на Слика 1.1 а) патот $\langle 2, 4, 5 \rangle$ е прост, додека патот $\langle 2, 5, 2, 5 \rangle$ не е прост. Ако патот почнува и завршува во исто теме тогаш велиме дека тој пат е ***циклус***. За еден циклус ќе речеме дека е ***едноставен или прост циклус*** ако сите темиња освен првото и последното теме се различни.



Слика 1. 3. Неориентиран несврзан граф

Ако во неориентиран граф постојат два различни патишта од едно до друго теме, тогаш тој граф има барем еден прост циклус. На пример, во графот на Слика 1.1 а) има два патишта од темето 2 до темето 5, едниот е $\langle 2, 5 \rangle$, а другиот $\langle 2, 4, 5 \rangle$, па $\langle 2, 4, 5, 2 \rangle$ е прост циклус. Од друга страна Графот на Слика 1. 2 b) е ацикличен, затоа што не содржи ниту еден циклус.

За ориентиран граф велиме дека е ***ацикчичен*** ако не содржи ниту еден циклус. Таквиот граф го нарекуваме ориентиран ацикличен граф или скратено се именува со ***оаг***. Графот на Слика 1.1 *b*) не е ацикличен, затоа што го содржи, на пример, циклусот $\langle 2, 6, 5, 2 \rangle$, но ако го графот индуциран од темињата 1, 2 и 3 ќе биде ацикличен. ***Тополошко подредување*** на ориентиран ацикличен граф е линеарно

подредување на неговите темиња така што за секое насочено ребро (u, v) во подредувањето темето u доаѓа пред темето v. Тополошко подредување е можно само ако и само ако графот нема циклуси, односно ако е ориентиран ацикличен график (оаг). Секој оаг има барем едно тополошко подредување.

Комплетен граф со n темиња, K_n , е едноставен неориентиран граф кој содржи точно едно ребро меѓу било кои две различни темиња. ***Циклус*** со n темиња, C_n, е неориентиран граф кој се состои од n темиња $v_1, v_2, \dots, v_n$ и ребра $\{v_1, v_2\}$, $\{v_2, v_3\}$, …, $\{v_{n-1}, v_n\}$, $\{v_n, v_1\}$.

Едноставен неориентиран граф се нарекува ***бипартен граф*** ако множеството на ребра V може да се подели на две множества V_1 и V_2, така што секое ребро поврзува едно теме од V_1 со теме од V_2. Парот од множества (V_1, V_2) се нарекува ***бипартиција*** на множеството темиња.

Еден неориентиран граф е сврзан ако постои пат меѓу секој пар на различни темиња од графот. Графовите на Слика 1. 2 се сврзани графови, додека графот на Слика 1. 3 не е сврзан. Неориентиран сврзан граф кој што нема циклус се нарекува ***дрво***, а ако не е сврзан се нарекува ***шума***. Така графот на Слика 1. 3 е шума, а подграфот од темињата *a, b* и *c* е дрво. Теорема која е очигледна но многу корисна во алгоритмите за наоѓање на најкраток пат во граф е теоремата за прост пат:

ТЕОРЕМА 1.5:

Меѓу секој пар темиња во сврзан граф постои едноставен пат.

Доказ: Пат меѓу било кои две темиња u и v мора да постои, нека е тоа $\langle v_0, v_1, \dots, v_n \rangle$. Ако овој пат не е прост некое теме се повторува двапати, пример $v_i = v_j$. Овој циклус можеме да го извадиме од патот, односно можеме да го извадиме делот $\langle v_i, \dots, v_{j-1} \rangle$ и тоа што ќе остане е пак пат од u до v. Ова можеме да го правиме се додека има циклус на патот и она што ќе остане ќе биде прост пат. □

Сврзана компонента на еден неориентиран граф G е сврзан подграф на G, кој не е подграф на друг сврзан подграф на G. Всушност

тоа е максимален сврзан подграф на G. Подграфот индуциран од темињата *a, b* и *c* на Слика 1. 3 е сврзана компонента. Граф кој не е сврзан има повеќе сврзани компоненти, а сврзаните графови се состојат само од една сврзана компонента. Графот на Слика 1. 3. има 2 сврзани компоненти.

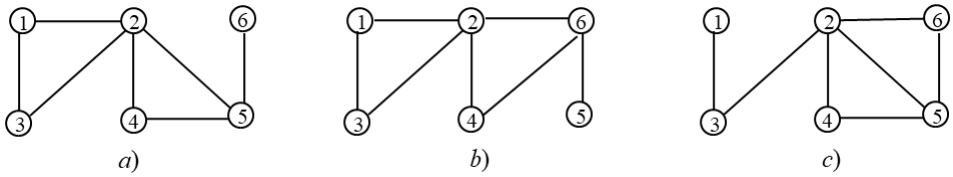
Ако во сврзаниот графот постои теме кое кога ќе го отстраниме, заедно со ребрата иницирани од него, ќе добиеме несврзан граф се нарекува ***теме на засек или точка на артикулација***. Точки на артикулација во графот од Слика 1.1. *a*) се темињата 2 и 5. Ако со тргање на едно ребро од сврзан граф, тој станува несврзан, тоа ребро се нарекува ***ребро на засек или мост***. На Слика 1.1. *a*) има само еден мост, реброто {5, 6}.

Ако во неориентиран граф има пат од темето u до темето v, тогаш истиот тој пат но наопаку е пат од темето v до темето u. Тоа не е случај кај ориентирани графови и затоа за нив дефинираме 2 типа на сврзаност, слаба и силна сврзаност. За еден ориентиран граф ќе речеме дека е ***силно сврзан*** ако за секој пар темиња u и v постои пат и од u до v и од v до u. Од друга страна за ориентиран граф велиме дека е ***слабо сврзан*** ако постои пат меѓу секој пар на различни темиња во соодветниот неориентиран граф (граф во кој секое ориентирано ребро ќе се замени со неориентирано ребро. Графот на Слика 1.1 *b*). Максимален цврсто сврзан подграф на ориентиран граф G, подграф во кој постои пат меѓу секој пар темиња се нарекува ***силно сврзана компонента или цврсто сврзана компонента***. Да потенцираме дека силно сврзана компонента не може да биде подграф од друга силно сврзана компонента. На пример во графот на Слика 1.1. *b*) една цврсто сврзана компонента е составена од темињата 2, 5 и 6. Еден ориентиран граф е цврсто сврзан ако и само ако се состои од една цврсто сврзана компонента.

Два едноставните графови $G_1 = (V_1, E_1)$ и $G_2 = (V_2, E_2)$ се ***изоморфни*** ако постои биекција $f: V_1 \to V_2$ со својството дека ако има ребро од u до v во G_1, тогаш во G_2 има ребро меѓу $f(u)$ и $f(v)$. Ваквата функција се нарекува изоморфизам. Со други зборови, два графа се изоморфни ако од едниот граф може да се добие другиот граф. На пример на Слика 1.4 графовите под *a*) и *b*) се изоморфни, и изоморфизмот е дефиниран со пресликувањето:

$$f: \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & 2 & 3 & 4 & 6 & 5 \end{pmatrix}.$$

Од друга страна, тие не се изоморфни со графот на Слика 1.4 *c*), иако имаат ист број на темиња ребра, имаат исто по едно теме со степен 1 и 4, едно теме со степен 3 и по 3 темиња со степен 2. Сепак, графот на Слика 1.4 *c*) има циклус со должина 4, а графовите графот на Слика 1.4. *a*) и *b*) немаат.



Слика 1.4. Три неориентирани графови со по 6 темиња и 7 ребра. Графовите под a) и b) се изоморфни меѓу себе, но не се изоморфни со графот под c)

Тешко е да се провери кога два графа се изоморфни и алгоритамот со груба сила би бил да се проверат сите $n!$ можни биекции дали се изоморфизам. За да се докаже дека два графа не се изоморфни, може да се најде некое својство кое го поседува едниот граф, а не го поседува другиот, а кое ќе биде непроменето со изоморфизам. Ваквите својства се нарекуваат граф инваријантни. На пример: ист број на ребра, ист број на темиња, ист број патишта со еднаква должина, исти циклуси и слично. Но дури и двата графа да ги поседуваат сите исти својства од овие не значи дека ќе бидат изоморфни. Сите познати алгоритми кои со сигурност можат да одговорат кога два графа се изоморфни, имаат експоненцијална сложеност и овој проблем спаѓа во познатата група од НП проблеми.

Прашања и задачи

1. Да се покаже теоремата за ракување, односно дека за секој неориентиран граф $G(V, E)$ важи

$$2|E| = \sum_{v \in V} \deg(v).$$

2. Колкав е најголемиот број на ребра во неориентиран едноставен граф со n темиња?

3. Докажи дека секој неориентиран граф $G(V, E)$ има парен број темиња со непарен степен.

4. Дали постои неориентиран едноставен граф со 5 темиња со степенски низи

   a. 1, 2, 3, 3, 4

   b. 0, 3, 3, 3, 3

   c. 2, 2, 3, 4, 5

   d. 1, 1, 2, 2, 2

5. Дали постои неориентиран едноставен сврзан граф со 5 темиња со степенски низи:

   a. 4, 4, 4, 4, 4

   b. 0, 3, 3, 3, 3

   c. 1, 1, 1, 1, 2

   d. 2, 2, 2, 2, 2

6. Да се покаже дека за секој ориентиран граф $G(V, E)$ важи

$$|E| = \sum_{v \in V} \deg^-(v) = \sum_{v \in V} \deg^+(v).$$

7. Колкав е најголемиот број на ребра во ориентиран едноставен граф со *n* темиња?

8. Нека е даден ориентиран граф $G(V,E)$. Формираме граф $G'(V',E')$ на следниов начин: Прво ги наоѓаме сите цврсто сврзани компоненти и за секокоја од нив ставаме теме во V'. Во E' ставаме ребро од темето u до темето v ако од цврсто сврзаната компонента која е репрезентирана со темето u постои барем едно ребро до цврсто сврзаната компонента која го репрезентира темето v. Покажи дека така добиениот граф е оаг, т.е. ориентиран ацикличен граф.

9. Нека u е теме со непарен степен во неориентиран граф. Покажи дека постои пат од *u* до некое друго теме со непарен степен.

10. Пат кој поминува низ секое ребро во графот точно по еднаш се нарекува Ојлеров пат. Ако патот завршува во темето од каде што почнал се нарекува Ојлеров циклус. Покажи дека за неориентиран сврзан граф $G(V,E)$ важи следново: во графот постои Ојлеров циклус ако и само ако сите темиња имаат парен степен.

11. Покажи дека во неориентиран сврзан граф $G(V,E)$ постои Ојлеров пат, а не постои Ојлеров циклус ако и само ако точно две темиња имаат непарен степен.

12. Комплементарен граф на неориентиран граф $G(V,E), \bar{G}(V',E')$ е граф за кој $V'=V, E'=\{\{u,v\}|u,v\in V, \{u,v\}\notin E\}$. Најди ги комплементарните графови на K_n и C_n!

13. Ако G има 15 ребра, а $\bar{G}$ има 13, колку темиња има $\bar{G}$?

14. Покажи дека темињата на мостовите се точки на артикулација ако и само ако не се со степен 1.

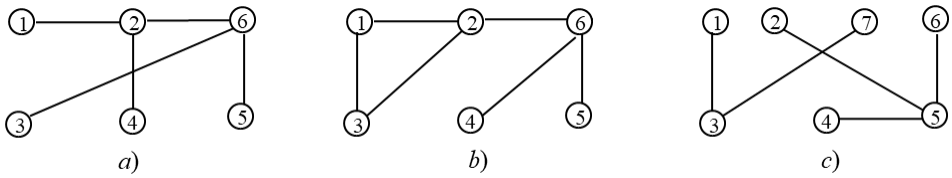
15. Покажи дека ако e е мост, тогаш постојат темиња u и v такви што секој пат од u до v минува низ e.

16. Покажи дека ако G содржи прост циклус, тогаш постојат темиња u и v меѓу кои има два различни патишта,.

17. Покажи дека ако има два различни патишта меѓу темиња u и v во графот G, тогаш G содржи прост циклус.

1.6 Дрва

Дрвата се структури кои многу често се појавуваат во дизајн на алгоритми, од една страна за да се решат проблеми со нив, од друга страна за да се решат проблеми со нивна помош. Но најчесто се користат како структури на податоци за чување на податоците со цел за нивен лесен и брз пристап, што најчесто и е срцето на ефикасноста на одредени алгоритми.



Слика 1. 5. а) е дрво, b) граф кој не е ни дрво ни шума, бидејќи го содржи циклусот ⟨1,2,3,1⟩. с) шума

Дрво претставува неориентиран сврзан граф во кој нема циклуси, нема повеќекратни ребра и нема алки, односно неориентиран ацикличен сврзан едноставен граф. Ако неориентираниот едноставен граф не е сврзан, но сепак е ацикличен се нарекува *дрво*. На Слика 1. 5, графот под *а*) е дрво, графот под *b*) на е дрво, тој е сврзан но го содржи циклусот $\langle 1, 2, 3, 1\rangle$. Графот под *с*) е ацикличен, но не е сврзан, затоа тој е шума.

Следната теорема кажува дека во дрво меѓу било кои две темиња постои единствен пат и со отстранување на било кое ребро, графот веќе нема да биде сврзан и обратно дека ако во некој граф помеѓу било кои темиња постои единствен пат или ако со отстранување на било кое ребро се добива граф кој што не е сврзан, тогаш тој граф е дрво. Исто така важи дека секое дрво има точно $|V| - 1$ но ребро, но уште повеќе дека секој сврзан граф со и има $|V| - 1$-но ребро или ацикличен граф со и има $|V| - 1$-но ребро, мора да биде дрво. На крај теоремата кажува дека ако во дрво се додаде

барем едно ребро, тогаш тоа ребро ќе формира циклус и обратно ако во некој граф со додавање на некое ребро секогаш се добива циклус, тогаш тој граф мора да биде дрво.

ТЕОРЕМА 1.6

За било кој неориентиран граф $G = (V, E)$ *следниве тврдења се еквивалентни:*

1. G *е дрво (сврзан и ацикличен).*

2. Било кои две темиња во G *се поврзани со единствен пат.*

3. G *е сврзан, но со отстранување на било кое ребро од* E*, ќе се добие несврзан граф.*

4. G *е сврзан и* $|E| = |V| - 1$.

5. G *е ацикличен и* $|E| = |V| - 1$.

6. G *е ацикличен, но ако се додаде било кое ребро во* E*, добиениот граф ќе содржи циклус.*

Доказ:

$1 \Rightarrow 2$: Ако G е дрво тогаш е сврзан граф, па било кои две темиња се поврзно со пат. Нека постојат две темиња *u* и *v* кои се поврзани со два различни патишта $p_1 = \langle u = v_0, v_1, \dots, v_n = v \rangle$ и $p_2 = \langle u = v_0, v'_1, \dots, v'_m = v \rangle$. Бидејќи овие два патишта се различни имаат барем едно теме кое во кое се разликуваат. Нека v_k е првото такво теме, т.е. нека $v_0 = v'_0, \dots, v_{k-1} = v'_{k-1}, v_k \neq v'_k$, а *x* е првото наредно теме во кое овие патишта се среќаваат (вакво теме мора да постои затоа што барем *v* е едно такво теме). Поточно, *x* е темето на потпатот од p_1 од v_k до v кое исто така припаѓа и на p_2. Сега имаме два патишта p'_1 и p'_2 од v_{k-1} до *x* кои немаат ниту едно друго исто теме освен v_{k-1} и *x*. Патот $p'_1 p'^R_2$, каде p'^R_2 е обратнот пат од p'_2, е

циклус. Значи G не би било дрво. Оттука следува дека претпоставката дека има два различни патишта е погрешна.

2 $\Rightarrow$ 3: Ако било кои две темиња се поврзани со единствен пат, тогаш за секое ребро $\{u, v\}$ тој единствен пат е реброто $\{u, v\}$. Ако го отстраниме тоа ребро, u и v веќе нема да бидат поврзани со пат, што значи графот веќе нема да биде сврзан.

3 $\Rightarrow$ 4: Доказот ќе го направиме со индукција. Едноставен граф со едно теме нема ребра и јасно е дека ќе го задоволува 4. Едноставен граф со две темиња има само едно ребро и јасно е дека го задоволува и 3, со отстранување на тоа ребро двете темиња веќе нема да се сврзани, а и 4 затоа што има едно ребро помалку отколку темиња. Нека сега имаме граф со $n > 2$ темиња, за кој што важи дека со отстранување на било кое ребро ќе се добие несврзан граф. Па нека отстраниме едно, било кое ребро. Тогаш ќе добиеме граф кој што не е сврзан и има 2 компоненти, едната со m темиња, другата со k, $m + k = n$. Бидејќи за секоја од компонентите важи 3, бидејќи ако не важи, тогаш нема да важи за целиот граф, од индуктивната претпоставка имаме дека во првата компонента има $m - 1$-но ребра, а во втората, $k - 1$-но ребро. Значи во тие две компоненти има вкупно $n - 2$ ребра, и плус реброто што го отстранивме се добива дека во целиот граф имало $n - 1$ ребро.

4 $\Rightarrow$ 5: Нека G е сврзан, $|E| = |V| - 1$ и има циклус, $\langle v_1, v_1, \dots, v_k \rangle$. Тој циклус се состои од точно k ребра. Со индукција ќе покажеме дека графот има барем $|V|$ ребра. Ако циклусот не е целиот граф, тогаш постои теме v_{k+1} кое не е во циклусот, но е поврзано со ребро во графот. Него можеме да го додадеме во графот заедно со неговото ребро. Така ќе добиеме сврзан подграф на G со $k + 1$ но ребро. Оваа постапка можеме да ја продолжиме и во i-тиот чекор ќе добиеме подграф со i ребра. Кога ќе го додадеме и последното теме ќе добиеме подграф на G со $|V|$ ребра има, што значи дека $|E| > |V| - 1$. Ова е контрадикција со претпоставката дека $|E| = |V| - 1$.

5 $\Rightarrow$ 6: Нека G е ацикличен со $|E| = |V| - 1$, но со додавање на ребро добиваме повторно ацикличен граф. Новодобиениот граф е шума и се состои од барем една сврзана компонента, $V_1, \dots, V_k, k \geq 1$. Уште повеќе, новодобиениот граф има $|V|$ ребра. Секоја компонента е

дрво и од 1 $\Rightarrow$ 4 мора да има $V_1 - 1$-но ребро. Така, вкупниот број на ребра во новиот граф е

$$|V_1| - 1 + \cdots . + |V_k| - 1 = |V| - k < |V|,$$

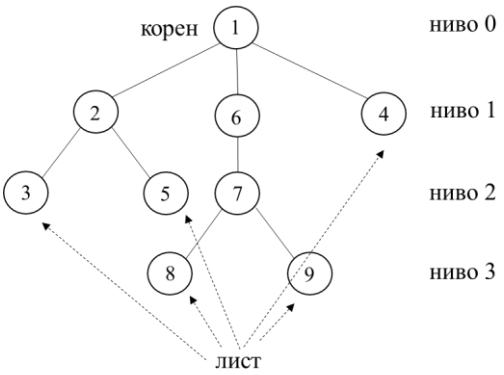
што е контрадикција со тврдењето дека новодобиениот граф има $|V|$ ребра. Оттука, мора со додавање на новото ребро да се добива циклус.

6 $\Rightarrow$ 1: Нека *G* е ацикличен и со додавање на било кое ребро се добива циклус. Треба да покажеме дека претходно графот бил и сврзан. Нека претпоставиме дека претходно графот не бил сврзан, т.е. се состоел од барем две сврзани компоненти. Тврдиме дека ако додадеме ребро од теме *u* од едната од тие компоненти до теме *v* во друга од тие компоненти, новиот граф нема да има циклус, затоа што ако има циклус, тогаш постоел пат од u до v кој не минува низ реброто $\{u, v\}$, што значи дека тие две темиња не биле во различни сврзани компоненти. Оттука следува дека пред да се додаде последното ребро, графот бил сврзан. □

Кореново дрво е дрво во кое едно теме е одделено од останатите и се означува како корен. Кореновите дрва ги цртаме така што најгоре го ставаме коренот, а останатите темиња доаѓаат подолу, со тоа што секој пат од коренот до останатите темиња е насочен одозгора – надолу, како на Слика 1. 6. На кореновото дрво на Слика 1. 6. коренот е темето 1.

Претци на темето v кое не е корен се сите темињата на патот од коренот до него, вклучувајќи го и коренот, со исклучок на самото теме, додека ***потомци*** на теме *v* се сите оние темиња на кои *v* им е предок. На темето 7 во кореновото дрво на Слика 1. 6 претци му се 6 и 1, а потомци 8 и 9.

Темето кое е последно на патот од коренот до темето v се нарекува ***родител*** на v. Коренот нема родители. ***Деца*** на дадено теме v се сите темиња на кои v им е родител. На темето 7 во кореновото дрво на Слика 1. 6. родител му е темето 6, а деца 8 и 9.



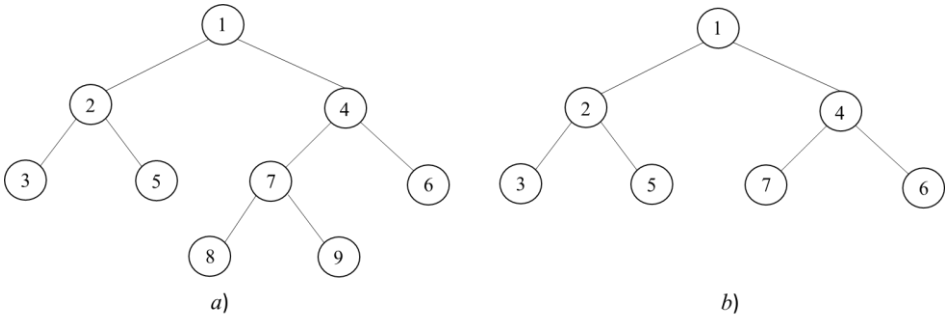
Слика 1. 6. Кореново дрво

Теме кое нема деца се нарекува ***лист***, додека темињата кои имаат деца се викаат ***внатрешни темиња***. Коренот се смета за внатрешно теме. Ако две темиња имаат ист родител, се нарекуваат сестрински темиња. Во дрвото на Слика 1. 6. Темињата 2, 4 и 6 се сестрински темиња.

Нека a е теме во дрво, ***поддрво со корен a*** е подграф од дрвото кој се состои од a, неговите потомци и сите ребра инцидентни со овие потомци. Поддрво со корен 2 на кореновото дрво од Слика 1. 6 е дрвото составено од темињата 2, 3 и 5.

Ниво на теме v во кореново дрво е должината на единствениот пат од коренот до него, односно бројот на ребра од коренот до него. Нивото на коренот е 0. Нивото на темето 2 од кореновото дрво на Слика 1. 6 е 1, додека на темето 8 е 3. ***Висина на кореново дрво*** е максимумот на нивоата на темињата, па дрвото од Слика 1. 6 има висина 3.

За едно кореново дрво велиме дека е ***подредено кореново дрво*** ако децата на секое внатрешно теме се подредени. Во ***подредено бинарно дрво***, ако внатрешно теме има две деца, тие се наречени ***лево дете*** и ***десно дете***. Поддрвото со корен во левото дете на едно теме се нарекува ***лево поддрво***, а поддрвото со корен во десното дете на едно теме се нарекува ***десно поддрво***.



Слика 1. 7. а) Целосно бинарно дрво, но не е комплетно. b) Комплетно бинарно дрво.

Едно кореново дрво се нарекува ***m-арно дрво*** ако секое внатрешно теме нема повеќе од m деца. m -арно дрво со $m = 2$ се нарекува ***бинарно дрво***, со $m = 3$ се нарекува ***тернарно дрво***. ***Целосно m -арно дрво*** е дрво во кое секое внатрешно теме има точно m деца, додека *комплетно m -арно дрво* е дрво на кое секое внатрешно теме има точно m деца и сите листови се на иста висина. Дрвото на Слика 1. 6. е тернарно дрво, но не е ниту целосно, ниту комплетно тернарно дрво. Дрвото на Слика 1. 7 а) е целосно бинарно дрво, но не е комплетно, додека дрвото на Слика 2.5.3 *b*) е комплетно, а со тоа и целосно.

Следнава Теорема ја дава врската меѓу бројот на листови, темиња и внатрешни темиња во целосно m-арно дрво:

ТЕОРЕМА 1.7.

Целосно m-арно дрво со i внатрешни темиња има $n = mi + 1$ темиња и $l = (m - 1)i + 1$ лисја.

Доказ: Јасно е дека за секое внатрешно теме имаме по m деца, а сите темиња освен коренот се деца на некое теме. Оттука $n = mi + 1$. Бидејќи бројот на листови е вкупниот број на темиња минус внатрешните темиња следува дека

$$l = i - n = i - (mi + 1) = (m - 1)i + 1.$$

□

Врз основа на оваа теорема лесно може да се изведе било која врска меѓу листовите, внатрешните темиња и вкупниот број на темиња во целосно m- арно дрво.

Напомнавме дека не постојат комплетни дрва за секој број на темиња и тоа е дадено со следнава теорема:

ТЕОРЕМА 1.8.

Целосно m-арно дрво со висина h има точно $\frac{m^{h+1}-1}{m-1}$ *темиња,* m^h *листови и* $\frac{m^h-1}{m-1}$ *внатрешни темиња.*

Доказ: Јасно е дека секое поддрво со корен во некое теме е исто така комплетно дрво. Тогаш за бројот на темиња ја имаме следнава рекурзија:

$$T[h] = 1 + mT[h-1].$$

со почетен услов $T[0] = 1$. Со решавање на оваа нехомогена рекурзивна равенка се добива дека $T[h] = \frac{m^{h+1}-1}{m-1}$ темиња. Рекурзијата која одговара на бројот на листови е:

$$L[h] = mL[h-1].$$

со почетен услов $L[0] = 1$, на која решение е $L[h] = m^h$. Оттука јасно е дека бројот на внатрешни темиња се $\mathrm{T}[h] - L[h] = \frac{m^h-1}{m-1}$.

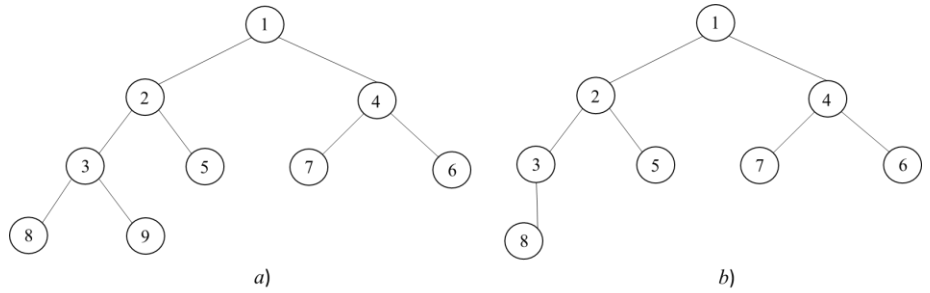
□

Врз основа на оваа теорема можеме да ја изведеме висината на комплетно m-арно дрво, ако е даден вкупниот број на темиња, бројот на листови или бројот на внатрешни темиња, што ќе биде оставено за вежба.

Кореновите дрва се една од најкорисните структури на податоци во дизајн на алгоритми, посебно бинарните дрва, затоа што

истите се наједноставни за користење. Структурата за работа со минимум и максимум базирана на подредено бинарно дрво, позната како бинарен куп е една од најкорисните структури која значително ги убрзува алгоритамот на Прим и алгоритамот на Дикстра кои се дадени во седмата глава. Тие се користат во многу други места за чување на податоци, бидејќи нудат брз и лесен пристап до податоците и во оваа книга ќе разгледаме некои проблеми во кои се бара да се дизајнира најоптималното дрво за чување на одреден тип на податоци. Целосните бинарни дрва се корисни во голем број апликации, но комплетните бинарни дрва се најдобра структура за брзо пронаоѓање на податоците.

Бидејќи не секогаш може да се дизајнира комплетно дрво, честопати се трудиме да дојдеме до дрво кое е блиску до комплетно, па таквите дрва ги нарекуваме скоро комплетни. *Скоро комплетно бинарно дрво* е комплетно пополнето на сите нивоа освен можеби на последното, кое е пополнето *од* лево до некое теме. Дрвото на Слика 1. 7 не е скоро комплетно, но ако темињата 8 и 9 се поместат како деца на темето 3, станува скоро комплетно, Слика 1. 8 *a*). Скоро комплетното бинарно дрво не мора да има само внатрешни темиња со по две деца, затоа што тоа понекогаш не е можно, Слика 1. 8 *b*).



Слика 1. 8. Скоро комплетни бинарни дрва

Важна карактеристика на скоро комплетните дрва е тоа што нивната висина е $\Theta(\log|V|)$ на кој факт се базраат многу брзи алгоритамски техники кои користат коренови дрва во некоја нивна рутина. Не е битно дали дрвата ќе бидат бинарни или тернарни, затоа што $\Theta(\log_a n) = \log_2 n$, па нема потреба да се усложнува работата и

секогаш е најдобро да се работи со бинарни дрва, затоа што тие се наједноставни. Ова е дадено со следнава теорема:

ТЕОРЕМА 1. 9.

Скоро комплетно m -арно дрво со n темиња има висина $\Theta(log\, n)$.

Доказ: Од Мастер теоремата имаме дека за секое поддрво со корен во некое теме, секое од *m*-те поддрва на тоа теме има скоро еднаков број на темиња, а првото такво поддрво има најголем број на темиња. Тоа поддрво е исто така скоро комплетно, па за висината на дрвото важи

$$H[n] = 1 + H\left[\left[\frac{n}{m}\right]\right]$$

Од Мастер теоремата $H[n]$ е $\Theta(\log n)$.

Прашања и задачи

1. Дали секој граф со n темиња и $n - 1$ ребро е дрво?
2. Дали секој ацикличен граф има помалку ребра од темиња?
3. Колкав е бројот на ребра во шума со 10 темиња и 4 дрва?
4. Колку дрва содржи шума со 10 темиња и 4 ребра?
5. Колкав е најголемиот степен кој може да го има теме во дрво со 10 темиња?
6. Колку најмногу темиња со степен 1 може да има дрво со 10 темиња?
7. Колку најмногу темиња со степен 1 може да има шума со 10 темиња?
8. Која е висината и арноста на кореновото дрво со корен 2 за дрвото на Слика 1. 5 *a*)?
9. Докажи дека во едно m -арно дрво со висина h постојат најмногу m^h лисја.
10. Докажи дека висината на комплетно m-арно дрво со l лисја е $h = \log_m l$.
11. Докажи дека ако едно m-арно дрво со висина h има l лисја тогаш, $h = \Omega(\log l)$.
12. Колкава е најголемата висина која може да ја има целосно бинарно дрво со n темиња?
13. Колкав е бројот на внатрешни темиња и листови во целосно тернарно дрво со 10 темиња?
14. Колкав е бројот на темиња и внатрешни темиња во целосно тернарно дрво со 7 листови?
15. Дали постои целосно тернарно дрво со 10 листови?

2 Еднодимензионални проблеми

Техниката на динамичко програмирање обично се применува во проблеми на оптимизација, односно во проблеми во кои се бара најдоброто решение во одредена ситуација, најчесто најкратко време, најмала цена или некоја друга дефинирана карактеристика. Овој тип на проблеми се такви што може да има многу решенија, односно до целта може да се стигне на повеќе начини, како на пример да се стигне од едно на друго место по некоја постпатна структура. Но секое вакво решение има некоја вредност, цена или време, како на пример време да се помине одреден пат во патната мрежа. Во оптимизационите проблеми се бара решение кое што има оптимална (најголема или најмала) вредност. Во примерот со патната структура тоа би било најмала цена или најкратко време. Во случај да постојат повеќе решенија кои што имаат оптимална вредност, обично се бара кое било од нив. Во една широка класа на проблеми на оптимизација, едно од оптималните решенија може да се најде користејќи го динамичкото програмирање.

Динамичко програмирање решава проблеми со комбинирање на решенијата на потпроблемите на кои може да се подели поголемиот проблем, и тоа може да се искористи само во ситуација кога потпроблемите не се независни и ја имаат истата природа како поголемиот проблем. Процедурата обично оди така што секој потпроблем се решава само по еднаш и добиениот резултат се зачувува, заради спречување на повторно пресметување на одговорот на потпроблемот секој пат кога е реповикан. Иако нема некој стриктен рецепт како овие проблеми се решаваат, сепак постапката на решавање може да се издефиренцира во четири редоследни чекори:

1. Карактеризирање на структурата на оптимално решение;
2. Рекурзивно дефинирање на оптималното решение;
3. Пресметување на вредноста на оптималното решение по принципот од долу кон горе;
4. Конструирање на оптимално решение од пресметаните податоци;

Чекорите од 1 – 3 се за, додека чекорот 4 се извршува само доколку се бара и реконструкција на едно оптимално решение. За да можеме да го реконструираме оптималното решение, обично треба да се запаметат некои податоци во текот на пресметките во чекорот 3.

Освен во проблеми со оптимизација, техниката на димнамичко програмирање се користи и за решавање на комбинаторни проблеми. Нив можеме да ги поделиме во две класи: проблеми со кои се брои и проблеми во кои се наоѓа редниот број на елемент од некоја низа или за даден реден број на елемент во некоја низа и правило како истата се генерира, да се најде тој елемент. Како илустрација да го разгледаме примерот со патишта во една патна мрежа. Проблемот да се најде најкраткиот пат од местото А до местото В е оптимизационен проблем, додека бројот на различни патишта, односно можности, да се стигне од местото А до местото В е стандарден комбинаторен проблем на броење. Ако пак имаме некое правило по кое се подредуваат различните патишта, тогаш е можно да имаме прашање да го најдеме k-тиот елемент во тоа подредување.

Најчесто идејата на која се базираат решенијата на вака поврзаните проблеми е многу слична, па една од главните во овој дел ќе се потрудиме и да ја објасниме ваквата врска помеѓу проблемите кои имаат слична природа.

Во оваа глава ќе разгледаме неколку поедноставни проблеми кои се базираат на рекурзивни равенки со една променлива, за чие пресметување ни требаат константен број на операции. Со првиот едноставен комбинаторен проблем сакаме да ја илустрираме постапката на изведување на рекурзивна равенка и начинот на

дизајнирање на рекурзивното решение. Овде ќе го спомнеме и концептот на мемоизација кој често се користи како стратегија на програмирање, но со истиот понатаму ретко ќе се занимаваме, затоа што тоа е повеќе начин на имплементација отколку развој на алгоритамската идеја. Вториот проблем кој ќе го разгледуваме е оптимизационен проблем и целта ни е да покажеме дека постои голема суштинска врска меѓу овие два типа на проблеми, со тоа што ќе ја потенцираме сличноста на решението на овој со првиот проблем. Третиот проблем, Време на извршување со две производствени ленти, репрезентира класа на оптимизациони проблеми кај кои оптималното решение се добива со избор од подобрата од две можности. Слична идеја ќе имаат и дел од проблемите кои ќе ги разгледуваме во наредното поглавје. Проблемот Шарено оро кој ќе го разгледаме во четвртиот дел од оваа глава, повторно се базира на идеја на комбинирање на две различни можности, но овде овие две можности или потпроблеми кои треба да се решат се суштински различни, суштински различно се решаваат, а различен е и начинот на нивно комбинирање. Последниот проблем, Телефони на шахисти, е генерализација проблемите чиј претставник е третиот проблем. Ова е комбинаторен проблем кој се базира на поголем систем од рекурзивни равенки.

Проблем 2. 1. Проблем на парички

Како прв проблем со кој ќе ја опишеме техниката на Динамичко програмирање избран е еден комбинаторен проблем, затоа што ваквите проблеми се базираат само на рекурзивна врска, а не на оптимизационо својство, што е значително поедноставно за анализа.

Дефинирање на проблемот

Пијалоците кои се порачуваат од автоматот за пијалоци, можат да се платат со железни парички, но само со монети од еден и два денари. Монетите се пуштаат во автоматот една по една. Ако цената на пијалокот кој го порачувате е n денари, на колку начини можете да ја платите точната сума, за дадено n?

Анализа на проблемот

Прво да анализираме што претставуваат различни начини на пуштање на низата парички. Ако на пример пијалокот кој го порачувате кошта 5 денари, тогаш можете да го платите со низата парички: 1, 1, 1, 1, 1, но и со 1, 2, 1, 1 или 1, 1, 1, 2, но постојат и други можности. Ова се различни начини на плаќање. Прашањето е колку вакви начини постојат? Можеме да забележиме дека во првиот и вториот случај последната паричка која се пушта во автоматот е еден денар и, пред таа паричка да се пушти, веќе имаме платено сума од 4 денари, по истиот принцип, со монети од еден или два денари. Од друга страна последната пуштена монета во третата комбинација е од два денари, па пред тоа била платена сума од 3 денари со монети од еден и два денари.

Нека бројот на начини со кои може да се плати сума од n денари го обележиме со $A[n]$. Тогаш сумата од 5 денари ќе може да се плати на $A[5]$ начини. Јасно е дека има само два начини да се избере последната монета, или еден, или два денари. Уште повеќе, секогаш кога последната монета е од еден денар, претходно сме платиле 4 денари, а тоа може да се направи на $A[4]$ начини. Слично, секогаш кога последната монета е од два денари, претходно сме платиле 3

денари, а тоа може да се направи на $A[3]$ начини. Оттука имаме дека $A[5] = A[4] + A[3]$.

Да се навратиме на рекурзивното решение на проблемот и да се потрудиме да ја пресметаме вредноста за 5. Користејќи дека $A[1] = 1$ и $A[2] = 2$, да пробаме да го пресметаме $A[5]$:

$$\begin{aligned} A[5] &= A[4] + A[3] = (A[3] + A[2]) + (A[2] + A[1]) \\ &= \big((A[2] + A[1]) + A[2]\big) + (A[2] + A[1]) \\ &= \big((2 + 1) + 2\big) + (2 + 1). \end{aligned}$$

Да напоменеме дека многу почесто почетните случаи почнуваат од 0 наместо од 1. Во оваа ситуација, бројот на начини да се плати сума од 0 денари е 1, а тоа е комбинацијата да не се пушти ниту една паричка. Многу често може да направи грешка при идентификацијата на големината на почетниот услов кога вредноста на n е 0. Така на пример во овој случај многумина повеќе би рекле дека $A[0]$ е 0, отколку што би рекле дека е 1. Поради тоа, секогаш е подобро да се провери дали оваа вредност одговара на рекурентната релација.

Овој начин на пресметување е со помош со рекурзивен алгоритам, кога функцијата која го решава проблемот се повикува самата себеси од помали вредности. Истото расудување можеме да го направиме и за било која друга сума на пари, n: бројот на начини на плаќање е еднаков на:

$$\begin{aligned} A[n] &= \text{број на начини со последна монета 1} \\ &\quad + \text{број на начини со последна монета 2} \\ &= A[n-1] + A[n-2]. \end{aligned} \tag{2. 1}$$

За ова да можеме да го пресметаме потребно е на рака, односно со груба сила, да го пресметаме бројот на начини за две најмали суми. Јасно дека сума од еден денар може да се плати само на еден начин, а сума од два денари на два начини (со низите 1,1 и 2).

Очигледно е дека равенката која ја добивме е линеарната хомогени равенка од втор ред од Пример 1.6, па решението може да се изведе и експлицитно, преку карактеристичниот полином на равенката и да се добие дека

$$A[n] = \frac{5+\sqrt{5}}{10}\left(\frac{1+\sqrt{5}}{2}\right)^n + \frac{5-\sqrt{5}}{10}\left(\frac{1-\sqrt{5}}{2}\right)^n.$$

Ако ја пресметуваме вредноста врз основа на оваа формула ќе ни биде потребно време O($\ln n$). Но мора да забележиме дека можат да се направат значителни нумерички грешки затоа што пресметките вклучуваат да се пресмета $\sqrt{5}$, кој не е цел број. Од друга страна, ваков пристап не може да се примени во случаи кога е тешко да се реши карактеристичната равенка. Заради тоа нема некоја голема придобивка со него, посебно што малку подоцна ќе дадеме техника со помош на матрици која има логаритамска сложеност и се базира на оваа идеја, а е многу попримениива.

Во општ случај, еден рекурзивен алгоритам врши сукцесивно повикување на функцијата се додека не го повика почетниот случај, кој посебно се пресметува со некој друг алгоритам. Псевдокодот на еден рекурзивен алгоритам во општ случај ја има формата дадена во П 2.1.:

П 2.1. Генерален псевдокод за рекурзија

```
1  функција rekurzija [листа од променливи];
2  ако {основен случај} тогаш {пресметки за основен случај}
3       инаку {
4                rekurzija [помали вредности или променливи];
5                {некои дополнителни пресметки }}
```

Рекурзивниот алгоритам кој ја имплементира формулата (2.1), без никакви модификации, е даден со следниот псевдокод П 2. :

П 2. 2. ПСЕВДОКОД ЗА РЕКУРЗИВЕН АЛГОРИТАМ ЗА ПРОБЛЕМ НА ПАРИЧКИ

```
1  функција A[n];
2  ако { n = 1 } тогаш {врати 1} инаку
3      ако { n = 2 } тогаш { врати 2} инаку A[n − 1] + A[n − 2];
```

Соодветниот код во програмските јазици Java и C++ базиран на овој рекурзивен алгоритам е даден со JAVA 2. 1 и JAVA 2. 1 :

JAVA 2. 1 ПРОБЛЕМ НА ПАРИЧКИ – РЕКУРЗИВЕН АЛГОРИТАМ

```
1 import java.util.*;
2
3 public class Main {
4     public static void main(String[] args) {
5         Scanner scn = new Scanner(System.in);
6         int n = scn.nextInt();
7         int[] A = new int[n];
8         if (n == 1)
9             System.out.println(1);
10        if (n == 2)
11            System.out.println(2);
12        A[0] = 1;
13        A[1] = 2;
14        for (int i = 2; i < n; i++)
15            A[i] = A[i - 1] + A[i - 2];
16        System.out.println(A[n - 1]);
17    }
18}
```

C++ 2. 1. Проблем на парички – рекурзивен алготирам

```
#include<iostream>
using namespace std;

int parichki(int n)
{
    if (n == 1)
        return 1;
    if (n == 2)
        return 2;
    return parichki(n - 1) + parichki(n - 2);
}

int main()
{
    int n;
    cin >> n;
    int res = parichki(n);
    cout << res;

return 0;
}
```

Да забележиме дека ако пресметките ги правиме со овој алгоритам, функцијата ќе ја повикуваме повеќе пати за една иста вредност. Ако се навратиме на пресметката за $A[5]$, ќе видиме дека вредноста за $A[3]$ се пресметува два пати, еднаш при замената за $A[5]$, а втор пат при замената за $A[4]$. Според тоа, бројот на операции кои би ги направиле е овој алгоритам ќе има експоненцијална сложеност, во овој случај $O\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$. Како можеме да го изброиме ова? Ако со $T(n)$ го обележиме бројот на операции потребни да се реши проблем со големина n, тогаш бројот на операции кои ни се потребни со овој рекурзивен алгоритам е еднаков на бројот на операции потребни да се реши проблем со големина $n-1$, односно $T(n-1)$, плус бројот на операции потребни да се реши проблем со големина $n-2$, $T(n-2)$, плус константен број на дополнителни пресметки, $O(1)$, па имаме:

$$T(n) = T(n-1) + T(n-2) + 1.$$

Оттука вредноста на $T(n)$ е сигурно поголема од вредноста на $A[n] = O\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$, или добивме експоненцијална сложеност.

Можеби изгледа дека оваа идеја не е корисна? Напротив, кога веќе сме ја пресметале вредноста на функцијата за одредена величина, нема потреба повторно да ја пресметуваме. Овде всушност доаѓа стратегијата со динамичко програмирање. Имено, наместо да ја пресметуваме функцијата директно за $A[5]$, можеме да ја пресметаме нејзината вредност за сите броеви до $A[5]$ и оние кои ни се потребни за пресметка на $A[5]$, ќе ги замениме во соодветната равенка. Со оваа стратегија пресметката изгледа вака:

$$A[3] = A[2] + A[1] = 2 + 1 = 3;$$
$$A[4] = A[3] + A[2] = 3 + 2 = 5;$$
$$A[5] = A[4] + A[3] = 5 + 3 = 8.$$

Со овој пристап, за пресметка на $A[5]$ ни требаа 3 операции собирање, додека со рекурзивниот метод ќе ни требаат 4. За поголеми вредности, овие разлики се многу позначителни. П 2. 3, го дава алгоритамот со динамичко програмирање:

П 2. 3 ПСЕВДОКОД ЗА АЛГОРИТАМ СО ДП ЗА ПРОБЛЕМ НА ПАРИЧКИ

1 $A[1] = 1;$
2 $A[2] = 2;$
3 **за** i = 3 **до** n **прави** $A[i] = A[i-1] + A[i-2];$
4 **печати** $A[n]$

Имплементациите на овој алгоритам во Java и C++ се дадени во продолжение, JAVA 2. 2 и C++ 2. 2. :

JAVA 2. 3 ПРОБЛЕМ НА ПАРИЧКИ ДП

```
1 import java.util.*;
2
3 public class Main {
4     public static void main(String[] args) {
5         Scanner scn = new Scanner(System.in);
6         int n = scn.nextInt();
7         int[] A = new int[n];
8         if (n == 1)
9             System.out.println(1);
10        if (n == 2)
11            System.out.println(2);
12        A[0] = 1;
13        A[1] = 2;
14        for (int i = 2; i < n; i++)
15            A[i] = A[i - 1] + A[i - 2];
16        System.out.println(A[n - 1]);
17    }
18}
```

C++ 2. 2. Проблем на парички ДП

```
#include<iostream>
using namespace std;

int parichki(int n)
{
    if (n == 1)
        return 1;
    if (n == 2)
        return 2;
    return parichki(n - 1) + parichki(n - 2);
}

int main()
{
    int n;
    cin >> n;
    int res = parichki(n);
    cout << res;

return 0;
}
```

Сложеноста на еден алгоритам можеме да ја определиме и од самата рекурзивна релација и од псевдокодот. Во псевдокодот П 2. 3 имаме само еден циклус во кој имаме константен број на операции, па оттука сложеноста на алгоритамот е $O(n)$. Од друга страна самиот псевдокод не ни е потрбен за да ја пресмеаме сложеноста, па истата понекогаш многу лесно можеме да ја определиме од самата рекурзивна равенка. Овде имаме функција од една променлива за чија пресметка ни се потребни само 3 операции, две повикувања на функцијата од помали инстанци и едно собирање. Оттука е јасно дека ќе имаме вкупно $const \cdot n$ операции, па сложеноста ќе биде $O(n)$.

Повеќекратно пресметување на една иста вредност може да се избегне и со рекурзивно решение. Овој метод е познат под името **мемоизација**. При мемоизација, прво е потребно за секој елемент од

доменот на функцијата, таа да се иницира на NILL. Наместо NILL може да се избере било која вредност која не може да се добие при пресметките во алгоритамот, најчесто -1 . Потоа, при секое повикување на рекурзијата треба да се провери дали вредноста е веќе пресметана: ако е пресметана ќе се земе пресметаната вредност, а ако не е ќе се пресмета преку рекурентната равенка. Оваа стратегија е полесна за имплементација затоа што работи веднаш над рекурзивната равенка и не е потребно претходно да се води сметка дали инстанците од кои ја повикуваме функцијата се веќе пресметани или не, што многу често може да се превиди при користење на стандардно динамичко програмирање од оздола па нагоре.

Мемоизацијата честопати може да ја зголеми ефикасноста и во смисла на меморија и во смисла на брзина, со тоа што ќе се избегнат пресметки за оние инстанци на функцијата кои нема да се користат. На пример, ако го имаме проблемот на парички монети од 4 и 5 денари, и треба да го пресметаме оптималното решение за 9 денари, воопшто нема да ни требаат оптималните решенија за 6, 7 и 8 па може воопшто да не ги пресметуваме. Но таквиот пристап побарува динамичка алокација на меморија, што нема сега да го разгледуваме.

Нашиот алгоритам со мемоизација е даден во П 2. 4:

П 2. 4 АЛГОРИТАМ СО МЕМОИЗАЦИЈА ЗА ПРОБЛЕМ НА ПАРИЧКИ

```
1  ако { n = 0 или n = 1 } тогаш { A[n] = 1};
2  за i = 2 до n прави A[i] = −1;
3  функција A[n]
4          ако A[n] < 0 тогаш  A[n − 1] + A[n − 2];
5  печати A[n].
```

Имплементацијата на алгоритамот со помошна мемоизација го даваме во двата програмски јазици Јава и C++.

```cpp
#include<iostream>
#include <vector>
using namespace std;

vector <int> A;

int getA(int n) {
    if (A[n] < 0)
        A[n] = getA(n-1) + getA(n-2);
    return A[n];
}

int main() {
    int n;
    cin >> n;
    if(n == 1)
        cout << 1;
    if(n == 2)
        cout << 2;
    if(n > 2)
    {
        A.resize(n);
        A[0] = 1;
        A[1] = 2;
        for(int i=2;i<n;i++)
        {
            A[i] = -1;
        }
        cout << getA(n-1);
    }

    return 0;
}
```

```
1 import java.util.*;
2
3 public class Main {
4     public static int[] A;
5
6     public static int getA(int n) {
7         if (A[n] < 0)
8             A[n] = getA(n - 1) + getA(n - 2);
9         return A[n];
10    }
11
12    public static void main(String[] args) {
13        Scanner scn = new Scanner(System.in);
14        int n = scn.nextInt();
15        if (n == 1)
16            System.out.println("1");
17        if (n == 2)
18            System.out.println("2");
19        if (n > 2) {
20            A = new int[n];
21            A[0] = 1;
22            A[1] = 2;
23            for (int i = 2; i < n; i++) {
24                A[i] = -1;
25            }
26        }
27        System.out.println(getA(n - 1));
28    }
29}
```

Интересно е што овој проблем може да се реши и побргу отколку во линеарно време. Генерално, рекурзивните проблеми можат да се забрзаат со слична идеја како што ќе опишеме подолу. Имено, рекурзијата во матрична форма може да се запише на следниов начин:

$$\begin{bmatrix} A[n] \\ A[n-1] \end{bmatrix} = \begin{bmatrix} A[n-1] + A[n-2] \\ A[n-1] \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} A[n-1] \\ A[n-2] \end{bmatrix}.$$

Ако истово го примениме $n - 1$ пати, ќе се добие:

$$\begin{bmatrix} A[n] \\ A[n-1] \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1} \begin{bmatrix} A[1] \\ A[0] \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1} \begin{bmatrix} 1 \\ 1 \end{bmatrix}.$$

Бидејќи степенувањето на матрица може да се направи во логаритамско време, најбрзото решение на овој проблем има сложеност O($\ln n$). Забележете, нам ни треба време O(2^3) за да ги извршиме множењата во матрицата, но тоа време е константно, па не влијае на целосната сложеност на алгоритамот. Секако, ако рекурзијата зависи од повеќе членови, ова време може да не биде незначително и треба да се има во предвид. Имајќи во предвид дека множењето на матрици има кубна сложеност, иако истиот може да се направи и во време O($m^2 \ln m$) [4], каде m е бројот на редици и колони во матрицата, ако $m = \Omega(\sqrt[3]{n})$, оваа стратегија не е од никаква помош.

Прашања и задачи

1. Дади приближна вредност за најголемото n за кое вредноста за бројот на начини на плаќање со монети од 1 и 2 денари може да се складира во меморија која собира до 10^{18} броеви.

2. Ако во една секунда можат да се направат 10^6 операции, дади приближна вредност за најголемото n за пресметување на бројот на начини на плаќање со монети од 1 и 2 денари, со линеарниот алгоритам. Дали може да се отпечати точната вредност за ова најголемо n? Дали може да се отпечати точната вредност за ова најголемо n по некој модул?

3. Ако во една секунда можат да се направат 10^6 операции, дади приближна вредност за најголемото n за пресметување на бројот на начини на плаќање со монети од 1 и 2 денари, со алгоритамот со степенување на матрици.

4. Да се даде рекурзивната врска за бројот на начини за плаќање во автомат со монети од 1, 2 и 5 денари.

5. Да се даде рекурзивната врска за бројот на низи од битови со должина n кои немаат последователна низа од две нули.

6. Матрица со големина $3 \times n$ треба да се поплочи со плочки со големина 3×1.

   a. Дади ја рекурзивната врска за бројот на начини на поплочување!

   b. Дади ги почетните услови!

7. Матрица со големина $2 \times n$ треба да се поплочи со плочки со големина 2×1, такви што половината со димензии 1×1 е црна, а другата половина со димензии 1×1 е бела.

   a. Дади ја рекурзивната врска за бројот на начини на поплочување!

   b. Дади ги почетните услови!

8. Од полето 0 до полето n треба да се стигне со чекори со должина 2 и 3. Чекор со должина 2 се прави со веројатност p, а со должина

3 со веројатност $1-p$. Да се даде рекурзивна равенка која ја пресметува веројатноста да се стигне до полето n.

Проблем 2. 2. Ред за билети

Наредниот проблем што ќе го разгледуваме е оптимизациониот проблем кој е аналог на претходниот. Имено, претходниот проблем можевме да го гледаме како број на начини елементи од дадена низа да се групираат по еден или два, додека овде проблемот ќе биде да се најде едно такво оптимално групирање.

Дефинирање на проблемот

Пред благајната во театар стојат n луѓе и чекаат да купат билети. На k-тиот човек во редот му треба t_k време за да купи билет, $k = \overline{1,n}, t_k > 0$. Секој човек може да се здружи со следниот во редот. Времето потребно за да k -тиот и $k+1$-от човек купат билет доколку се здружат, изнесува p_k, $k = \overline{1,n-1}$. Со тоа купувањето може, а и не мора, да се забрза. Да се одреди таков начин на здружување на луѓето во кој што вкупното време потребно сите n луѓе да купат билет да биде минимално.

Влезни податоци се бројот n и низите t_k и p_k. Како излез треба да се испишат редните броеви на оние луѓе кои се здружуваат со следниот во редот.

Анализа на проблемот

Да забележиме дека еден начин да се реши проблемот е да се разгледаат сите можни спојувања на двајца луѓе и за секое спојување да се пресмета вкупното време на чекање. На пример ако имаме четворица во редот, тогаш тие можат да ги купат картите по следниве принципи

- Да нема спојување. Ова може да го обележиме со низата: 1, 1, 1, 1.
- Да се спојат само првиот и вториот. Ова може да го обележиме со низата: 2, 1, 1.
- Да се спојат само вториот и третиот. Ова може да го обележиме со низата: 1, 2, 1.

- Да се спојат само третиот и четвртиот. Ова може да го обележиме со низата: 1, 1, 2.
- Да се спојат првиот и вториот, па третиот и четвртиот. Ова може да го обележиме со низата: 2, 2.

Секоја од овие варијанти си има свое вкупно време на чекање, па наивниот алгоритам би бил да ги пресметаме сите нив и да го најдеме најмалото меѓу нив. Но, тоа значи да се генерираат сите вакви низи, што е всушност друг рекурзивен проблем. Поврзаноста на овој проблем со претходниот е токму во овие низи од единици и двојки, чија вкупна сума треба да биде n. Имено, проблемот со монетите е проблем во кој треба да се изброи колку вакви низи постојат, додека во овој проблем треба да се определи една од нив - онаа која има најоптимално време на чекање. Оттука, за да го најдеме оптималното решение со користење на наивниот алгоритам, од претходната задача, треба да генерираме $O\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$ комбинации на здружување, што само по себе значи дека алгоритамот има сложеност $\Omega\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$.

Бидејќи двата проблеми се аналогни, размислувањето при дизајнот на решенијата ќе биде аналоген. Повторно треба да анализираме како ќе се намали проблемот во зависност од тоа дали последно купил билет само еден човек или се групирал со тој пред него, односно дали последниот се здружил со претпоследниот или не.

Размислувањето оди вака: При оптималното здружување можни се два случаи:

- Последен да купил само еден човек или
- Последни на благајната да биле последните двајца луѓе во редицата.

Без разлика која од овие можности води до оптимално решение, битно да се увиди дека проблемот ќе се намали на потполно истиот проблем, само со редица од помал број на луѓе. Подетално, ако во оптималното решение последниот човек купил сам билет, тогаш претходните $n - 1$ во редицата требало исто така да

се групираат најдобро можно. Ако не е така, тогаш ние би ги групирале подобро, односно да завршат за пократко време, па бидејќи времето за последниот човек да купи билет е константно, t_n, и вкупното време би било пократко. Исто, ако во оптималното решение последните двајца се здружиле, тогаш претходните n – 2-ца во редицата требало да се групираат најдобро можно, затоа што ако можеме да ги групираме да завршат за пократко време, вкупното време би било исто така пократко. Ова својство кое го поседува проблемот се нарекува **оптимално својство**.

Секој оптимизационен проблем кој се решава со динамичко програмирање мора да го поседува ова оптимално својство, ако не го поседува, тогаш таквиот проблем едноставно не може да се реши со оваа техника.

Да го дефинираме оптималното својство за овој проблем поконкретно:

> *Ако во оптималното групирање на луѓето во редица со должина n, човекот кој е k-ти во редицата купува сам или со тој пред него, $k < n$, тогаш луѓето во подредицата од првиот до k -тиот човек се исто здружени на оптимален начин.*

Оптималното својство секогаш треба да се потрудиме да го докажеме, затоа што не секогаш нашата идеја мора да биде точна. Докажувањето обично оди по сличен терк како и во оваа задача, и е познато како копирај-залепи техника.

Доказ на оптималното својство: Нека сме нашле некое оптимално здружување на редицата од n луѓе. Бидејќи е оптимално, нема начин луѓето да се здружат така што ќе им биде потребно пократко време. Јасно, во ова оптималното здружување k-тиот човек купува сам или со тој пред него, и нека претпоставиме дека претходните $k - i, i = 1,2$ луѓе не биле оптимално здружени, т.е. постои подобар начин тоа да се направи. Тогаш здружувањето за првите $k - i$ во редицата можеме да го замениме овој подобар начин на здружување, а останатите во редицата ќе ги здружиме на истиот начин како во оптималното решение кое сме го нашле. Бидејќи во ова нова распределба времето на чекање на првите $k - i$ луѓе ќе биде

пократко, а за останатите исто како и претходно, излегува дека ќе добиеме пократко време на чекање, што е контрадикција со претпоставката дека нема начин луѓето да се здружат така што ќе им биде потребно пократко време. Со ова се докажува горното оптимално својство. □

Конечно, да ја изведеме рекурзивната равенка. Нека оптималното време за чекање на редица од n луѓе.го обележиме со $A[n]$. Тогаш:

- Ако во најдоброто решение последен купил само еден човек, тогаш најмалото време на чекање е

$$A[n] = A[n-1] + t_{n-1}.$$

- Ако во најдоброто решение последни на благајната да биле последните двајца луѓе во редицата, тогаш најмалото време на чекање е

$$A[n] = A[n-2] + p_{n-2}.$$

Можен е или едниот или другиот случај, па најмалото време кое го бараме е помалото од овие две можности, односно:

$$A[n] = min\{A[n-1] + t_n, A[n-2] + p_{n-1}\}, n \geq 2.$$

Бидејќи рекурзивната равенка се враќа за два члена наназад, за редици со големина 0 и 1 мора да се пресмета рачно. (Ако ви е проблем да пресметате за 0, тогаш може да се пресмета за еден и два). Јасно, ако нема луѓе во редицата, времето кое е потребно е 0, ако има само еден човек, единствен начин на купување е тој самиот да купи, па оптималното време е t_1. Оттука почетните услови се: $A[0] = 0$ и $A[1] = t_1$.

Ако за пресметување користиме рекурзивен алгоритам, тогаш слично како и во претходниот проблем, времето на работа може да се пресмета со следнава рекурентна релација:

$$T(n) = T(n-1) + T(n-2) + 1 = \mathrm{O}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right).$$

Решението со динамичко програмирање е техника во која решенијата се добиваат со така наречен од долу нагоре пристап, односно прво се решаваат помалите, а потоа поголемите проблеми, при што во решението на помалите проблеми се користат веќе пресметаните вредности за помалите проблеми. Ова е илустрирано со примерот претставен во Табела 2. 1. Во првата редица е даден редниот број на човекот во редицата, а во втората редица е дадено времето кое е потребно секој да купи билет сам, додека в третата е дадено времето потребно да купи билет еден човек со тој после него. Последните две редици од Табела 2. 2. ги даваат пресметаните оптимални вредности и даваат ознака на кои места има спојување на луѓе од редицата.

Табела 2. 3. Илустрација на алгоритамот за проблемот „Ред за билети“

k	1	2	3	4	5	6	7	8	9	10	11	12
t_k	2	3	5	4	6	2	3	4	3	3	5	4
p_k	6	4	7	8	9	7	7	6	4	8	8	
$A[k]$	2	5	6	10	14	16	19	23	25	27	32	35
$C[k]$	0	0	1	0	1	0	0	0	1	1	0	1

Пресметките се прават чекор по чекор при што се пополнува последната редица во табелата, во која k-тиот елемент е најмалото време што можат да го потрошат првите k во редицата.

Ќе изведеме како се добиваат првите неколку членови од $A[k]$. Јасно, $A[0] = 0$ и $A[1] = 2$. Сега,

$$A[2] = min\{A[1] + 3, A[0] + 6\} = min\{2 + 3, 0 + 6\} = 5.$$

$$A[3] = min\{A[2] + 5, A[1] + 4\} = min\{5 + 5, 2 + 4\} = 6.$$

$$A[4] = min\{A[3] + 4, A[2] + 7\} = min\{6 + 4, 5 + 7\} = 10.$$

Да забележиме дека минимумот за $A[2]$ и $A[4]$ се добива кога вториот, односно четвртиот човек, би купувале сами, односно нема да се здружат со тој пред нив, додека минимумот за $A[3]$ се добива во случај кога третиот би се здружил со вториот. Ова треба да го оставиме како трага за да можеме да го реконструираме решението, односно да дадеме предлог за едно најдобро здружување. Затоа воведуваме нова функција, $C[k]$, во која ќе паметиме меѓу кои луѓе ќе има здружување. Имено, за сите k за кои е подобро k -тиот човек да се здружи со тој пред него, низата $C[k]$ ќе прима вредност 1, а ако не треба да се здружат $C[k]$ ќе прими вредност 0. Оттука,

- $C[1] = 0$, бидејќи ако имаме низа од само еден елемент, тој не може да се здружи со тој пред него.
- $C[2] = 0$, бидејќи помала вредност се добива кога вториот не се здружува со првиот
- $C[3] = 1$, бидејќи помала вредност се добива кога третиот ќе се здружи со вториот
- $C[4] = 0$, бидејќи помала вредност се добива кога четвртиот не се здружува со третиот.

Псевдокодот е даден во П 2. 5 :

П 2. 5 Алгоритам за Ред на билети

```
1 Внеси ги низите t_k и p_k, k = 1,n−1.
2 A[0] = 0;
3 A[1] = 1;
4 за k = 2 до n прави
5    ако A[k − 1] + t_k > A[k − 2] + p_{k−1} тогаш
6         {
7           A[k] = A[k − 2] + p_{k−1};
8           C[k] = 1
```

```
9          }
10         инаку
11         {
12           A[k] = A[k − 1] + t_k;
13           C[k] = 0;
14         }
15 печати A[k];
```

Сложеноста на алгоритамот лесно може да се пресмета анализирајќи ги циклусите во алгоритамот. Во првиот дел, кога се пресметува оптималното време се јавува само еден циклус од 1 до n и во тој циклус се прават константен број операции, најмногу 4. Значи сложеноста на ова е $O(n)$, каде n е бројот на луѓе во редицата. Од друга страна, во делот од алгоритамот за реконструкција повторно имаме само еден циклус, кој се повторува најмногу n пати. Оттука, сложеноста на целиот алгоритам е $O(n)$.

Откога ќе се пресмета минималното време и ќе се зачуваат вредностите за $A[k]$ и $C[k]$, врз база на $C[k]$може да се реконструира оптималното решение. Прво ќе го дадеме псевдокодот за реконструкција, а потоа ќе го објасниме како истиот работи. Да напоменеме дека псевдокодот П 2. 6. кој го даваме ќе го печати решението во обратен редослед.

П 2. 6 АЛГОРИТАМ ЗА РЕД НА БИЛЕТИ

```
1  Внеси ги низите t_k и p_k, k = 1, n − 1.
2  A[0] = 0;
3  A[1] = 1;
4  за k = 2 до n прави
5     ако A[k − 1] + t_k > A[k − 2] + p_{k−1} тогаш
6          {
7            A[k] = A[k − 2] + p_{k−1};
8           C[k] = 1
9          }
10         инаку
```

```
11        {
12          A[k] = A[k − 1] + t_k;
13          C[k] = 0;
14        }
15  k = n;
16  додека k > n прави
17    ако C[k] = 0 тогаш k = k − 1 инаку
18        {
19          печати „треба да се спојат“ k − 1 „и“ k;
20          k = k − 2;
21        }
```

За нашиот пример реконструкцијата на едно оптимално решение оди на следниов начин:

- Затоа што $C[12] = 1$, 12-тиот и 11-тиот во редицата треба да купат билет заедно. Ова оптимално решение се добило од оптималното решение до 10-тиот во редицата, плус p_{11}.
- Во ваква ситуација кога се здружуваат двајца, C го повикуваме за два чекори наназад. Бидејќи $C[12 - 2] = C[10] = 1$, во оптималното решение 10-тиот се здружува со 9-тиот.
- Повторно се здружиле двајца, па треба да го повикаме $C[8]$. Затоа што $C[8] = 0$, во оптималното решение 8-тиот купува сам.
- Сега имаме ситуација кога еден човек купува сам, па функцијата C ја повикуваме за еден чекор наназад. Бидејќи $C[8 - 1] = C[7] = 0$, следува дека во оптималното решение и 7-тиот купува сам.
- Повторно ја повикуваме функцијата C за еден чекор наназад, и бидејќи е $C[6] = 0$ следува дека во оптималното решение и 6-тиот купува сам.
- Бидејќи $C[5] = 1$, во оптималното решение 4-тиот и 5-тиот се здружуваат.

- $C[3] = 1$, па во оптималното решение се здружуваат 2-от и 3-от се здружуваат.
- На крај $C[1] = 0$, па првиот купува сам.

Алгоритамот за реконструкција нема да ја зголеми сложеноста на алгоритамот, затоа што и овде повторно имаме само еден циклус, кој се повторува најмногу n пати. Оттука, сложеноста на целиот алгоритам е $O(n)$.

Во продолжение ги даваме целосните решенија во програмските јазици C++ и Јава, , C++ 2. 4. и JAVA 2. 5. соодветно. Во двете решенија како излез освен оптималното време за купување на билетите се печати и една реконструкција, односно еден начин како тоа оптимално решение може да се добие.

C++ 2. 4. РЕД ЗА БИЛЕТИ

```
1  #include <iostream>
2  #include <algorithm>
3
4  using namespace std;
5
6  const int MAX = 1001;
7
8  long long t[MAX];
9  long long p[MAX];
10 long long a[MAX];
11 long long c[MAX];
12
13 int n;
14
15 void print_pairs(int i)
16 {
17    if(i < 2)
18       return;
19
20    if(c[i] == 1)
```

```
21    {
22       print_pairs(i-2);
23       cout << "Treba da se spojat " << (i-1) << " i "
<< i << endl;
24    }
25    else
26    {
27       print_pairs(i-1);
28    }
29 }
30
31 int main()
32 {
33    cin >> n;
34
35    for (int i=1; i<=n; i++)
36       cin >> t[i];
37    for (int i=1; i<n; i++)
38       cin >> p[i];
39
40    a[0] = 0;
41    a[1] = t[1];
42    c[1] = 0;
43
44    for (int i=2; i<=n; i++)
45    {
46
47       if (a[i-1] + t[i] > a[i-2] + p[i-1])
48       {
49          a[i] = a[i-2] + p[i-1];
50          c[i] = 1;
51       }
52       else
53       {
54          a[i] = a[i-1] + t[i];
55          c[i] = 0;
56       }
57    }
58
59    cout << "Vkupno vreme " << a[n-1] << endl;
60    print_pairs(n);
```

```
61      return 0;
62  }
```

JAVA 2. 5. РЕД ЗА БИЛЕТИ

```
1   import java.util.*;
2
3   public class Main {
4
5       public static int MAX = 1001;
6
7       public static long [] t = new long[MAX];
8       public static long [] p = new long[MAX];
9       public static long [] a = new long[MAX];
10      public static long [] c = new long[MAX];
11
12      public static int n;
13
14      public static void print_pairs(int i)
15      {
16          if(i < 2)
17              return;
18
19          if(c[i] == 1)
20          {
21              print_pairs(i-2);
22              System.out.println("Treba da se spojat "+
(i-1) + " i " + i);
23          }
24          else
25          {
26              print_pairs(i-1);
27          }
28      }
29
30      public static void main(String args[]) {
31          Scanner sc = new Scanner(System.in);
32          n = sc.nextInt();
33
34          for (int i=1; i<=n; i++)
35              t[i] = sc.nextInt();
36          for (int i=1; i<n; i++)
```

```
37          p[i] = sc.nextInt();
38
39       a[0] = 0;
40       a[1] = t[1];
41       c[1] = 0;
42
43       for (int i=2; i<=n; i++)
44       {
45
46          if (a[i-1] + t[i] > a[i-2] + p[i-1])
47          {
48             a[i] = a[i-2] + p[i-1];
49             c[i] = 1;
50          }
51          else
52          {
53             a[i] = a[i-1] + t[i];
54             c[i] = 0;
55          }
56       }
57
58       System.out.println("Vkupno vreme " + a[n-1]);
59       print_pairs(n);
60    }
61 }
```

Прашања и задачи

1. Колкава е најголемата вредност на броевите t_i и p_i во проблемот за ред на билети, ако тие се приближно исти, за да вредноста за $A[n]$ ја собере во мемориска локација од 10^{18}, кога $n = 10^6$?

2. Дали проблемот може да се реши со временска сложеност помала од $O(n)$?

3. Дали ако знаеме дека времето да се спојат двајца е секогаш помало од времето тие да купуваат билети посебно би можеле да дизајниираме поефикасен алгоритам?

4. Нека во проблемот Ред за билети освен што секој што чека ред може да купува сам, може да се спојат двајца или тројца последователни луѓе. Нека времето i-тиот купувач да купи сам е t_i, времето ако се спои со тој после него е p_i, а времето да се спои со двајцата после него е q_i. Дади ја рекурзивната равенка која го решава овој проблем!

5. Секој од броевите од една низа цели броеви може да се групира со тој после него, и групата да се замени со разликата меѓу тие броеви. Треба да се најде оптималното групирање, така што добиениот збир да биде најмал. На пример за низата: 7, 8, 1, 2, 3 може оптимално да се групира како $(7 - 8) + 1 + (2 - 3) = -1$. Да се даде рекурзивна равенка врз основа на која може да се дизајнира решение со динамичко програмирање!

Проблем 2. 3. Време на извршување со две производствени ленти

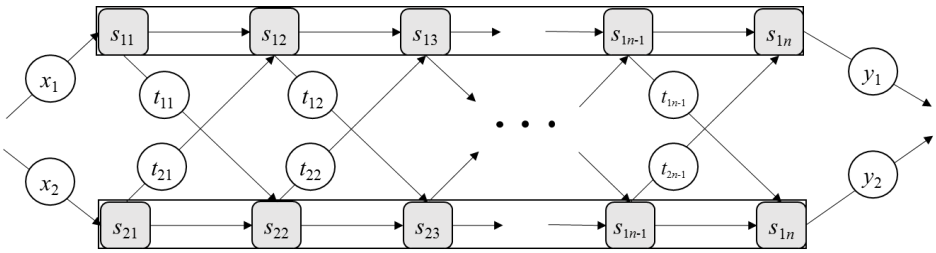
Во овој дел ќе разгледаме оптимизационен проблем, кој што главниот проблем повторно го намалува на два потпроблеми, но во овој случај потпроблемите не се од сосема ист тип, како што беше случај во претходниот проблем. Варијанта односно модификација на овој проблем е познатиот скриен Марков модел [5] [6], кој е статистички модел кој се користи на многу места, како финансиска математика [7], препознавање и синтеза на говор [8], машински превод, предикција на гени и на многу други места.

Дефинирање на проблемот

Во една фабрика која произведува автомобили има две производствени линии кои се состојат од повеќе станици во кои се монтираат различни делови од автомобилот, зависно бројот на станицата. Шасијата на автомобилот влегува во некоја од производствените линии, во секоја станица му се додаваат одредени делови и на крај, завршениот автомобил излегува на крајот од линијата. Двете линии се аналогни една на друга, односно и двете имаат по n станици нумерирани $i = 1, 2, \dots, n$ и i-тата станица на првата линија ја работи потполно истата работа како i-тата станица на втората линија. Нека ја означиме i-тата станица на првата линијата со a_{1i}, а на втората со $a_{2i}, i = \overline{1,n}$. Станиците биле изградени во различно време и со различна технологија, па времето да се сработи една иста работа на двете линии е различно. Дадено ни е времето на производство за секоја од станиците s_{1i} и $s_{2i}, i = \overline{1,n}$ и нека тие соодветно се означени со $p_{1,i}$ и $p_{2,i}$. Кога шасијата ќе влезе во некоја производствена линија, почнува да се движи од првата станица до секоја наредна станица. Но за да влезе на првата станица ќе потроши време, x_1, а за да влезе на втората станица ќе потроши време, x_2. При производството е потребно и време за да излезе готовиот производ од некоја од станиците, па за да излезе од првата станица ќе потроши време, y_1, а за да излез на втората станица ќе потроши време, s_2. Нормално, штом шасијата еднаш ќе влезе во некоја од

производствените линии оди само низ таа линија и времето за да се премине од една станица во наредната на иста линија е занемарливо.

Повремено доаѓа специјална нарачка, па купувачот сака автомобилот да му биде готов за што е можно побрзо. На забрзаните нарачки автомобилот повторно мора да ги помине сите n станици, но менаџерот на фабриката може да избере после секоја станица да го пренасочи недовршениот автомобил од едната на другата линија. Времето за пренесување на шасијата од една производна линија на друга не е занемарливо и во задачата ни е дадено. Нека со $t_{1,i}$ го обележиме времето потребно недовршениот автообил да помине од i-тата станица на првата производствена линија до $i+1$-вата станица на втората производствена линија, а со $t_{2,i}$ да го обележиме времето потребно недовршениот автомобил да помине од i -тата станица на втората производствена линија до $i+1$ -вата станица на првата производствена линија за $i=\overline{1,n-1}$. Проблемот е кои станица од првата линија, а кои од втората производствена линија да се изберат, со цел на се минимизира вкупното време на производство на еден автомобил. На Слика 2. 1. се илустрирани двете производствени линии.



Слика 2. 1. Две производствени линии.

Во примерот на Слика 2. 1. најбрзото време се за кое може да се произведе еден автомобил е да помине преку станиците 1, 3 и 6 од првата линија и станиците 2, 4 и 5 од втората линија.

Анализа на проблемот

Очигледниот „алгоритам со груба сила“ би го пресметал ова со генерирање на сите низи од единици и двојки со должина n (за

разлика од претходниот пример каде сумата на таквите низи требаше да биде n, овде должината е n). единицата на i-та позиција од таквата низа ќе означува дека се поминува низ со a_{1i}, додека двојката ќе означува дека се поминува низ со a_{2i}. При вака дадена листа лесно ќе пресметаме колку долго ќе ни биде потребно шасијата да помине низ фабриката. Но ова има сложеност $O(2^n)$ затоа што постојат 2^n такви низи.

Првиот чекор што треба да го направиме за да ја решиме задачата со е да ја карактеризираме структурата на оптималното решение:

> *Ако во најбрзиот начин да се произведе автомобилот шасијата поминува преку станицата* $a_{ki}, k = 1,2$ *тогаш времето кое во тоа оптимално решение било потребно да се стигне од почеток* a_{ki} *е најмалото време да се стигне од почеток* a_{ki}.

Доказ на оптималната потструктура: Нека сме нашле некое оптимално решение за линии со n станици и нека во ова оптимално решение се поминува низ станицата a_{1i}, но времето да се стигне до таа станица не е најмалото можно, т.е. постои подобар начин. Тогаш стариот начин за стигање до a_{1i}, можеме да го замениме со овој подобар начин, а понатаму да си продолжиме на истиот начин како во оптималното решение кое сме го нашле. Со тоа ќе добиеме пократко време на производство, што е контрадикција. Потполно исто би било ако оптимално решение поминува низ станицата a_{2i}, Со ова се докажува горното оптимално својство. □

Конечно, да ја изведеме рекурзивната равенка. Нека оптималното време да се стигне до станицата a_{1i}, го обележиме со $A_1[i]$, а до станицата a_{2i}, со $A_2[i]$. Тогаш, најкраткото време до a_{1i} е или преку a_{1i-1} или преку a_{2i-1}. Во првиот случај времето да се стигне до a_{1i} е времето да се стигне до $a_{1i-1}, A_1[i-1]$ плус времето да се задржи во a_{1i-1}, s_{1i-1}, а во вториот случај времето да се стигне до $a_{2i-1}, A_2[i-1]$ плус времето да се задржи во a_{2i-1}, s_{2i-1} и плус времето да се префрли од a_{2i-1} до a_{1i}, t_{2i-1}. Оттука ја имаме следнава рекурзивна релација:

$$\begin{cases} A_1[i] = \min\{A_1[i-1] + s_{1i-1}, A_2[i-1] + s_{2i-1} + t_{2i-1}\} \\ A_2[i] = \min\{A_2[i-1] + s_{2i-1}, A_1[i-1] + s_{1i-1} + t_{1i-1}\} \end{cases}.$$

Јасно дека на почеток $A_1[1] = x_1$, а $A_2[1] = x_2$. На крај за излезот имаме дека минималното време да се изработи автомобилот е:

$$B = \min\{A_1[n] + y_1, A_2[n] + y_2\}.$$

За да можеме да направиме реконструкција на оптималното решение потребно е повторно да зачуваме одредени вредности. Затоа ќе дефинираме две нови низи $C_k[i], k = 1,2, i = \overline{2,n}$, во која ќе зачуваме 1 ако набрзиот пат до станицата a_{ki} е преку станицата $a_{1,i-1}$ и 2 ако најбрзиот пат до станицата a_{ki} е преку станицата $a_{2,i-1}$

Псевдокодот за оптималното здружување, заедно со пресметување на низата за која се користи за реконструкција на оптималното решение е дадено со П 2. 7. Во решението не се печати ништо, но од пресметаните вредности може да се отпечатат и оптималното решение и патеката за кое истото се добива, што е дадено со П 2. 8.

П 2. 7. ПСЕВДОКОД ЗА ПРОИЗВОТСТВЕНА ЛЕНТА

```
1  Внеси ги низите t_ki и s_ki, k = 1, 2, i = 1,n.
2  A_1[1] = x_1;
3  A_2[1] = x_2;
4  за i = 2 до n прави
5      ако A_1[i − 1] + s_1i−1 > A_2[i − 1] + s_2i−1 + t_2i−1 тогаш
6          {
7           A_1[i] = A_2[i − 1] + s_2i−1 + t_2i−1;
8           C_1[i] = 2
9           }
10           инаку
11                {
```

```
12              A_1[i] = A_1[i − 1] + s_{1i−1};
13              C_1[i] = 1
14             }
15    ако A_2[i − 1] + s_{2i−1} > A_1[i − 1] + s_{1i−1} + t_{1i−1} тогаш
16       {
17        A_2[i] = A_1[i − 1] + s_{1i−1} + t_{1i−1};
18        C_2[i] = 1
19       }
20         инаку
21              {
22               A_2[i] = A_2[i − 1] + s_{2i−1};
23               C_2[i] = 2
24              }
25     ако A_1[n] + s_{1i} + y_1 > A_2[n] + s_{2i} + y_2 тогаш
26         {
27          B = A_2[n] + s_{2i} + y_2;
28           B_1 = 2
29          }
30          инаку
31               {
32                B = A_1[n] + s_{1i} + y_1;
33                B_1 = 1
34               }
```

Рекурзивната функција зависи само од една променлива, а нејзината рекурзивна равенка повикува константен број на помали инстанци од функцијата. Затоа и во алгоритамот има само еден циклус во кој има константен број на пресметки. Оттука сложеноста е $O(n)$, каде n е бројот на станици. Реконструкцијата секогаш побарува помалку пресметки, затоа што се базира на веќе пресметаните вредности во основниот алгоритам. Во овој пример во најлош случај ќе ги побарува сите вредности за C што ќе се сработи со само еден циклус, па целосната сложеност на алгоритамот е $O(n)$.

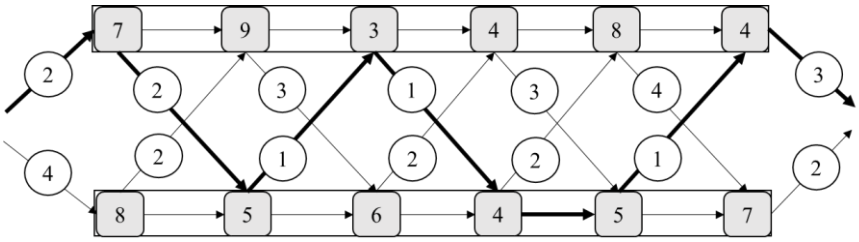
Во Табела 2. 4 се дадени вредностите кои се зачувуваат за функциите A_1, A_2, C_1 и C_2 за примерот илустриран на Слика 2. 2. Вредностите за B и за B_1 се пресметуваат посебно на крај и тие се:

$$B = \min(31 + 4 + 3, 30 + 7 + 2) = \min(38, 39) = 38,$$

$$B_1 = 1.$$

Табела 2. 4. Вредностите за кои се зачувуваат за A_k и C_k примерот илустриран на Слика 2. 2.

i	1	2	3	4	5	6
A_1	2	$min(9,14)$ =9	$min(18,17)$ =17	$min(20,24)$ =20	$min(24,27)$ =24	$min(35,31)$ =31
C_1		1	2	1	1	2
A_2	4	$min(11,12)$ =11	$\min(21,16)$ =16	$min(21,22)$ =21	$min(27,25)$ =25	$min(39,30)$ =30
C_2		1	2	1	2	2



Слика 2. 2. Илустрација на патеката на оптималното време на производство за две производствени линии.

Според табелата дешифрирањето на едно оптимално решение би одело на следниот начин:

- Бидејќи $B_1 = 1$, последната, 6-та станица е од првата линија;
- Бидејќи $C_1[6] = 2,$ 5-тата станица е од втората линија;
- Бидејќи $C_2[5] = 2,$ 5-тата станица е од втората линија;
- Бидејќи $C_2[4] = 1,$ 5-тата станица е од првата линија;
- Бидејќи $C_1[3] = 2,$ 5-тата станица е од втората линија;

- Бидејќи $C_1[2] = 1,$ 5-тата станица е од првата линија.

Во псевдокодот за реконструкција на едно оптимално решение што го даваме овде, П 2. 8, станиците се печатат во обратен редослед, како што е објаснето погоре.

П 2. 8. ПСЕВДОКОД ЗА РЕКОНСТРУКЦИЈА ЗА ПРОБЛЕМОТ НА ПРОИЗВОТСТВЕНА ЛЕНТА

```
1  i = n;
2  ако B_1 = 1 тогаш
3      {
4       печати 1;
5        k = 1;
6       }
7       инаку
8             {
9              печати 2;
10             k = 2;
11             }
12 додека i > 1 прави {
13              ако C_k[i] = 1 тогаш
14                              {
15                               печати 1;
16                                k = 1;
17                              }
18                         инаку
19                              {
20                                печати 2;
21                                k = 2;
22                               }
23                     i = i – 1.
24                }
```

На крај на овој дел ги презентираме програмите во Јава и C++, JAVA 2. 6 и C++ 2. 5.

JAVA 2. 6 ПРОИЗВОДНА ЛЕНТА

```
1 import java.util.*;
2
3 public class Main {
4     public static void main(String[] args) {
5         Scanner scn = new Scanner(System.in);
6         int n = scn.nextInt();
7         int x1, x2, y1, y2, B, B1, k;
8         int[][] s = new int[2][n];
9         int[][] A = new int[2][n];
10        int[][] C = new int[2][n - 1];
11        int[][] t = new int[2][n - 1];
12
13        x1 = scn.nextInt();
14        x2 = scn.nextInt();
15        y1 = scn.nextInt();
16        y2 = scn.nextInt();
17
18        for (int i = 0; i < n; i++)
19            s[0][i] = scn.nextInt();
20
21        for (int i = 0; i < n; i++)
22            s[1][i] = scn.nextInt();
23
24        for (int i = 0; i < n - 1; i++)
25            t[0][i] = scn.nextInt();
26
27        for (int i = 0; i < n - 1; i++)
28            t[1][i] = scn.nextInt();
29
30        A[0][0] = x1;
31        A[1][0] = x2;
32
33        for (int i = 1; i < n; i++) {
34            if (A[0][i - 1] + s[0][i - 1] > A[1][i - 1] +
s[1][i - 1] + t[1][i - 1]) {
35                A[0][i] = A[1][i - 1] + s[1][i - 1] +
t[1][i - 1];
36                C[0][i - 1] = 2;
37            } else {
38                A[0][i] = A[0][i - 1] + s[0][i - 1];
39                C[0][i - 1] = 1;
40            }
```

```
41
42              if (A[1][i - 1] + s[1][i - 1] > A[0][i - 1] +
s[0][i - 1] + t[0][i - 1]) {
43                  A[1][i] = A[0][i - 1] + s[0][i - 1] +
t[0][i - 1];
44                  C[1][i - 1] = 1;
45              } else {
46                  A[1][i] = A[1][i - 1] + s[1][i - 1];
47                  C[1][i - 1] = 2;
48              }
49          }
50
51          if (A[0][n - 1] + s[0][n - 1] + y1 > A[1][n - 1]
+ s[1][n - 1] + y2) {
52              B = A[1][n - 1] + s[1][n - 1] + y2;
53              B1 = 2;
54          } else {
55              B = A[0][n - 1] + s[0][n - 1] + y1;
56              B1 = 1;
57          }
58
59          System.out.println(B);
60
61          // Pechati pateka vo obraten redosled
62          if (B1 == 1) {
63              System.out.print("1 ");
64              k = 0;
65          } else {
66              System.out.print("2 ");
67              k = 1;
68          }
69
70          for (int i = n - 2; i >= 0; i--) {
71              if (C[k][i] == 1) {
72                  System.out.print("1 ");
73                  k = 0;
74              } else {
75                  System.out.print("2 ");
76                  k = 1;
77              }
78          }
79      }
80}
```

```
1  #include<iostream>
2  #include <vector>
3  using namespace std;
4
5  int main() {
6     int n;
7     cin >> n;
8     int x1, x2, y1, y2, B, B1, k;
9     int s [2][n];
10    int A [2][n];
11    int C [2][n - 1];
12    int t [2][n - 1];
13
14    cin >> x1;
15    cin >> x2;
16    cin >> y1;
17    cin >> y2;
18
19    for (int i = 0; i < n; i++)
20       cin >> s[0][i];
21
22    for (int i = 0; i < n; i++)
23       cin >> s[1][i];
24
25    for (int i = 0; i < n - 1; i++)
26       cin >> t[0][i];
27
28    for (int i = 0; i < n - 1; i++)
29       cin >> t[1][i];
30
31    A[0][0] = x1;
32    A[1][0] = x2;
33
34    for (int i = 1; i < n; i++) {
35       if (A[0][i - 1] + s[0][i - 1] > A[1][i - 1] + s[1][i
- 1] + t[1][i - 1]) {
36          A[0][i] = A[1][i - 1] + s[1][i - 1] + t[1][i - 1];
37          C[0][i - 1] = 2;
38       } else {
39          A[0][i] = A[0][i - 1] + s[0][i - 1];
40          C[0][i - 1] = 1;
41       }
42
43       if (A[1][i - 1] + s[1][i - 1] > A[0][i - 1] + s[0][i
- 1] + t[0][i - 1]) {
44          A[1][i] = A[0][i - 1] + s[0][i - 1] + t[0][i - 1];
```

```
45          C[1][i - 1] = 1;
46       } else {
47          A[1][i] = A[1][i - 1] + s[1][i - 1];
48          C[1][i - 1] = 2;
49       }
50    }
51
52    if (A[0][n - 1] + s[0][n - 1] + y1 > A[1][n - 1] +
s[1][n - 1] + y2) {
53       B = A[1][n - 1] + s[1][n - 1] + y2;
54       B1 = 2;
55    } else {
56       B = A[0][n - 1] + s[0][n - 1] + y1;
57       B1 = 1;
58    }
59
60    cout << B << endl;
61
62    // Pechati pateka vo obraten redosled
63    if (B1 == 1) {
64       cout << "1 ";
65       k = 0;
66    } else {
67       cout << "2 ";
68       k = 1;
69    }
70
71    for (int i = n - 2; i >= 0; i--) {
72       if (C[k][i] == 1) {
73          cout << "1 ";
74          k = 0;
75       } else {
76          cout << "2 ";
77          k = 1;
78       }
79    }
80
81    return 0;
82 }
```

Прашања и задачи

1 Нека во проблемот за време на извршување на производтствена лента е додадено време за премин од i-тата до $i+1$-та станица од j -тата до другата лента, $r_{ji}, j = 1,2, i = \overline{1,n}$. Дади ја рекурзивната равенка за оваа преформулација на проблемот.

2 Нека во проблемот за време на извршување на производтствена лента има 3 наместо 2 ленти и времиња на премин од k-тата до $k+1$ -та станица, од i -тата лента до j -тата лента, $t_{ijk}, i, j = 1,2,3;\ k = \overline{1,n}$ и времиња на стигање и напуштање на i-тата лента x_i и $y_i, i, j = 1,2,3$ соодветно. Дади ја рекурзивната равенка за оваа преформулација на проблемот.

3 Нека веројатноста да врне $i+1$-от ден, ако врнело i-тиот ден е p_{1i}, а да врне $i+1$-от ден, ако не врнело i-тиот ден е p_{2i}. Од друга страна нека веројатноста да не врне $i+1$ -от ден, ако врнело i-тиот ден е q_{1i}, а да не врне $i+1$-от ден, ако не врнело i-тиот ден е q_{2i}. Веројатноста првиот ден да врнело нека е x, а да не врнело y.

   a. Дади алгоритам за пресметување на веројатноста на најверојатната можна низа од настани врнело и не врнело во текот на n последователни дена.

   b. Како ќе пресметаш која е најверојатниот број на денови во кои врнело за претходниот проблем од задача 3?

Проблем 2. 4. Шарено оро

Наредниот пример е илустрација на алгоритам со динамичко програмирање, со кој проблемот се сведува на два потпроблеми, но од различен тип. Едниот потпроблем е оригиналниот потпроблем со помала големина, додека другиот потпроблем може аналитички да се реши. Овој проблем ќе го искористиме за да илустрираме на што треба да се внимава кога ќе бројот кој треба да се отпечати е толку голем за да не може да го собере во една мемориска локација, па треба да се отпечати по некој даден модул.

Дефинирање на проблемот

Група од $n > 1$ деца наредени во редица и фатени за рака играат оро. На почеток едно од децата е на чело на орото, но во текот на играњето последниот од орото доаѓа на почеток односно станува ороводец, а другите не се менуваат. За орото да биде поинтересно, децата треба да се облечат во маички во $m > 1$ различни бои и треба да се наредат така да за цело време додека трае играњето нема две деца едно до друго во маички од иста боја. На колку различни начини може децата да се распределат на почеток? Ако бројот на начини на прераспределување е голем, треба да се испечати по модул 100000007.

Анализа на проблемот

Рекурзивната врска која треба да ја изведеме мора да зависи од бројот на деца, n. Она што веднаш се забележува е тоа дека не само што две деца едно до друго не се во маички со иста боја, туку потребно е, и првото, и последното дете од почетната конфигурација да не се во маички од иста боја. Нека бројот на вакви распределувања на n деца го обележиме со $A[n]$. Јасно е дека ако се бара две деца едно до друго да не се во истобојни маички, тогаш бројот на такви распределувања е

$$B[n] = m \cdot (m-1)^{n-1},$$

затоа што бојата на маичката на ороводецот може да се избере на m начини, на тој после него може да се избере било која боја освн бојата на првиот, т.е. на $m-1$ начини, на третиот може да се избере било која боја освен бојата на вториот, т.е. пак на $m-1$ начини. Ако продолжиме вака можеме да заклучиме дека бојата на i-тиот во орото може да се избере од било која боја освен бојата која ја носи $i-1$-от. Но во ваквиот избор се вклучени и подредувањата во кои првиот и последниот во орото имаат мачки во иста боја, па ваквите комбинации треба да ги отстраниме, односно да ги одземеме. Оттука ја имаме следнава релација:

$$A[n] = \#\text{ во редица} - \#\text{ во редица прв и последен иста боја},$$

каде што „# во редица“ е бројот на начини децата да се подредат во редица, така да две соседни деца не се со истобојни маички, кое го обележавме со $B[n]$, а „# во редица прв и последен иста боја“ е бројот на начини децата да се подредат во редица, така да две соседни деца не се со истобојни маички, но првиот и последниот да имаат маички во иста боја.

Кога соседни деца не се облечени во маички во иста боја, а првиот и последниот се во иста боја, тогаш, ако го тргнеме последниот од орото ќе добиеме оро во кое од една страна нема две деца едно до друго со иста боја, а од друга страна првиот и последниот исто така имаат маички од различна боја. Тоа е ситуацијата која ни се бара во задачата, само за оро пократко за едно дете, т.е. бројот на вакви подредувања е $A[n-1]$. Па рекурзивната релација е следнава:

$$A[i] = m \cdot (m-1)^{i-1} - A[i-1].$$

Останува да се определи уште почетниот услов. Да забележиме дека рекурзијата не важи за 2 деца, затоа што во тој случај последното дете е всушност второто дете, па ако децата наредени во редица не се со маички од иста боја, тогаш првиот и последниот не можат да имаат маички во иста боја. Затоа во задачата се бара $n > 1$. Ако имаме две деца, тогаш за првото може да

избереме маичка од било која од m -те бои, а второто, т.е. последното било која од останатите, па $A[2] = m(m-1)$.

Уште една работа која треба да се искоментира во врска со овој проблем е печатењето на решението по модул $10^9 + 7$. Имено, во секој чекор кога се повикува рекурзивната релација потребно е и двата члена во равенката да се пресметаат по модул $10^9 + 7$, но и добиената разлика да се пресмета по истиот модул. Имено, бидејќи равенката содржи одземање, во некои програмски јазици, како на пример C++ може да се случи модулот да добие негативна вредност, па испечатеното решение да не е точно. За да не се случи тоа после пресметката е добро да се додаде $10^9 + 7$ и од добиениот резултат повторно да се побара модул. Решението е дадено со следниов псевдокод, П 2. 9:

П 2. 9. ПСЕВДОКОД ЗА ШАРЕНО ОРО

```
1 Внеси ги n и m;
2 A[2] = m(m − 1);
3 B[2] = m(m − 1);
4 за i = 3 до n прави
5    {
6      B[i] = mod(B[i − 1] ∗ (m − 1), 100000007);
7      A[i] = mod(B[i] − A[i − 1] + 100000007,100000007)
8    }
9 печати A[n].
```

Псевдокодот содржи само еден циклус, па јасно е дека сложеноста е $O(n)$, каде n е бројот на деца во редицата. Но, да продискутираме како може лесно да се излажеме и да дизајнираме многу посложено решение. Имено, проблемот се решава со рекурзивната функција, $A[i]$, со една променлива, која се повикува од една помала инстанца, но и член кој дополнително треба да се пресмета. Овој член е степенска функција, па исто така може да се пресмета со динамичко програмирање, заедно со вредноста на функцијата $A[i]$. Овде тој се пресметува во нова функција $B[i]$.

Имено, можно е да дизајнираме такво решение во кое за секоја вредност на $A[i]$, членот $m \cdot (m-1)^{i-1}$ ќе го пресметуваме посебно, што би ни дало алгоритам кој работи во квадратно време, или ако оваа пресметка ја забрзаме, време $O(n \log n)$. Затоа ова увидување дека двата терма во рекурзивната релација можат да се пресметуваат истовремено, во рамките на ист циклус, е од клучно значење. Всушност може да се направи и посебен код за пресметување на сите $m \cdot (m-1)^{i-1}$, како посебна потпрограма, што исто така не би делувало на зголемување на сложеноста.

Овој алгоритам не може да се забрза до логаритамско време на начинот како што направивме забрзување во проблемот на парички (Проблем 2.1), но може да се изведе аналитичка формула, врз основа на која таквото решение е можно. Тоа ќе биде оставено за читателот како вежба.

На крај да ги дадеме решенијата во Јава и C++:

JAVA 2. 7 ШАРЕНО ОРО

```
1 import java.util.*;
2
3 public class Main {
4     public static void main(String[] args) {
5         Scanner scn = new Scanner(System.in);
6         int n = scn.nextInt();
7         int m = scn.nextInt();
8
9         int[] A = new int[n + 1];
10        int[] B = new int[n + 1];
11
12        A[2] = B[2] = m * (m - 1);
13
14        for (int i = 3; i <= n; i++) {
15            B[i] = (B[i - 1] * (m - 1)) % 100000007;
16            A[i] = (B[i] - A[i - 1] + 100000007) %
100000007;
17        }
18
19        System.out.println(A[n]);
20    }
21}
```

C++ 2. 7 ШАРЕНО ОРО

```
1   #include<iostream>
2  using namespace std;
3
4  int main() {
5     int n;
6     cin >> n;
7     int m;
8     cin >> m;
9
10    int A [n + 1];
11    int B [n + 1];
12
13    A[2] = B[2] = m * (m - 1);
14
15    for (int i = 3; i <= n; i++) {
16       B[i] = (B[i - 1] * (m - 1)) % 100000007;
17       A[i] = (B[i] - A[i - 1] + 100000007) % 100000007;
18    }
19
20    cout << A[n];
21
22    return 0;
23 }
```

Прашања и задачи

1. Анализирај дали проблемот Шарено оро е ист со проблемот на број на начини n деца да се распределат во круг, така да две соседни деца немаат иста маичка?

   a. Ако децата ги сметаме за исти, односно не интересира само бојата на маичката.

   b. Ако децата ги сметаме за различни.

2. Функцијата $A[i]$ во проблемот Шарено оро може да се изрази со помош на степенски ред.

   a. Најди го таквиот облик на функцијата!

   b. Каква сложеност би имал алгоритам кој се базира на формулата изведена под а.?

3. Една низа со должина n е $m - k$ е убава, $3 < m, k < n$ ако е составена од природни броеви помали од m и во секоја подниза со должина 3 има барем еден број стриктно поголем од k.

   a. За дадени n, m и k да се даде рекурзивна равенка врз база на која може да се дизајнира алгоритам со динамичко програмирање кој го пресметува бројот на $m - k$ убави низи.

   b. Направи најбрз можен алгоритам кој го решава овој проблем!

Проблем 2. 5. Телефонски броеви на шахисти

Во претходниот пример имавме алгоритам базиран на две поврзани рекурентни релации. Наредниот пример ќе ни илустрира ситуација со повеќе вакви релации.

Дефинирање на проблемот

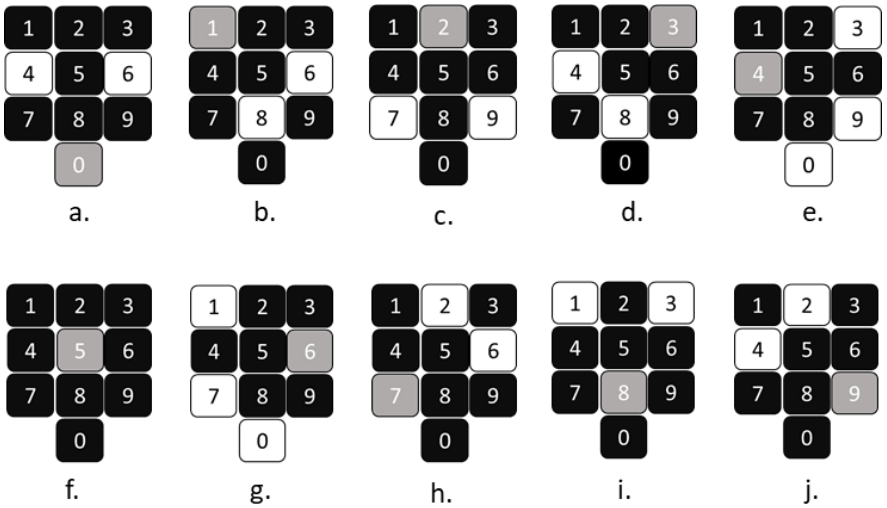
Шахистите се специфични луѓе, па сакаат се да им е поврзано со шахот. Затоа тие бараат нивните телефонски броеви на нумеричката тастатура на телефон (Слика 2. 3.) да може да се притискаат следејќи го движењето на фигурата коњ. Попрецизно, после секоја притисната цифра на тастатурата, тие сакаат следната цифра на бројот да биде на копче на кое би можела да скокне фигурата коњ (две полиња во еден правец и едно нормално на тој правец). Да се најде колку вкупно такви n-цифрени броеви постојат!

Слика 2. 4. Телефонска тастатура. Стрелките го покажуваат начинот на движење на коњ во шахот.

Анализа на проблемот

Јасно е дека рекурзивната врска ќе зависи од должината на телефонскиот број, но интересна забелешка во овој проблем е фактот што бројот на цифри кои можат да дојдат како наредна цифра зависи од претходната цифра. Имено, после бројот 4 можеме да го притиснеме копчето на бројот 3, 9 или 0, односно имаме три можности, додека после копчето со број 1 можеме да избереме само две можности, копчето со број 6 и копчето со број 8. Поради ова, не

можеме да изведеме една единствена рекурзивна равенка која ќе зависи од должината на бројот, туку треба посебно да се разгледа секоја завршна цифра и за секоја цифра како последна да се изведе посебна равенка.



Слика 2. 5. Илустрација на рекурзивните равенки кои го опишуваат последното движење. Сивите копчиња ја претставуваат последната цифра на бројот, додека белите копчиња ја претставуваат претпоследната цифра. a. Пред 0 треба да се притисне 4 или 6. b. Пред 1 треба да се притисне 6 или 8. c. Пред 2 треба да се притисне 7 или 9. d. Пред 3 треба да се притисне 4 или 8. e. Пред 4 треба да се притисне 3, 9 или 0. f. Пред 5 нема цифра. g. Пред 6 е 1, 7 или 0. h. Пред 7 е 2 или 6. i. Пред 8 е 1 или 3. j. Пред 9 е 2 или 4.

Со $A_k[i], k = \overline{0,9}$, ќе го обележиме бројот на телефонски броеви со должина i кои завршуваат на цифрата k. Јасно е дека бројот на броеви со должина i кои заавршуваат на некоја цифра k можеме да го добиеме од броевите кои имаат должина $i-1$, од кои со коњски скок може да се стигне до цифрата k. На Слика 2. 5 е илустрирано како се изведува секоја од равенките. На пример, ако последна цифра во бројот е 0, тогаш пред неа е некоја од цифрите 4 или 6. Оттука,

$$A_0[i] = A_4[i-1] + A_6[i-1].$$

Да забележиме до цифрата 5 не може да се стигне со коњски скок од ниту една друга цифра, па ако последна цифра е 5, тогаш

$A_5[i] = 0$ за секоја вредност на i различна од 1. Копчиња пред кое можат да стојат три вредности се 4 и 6, па за нив имаме:

$$A_4[i] = A_0[i-1] + A_3[i-1] + A_9[i-1],$$
$$A_6[i] = A_0[i-1] + A_1[i-1] + A_7[i-1].$$

Целиот систем од рекурзивни равенки е следниов:

$$\begin{cases} A_0[n] = A_4[i-1] + A_6[i-1] \\ A_1[n] = A_6[i-1] + A_8[i-1] \\ A_2[i] = A_7[i-1] + A_9[i-1] \\ A_3[i] = A_4[i-1] + A_8[i-1] \\ A_4[i] = A_0[i-1] + A_3[i-1] + A_9[i-1] \\ A_5[i] = 0 \\ A_6[i] = A_0[i-1] + A_1[i-1] + A_7[i-1] \\ A_7[i] = A_2[i-1] + A_6[i-1] \\ A_8[i] = A_1[i-1] + A_3[i-1] \\ A_9[i] = A_2[i-1] + A_4[i-1] \end{cases}$$

Да ги определиме иницијалните вредности за секоја од променливите. Најмалите шаховски броеви би имале една цифра, т.е. $i = 1$. Треба да се увиди дека секој едноцифрен број го има бараното својство, бидејќи после притискање на првата цифра, не се притиска друга цифра. Оттука,

$$\forall k = \overline{0,9}, A_k[1] = 1.$$

Вредностите на функцијата $A_k[i]$за првите неколку должини на телефонски броеви се табелирани во Табела 2. 3.

Табела 2. 5. Илустрација на алгоритамот за проблемот „телефонски броеви на шахисти“

n	0	1	2	3	4	5	6	7	8	9

1	1	1	1	1	1	1	1	1	1	1
2	2	2	2	2	3	0	3	2	2	2
3	6	5	4	5	6	0	6	4	5	5
4	12	10	10	10	16	0	16	10	10	10
5	32	26	20	26	32	0	32	26	20	26
6	64	52	52	52	84	0	84	52	52	52

Нека вкупниот број на шаховски броеви со должина n го обележиме со $A[n]$. Тогаш $A[n]$ е збир на сите броеви со должина n кои завршуваат на некоја од десетте цифри, односно,

$$A[n] = A_0[n] + A_1[n] + A_2[n] + A_3[n] + A_4[n] + A_5[n] + A_6[n] + A_7[n] + A_8[n] + A_9[n].$$

Псевдокодот на алгоритамот е даден во П 2. 10.:

П 2. 10. ПСЕВДОКОД ЗА ТЕЛЕФОНСКИ БРОЕВИ НА ШАХИСТИ

```
1 за k = 0 до 9 прави A_k[1] = 1;
2 за i = 2 до n прави
3    за k = 0 до 9 прави
4       пресметај го A_k[i] од формула (2);
5 B=0;
6 за k = 0 до 9 прави B = B + A_k[n];
7 испечати го B.
```

Решението вклучува два вгнездени циклуси, но едниот од нив е со константна должина, 10, па алгоритамот има сложеност $O(10n)= O(n)$. Истото повторно може да се забрза до логаритамска сложеност имајќи во предвид дека системот во матрична форма може да се запише на следниов начин:

$$\begin{bmatrix} A_0[n] \\ A_1[n] \\ A_2[n] \\ A_3[n] \\ A_4[n] \\ A_5[n] \\ A_6[n] \\ A_7[n] \\ A_8[n] \\ A_9[n] \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} A_0[n-1] \\ A_1[n-1] \\ A_2[n-1] \\ A_3[n-1] \\ A_4[n-1] \\ A_5[n-1] \\ A_6[n-1] \\ A_7[n-1] \\ A_8[n-1] \\ A_9[n-1] \end{bmatrix} = .$$

$$= \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}^n \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

Секако, се претпоставува дека бројот на начини може да не го собере во една мемориска локација, и зависно од проблемот, многу често во задачите од оваа област се бара решението да се испечати по модул од некој број. Во тој случај во при секоја пресметка во редовите 4 и 6 од псевдокодот собирањето треба да биде по модул од тој број.

Решенијата кои ги даваме во Јава и C++ се со динамичко програмирање. Решението во јава е дадено со кодот JAVA 2. 8

JAVA 2. 8 ТЕЛЕФОНСКИ БРОЕВИ ЗА ШАХИСТИ

```
1 import java.util.*;
2
3 public class Main {
4     public static void main(String[] args) {
5         Scanner scn = new Scanner(System.in);
6         int n = scn.nextInt();
```

```
7
8           int[][] A = new int[n + 1][10];
9
10          for (int k = 0; k <= 9; k++)
11              A[1][k] = 1;
12
13          for (int i = 2; i <= n; i++) {
14              A[i][0] = A[i - 1][4] + A[i - 1][6];
15              A[i][1] = A[i - 1][6] + A[i - 1][8];
16              A[i][2] = A[i - 1][7] + A[i - 1][9];
17              A[i][3] = A[i - 1][4] + A[i - 1][8];
18              A[i][4] = A[i - 1][0] + A[i - 1][3] + A[i -
1][9];
19              A[i][5] = 0;
20              A[i][6] = A[i - 1][0] + A[i - 1][1] + A[i -
1][7];
21              A[i][7] = A[i - 1][2] + A[i - 1][6];
22              A[i][8] = A[i - 1][1] + A[i - 1][3];
23              A[i][9] = A[i - 1][2] + A[i - 1][4];
24          }
25
26          int B = 0;
27
28          for (int k = 0; k <= 9; k++)
29              B = B + A[n][k];
30
31          System.out.println(B);
32      }
33}
```

Решението во C++ е дадено со:

C++ 2.7 ТЕЛЕФОНСКИ БРОЕВИ ЗА ШАХИСТИ

```
1   #include<iostream>
2  using namespace std;
3
4  int main() {
5     int n;
6     cin >> n;
7
```

```
8     int A [n + 1][10];
9
10    for (int k = 0; k <= 9; k++)
11       A[1][k] = 1;
12
13    for (int i = 2; i <= n; i++) {
14       A[i][0] = A[i - 1][4] + A[i - 1][6];
15       A[i][1] = A[i - 1][6] + A[i - 1][8];
16       A[i][2] = A[i - 1][7] + A[i - 1][9];
17       A[i][3] = A[i - 1][4] + A[i - 1][8];
18       A[i][4] = A[i - 1][0] + A[i - 1][3] + A[i - 1][9];
19       A[i][5] = 0;
20       A[i][6] = A[i - 1][0] + A[i - 1][1] + A[i - 1][7];
21       A[i][7] = A[i - 1][2] + A[i - 1][6];
22       A[i][8] = A[i - 1][1] + A[i - 1][3];
23       A[i][9] = A[i - 1][2] + A[i - 1][4];
24    }
25
26    int B = 0;
27
28    for (int k = 0; k <= 9; k++)
29       B = B + A[n][k];
30
31    cout << B;
32
33    return 0;
34 }
```

Прашања и задачи

1. За даден природен број n, да се пресмета бројот на n цифрени броеви за кои важи дека сите соседни цифри имаат апсолутна разлика еднаква на 1. На пример такви броеви се 321 и 323, но 421 не е.

   a. Да се даде рекурзивната релација!

   b. Да се определат почетните услови!

2. Дадена е шаховска табла со димензии 8×8 и на неа фигура коњ на позиција (a, b). Коњот треба да направи точно k чекори, така што во секој чекор случајно се бира во која од осумте насоки ќе се придвижи. Ако се наоѓа на позиција (x, y)ќе се придвижи до некоја од позициите $(x+2, y+1)$, $(x+2, y-1)$, $(x-2, y+1)$, $(x-2, y-1)$, $(x+1, y+2)$, $(x+1, y-2)$, $(x-1, y+2)$ или $(x-1, y-2)$ Треба да се определи веројатноста коњот да остане на таблата за време на сите k чекори (не може да се врати на таблата ако веќе излезе од неа). Да се даде решение на проблемот!

3. Нека веројатноста да врне $i+1$-от ден, ако врнело i-тиот ден е p, веројатноста да не врне $i+1$-от ден, ако врнело i-тиот ден е q. Да не врне $i+1$-от ден, ако не врнело *i*-тиот ден е q_{2i}. Веројатноста првиот ден да врнело нека е x, а да не врнело y.

   a. Дади линеарен алгоритам за пресметување на веројатноста да врне во n-тиот ден.

   b. До која сложеност може да се забрза претходниот алгоритам?

4. Нека еден систем може да се најде во некоја од m состојби и нека веројатноста да се премине од состојба i во состојба j е p_{ij}. Системот на почеток е во состојба 1. Дади алгоритам за пресметување на веројатностите системот да е во состојба k после n чекори.

5. Да се најде бројот на низи од битови со должина n кај кои секоја низа од нули има барем три нули.

a. Да се даде рекурзивна равенка во матрична форма!

b. Да се најде рекурзивна равенка во која не е во матрична форма за бројот на низи од битови со должина n кај кои секоја низа од нули има барем три нули!

3 Дводимензионални проблеми

Во еднодимензионалните проблеми од динамичко програмирање имаме една или константен број на низи, кои одговараат на некои функции, чии вредности треба да се определат. За разлика од нив, дводимензионалните проблеми може да бидат од два типа. Првиот, најчесто поедноставен тип на проблеми се оние кои се базираат на рекурзивна функција од две променливи, за кои бројот на пресметки кои се потребни да се пресмета една вредност е константен. Во решенијата на овие проблеми се определуваат елементите во матрица која одговара на таа дводимензионална рекурзивна врска. Во ова глава ќе дадеме два такви проблеми. Во првиот проблем, Пат низ матрица, всушност ќе разгледаме две варијанти на истиот проблем, едната комбинаторен проблем, а втората оптимизационен. Идејата за решение на овие проблеми едноставно се базира на определување на подобрата од две можности, слично како проблемот 2.3. за време на производство на две произвотствени ленти. Вториот проблем, најдолга заедничка подниза на прв поглед изгледа многу поразличен проблем од проблемот за пат низ матрица, но всушност идејата ни е да илустрираме како истиот многу лесно може да се сведе на проблемот за пат низ матрица. Вториот тип на проблеми се проблеми кои се базираат на еднодимензионална рекурзивна функција, но за да истата се определи се потребни $O(n)$ операции. Како репрезент од овој тип на проблеми го даваме познатиот проблем на ранец, каде ќе анализираме и какви алчни стратегии можат да не излажат да дадеме погрешно решение на проблемот.

Последниот проблем кој ќе го разгледаме во ова поглавје е со многу посложена структура. Истиот може да се анализира со повеќе приоди, суштински многу различни еден од друг, и нашата идеја е да прикажеме како сите овие идеи водат до точни решенија, иако формулите на кои ќе се базираат се сосема различни. Со еден од овие

приоди исто така сакаме да ја прикажеме идејата за редукција на димензионалноста, техника со која израз кој на прв поглед изгледа дека мора да се пресметува во време $O(n)$, всушност може да се намали на решение со константна сложеност. Оваа техника е всушност аналог на алчната стратегија кај оптимизациони проблеми, на што повеќе ќе се задржиме на шестата глава.

Проблем 3. 1. Пат низ матрица

Со првиот проблем ја илустрираме идејата за опишување на проблем со дводимензионална рекурзивна функција над дискретно множество и во ситуација на комбинаторен проблем и во ситуација на оптимизационен проблем.

Дефинирање на проблемот

Дадена матрица од ред $n \times m$ во која има препреки и може да се движите само надолу или надесно за еден чекор, од едно поле на соседното. Движењето почнува од горниот лев агол, за кој сметаме дека се наоѓа на полето (1, 1) а завршува во долниот десен агол. Дадени се димензиите на матрицата, и позициите на секое поле во матрицата на која не може да се настапне.

a. Да се пресмета бројот на можни патишта!

b. Нека на секое поле (i, j) во матрицата има позитивен број на поени, s_{ij}, кои се освојуваат ако се помине низ тоа поле. Да се најде патот по кој ќе се соберат највеќе поени!

Анализа на проблемот

Идејата на динамичкото програмирање во двете ситуации е иста и се базира на фактот дека сите патишта до позицијата (n, m) доаѓаат или од горе, со поместување од позицијата $(n-1, m)$ до (n, m) или од лево со поместување од позицијата $(n, m-1)$ до (n, m). Во првиот случај се работи за тоа дека бројот на патишта до (n, m) е еднаков на бројот на патишта до $(n-1, m)$ плус бројот на патишта до $(n, m-1)$, а во другата ситуација патот со максимален број на собрани поени до позицијата (n, m) е или патот со максимален број на собрани поени до $(n-1, m)$ плус бројот на поени во полето (n, m), ако може да се стигне до полето $(n-1, m)$, или патот со максимален број на собрани поени до $(n, m-1)$, плус бројот на поени во полето (n, m), ако може да се стигне до полето $(n, m-1)$. Ќе ги изведеме решенијата за секоја задача посебно.

1	1	1	0	0
1	2	3	3	3
1	3	0	3	6
0	3	0	3	9
0	3	3	6	15

Слика 3. 1. Матрица во која некои полиња се забранети за посета. Во секое од полињата е даден бројот на начини на кои мое да се стигне од полето налево најгоре до тоа поле, ако се дозволени движења само надесно и надолу.

Решение на проблемот а.

Нека во задачата под а., со $A[i,j]$ го обележиме бројот на патишта до позицијата (i,j). Тогаш $A[i,j] = 0$ ако на позицијата (i,j) има препрека, ако нема тогаш:

$$A[i,j] = A[i-1,j] + A[i,j-1]. \qquad (3.1)$$

Да забележиме дека во полињата од првата редица може да се стигне само со чекор надесно од полето лево од таа позиција, а во полињата од првата колона може да се стигне само со чекор надолу од полето над таа позиција. Оттука, за полињата во кои е дозволено да се стапне имаме:

$$A[1,j] = A[1,j-1];$$
$$A[i,1] = A[i-1,1].$$

Псевдокодот на алгоритамот е следниов:

П 3. 1. ПСЕВДОКОД ЗА БРОЈ НА ПАТИШТА ВО МАТРИЦА

```
1  A[1,1] = 1;
2  за секоја препрека стави A[i,j] = 0;
3  ако A[1,1] тогаш
4  за j = 2 до m прави
```

```
5    ако A[1,j] ≠ 0 тогаш A[1,j] = A[1,j − 1]
6  за i = 2 до n прави
7     ако A[i,1] ≠ 0 тогаш A[i,1] = A[i − 1,1];
8  за j = 2 до m прави
9      за i = 2 до m до n прави
10        ако A[i,j] ≠ 0 тогаш A[i,j] = A[i,j − 1] + A[i − 1,j]
11 печати A[n,m].
```

На Слика 3. 1 е даден пример за матрица со недозволени полиња за движење, кои се обоени во црна боја. Во секое поле е даден бројот на патишта до тоа поле. Вкупниот број на патишта во овој пример е 15.

Во прилог ги даваме решенијата на овој проблем во програмските јазици C++ и Јава.

C++ 2. 1 БРОЈ НА ПАТИШТА ВО МАТРИЦА СО ПРЕПРЕКИ

```
#include<iostream>
#include <string.h>
using namespace std;

int main() {
   int n;
   cin >> n;
   int m;
   cin >> m;
   int A [n][m];
   memset(A, 0, sizeof(A));

   for (int i=0; i<n; i++)
         for (int j=0; j<m; j++)
            cin >> A[i][j];

   if(A[0][0] != 0)
      A[0][0] = 1;

   for (int j=1;j<m;j++)
      if(A[0][j] != 0)
         A[0][j] = A[0][j-1];
```

```
23
24    for (int i=1;i<n;i++)
25       if(A[i][0] != 0)
26          A[i][0] = A[i-1][0];
27
28    for(int j=2;j<m;j++)
29       for(int i=2;i<n;i++)
30          if(A[i][j] != 0)
31             A[i][j] = A[i][j-1] + A[i-1][j];
32
33    cout << A[n-1][m-1];
34
35    return 0;
36 }
```

JAVA 2. 1 БРОЈ НА ПАТИШТА ВО МАТРИЦА СО ПРЕПРЕКИ

```
1 import java.util.Scanner;
2
3 public class Main {
4
5  public static void main(String[] args)
6  {
7     Scanner scn = new Scanner(System.in);
8     int n = scn.nextInt();
9     int m = scn.nextInt();
10    int A [][] = new int[n][m];
11
12    for (int i=0; i<n; i++)
13        for (int j=0; j<m; j++)
14           A[i][j] = scn.nextInt();
15
16    if(A[0][0] != 0)
17        A[0][0] = 1;
18
19    for (int j=1;j<m;j++)
20        if(A[0][j] != 0)
21            A[0][j] = A[0][j-1];
22
23    for (int i=1;i<n;i++)
24        if(A[i][0] != 0)
25            A[i][0] = A[i-1][0];
```

```
26
27      for(int j=2;j<m;j++)
28          for(int i=2;i<n;i++)
29              if(A[i][j] != 0)
30                  A[i][j] = A[i][j-1] + A[i-1][j];
31
32      System.out.println(A[n-1][m-1]);
33  }
34}
```

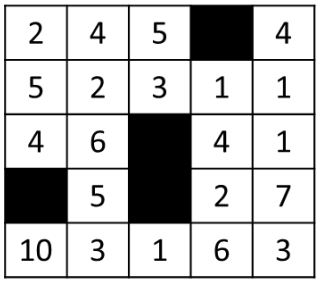
Решение на проблемот b.

Јасно е дека наивниот алгоритам или алгоритамот со „груба сила“ кој го решава проблемот под б) е да ги најдеме сите 15 патишта, за секој од нив да пресметаме колку поени ќе се соберат по тој пат и да видиме кој е најдобар. Тоа би била поголема од експоненцијална сложеност, затоа што толкав е бројот на патишта,во најлош случај, $\binom{n+m}{n}$, бидејќи треба да се направат вкупно $n + m$ чекори, од кои n се во лево.

За да го решиме проблемот со динамичко програмирање, прво треба да го карактеризираме оптималното својство кое гласи:

> *Ако патот со најголем број поени од почетната позиција до позицијата* (n, m) *поминува низ* (i, j)*, тогаш потпатот до* (i, j) *е патот со најголем број поени од почетната позиција до позицијата* (i, j)*.*

Доказ на оптималното својство: Ако не е така, односно оптималниот пат до (n, m) поминува низ (i, j), но потпатот до (i, j) не е патот со најголем број на поени, тогаш можеме да го замениме со поголем пат p_1. Ако сега на патотp_1 му го додадеме стариот пат од (i, j) до (n, m) ќе добиеме пооптимален пат, што е контрадикција. □



2	4	5		4
5	2	3	1	1
4	6		4	1
	5		2	7
10	3	1	6	3

a.

2	6	11	0	0
7	9	14	15	16
11	17	0	19	20
0	22	0	21	28
0	25	26	32	35

b.

Слика 3. 2. А. Матрица со препреки во која се дадени бројот на поени кои можат да се освојат на соодветната позиција, со вредностите s_{ij} б. Матрицата $A[i,j]$ која го дава максималниот број на поени кои можат да се освојат до соодветната позиција.

Нека со $A[i,j]$ го обележиме патот со најголем број поени до позицијата (i,j). Тогаш $A[i,j]=0$, ако на позицијата (i,j) има препрека, затоа што претпоставуваме дека на таа препрека не може да се стигне, што може да се смета дека од таму не можат да се соберат поени. Ако нема препрека, тогаш

$$A[i,j]=\max(A[i-1,j]+1, A[i,j-1]+1).$$

Бидејќи во полињата од првата редица може да се стигне само со чекор надесно од полето лево од таа позиција, а во полињата од првата колона може да се стигне само со чекор надолу од полето над таа позиција, за полињата од овде во кои е дозволено да се стапне имаме:

$$A[1,j]=A[1,j-1]; A[i,1]=A[i-1,1].$$

Псевдокодот на алгоритамот е следниов:

П 3. 2. ПСЕВДОКОД ЗА ПАТ ВО МАТРИЦА СО НАЈГОЛЕМА ВРЕДНОСТ

```
1 A[1,1] = 1;
2 за секоја препрека стави A[i,j] = 0;
3 за j = 2 до m прави
4     ако A[1,j] ≠ 0 тогаш A[1,j] = A[1,j-1] + s_{1j}
5 за i = 2 до n прави
6     ако A[i,1] ≠ 0 тогаш A[i,1] = A[i-1,1] + s_{i1};
7 за j = 2 до m прави
```

```
8   за i = 2 до n прави
9     ако A[i,j] ≠ 0 тогаш
10              A[i,j] = max(A[i,j − 1], A[i − 1,j]) + s_ij
11  печати A[n,m].
```

Ако задачата би ја решавале со рекурзија тогаш сложеноста на двата алгоритми ќе биде експоненцијална, но со динамичко програмирање сложеноста е $O(nm)$, од една страна затоа што решението има два циклуси кои се вгнездени еден во друг, а од друга страна затоа што трба да се пополнат сите полиња на матрица со димензија $n \times m$ и за секое поле имаме константен број на операции.

За реконструкција на оптималното решение можеме да оставаме трага, но овде ќе покажеме како може тоа да го направиме со помош на пресметаната матрица A. Имено, ако од вредноста пресметана со динамичкото програмирање се одземе вредноста во оригиналната матрица, добиената вредност ќе се поклопи со некое од соседните полиња во матрицата A. Тоа ќе значи дека оптималната вредност е добиена од таа страна. На пример, 35 – 3 = 32, па значи дека во оптималното решение последниот чекор бил на лево. Во реконструкцијата ги печатиме полињата на оптималниот пат од позади нанапред. Да забележиме дека сите патишта се со должина $n + m$, па во оваа реконструкција имаме константно толку чекори.

П 3. 3. ПСЕВДОКОД ЗА ПАТ ВО МАТРИЦА СО НАЈГОЛЕМА ВРЕДНОСТ

```
1  i = n; j = m;
2  додека i > 1 или j > 1 прави {
3        ако A[i,j] − s_ij = A[i,j − 1] тогаш
4            {
5             печати (i,j − 1);
6             j = j − 1;
7             }
8             инаку
9               {
```

```
10          (i − 1, j);
11          i = i − 1;
12        }.
```

Реконструкцијата во оваа ситуација може да се направи и на тој начин што придвижувањето наназад ќе биде кон полето со поголема вредност.

JAVA 2. 2 МАКСИМАЛЕН ПАТ ВО МАТРИЦА

```
1 import java.util.Scanner;
2
3 public class Main {
4
5     public static void printMatrix(int[][] matrix) {
6         for (int row = 0; row < matrix.length; row++) {
7             for (int col = 0; col < matrix[row].length;
col++) {
8                 System.out.printf("%4d", matrix[row][col]);
9             }
10            System.out.println();
11        }
12    }
13
14
15 public static void main(String[] args)
16 {
17    Scanner scn = new Scanner(System.in);
18    int n = scn.nextInt();
19    int m = scn.nextInt();
20    int s [][] = new int[n][m];
21    int A [][] = new int[n][m];
22
23    for (int i=0; i<n; i++)
24        for (int j=0; j<m; j++)
25           s[i][j] = scn.nextInt();
26
27    if(s[0][0] != 0)
28        A[0][0] = s[0][0];
29
30    for (int j=1;j<m;j++)
```

```
31          if(s[0][j] != 0 && A[0][j-1] != 0)
32              A[0][j] = A[0][j-1]+s[0][j];
33
34      for (int i=1;i<n;i++)
35          if(s[i][0] != 0 && A[i-1][0] != 0)
36              A[i][0] = A[i-1][0]+s[i][0];
37
38      for(int j=1;j<m;j++)
39          for(int i=1;i<n;i++)
40              if(s[i][j] != 0)
41                  if(A[i][j-1] > A[i-1][j])
42                  {
43                      if(A[i][j-1] != 0)
44                          A[i][j] = A[i][j-1] + s[i][j];
45                  }
46                  else
47                  {
48                      if(A[i-1][j] != 0)
49                          A[i][j] = A[i-1][j] + s[i][j];
50                  }
51
52      System.out.println(A[n-1][m-1]);
53      printMatrix(A);
54
55      //pechati pateka
56      int i=n-1;
57      int j=m-1;
58      System.out.println(i+" "+j);
59      while(i>0||j>0)
60      {
61          if(j>0 && A[i][j] - s[i][j] == A[i][j-1])
62          {
63              System.out.println(i+" "+(j-1));
64              j--;
65          }
66          else
67          {
68              System.out.println((i-1)+" "+j);
69              i--;
70          }
71      }
```

```
72 }
73}
```

C++ 2. 2 МАКСИМАЛЕН ПАТ ВО МАТРИЦА

```
1 #include<iostream>
2 #include <string.h>
3 using namespace std;
4
5
6 int main() {
7  int n;
8  cin >> n;
9  int m;
10 cin >> m;
11 int s [n][m];
12 int A [n][m];
13 memset(A, 0, sizeof(A));
14
15 for (int i=0; i<n; i++)
16     for (int j=0; j<m; j++)
17        cin >> s[i][j];
18
19 if(s[0][0] != 0)
20     A[0][0] = s[0][0];
21
22 for (int j=1;j<m;j++)
23     if(s[0][j] != 0 && A[0][j-1] != 0)
24         A[0][j] = A[0][j-1]+s[0][j];
25
26 for (int i=1;i<n;i++)
27     if(s[i][0] != 0 && A[i-1][0] != 0)
28         A[i][0] = A[i-1][0]+s[i][0];
29
30 for(int j=1;j<m;j++)
31     for(int i=1;i<n;i++)
32         if(s[i][j] != 0)
33             if(A[i][j-1] > A[i-1][j])
34             {
35                 if(A[i][j-1] != 0)
36                     A[i][j] = A[i][j-1] + s[i][j];
37             }
38             else
39             {
40                 if(A[i-1][j] != 0)
```

```
41                      A[i][j] = A[i-1][j] + s[i][j];
42              }
43
44 cout << A[n-1][m-1] << endl;
45
46 //pechati pateka
47 int i=n-1;
48 int j=m-1;
49 cout << i << " " << j << endl;
50 while(i>0||j>0)
51 {
52     if(j>0 && A[i][j] - s[i][j] == A[i][j-1])
53     {
54         cout << i << " " << (j-1) << endl;
55         j--;
56     }
57     else
58     {
59         cout << (i-1) << " " << j << endl;
60         i--;
61     }
62 }
63
64 return 0;
65}
```

Прашања и задачи

1. Како ќе се преправи рекурзивната релација во проблемот за најкраток пат низ матрица, за пат со најмала вредност?

2. Како ќе се преправи рекурзивната релација во проблемот за најкраток пат низ матрица, ако е дозволено да се движите и дијагонално десно - надолу?

3. Матрица од префиксни суми на дадена матрица $A[i,j]$ е матрица $B[i,j]$ кај која секој член се добива со собирање на сите елементи во подматрицата од $A[i,j]$ составена од елементите кои се во редиците помали или еднакви на i и колоните помали или еднакви на j, поточно

$$B[i,j] = \sum_{x=1}^{i} \sum_{y=1}^{j} A[x,y]$$

   Да се даде рекурзивна релација над која може да се конструира алгоритам кој го решава овој проблем во квадратно време.

4. Да се модифицира решението на проблемот за пат во матрица со најголема вредност, за проблем за наоѓање на пат со најголема вредност кој почнува од првата редица, а завршува во последната редица!

5. Модифицирај го решението на проблемот за пат во матрица, за да пресметаш најдолг растечки пат, почнувајќи од полето кое е најлево и најгоре и смееме да се движиме на десно или надолу, но само кон поле во кое елементот има поголема вредност од моменталната позиција во која се наоѓаме.

6. Рамнокрак правоаголен триаголник со должина на катетите n е пополнет со броеви на следниот начин: во првиот ред се распоредени n броеви, во вториот $n - 1$ броеви, итн. во n-тиот ред има 1 број. Од секое поле на триаголникот дозволено е движење десно и долу од тоа поле. Треба да се најде пат од правиот агол до хипотенузата на триаголникот така што збирот на броевите по должината на патот да биде минимален. Односно, поаѓајќи од првиот елемент во првиот ред треба да се

најде минималниот пат до последното поле на било кој ред на триаголникот. Напишете псевдокод кој ќе ја печати вредноста на минималниот пат и позицијата на крајното поле од тој пат.

Проблем 3. 2. Најдолга заедничка подниза

Алгоритамот кој ќе го разгледаме во овој дел се сведува на претходната задача, и има значителна апликативна вредност во биологија, кога се споредуваат ДНК низи на два (или повеќе) различни организми. Една цел заради која две ДНК низи се споредуваат колку се "слични", е да се види колку се тесно поврзани двата организми. Сличноста може да биде дефинирана на различни начини, дали една низа е подниза на друга, што е едноставен алгоритамски проблем или бројот на промени за да едната низа се претвори во другата, што е повторно проблем со динамичко програмирање кој е даден како вежба на читателот. Во нашиот проблем треба да се најде трета низа, што е можно подолга, со бришење на дел од буквите и од едната и од другата низа. Овој алгоритам е и основа за пресметување на сличност и на други објекти и е познат како „динамичко искривување во времето“ кој за првпат е предложен како алгоритам за препознавање на говор. [9].

Дефинирање на проблемот

Подниза на дадена низа се добива со изоставање на нула или повеќе елементи. Формално, за дадена низа $X = (x_1, x_2, \cdots x_m)$, друга низа $Z = (z_1, z_2, \cdots z_n)$ е подниза, ако постои една строго растечка низа $(i_1, i_2, \cdots i_k)$ од индексите на X такви што за сите $j = 1, 2, \ldots, k$, имаме $x_{i_j} = z_j$. На пример, $Z = (B, C, D, B)$ е подниза на $X = (A, B, C, D, A, B)$, каде соодветната низа од индекси на X кои се во Z е 2, 3, 5, 7. За две дадени низи X и Y, велиме дека низата Z е заедничка подниза на X и Y ако е подниза и на двете низи X и Y. На пример, ако $X = (A, B, C, D, A, B)$ и $Y = (B, D, C, B, A, B)$, низата (B, C, A)е заеднички подниза и на двете, но не е најдполгата, бидејќи има должина 3. Низата (B, C, B, A) е исто така заедничка за двете низи X и Y и има должина 4 и бидејќи не постои заедничка подниза со должина 5 или поголема, таа е најдолга заедничка подниза. Во проблемот за најдолга заедничка подниза за две низи треба да ја најдеме нивната заедничка најдолга подниза.

Анализа на проблемот

Ако го решаваме овој проблем со наивниот алгоритам со груба сила, како и во претходниот, треба да се нумерираат сите поднизи на X и се потоа треба да се провери секоја низа за да се види дали е подниза и на Y, при што во секој чекор се зачувува најдолгата до тогаш најдена низа. Ако X има m елементи, тогаш имаме 2^m поднизи на X, па за ова ни треба експоненцијално време. Но и овој проблем има оптимална подструктура, па може да се реши со динамичко програмирање. Отпималната потструктура е дадена со следново својство:

> *Нека се договориме дека со U^i ќе ја обележуваме поднизата $(u_1, u_2, \cdots u_i)$ од низата $U = (u_1, u_2, \cdots u_m)$ за $i \leq m$. За дадени две низи $X = (x_1, x_2, \cdots x_m)$ и $Y = (y_1, y_2, \cdots y_n)$, нека $Z = (z_1, z_2, \cdots z_k)$ е било која нивна најдолга заедничка подниза.*
>
> - Ако $x_m = y_n$, тогаш $z_k = x_m = y_n$ и Z^{k-1} е најдолга заедничка подниза на X^{m-1} и Y^{n-1}.
> - Ако $x_m \neq y_n$ и $z_k \neq x_m$ следува дека Z е најдолга заедничка подниза на X^{m-1} и Y.
> - Ако $x_m \neq y_n$ и $z_k \neq y_n$ следува дека Z е најдолга заедничка подниза на X и Y^{n-1}.

Доказ на оптималното својство: Ако последните две букви во низите се еднакви, $x_m = y_n$, тогаш овој елемент мораме да го ставиме во најдолгата заедничка подниза, односно ќе имаме $z_k = x_m = y_n$, бидејќи ако тој елемент не е во таа низа, тогаш можеме да го додадеме и да добиеме подолга низа. Тогаш остатокот од низата Z^{k-1} мора да биде најдолга заедничка подниза на низите X и Y без последните елементи, односно низите *на* X^{m-1} и Y^{n-1}.

Ако пак $x_m \neq y_n$, тогаш z_k мора да се разликува барем од еден од x_m и y_n. Ако z_k се разликува од x_m, тогаш x_m не е во Z, па следува дека Z е најдолга заедничка подниза на X^{m-1} и Y^n. Слично, ако z_k се разликува од y_n, тогаш y_n не е во Z, па следува дека Z е најдолга заедничка подниза на X^m и Y^{n-1}. □

Од оптималното својство следува дека има или еден или два потпроблеми кои треба да се разгледаат за да се најде најдолгата заедничка подниза на X и Y.

- Ако $x_m = y_n$, треба да ја пресметаме најдолгата заедничка подниза на X^{m-1}и Y^{n-1} и на неа да го додадеме $x_m = y_n$,
- Ако $x_m \neq y_n$, треба да решиме два потпроблеми: да ги пресметаме најдолгата заедничка подниза на X^{m-1} и Y и најдолгата заедничка подниза на X и Y^{n-1} и да ја земе подолгата од нив како најдолгата заедничка подниза на X и Y.

Нека со $A[i,j]$ ја обележиме должината на најдолгата заедничка подниза на X^i и Y^j. Ако или $i = 0$ или $j = 0$, една од низите има должина 0, па најдолгата заедничка подниза има должина 0. Во другите случаи ја намалуваме низата на помали низи. Се добива следнава рекурзивна формула:

$$A[i,j] = \begin{cases} 0, & i = 0 \text{ или } j = 0 \\ A[i-1, j-1] + 1, & i,j > 0 \text{ и } x_i = y_j \\ \max(A[i, j-1], A[i-1, j]), & i,j > 0 \text{ и } x_i \neq y_j \end{cases} . \quad (3.2)$$

Оттука лесно може да се најде рекурзивно решение кое ќе има експоненцијална сложеност.

Решението со динамичко програмирање ќе биде такво што ќе ги пресметуваме вредностите на $A[i,j]$ од помали кон поголеми вредности. Имено, прво ќе се пресметаат ситуациите за основниот случај, а потоа ќе се искористи рекурзивната врска за поголемите случаи. Всушност бидејќи $A[i,j]$ се елементи од матрица, овие пресметани вредности можеме да ги памтиме во матрица, каде што едната од низите ќе ги претставува редиците, а другата колоните, како што е дадено на Слика 3. 3. Всушност задачата се сведува на задачата за најкраток пат во матрица, со таа разлика што овде немаме препреки и во една ситуација на бројот кој е дијагонално горе лево од полето кое треба да го пресметаме додаваме 1, а во друга ситуација го земаме максимумот од полето над и полето лево од полето за кое ја пресметуваме вредноста. Во табелата на секое

поле е дадена стрелка која покажува од кое поле се добива оптималната вредност.

	λ	**B**	D	**C**	A	**B**	**A**
λ	0	0	0	0	0	0	0
A	0	0↑	0↑	0↑	↖1	1↑	↖1
B	0	↖1	1	1	1↑	↖2	2
C	0	1↑	1↑	↖2	2	2↑	2↑
B	0	↖1	1↑	2↑	2↑	↖3	3
D	0	1↑	↖2	2↑	2↑	3↑	3↑
A	0	1↑	2↑	2↑	↖3	3↑	↖4
B	0	↖1	2↑	2↑	3	↖4	4↑

Слика 3. 3. Илустрација на алгоритамот за најдолга заедничка подниза на низите во првата редица и првата колона. Здебелените букви ја даваат најдолгата заедничка подниза. Стрелките покажуваат од кое поле се добива оптималното решение.

Во нашиот псевдокод, во случајот кога карактерите не се еднакви, а оптималната подниза до позицијата над соодветното поле е иста со таа лево од него, се зема вредноста која е над полето. Тоа не менува во вредноста на оптималното решение, но при реконструкцијата, ако го земаме полето лево наместо полето над, може да се добие друга оптимална подниза.

Псевдокодот на алгоритамот за пресметка на оптималното решение е следниов:

П 3. 4. ПСЕВДОКОД ЗА НАЈДОЛГА ЗАЕДНИЧКА ПОДНИЗА

```
1 внеси ги низите X и Y;
2 за i = 0 до m прави A[i, 0] = 0;
3     за j = 0 до n прави A[0, j] = 0;
4 за i = 0 до m прави
5     за j = 0 до n прави
6         ако x_i = y_j тогаш
```

```
7          {
8            A[i,j] = A[i − 1, j − 1] + 1;
9            B[i,j] = koso;
10         }
11          инаку ако A[i − 1, j] ≥ A[i, j − 1] тогаш
12                {
13                 A[i,j] = A[i − 1, j];
14                 B[i,j] = gore;
15                }
16                 инаку
17                    {
18                     A[i,j] = A[i, j − 1];
19                     B[i,j] = levo;
20                    }
21 печати A[n, m].
```

И временската и мемориската сложеност на алгоритамот е $\Theta(mn)$. Имено, пресметуваме елементи на една дводимензионална матрицаматрица со димензии $m \times n$, што значи дека тоа е меморијата која ни е потребна за да ги зачуваме сите потребни вредности. Од друга страна и временската сложеност е иста бидејќи вредноста во секое поле од таа матрица се добива со константен број на пресметки и константен број на споредби.

За реконструкција на едно оптимално решение ќе ја користиме низата B. Во овој пример ќе покажеме како може да се направи реконструкција со рекурзија. За таа цел дефинираме рекурзивна функција $pnzp(B, X, i, j)$, каде B е матрицата која ја зачувува трагата од алгоритамот за пресметка на оптималното решение, а X е првата низа. Следниот псевдокод треба да се повика за последното поле во матрицата, односно $i = m$ и $j = n$ и исто како и во другите проблеми оптималното решение го печати од назад на напред:

П 3. 5. ПСЕВДОКОД ЗА РЕКОНСТРУКЦИЈА НА НАЈДОЛГА ЗАЕДНИЧКА ПОДНИЗА

```
1  pnzp (B, X, i, j)
2  ако i = 0 или j = 0 тогаш застани инаку
3      ако B[i, j] = koso тогаш
4          {
5            pnzp(B, X, i − 1, j − 1);
6            печати x_i
7          }
8          инаку
9              ако B[i, j] = gore тогаш pnzp(B, X, i − 1, j);
10                 инаку pnzp(B, X, i, j − 1).
```

Бидејќи алгоритамот за реконструкција на едно оптимално решение е рекурзивен, неговата временска сложеност се пресметува со помош на рекурзија. Затоа нека со $T[i+j]$ го обележиме времето на работа на алгоритамот повикан од (i, j). Кога се повикува функцијата $pnzp\ (B, X, i, j)$ се прават константен број на пресметки и во една ситуација и двата параметри i и j се намалуваат за еден, но во другите две ситуации само еден од нив се намалува за еден. Оттука, во најлош случај цело време би се намалувал само еден параметар, па

$$T[i+j] = T[i+j-1] + 1.$$

Ова е линеарна нехомогена рекурзивна равенка, па решението е:

$$\begin{aligned} T[m+n] &= T[m+n-1] + 1 = T[m+n-2] + 2 \\ &= \cdots = T[0] + m + n. \end{aligned}$$

Оттука, овој дел има помала сложеност отколку добивањето на вредноста на оптималното решение и е $O(m+n)$.

Кодот за решението во C++ е дадено со C++ 2. 3 :

C++ 2. 3 НАЈДОЛГА ЗАЕДНИЧКА ПОДНИЗА

```
1   #include <iostream>
2
3   using namespace std;
4
5   int main()
6   {
7      string a, b;
8      getline(cin, a);
9      getline(cin, b);
10
11     int A[a.length() + 1][b.length() + 1];
12     for (int i = 0; i <= a.length(); i++)
13     {
14        A[i][0] = A[0][i] = 0;
15     }
16
17     for (int i = 0; i < a.length(); i++)
18        for (int j = 0; j < b.length(); j++)
19           if (a[i] == b[j])
20              A[i + 1][j + 1] = A[i][j] + 1;
21           else
22              A[i + 1][j + 1] = max(A[i + 1][j], A[i][j +
1]);
23
24     cout << A[a.length()][b.length()] << endl;
25     string result = "";
26     for (int x = a.length(), y = b.length(); x != 0 && y !=
0;)
27     {
28        if(a[x - 1] == b[y - 1])
29        {
30           result = a[x - 1] + result;
31           x--;
32           y--;
33        }
34        else if (A[x][y] == A[x][y - 1])
35           y--;
36        else if (A[x][y] == A[x - 1][y])
37           x--;
38     }
39
40     cout << result << endl;
```

```
41
42     return 0;
43 }
```

Решението во јава е дадено со JAVA 2. 3 :

JAVA 2. 3 НАЈДОЛГА ЗАЕДНИЧКА ПОДНИЗА

```
1 import java.util.*;
2
3 public class Main
4 {
5 public static void lcs(String a, String b)
6 {
7 int[][] A = new int[a.length() + 1][b.length() + 1];
8
9            // pravata redica i prvata kolona vekje se
inicijalizirani na 0
10
11       for (int i = 0; i < a.length(); i++)
12          for (int j = 0; j < b.length(); j++)
13             if (a.charAt(i) == b.charAt(j))
14                A[i + 1][j + 1] = A[i][j] + 1;
15             else
16                A[i + 1][j + 1] = Math.max(A[i + 1][j],
A[i][j + 1]);
17
18       System.out.println(A[a.length()][b.length()]);
19       String result = "";
20       for (int x = a.length(), y = b.length(); x != 0 &&
y != 0;)
21       {
22          if(a.charAt(x - 1) == b.charAt(y - 1))
23          {
24             result = a.charAt(x - 1) + result;
25             x--;
26             y--;
27          }
28          else if (A[x][y] == A[x][y - 1])
29             y--;
30          else if (A[x][y] == A[x - 1][y])
31             x--;
32       }
33
34       System.out.println(result);
35    }
36
```

```
37    public static void main(String[] args)
38    {
39       Scanner scanner = new Scanner(System.in);
40       lcs(scanner.next(), scanner.next());
41    }
42 }
```

Прашања и задачи

1. Палиндром е збор кој исто се чита нанапред и наназад. За даден стринг треба да се најде должината на најдолгиот подстринг кој е палиндром. Подстринг на даден стринг е стринг кој се добива со бришење на дел од карактерите. На пример, ако е даден зборот “BBABCBCAB”, тогаш најдолгиот палиндром има должина 7, а тоа е подстрингот “BABCBAB”. Како можеш да го искористиш решението на проблемот за наоѓање на најдолга заедничка подниза за да го решиш овој проблем?

2. Да се најде најдолга повторувачка низа на дадена низа $X[i], i = \overline{1,n}$. Тоа е низа која се јавува двапати во оригиналната низа, но ниту еден член не е заеднички. Попрецизно, да се најдат најдолгите поднизи на $X[i], Y[j]$ и $Z[j], j = \overline{1,k}$, и $k \leq n$, такви што за секое j важи $Y[j] = Z[j]$, но $Y[j] = X[k]$, а $Z[j] = X[k']$ за $k \neq k'$.

   a. Како треба да го модифицираш решението на проблемот за наоѓање на најдолга заедничка подниза за да го решиш овој проблем?

   b. Дади ја рекурзивната релација!

3. Да се најде најдолга растечка подниза од дадена низа од броеви. Попрецизно, да се исфрлат што помалку елементи од дадената низа $X[i], i = \overline{1,n}$ од различни броеви, за да низата која ќе остане биде растечка.

   a. Покажи дека со мала модификација овој проблем се сведува на проблемот за најголема заедничка подниза.

   b. Која е временската и мемориската сложеност на решението под а.?

   c. Како можеш да го модифицираш решението за да добиеш мемориска сложеност $O(n)$?

4. Модифицирај го решението на проблемот за наоѓање на најдолга растечка подниза за да го решиш проблемот за определување на растечка подниза со максимална сума!

a. Дади ја рекурзивната равенка!

b. Дади ја оптималната потструктура!

5. Да се најде најкраткиот можен стринг Z кој е комбинација од два стринга X и Y, така да во новиот стринг Z стринговите X и Y се потстрингови.

   a. Дади ја оптималната потструктура на проблемот!

   b. Докажи ја оптималната потструктура!

   c. Дади ја рекурзивната равенка која го решава проблемот!

6. Дадени се два стринга X и Y . Над секој од нив можат да се извршуваат следниве операции: вметнување и отстранување на карактер во стрингот и замена на еден карактер со друг. Сите горенаведени операции се со еднаква цена. Треба да се направат двата стринга исти со најмал број на операции.

   a. Покажи дека двата стринга можат да се направат исти со најмал број на операции, така да се прават операции само на првиот стринг.

   b. Дади ја оптималната потструктура на проблемот!

   c. Дади ја рекурзивната релација за оптималното решение!

7. Модифицирај ја рекурзивната релација за претходната задача, за да стрингот X го претвориш во стрингот Y за најмала цена, ако операцијата вметнување кошта цена a, операцијата отстранување кошта цена b, а операцијата замена чини c.

Проблем 3. 3. Проблем на ранец

Проблемот на ранец кој ќе го разгледаме во ова поглавје е еден од најкористените примери со кои може да се објасни идејата на динамичкото програмирање. За овој проблем постојат неколку познати варијанти, сите предмети да се различни, да има повеќе предмети од ист тип, предметите да можат да се делат и слично. Овде ќе ја разгледаме варијантата кога имаме неколку типови на предмети и неограничена количина од секој од нив. Останатите варијанти се дадени како вежба.

Овој проблем ќе го искористиме и за да објасниме и како лесно можеме да дојдеме до заблуда во оптимизациони проблеми и да понудиме неточно решение.

Дефинирање на проблемот

Крадецот влегува во просторија во која што се чуваат скапоцени предмети. Тој носи ранец во кој има носивост n, односно во него можат да сместат предмети ко имаат тежина најмногу n единици Во просторијата има вкупно m типови на предмети и сметаме дека предметите се повеќе отколку што може да се собере во ранецот, и уште повеќе претпоставуваме дека од секој тип на предмети има доволна количина, така да крадецот ако сака може да го наполни ранецот само со еден тип на предмети. За секој тип на предмет позната е неговата вредност v_k и неговата тежина $t_k, k = \overline{1,m}$. Сите големини се целобројни. Целта на крадецот е да украде што е повредна стока и нашиот проблем е да се одредат предметите кои треба да ги стави во ранецот и нивната вкупна вредност.

Анализа на проблемот

Прва идеја која би ни паднала напамет е да го пополниме ранецот колку што е можно повеќе, но да забележиме дека ранецот кој што е оптимално пополнет не мора да биде пополнет до врвот, или попрецизно, збирот на волумените на предметите ставени во ранецот не мора да биде еднаков на волуменот на ранецот. Важно е

тој збир да не е поголем од волуменот на ранецот, а во исто време збирот на вредностите на тие предмети да биде максимален. Ова можеме да го прикажеме на еден едноставен пример. Нека ранецот има носивост од $n = 7$ единици и нека постојат $m = 3$ типови на предмети такви што предметите од првиот тип вредат 3 и тежат 3, предметите од вториот тип имаат вредност 4 и тежат 4, а предметите од третиот тип имаат вредност 8 и тежат 5. Ранецот може да се пополни до врвот така што во него крадецот ќе ги стави по еден предмет од првиот и вториот тип, бидејќи $t_1 + t_2 = 3 + 4 = 7 = n$. Но, ваквото пополнување не е оптимално бидејќи вредноста која би се ставила во ранецот е $v_1 + v_2 = 3 + 4 = 7$, додека ако во ранецот се стави само третиот предмет, кој има тежина 5, се добива ранец со повредна содржина $v_3 = 8$. Ова значи дека не мора да е трудиме во ранецот да ставиме предмети со поголема тежина, т.е. да го пополниме што е можно повеќе во тежина и ваквата стратегија нема да работи.

Наредна идеја која може да ни падне напамет е во ранецот прво да ставиме од највредните предмети, се додека може да се стави некој од нив, па потоа остатокот да го пополниме со предмети од останатите типови. Значи, би требало типовите на предмети да ги наредиме по нивната вредност, и во секој момент, кога во ранецот ни се ставени одреден број на предмети и останало да се пополни одредена носивост, да се избере највредниот предмет таков што неговата тежина е помала отколку она што може да се стави во ранецот, односно, да може да го собере во веќе до некаде пополнетиот ранец. Ова навистина изгледа логично, но веднаш со пример можеме да видиме дека и оваа стратегија не е добра. Нека ранецот пак има капацитет од $n = 7$ тежински единици и нека постојат $m = 3$ предмети такви што првиот тип предмети имаат вредност 4 и тежина 3, вториот тип вредност 5 и тежина 4, а третиот тип предмети имаат вредност 8 и тежина 5. Тогаш со оваа стратегија прво во ранецот би ставиле еден предмет од третиот тип кој ќе пополни 5 единици и веќе нема да можеме во ранецот да ставиме ништо. Вредноста која ќе ја добиеме ќе биде 8, па, секако е подобро да ставиме по еден предмет од првиот и вториот тип со што би се добила содржина вредна $v_1 + v_2 = 4 + 5 = 9$. Ваквата стратегија се

нарекува алчна стратегија, која е разработена во шестата глава од оваа книга.

Претходниот пример ни разјаснува дека најскапиот предмет не мора да биде добро решение за во ранецот, затоа што неговата цена по единица тежина може да биде мала. Па се добива впечаток дека до решението се доаѓа така што се одредува k за кое v_k/t_k најголемо, па прво ранецот се полни со предмети од оној тип за кој овој однос е најголем. Ова е повторно една идеја која користи алчна стратегија и на прв поглед изгледа дека навистина сме го решиле проблемот, но сепак не е така. На пример, нека пак имаме ранец со носивост $n = 7$ и нека имаме $m = 3$ типови на предмети такви што првиот тип се со вредност 3 и тежина 3, вториот со вредност 4 и тежина 4, а третиот со вредност 6 и тежина 5. Наведената идеја наложува прво да се стават предмети од третиот тип, како највреден по единица тежина. Според тоа во ранецот би се ставил само еден од предмет, а вредноста на ранецот би била еднаква на 6. Лесно може да се согледа дека ако избереме по еден предмет од првите два типови ќе ни даде вредност на ранецот 7, што е подобро, а воедно и оптимално решение. Значи ни оваа идеја не добра, па мораме да се насочиме кон поинакво размислување, и некако да се откажеме од користење на алчна стратегија. Бидејќи овие три пристапи паднаа во вода, ни изгледа разочарувачки и невозможно дека проблемот има шанса да се реши со избегнување да се разгледаат сите можни варијанти. Но сепак ќе видиме дека може да се најде решение, кое секако нема да биде толку брзо како овие претходни идеи, но сепак ќе биде далеку побргу отколку да се разгледуваат сите можни пополнувања на ранецот.

Проблемот го има следново оптимално својство:

> *Ако при оптимално пополнување на ранецот последен избран предмет е од тип k, тогаш претходно избраните предмети оптимално го пополнуваат ранецот со капацитет $n - t_k$.*

Доказ на оптималното својство: Својството лесно се докажува со користење на копирај - залепи техниката. Имено да претпоставиме дека сме го нашле оптималниот начин за пополнување на ранец со големина n, така што последен е ставен производ од k-тиот тип, но

ранецот со капацитет $n - t_k$ не е оптимално пополнет, т.е. може да се пополни со производи чија вкупна цена е поголема отколку таа што сме ја добиле по нашето пополнување. Тогаш, наместо нашето пополнување на ранецот со капацитет $n - t_k$, можеме да го пополниме на овој подобар начин, и пак на крај да го ставиме k-тиот производ. На тој начин ќе добиеме пополнување на целиот ранец кое е подобро отколку она за кое сметавме дека е оптимално, што е контрадикција.

Сега ќе докажеме дека за секој цел број n и дадените типови на предмети, кои не се менуваат, може да се најде најдобро пополнување на ранецот со капацитет n. Оптималното решение за $n = 0$ е празен ранец. Нека се познати сите оптимални решенијата за ранци со капацитет $i < n$ и нека е $A[i]$ е вкупната вредност за оптимално пополнување на ранец со големина i, а $C[i]$ низата на редни броеви на предметите кои во тој случај би биле ставени во ранецот. Нека како последен предмет се става предмет од тип k, $k = \overline{1, m}$. Секако, за да воопшто може да го ставиме тој предмет мора да важи $n \geq t_k$. Најдоброто пополнување во ваков случај може да се добие како:

$$A[n - t_k] + v_k.$$

Ова ќе го пробаме за секој од m-те дадени предмети и тврдиме дека:

$$A[i] = \max_{l \leq k \leq m} (A[i - t_k] + v_k) \qquad (3.3)$$

е оптималното пополнување на ранец со капацитетот *i*. Ова следува директно од оптималната на подструктура на проблемот која ја покажавме претходно и тоа што само еден предмет мора да биде последен, а ние го испробувавме секој тип на предмети како последен.

Ако максималната вредност во (3. 3) се добива за предмет од тип k, тогаш во ранецот ќе додадеме еден таков предмет, кој ќе го обележиме со x_k. Па

$$C[i] = C[i - t_k] \cup \{x_k\}.$$

Да забележиме дека нема потреба да се памети целото множество $C[i]$, т.е. доволно е да се запамети само последниот член на низата. Тогаш сите елементи може да се најдат во обратен редослед како:

$$a = C[n], b = C[n - t_a], c = C[n - t_a - t_b].$$

Врз основа на дадените рекурзивни равенки, можеме да направиме рекурзивен алгоритам кој дава едно оптимално пополнување на ранецот. Во псевдокодот на рекурзивниот алгоритам, П 3. 6. кој го даваме овде не се печатат предметите кои ќе се најдат во ранецот при едно негово оптимално пополнување и тоа ќе биде оставено на читателот како вежба.

П 3. 6. РЕКУРЗИВЕН АЛГОРИТАМ ЗА ПРОБЛЕМ НА РАНЕЦ

```
1 RanecRek[i];
2 B = 0;
3 ако i > 0 тогаш
4     за k = 1 до m прави
5         ако t_k ≤ i тогаш
6             ако RanecRek [i − t_k] + v_k > B тогаш
7                 B = RanecRek[[i − t_k] + v_k];
8 врати B.
```

Од самиот алгоритам се гледа дека неговата сложеност е многу голема. Ако $F(n)$ бројот на чекори за проблем со големина n, тогаш рекурзивната равенка за сложеноста би била:

$$F(n) = \sum_k F(n - t_k) + 1.$$

Бидејќи има сигурно барем два предмети, сложеноста е барем $O(2^n)$. Комплексноста на рекурзивното решение е пред се заради голем број на повикувања на $RanecRek$ за иста вредност. Тоа можеме да го надминеме така што ќе ги запаметиме вредностите за оптимално пополнување на ранци, пополнувајќи табели за

оптималната вредност на ранецот, т.е. вредности за $A[i]$ и $C[i]$. Табелата почнуваме да ја пополнуваме за ранец со големина 0, и потоа користејќи ги оптималните вредности за помалите ранци, ја пресметуваме оптималната за поголемите.

Да го разгледаме решението на ранец со волумен 14 и три типа на производи со вредности 2, 3 и 6, и тежини 2, 4 и 5 соодветно. Нека. Во Табела 3. 1. се дадени сите капацитети помали од 15.

Табела 3. 1. Оптималното пополнување на ранци со големини од 0 до 14.

i	$v_1 = 2, t_1 = 2$	$v_2 = 3, t_2 = 4$	$v_3 = 6, t_3 = 5$	$A[i]$	$C[i]$
0				0	
1				0	
2	2+*A*[0]=2			2	1
3	2+*A*[1]=2+0=2			2	1
4	2+*A*[2]=2+2=4	3+*A*[0]=3+0=3		4	1,1
5	2+*A*[3]=2+2=4	3+*A*[1]=3+0=3	6+*A*[0]=6+0=6	6	3
6	2+*A*[4]=2+4=6	3+*A*[2]=3+2=5	6+*A*[1]=6+0=6	6	1,1,1
7	2+*A*[5]=2+6=8	3+*A*[3]=3+2=5	6+*A*[2]=6+2=8	8	3,1
8	2+*A*[6]=2+6=8	3+*A*[4]=3+4=7	6+*A*[3]=6+2=8	8	1,1,1,1
9	2+*A*[7]=2+8=10	3+*A*[5]=3+6=9	6+*A*[4]=6+4=10	10	3,1,1
10	2+*A*[8]=2+8=10	3+*A*[6]=3+6=9	6+*A*[5]=6+6=12	12	3,3
11	2+*A*[9]=2+10=12	3+*A*[7]=3+8=11	6+*A*[6]=6+6=12	12	3,1,1,1
12	2+*A*[10]=2+12=14	3+*A*[8]=3+8=11	6+*A*[7]=6+8=14	14	3,3,1
13	2+*A*[11]=2+12=14	3+*A*[9]=3+10=13	6+*A*[8]=6+8=14	14	3,1,1,1,1
14	2+*A*[12]=2+14=16	3+*A*[10]=3+12=15	6+*A*[9]=6+10=16	16	3,3,1,1

Псевдокодот на алгоритамот е:

П 3. 7. АЛГОРИТАМ ЗА ПРОБЛЕМ НА РАНЕЦ

```
1 A[0] = C[0] = 0;
2 за i од 1 до n прави
3   {
4     A[i] = 0 = C[i] = 0;
5     за k од 1 до m прави
6       ако  t_k ≤ i тогаш
```

```
7          ако A[i − t_k] + v_k > A[i] тогаш
8            {
9             A[i] = A[i − t_k] + v_k;
10            C[i] = k;
11            }
12  печати A[n].
```

Со П3.8. предметите се печатат во обратен редослед:

П 3. 9. РЕКОНСТРУКЦИЈА НА ОПТИМАЛНОТО РЕШЕНИЕ ВО ПРОБЛЕМ НА РАНЕЦ

```
1 i = n;
2 додека C[i] > 0 прави
3  {
4   печати C[i];
5   i = i − t_{C[i]};
6  }
```

Сложеноста може лесно да се пресмета броејќи го бројот на операции во циклусите. Имаме два вгнездени циклуси, еден од 1 до n, а втор од 1 до m, во кои има константен број на операции, па сложеноста е $O(nm)$. Реконструкцијата користи само еден циклус, па оттука целата сложеност е $O(nm)$.

JAVA 2. 4 ПРОБЛЕМ НА РАНЕЦ

```
1  import java.util.Scanner;
2
3  public class Main {
4  public static void main(String[] args)
5  {
6        Scanner inp = new Scanner(System.in);
7        int m = inp.nextInt();
8        int v[] = new int[m];
9        int t[] = new int[m];
10
11       for (int i=0; i<m; i++)
12          v[i] = inp.nextInt();
13       for (int i=0; i<m; i++)
```

```
14          t[i] = inp.nextInt();
15
16       int n = inp.nextInt();
17
18       int A[] = new int[n];
19
20       for (int i=1; i<n; i++)
21       {
22          for (int k=0; k<m; k++)
23          {
24             if(t[k]<=i)
25                if (A[i-t[k]]+v[k] > A[i])
26                   A[i] = A[i-t[k]] + v[k];
27          }
28       }
29
30       System.out.println(A[n-1]);
31    }
32 }
```

C++ 2. 4 Проблем на ранец

```
1  #include<iostream>
2  #include <string.h>
3  using namespace std;
4
5  int main() {
6     int m;
7     cin >> m;
8     int v [m];
9     int t [m];
10
11    for (int i=0; i<m; i++)
12       cin >> v[i];
13    for (int i=0; i<m; i++)
14       cin >> t[i];
15
16    int n;
17    cin >> n;
18
19    int A [n];
20    memset(A, 0, sizeof(A));
21
22    for (int i=1; i<n; i++)
23    {
24       for (int k=0; k<m; k++)
25       {
```

```
26          if(t[k]<=i)
27             if (A[i-t[k]]+v[k] > A[i])
28                A[i] = A[i-t[k]] + v[k];
29       }
30    }
31
32    cout << A[n-1];
33
34    return 0;
35 }
```

Прашања и задачи

1. Во проблемот познат како 0-1 проблем на ранец, наместо неограничена количина од секој тип на предмети има само по еден предмет од секој тип.

    a. Како ќе ја модифицираш рекурзивната релација за да го решиш овој проблем?

    b. Која е сложеноста на вашето решение?

2. Во проблемот познат како фракционен проблем на ранец има ограничена количина на предмети, но од истите можеме да земеме колкав сакаме дел, како на пример ориз, грав, брашно.

    a. Како ќе се реши овој проблем?

    b. Која е сложеноста на вашето решение?

3. Како ќе се промени рекурзивната равенка ако наместо еден имаме два ранци со капацитети n_1 и n_2?

4. За дадено множество од n позитивни броеви, потребно е да се подели во две подмножества, такви што разликата на сумите на елементите од двете множества да биде најмала можна, т.е. апсолутната вредност на разликата во сумите да биде најмала.

    a. Која е сличноста на овој проблем со проблемот на ранец?

    b. Дади алгоритам кој ќе го реши проблемот.

    c. Која е сложеноста на алгоритамот?

5. Треба да се врати кусур од n денари, при што на располагање се неограничен број на монети со вредности од множеството $\{s_1, s_2, \dots, s_k\}$. На колку начини може да се направи тоа?

    a. На кои потпроблеми може да се подели проблемот над множество $\{s_1, s_2, \dots, s_i\}, i \leq k$ и сума од m денари?

    b. Да се даде рекурзивната равенка за алгоритам кој го решава овој проблем!

    c. Да се определи сложеноста на решението.

6. Имаме n лица и две идентични машини за гласање. За секое лице е дадено колку време ми е потребно да гласа на било машина. Само едно лице може да гласа на секоја од машините.

    a. Како ќе се реши проблемот да се израчуна кое е минималното време сите да гласаат?

    b. Како ќе се реши проблемот да се определи дали сите можат да гласаат ако е дадено максималното време за кое треба сите да завршат со гласањето?

Проблем 3. 4. Правилно распределување парови на загради

Ова е еден интересен комбинаторен проблем, од два аспекти. Прво, има повеќе приоди за решавање на истиот за решавање. Овде ќе разгледаме три различни пристапи кои суштински се разликуваат, се разликува идејата за генерирање на комбинациите, а и самата рекурзивна равенка која го опишува проблемот, но сите водат до алгоритам со иста временска сложеност. Главната идеја на решението е во три димензии, но можеме да примениме техника наречена редукција на димензионалноста, со која што проблемот ќе го намалиме на две димензии. Посебно тоа ќе го видиме во првиот пристап, каде што прво ќе изведеме рекурзивна релација над која може да се конструира кубен алгоритам, а потоа ќе покажеме како тоа може да се редуцира на две димензии.

Вториот аспект поради кој овој проблем е интересен е тоа што се јавува во повеќе апликации. Ние овде ќе ја разгледаме варијантата на број на правилно распределени загради, но истиот број на распоредувања се јавува и на други места, кои се дадени како вежба за студентите.

Дефинирање на проблемот

За дадено n да се најде бројот на стрингови со должина $2n$, кои се состојат од n правилно распоредени парови на загради. Заградите се правилно распоредени ако за секое k од 1 до n, k-тата отворена заграда стои пред k-тата затворена заграда ако имаме три пара на загради, правилно можеме да ги распределиме на следниве начини: ((())), (())(), (()()), ()(()), ()()(), ()(()).

Анализа на проблемот

Различните пристапи за броење на бројот на правилно распределени загради се базираат на различните начини на нивно генерирање. Во петтото поглавје овој проблем ќе го разгледуваме од поинаков агол, и ќе видиме

дека секој пристап соодветствува на соодветен начини на подредување на комбинациите.

Прв начин:

Првиот пристап се базира на следнава забелешка:

> *Последната отворена заграда мора да биде на некоја од позициите од* n *до* $2n - 1$*. Да претпоставиме дека таа е на позицијата* k*. Тогаш претпоследната отворена заграда мора да биде на некоја позиција од* $n - 1$ *до* $k - 1$*.*

Нека со $A[m, k]$ го означиме бројот на стрингови кои се состојат од m правилно распоредени парови на загради, на кои последната отворена заграда е на позицијата k. Тогаш сите загради после неа се затворени загради и ако ја тргнеме неа заедно со затворената заграда веднаш после неа, ќе добиеме $n - 1$ парови на правилно распоредени загради. Последната отворена заграда после тргањето на овие две загради мора да биде на позиција пред k, но бидејќи ќе останат $n - 1$ парови на загради, таа позиција не може да биде порано од $n - 1$ -та позиција. Оттука ја добиваме следнава рекурентна врска:

$$A[m, k] = A[m-1, k-1] + A[m-1, k-2] + \cdots$$
$$\ldots + A[m-1, m-1]. \quad (3.4)$$

Ако пресметката ја правиме по оваа формула, тогаш за тоа ќе ни требаат $O(m - k)$ операции за тоа. Но, ако забележиме дека

$$A[m-1, k-2] + A[m-1, k-3] + \cdots + A[m-1, m-1]$$

е всушност еднакво на $A[m, k-1]$, рекурентната врска се сведува на:

$$A[m, k] = A[m-1, k-1] + A[m, k-1], \quad (3.5)$$

за што ни треба само една операција.

Останува да се изведе основниот случај, кога $m = 1$. Тогаш има само еден правилен распоред на загради, (), па $A[1,1] = 1$. Целосната формула е следнава:

$$A[m,k] = \begin{cases} 1, & m = 1, k = 1 \\ 0, & k < m \text{ или } k \geq 2m \\ A[m-1,k-1] + A[m,k-1], & m \leq k < 2m \end{cases} \quad (3.6)$$

Но ова не е бројот на сите правилни распореди на загради, туку само на оние на кои последната отворена заграда е на k-та позиција. Бараниот број е сума по сите k за $A[n,k]$.

Да забележиме дека вредностите 0 кога $k < m$ или $k \geq 2m$ мораме да ги запишеме затоа што се случува рекурзијата да се сведе до таква ситуација, па ако вредноста не е ставена може да се добие грешка во пресметките или програмата да не работи. Од друга страна може да се забележи дека $A[m,m] = 1$ и со тоа може да се избегнат дополнителни пресметки. Псевдокодот за решавање на овој проблем со првиот принцип е даден во:

П 3. 10. ПРАВИЛНО РАСПРЕДЕЛУВАЊЕ НА ЗАГРАДИ – ПРВ НАЧИН

```
1  внеси го n;
2  за m од 1 до n прави
3     за k од 1 до 2n – 1 прави A[m,k] = 0;
4  за k од 1 до 2n – 1 прави A[m,m] = 1;??
5  за m од 2 до n прави*
6        за k од m + 1 до 2m – 1 прави
7              A[m,k] = A[m – 1,k – 1] + A[m,k – 1];
8  s = 0;
9  за k од n до 2n – 1 прави s = s + A[n,k];
10 печати s.
```

Сложеноста на овој алгоритам е $O(n^2)$ затоа што имаме два вгнездени циклуси еден во друг. Во ова решение се пополнува матрица од ред $n \times (2n-1)$, иако не мора да ги пополниме сите

полиња во неа, туку само полата. Матрицата која се добива за 5 парови загради е:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 2 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 3 & 5 & 5 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 4 & 9 & 14 & 14 \end{bmatrix}.$$

Вкупниот број на правилно распоредени загради е 1+4+9+14+14=42.

ВТОР НАЧИН:

Вториот пристап се базира на следнава забелешка:

> *Во еден стринг од правилно напишани загради, во секој префикс, односно подстринг кој почнува од првата заграда, бројот на отворени загради е поголем или еднаков на бројот на затворени загради. Нека првата позиција кога бројот на отворени и затворени загради се изедначува е* $2k$*. Оваа позиција е парна затоа што овде мора да има парен број на загради, полата отворени, а полата затворени. Тогаш подстрингот од меѓу првата и* k*-тата заграда е стринг од правилно напишани* $k-1-$ *ен пар на загради. Исто така и подстрингот од* $k+1$*-та до последната заграда е стринг од правилно напишани загради, во кој се останатите* $n-k-1$ *парови загради. Оттука, бројот на стрингови од правилно запишани загради со должина* $2n$ *во кои бројот на отворени и затворени загради првпат се изедначува на позицијата* $2k$ *е бројот на стрингови од правилно запишани загради со должина* $2k-2$ *по на стрингови од правилно запишани загради со должина* $2(n-k-1)$

Нека со $A[i]$ го означиме бројот од правилно напишани i парови загради, односно стринг со должина $2i$. Бројот на стрингови

од правилно запишани загради со должина $2n$ во кои бројот на отворени и затворени загради првпат се изедначува на позицијата $2k$, односно првата заграда се затвора со затворената заграда на позиција $2k$ е

$$A[k-1] \cdot A[n-k]$$

Бидејќи првата позиција кога бројот на отворени и затворени загради се изедначува може да биде било која парна позиција од 2 до $2n$, имаме:

$$A[n] = A[0] \cdot A[n-1] + A[1] \cdot A[n-2] + \cdots + A[n-1] \cdot A[0]. \quad (3.7)$$

$A[0]$ означува празен стринг и тој се добива кога префиксот на стрингот е () или кога првиот пат кога има изедначување е на крај, т.е. позиција $2n$.

Почетните услови се бројот на празни стрингови, кој еден, т.е. празниот стринг и стрингови со еден пар загради, кој е исто 1, т.е. стрингот (). Оттука $A[0] = A[1] = 1$. Псевдокодот би бил следниот:

П 3. 11. ПРАВИЛНО РАСПОРЕДУВАЊЕ НА ЗАГРАДИ – ВТОР НАЧИН

```
1 внеси го n;
2 A[0] = 1;
3 A[1] = 1;
4 за i од 2 до n прави
5  {
6    A[i] = 0
7    за k од 1 до i прави A[i] = A[i] + A[k − 1] · A[n − k];
8  }
9 печати A[n, k];
```

Сложеноста на овој алгоритам е $O(n^2)$ затоа што и овде имаме два вгнездени циклуси еден во друг. Но во ова решение не се пополнува матрица од туку само низа, што значи дека има помала мемориска сложеност од претходниот начин. Елементите од низата кои се добиваат за 5 парови на правилно распределени загради се:

$$A[0] = 1;$$

$$A[1] = 1;$$

$$A[2] = A[0]A[1] + A[1]A[0] = 1 + 1 = 2;$$

$$A[3] = A[0]A[2] + A[1]A[1] + A[2]A[0] = 2 + 1 + 2 = 5;$$

$$A[4] = A[0]A[3] + A[1]A[2] + A[2]A[1] + A[3]A[0]$$
$$= 5 + 2 + 2 + 5 = 14;$$

$$A[5] = 2 \cdot A[0]A[4] + 2 \cdot A[1]A[3] + A[2]A[2]$$
$$= 2 \cdot 14 + 2 \cdot 5 + 4 = 42.$$

Интересно е да се напомене дека равенката (3. 7) има експлицитно решение, кое е познато како Каталионов број. Овој број е еднаков на $A[n] = \frac{1}{n+1}\binom{2n}{n}$ и покажано е дека неговата приближна вредност е $\frac{4^n}{n^{3/2}\sqrt{\pi}}$, што значи дека бројот на вакви поставувања на загради е $\Omega(2^n)$. Овде нема да објаснуваме како се доаѓа до ова решение, а читателите кои сакаат да се информираат повеќе за овој број можат да прочитаат во [10], [11], [12].

ТРЕТ НАЧИН:

Третиот пристап ја имитира градбата на стрингот, заграда по заграда. Имено, да претпоставиме дека во даден момент имаме напишано дел од заградите, водејќи се по правилото на правилно запишани загради. Тогаш во тој момент бројот на отворени загради е поголем или еднаков на бројот на затворени загради. Ние ќе броиме уште на колку начини ќе можеме да ги додадеме останатите загради. На пример ако треба да ставиме 3 пара загради, а во даден момент имаме напишано „(()“, тогаш еден пар загради е целосно напишан, на еден пар треба да се допише само затворената заграда, а еден пар целосно треба да се изнапише. Целта е да изброиме уште на колку начини можеме да додадеме една отворена и две затворени загради,така да се добие стринг од правилно запишани загради. Рекурзијата која овде ќе ја градиме ќе оди по бројот на парови

загради кои треба целосно да се допишат и бројот на дополнителни затворени загради кои треба да се допишат. Така, ако треба да се допишат i целосни парови загради и да се допишат уште j затворени загради, со $A[i,j]$ ќе го обележиме бројот на начини на кои може да се направи тоа. Во горниот пример, бројот на правилно запишани 3 пара загради кои започнуваат со „((“ е $A[1,1]$, затоа што на овој стринг треба да се додаде уште цел пар загради и една затворена заграда. Оттука бројот на начини на кои треба да се запишат 3 пара загради е $A[3,0]$, затоа што ништо не сме почнале да пишуваме, па треба да се допишат само сите 3 пара загради (затворени загради за допишување нема).

Пак да се навратиме на моментот кога имаме запишано дел од заградите, и треба да се допишат уште i целосни парови загради и j затворени загради, што рековме дека може да се направи на $A[i,j]$ начини. За наредната заграда имаме два случаи:

- Да ставиме „(“, што значи дека на така добиениот стринг ќе треба целосно да се допише еден пар загради помалку, т.е. уште $i-1$ пар загради, и една затворена заграда повеќе, т.е. уште $j+1$ затворени загради, што може да се направи на $A[i-1,j+1]$ начини.
- Да ставиме „)“, што затвара една отворена заграда, па бројот на загради кои целосно треба да се допишат останува ист, i, а бројот на затворени загради кои треба да се допишат се намалува за еден, т.е. на така добиениот стринг ќе треба да се додадат уште $j-1$ затворени загради, што може да се направи на $A[i,j-1]$ начини..

Оттука,

$$A[i,j]=A[i-1,j+1]+A[i,j-1].$$

Секако, ако во стрингот кој до некаде ни е напишан веќе се ставени сите отворени загради, тогаш не треба да се допишуваат цели парови загради и во таква ситуација треба само до крај да се допишат затворени загради, што може да се направи само на еден начин. Ова е основниот случај, кој во нашата рекурзија се бележи со $A[0,j]$ и $A[0,j]=1$. Од друга страна, ако во стрингот кој до некаде

ни е напишан бројот на отворени затворени загради е ист, тогаш не може наредна да биде затворена заграда, па во оваа ситуација го имаме само првиот член, т.е. $A[i,0] = A[i-1,1]$.

Целосната равенка следнава:

$$A[i,j] = \begin{cases} 1, & i = 0 \\ A[i-1,1], & j = 0 \\ A[i-1,j+1] + A[i,j-1]., & \text{инаку} \end{cases}. \qquad (3.8)$$

Пресметката почнува од $A[n,0]$, бидејќи на почеток треба да се стават сите n парови загради и немаме ниедна затворена заграда за додавање.

Да видиме како оваа рекурзија ќе работи на случај со три пара загради:

$$\begin{aligned} A[3,0] = A[2,1] = A[1,2] + A[2,0] &= A[0,3] + A[1,1] + A[1,1] \\ &= 1 + 2(A[0,2] + A[1,0]) = 1 + 2(1 + A[0,1]) \\ &= 1 + 2 \cdot 2 = 5. \end{aligned}$$

И овој алгоритам работи во време $O(n^2)$, а псевдокодот е следниот:

П 3. 12. ПРАВИЛНО РАСПОРЕДУВАЊЕ НА ЗАГРАДИ

```
1 внеси го n;
2 за j од 1 до n прави A[0,j] = 1;
3   за i од 1 до n прави
4    {
5      A[i,0] = A[i-1,1];
6      за j од 1 до i прави
7            A[i,j] = A[i-1,j+1] + A[i,j-1];
8   }
9 печати A[n,0];
```

Кодовите во Јава и C++ кои ги даваме овде се итеративни решенија кои се базираат на третиот пристап.

JAVA 2. 5 ПРАВИЛНО ПОСТАВУВАЊЕ НА ЗАГРАДИ

```
1 import java.util.*;
2
3 public class Main
4 {
5  public static void main(String[] args) {
6         Scanner scn = new Scanner(System.in);
7         int n = scn.nextInt();
8         int [][] A = new int[n+1][n+1];
9
10        for(int j=0; j<=n; j++)
11            A[0][j] = 1;
12
13        for(int i=1; i<=n; i++)
14        {
15            A[i][0] = A[i-1][1];
16            for(int j=1; j<n; j++)
17            {
18                A[i][j] = A[i-1][j+1] + A[i][j-1];
19            }
20        }
21
22    System.out.println(A[n][0]);
23 }
24}
```

C++ 2. 5 ПРАВИЛНО ПОСТАВУВАЊЕ НА ЗАГРАДИ

```
1   #include<iostream>
2  using namespace std;
3
4  int main() {
5     int n;
6     cin >> n;
7     int A [n+1][n+1];
8
9     for(int j=0; j<=n; j++)
10       A[0][j] = 1;
11
12    for(int i=1; i<=n; i++)
13    {
14       A[i][0] = A[i-1][1];
15       for(int j=1; j<n; j++)
16       {
17          A[i][j] = A[i-1][j+1] + A[i][j-1];
18       }
19    }
20
21    cout << A[n][0];
22
23    return 0;
24 }
```

Прашања и задачи

1. Еден полжав се движи по вертикален столб надолу и нагоре. Тој се придвижува по точно еден сантиметар секоја минута и тоа или нагоре или надолу. Движењето му почнува од дното на столбот и во текот на неговото движење може да се врати пак на почетната позиција, но никако не го напуштил столбот. На колку различни начини можел да се движи полжавот, ако после n минути

   a. се наоѓа на почетната позиција

   b. се наоѓа на позиција m.

2. Знаеме дека множењето е асоцијативна операција, па заради тоа производ од облик $x_1 \cdot x_2 \cdot \ldots \cdot x_n$, од n броеви може да се пресмета на различни начини. На пример, ако се дадени четири броеви, има пет различни начини за пресметување на производот: $\big(\big((x_1\ x_2)x_3\big)x_4\big)$, $\big(\big(x_1(x_2\ x_3)\big)x_4\big)$, $\big(x_1\big((x_2\ x_3)x_4\big)\big)$, $\big((x_1\ x_2)(x_3\ x_4)\big)$ и $\big(x_1\big(x_2(x_3\ x_4)\big)\big)$. На колку начини може да се направи ова?.

3. Покажи дека бројот на целосни бинарни дрва со n внатрешни темиња е ист со бројот на правилно распределени n парови на загради!

4. Околу округла маса седа $2n$ луѓе. Тие симултано треба да се поздраваат со некој друг од масата, но така да нема преклопување на раце. Објасни дека бројот на начини на кои може да се направи ова поздравување е Каталановиот број!

5. Триангулација на конвексен полигон е постапка во која тој ќе се подели на триаголници, така да секое теме од секој од триаголниците е теме на полигонот.

   a. Дади ја рекурзивната равенка за бројот на различни триангулации на полигон со n темиња, ако сите темиња ги гледаме како различни.

   b. Која е врската меѓу овој број и бројот на правилно напишани загради?

6. Покажи дека бројот на патишта во $n \times n$ матрица кои почнуваат од горниот лев агол, а завршуваат во долниот десен агол и цело време се под дијагоналата (може и да ја допираат) е ист со бројот на правилно распределени загради.

    a. Покажи дека бројот на патишта во $(n+1) \times (n-1)$ матрица е еднаков на бројот на патишта во $n \times n$ матрица кои поминуваат низ дијагоналата.

    b. Користејќи го резултатот под а. Покажи дека Каталановиот број е еднаков на $\frac{1}{n+1}\binom{2n}{n}$.

4 Повеќедимензионални проблеми

Има две генерални групи на повеќедимензионални проблеми од динамичко програмирање. Првиот тип на проблеми се проблеми кои се опишуваат функции од повеќе од две променливи, па ваквите проблеми се базираа на пресметување на повеќедимензионална рекурзивна равенка, за чија што пресметка се потребни константен број на операции. Овој тип на проблеми се слични на генерализација во проблемот на пат низ матрица во повеќедимензионална матрица, па не се многу интересни за анализа. Од друга страна, проблемите во кои рекурзивната функција зависи од една или две променливи но нејзино пресметување има барем линеарна сложеност се такви што побаруваат подлабока анализа, па сите проблеми кои овде ќе ги разгледаме се од овој тип.

Во оваа глава ќе разгледаме четири проблеми, еден комбинаторен и три оптимизациони проблеми. Сите четири проблеми се карактеризираат со пресметување на дводимензионална функција, со тоа што во првите два проблеми едната од димензиите е значително помала од влезот. Во секој од нив, за пресметка на секоја инстанца на функцијата се потребни $O(n)$ операции.

Во првиот проблем е потребно влезната низа од податоци оптимално да се подели во неколку групи, додека во вториот, комбинаторен проблем, е потребно да се избројат бројот на поднизи со одредена карактеристика и димензија, во овој случај треба да бидат растечки поднизи.

Третиот и четвртиот проблем се оптимизациони проблеми кои се базираат на некоја апликација на Каталионовиот број, кој што претходно го сретнавме во правилно распределување на загради, но во вежбите во делот 3.4. дадовме уште неколку места каде истиот

број се јавува. Имено, во овие проблеми секој елемент од множеството распореди кое одговара на една апликација на Катлиновиот број има одредена вредност и нашата задача е да го определиме оној што има оптимална вредност. Генерално, во двата проблеми идејата за решавање е иста, а разликата е во тоа што пресметката на цената на секоја инстанца е различна.

Проблем 4. 1. Оптимална партиција на низа на конечен број сегменти

Да разгледаме проблем во кој имаме напишан роман кој се состои од n глави, и нормално секоја од главите може да има различен број на страници. Сакаме романот да се издаде во точно m томови така што најдебелиот том да биде што е можно потенок. При тоа, јасно, главите не може да се делат, односно не може еден дел од некоја глава да биде во еден том, а друг дел во наредниот том. Секако, главите мора да бидат наредени по редоследот по кој се напишани.

Дефинирање на проблемот

Дадена е низа од n броеви, и нека j-тиот елемент од низата го обележиме со x_j. Низата треба да се подели на точно m делови, или на m партиции, така што сумата на елементите во партицијата со најголема сума биде минимална. Проблемот е при дадена низа $x_j, j = \overline{1,n}$ и даден број на партиции m, да се најде сумата на броевите во партицијата со најголема сума, како и една таква оптимална партиција.

Анализа на проблемот

Во овој проблем имаме две различни величини кои се менуваат, односно можат да растат, од една страна бројот на глави, а од друга страна бројот на томови од книгата. Затоа прашањето од која од овие димензии треба да зависи рекурзивната равенка што треба да ја поставиме, и одговорот е по двете. Имено, јасно е дека ако во оптималното решение го отстраниме последниот том, тогаш отстрануваме и неколку глави, односно ако го намалиме бројот на томови, се подразбира дека сме го намалиле и бројот на глави. Затоа идејата за решавање на проблемот ќе ја започнеме со отстранување на последниот том. Нека во оптималната распределба на главите по томови, во последниот том се ставени последните k глави. Тогаш, бројот на страни во овој том би бил:

$$b_m = x_{n-r+1} + x_{n-r+2} + \cdots + x_n$$

Има две можности:

- Последниот том да не е томот со најмногу страници. Во овој случај томот со најмногу страници е некој том порано, односно во оптималното распоредување на $n-r$ глави во $m-1$ томови, некој том има повеќе од b_m страни.
- Последниот том да биде најголемиот том. Во овој случај претходните томови имаат помалку од b_m страни.

Оптималната потструктура на овој проблем е следнава:

> *Ако во оптималното разместување i-те први томови се составени од j-те први глави, $i < j$, тогаш постои оптимално разместување на сите n глави во сите m томови кое го вклучува оптималното разместување на првите j глави во i томови.*

Доказ на оптималното својство: Нека во оптималното разместување максималниот број на страни во некој том е M и првите i томови биле разместени во j-те први глави, но не оптимално (има подобар начин кој ќе го намали бројот на страни во првите i томови. Можни се два случаи за тоа каде се наоѓа најдебелиот том: или е во првите i томови, а во останатите, сите томови се со помал број на страници, или е во останатите. Во првиот случај, во првите i томови има том со дебелина M. Но, тие не се разместени оптимално, па значи можеме да ги разместиме пооптимално, и да го намалиме бројот на страни во најдебелиот том, односно да ги наредиме главите така да бројот на страни во сите тие томови е стриктно помал од M, што е контрадикција на тврдењето дека во оптималното разместување максималниот број на страни во некој том е M. Ако пак томот со најмногу страници е во останатите глави, тогаш воопшто не е проблем првите j глави оптимално да ги наредиме во i-те томови, затоа M нема да се смени. □

Она што овде се јавува како проблем е тоа што не знаеме колкав е бројот на глави во последниот том. Затоа треба да ги провериме сите можности за број на глави во последниот том.

Нека со $A[i,j]$ го обележиме максималниот број на страници во некој том ако во првите i томови се разместени првите j глави од книгата. Јасно е дека $i \leq j$, затоа што во i томови неможат да се разместат повеќе од i глави од книгата. Тогаш имаме:

$$A[i,j] = \begin{cases} \sum_{r=1}^{j} x_r\,, & i = 1 \\ \min\limits_{k<j} \left\{ \max\left\{ A[i-1,k], \sum_{r=k+1}^{j} x_r \right\} \right\}, & 1 < i \leq j \end{cases} \quad (3.1)$$

Ако врз основа на оваа равенка градиме рекурзивен алгоритам, тогаш тој би имал експоненцијална сложениот, затоа што се повикуваме на барем два чекори наназад. Алгоритамот со динамичко програмирање ќе ги пресметува решенијата одоздола нагоре, постепено почнувајќи од $i = 1$, односно од една глава. За секој број на глави i постепено ќе ја пресметува вредноста за $A[i,j]$ почнувајќи од $j = 1$ до $j = n$. Да забележиме дека ако сакаме да ги намалиме пресметките кога $i < m$ не мора да одиме до $j = n$, туку може да престанеме до $j = n - 1$. Слично, за $j = m$ потребно е да израчунаме само за $i = m$ глави.

Како и во другите случаи, и овде ќе користиме две табели, една онаа во која ќе ги паметиме вредностите на $A[i,j]$, а втората таа која ќе ја користиме за реконструкција на оптималното решение. Во втората табела, $C[i,j]$, ќе ја запишуваме вредноста на главата со која започнува последниот том. Почетните услови се кога имаме само еден том, т.е. кога $i = 1$. Тогаш знаеме дека сите глави мора да се во тој том, па бројот на страници во него е еднаков на бројот на страници во книгата. Тој том секогаш ќе започнува со првата глава.

Работата на алгоритамот е илустрирана со Табела 4. 1. Јасно, ако имаме само еден том, тој почнува со првата глава, па $C[1,j] = 1$ за сите j, а $A[1,j]$ е збирот на страните на сите глави до j-тата глава.

За два тома ќе дадеме неколку пресметки:

$$A[2,2] = \max\{A[1,1], x_2\} = \max\{20,25\} = 25;$$

$$C[2,2] = 2;$$

$$A[2,3] = \min\{\max\{A[1,1], x_2 + x_3\}, \max\{A[1,2], x_3\}\} =$$

$$= \min\{\max\{20, 55\}, \max\{45,30\}\} = \min\{55,45\} = 45.$$

Минимумот се добива кога последниот том започнува од третата глава, па

$$C[2,3] = 3;$$

Останатите чекори за распределување во две глави нема да ги даваме, ќе го дадеме само последниот чекор. Со последниот чекор ќе илустрираме дека за последната глава не мора да се проверуваат сосема сите можности за главите кои ќе се стават во него:

$$\max\{A(2,9), x_{10}\} = \max\{101,12\} = 101;$$

$$\max\{A(2,8), x_9 + x_{10}\} = \max\{90,29\} = 90;$$

$$\max\{A(2,7), x_8 + x_9 + x_{10}\} = \max\{75,61\} = 75;$$

$$\max\{A(2,6), x_7 + x_8 + x_9 + x_{10}\} = \max\{75,71\} = 75;$$

$$\max\{A(2,5), x_6 + x_7 + x_8 + x_9 + x_{10}\} = \max\{63,95\} = 95.$$

Бидејќи во последниот чекор сумата на последните глави го надминува минимумот кој е добиен претходно, тоа ќе се случува и ако додаваме дополнителни глави во последниот том. Заради тоа овде треба да прекинеме, бидејќи подобро решение од 75 нема да се добие.

Како што гледаме можно е во повеќе од една ситуација се добие минималното решение. Која од тие ситуации ќе ја одбереме при реконструкцијата на оптималното решение не е битно. Во нашиот псевдокод како маркер за реконструкција се става последното добиено по овој редослед. Така во овој пример последниот том ќе почне од седмата глава.

Табела 4. 1. Илустрација на алгоритамот за проблемот „Роман“ за оптимално распоредување на 10 глави во 3 тома

j	1	2	3	4	5	6	7	8	9	10
x_j	20	25	30	15	18	24	10	32	17	12

$A[1,j]$	20	45	75	90	108	132	142	174	191	203
$C[1,j]$	1	1	1	1	1	1	1	1	1	1
$A[2,j]$		25	45	45	63	75	75	90	101	108
$C[2,j]$		2	3	3	3	4	4	5	5	6
$A[3,j]$			30	45	45	45	52	66	75	75
$C[3,j]$			3	4	4	4	5	6	7	7

Псевдокодот за алгоритамот е следниов:

П 4. 1. ПСЕВДОКОД ЗА ОПТИМАЛНА ПАРТИЦИЈА ЗА НИЗА НА КОНЕЧЕН БРОЈ СЕГМЕНТИ

```
1   Внеси го бројот на томови m и низата од број на страници во
    секоја глава x_j
2   A[1,1] = x_1;
3   за j од 2 до n прави A[1,j] = A[1,j − 1] + x_j;
4   за i од 2 до m прави
5     {
6      за j од i до n прави
7       {
8        k = j;
9        suma = x_k;
10       додека suma ≤ A[i − 1, k − 1] и k ≥ i прави
11             {
12               A[i,j] = A[i − 1, k − 1];
13               C[i,j] = k;
14               k = k − 1;
15               suma = suma + x_k ;
16             }
17         ако suma ≤ A[i,j] тогаш
18              {
```

```
19                  A[i,j] = suma;
20                  C[i,j] = k;
21                }
22          }
23    }
24  врати A[m,n].
```

Во предложениот псевдокод не бараме минимум по сите можности за последниот том, како што е опишано во рекурзивната равенка, туку во **додека** циклусот во редовите 10-16, во последниот том додаваме елементи се додека тој не стане поголем од најголемиот том од претходните глави. Кога ќе стане поголем, во редовите 17-21 споредуваме која од двете опции е подобра, дали со последната додадена глава или без неа. Имено во чекорите во редовите 10-16 во последниот том се додаваат глави се до некое k такво што

$$\sum_{r=k+1}^{j} x_j \leq A[i,k] \text{ и } A[i,k-1] < \sum_{r=k}^{j} x_j . \qquad (3.2)$$

Тогаш оптималното решение во последниот том ги вклучува или од $k+1$-та до j-тата глава, ако

$$A[i,k] < \sum_{r=k}^{j} x_j$$

или од k-тата до j-тата глава, ако

$$A[i,k] > \sum_{r=k}^{j} x_j .$$

Тоа е секогаш така, како што може да се види од примерот кој го разгледавме, но доказот ќе го оставиме на читателот да го докаже како прашање 3 од прашањата и задачите во оваа глава.

Реконструкцијата оди на тој начин што од табелата за C читаме која е првата глава од која треба да почне последниот додаде во последниот, m-тиот том. Во нашиот случај процедурата оди на следниов начин:

- $C[3,10] = 7$ па последниот, третиот том почнува со седмата глава, бројот здебелен во Табела 4. 1. Третиот том ги има од седмата до десеттата глава, со вкупно 71 страна.
- Понатаму, треба да се поделат оптимално првите 6 глави во 2 тома. Здебелениот број во табелата за два тома е $C[2,6] = 4$, што кажува дека вториот том треба да почне од четвртата глава.
- На крај гледаме за еден том и 4 глави, $C[1,3] = 1$ и добиваме дека тој треба да почне од првата глава.

Рекурзивната процедура за печатење на оптималното решение е дадена со П 4. 2:

П 4. 2. РЕКОНСТРУКЦИЈА НА ЕДНО ОПТИМАЛНО РЕШЕНИЕ ЗА ОПТИМАЛНА ПАРТИЦИЈА ЗА НИЗА НА КОНЕЧЕН БРОЈ СЕГМЕНТИ

```
1  PechatiTomovi(m, n)
2  ако m = 1 тогаш печати 1"-виот том почнува со глава" 1
3      инаку
4          {
5           k = C[m, n];
6           PechatiTomovi(m – 1, k – 1);
7           печати m "-тиот том почнува со глава" k.
8          }
```

На крај да ја анализираме сложеноста на алгоритамот. Во псевдокодот на „роман со најмал број на страници во најтешкиот том" се јавуваат три вгнездени циклуси, надворешниот е по сите томови, внатрешниот е по сите глави, а третиот вгнезден циклус може сепак да се движи најмногу до бројот на глави, односно n. Иако во пракса последниот вгнезден циклус нема да се движи до n, сепак

не можеме да докажеме дека бројот на влегувања во него не е $O(n)$. Оттука сложеноста на даденото решение е $O(mn^2)$.

Сложеноста може да се пресмета и директно од самата рекурзивна равенка. Имено, да забележиме дека за секоја пресметка на вредност на функцијата $A[i,j]$ потребно е да се определи минимумот од $O(j)$ елементи, кој во најлош случај може да биде $O(n)$. Сложеноста произлегува од фактот што треба да се пополни табела со димензии $m \times n$ така што за секое поле се потребни $O(n)$.

Во решенијата во јава и c++ се пресметува дебелината на најдебелиот том и се печати една оптимална поделба. Прво го даваме решението во c++

C++ 4. 1 ОПТИМАЛНА ПАРТИЦИЈА ЗА НИЗА НА КОНЕЧЕН БРОЈ СЕГМЕНТИ.

```
#include<iostream>
#include<bits/stdc++.h>
using namespace std;

void pechatiTomovi(int *C, int m, int n, int k)
{
    if(m == 0)
    {
       cout << "1-viot tom pochnuva so glava 1" << endl;
    }
    else
    {
       int index = (m * n + k);
       k = *(C + index);
       pechatiTomovi(C, m-1, n, k-1);
       cout << (m+1) <<"-tiot tom pochnuva so glava " <<
(k+1) << endl;
    }
}

int main()
{
    int n;// broj na glavi
    cin >> n;
    int m;// broj na tomovi
    cin >> m;
```

```
26    int x[n];/ stranici po glava
27    for (int i = 0; i < n; i++)
28       cin >> x[i];
29    int A[m][n];// strani
30    int C[m][n];//reden broj na glava
31    A[0][0] = x[0];
32    C[0][0] = 1;
33    // ako imame samo eden tom site glavi vleguvaat vo nego
34    for (int j = 1; j < n; j++)
35    {
36       A[0][j] = A[0][j-1] + x[j];
37       C[0][j] = 1;
38    }
39
40    for (int i = 1; i < m; i++)
41    {
42       for (int j = i; j < n; j++)
43       {
44          int k = j;
45          int suma = x[k];
46          A[i][j] = INT_MAX;
47          while (suma <= A[i-1][k-1])
48          {
49             A[i][j] = A[i-1][k-1];
50             C[i][j] = k;
51             k--;
52             suma += x[k];
53          }
54
55          if (suma <= A[i][j])
56          {
57             A[i][j] = suma;
58             C[i][j] = k;
59          }
60       }
61    }
62
63    cout << A[m - 1][n - 1] << endl;
64    pechatiTomovi((int *)C, m-1, n, n-1);
65
```

```
66      return 0;
67  }
```

Решението во Јава е дадено со JAVA 4. 1 :

JAVA 4. 1 ОПТИМАЛНА ПАРТИЦИЈА ЗА НИЗА НА КОНЕЧЕН БРОЈ СЕГМЕНТИ.

```
1  import java.util.Scanner;
2
3  public class Main {
4      public static void pechatiTomovi(int[][] C, int m, int
n) {
5          if (m == 0) {
6             System.out.println("1-viot tom pochnuva so glava
1");
7          } else {
8              int k = C[m][n];
9              pechatiTomovi(C, m - 1, k - 1);
10             System.out.println((m + 1) + "-tiot tom pochnuva
so glava " + (k + 1));
11         }
12     }
13
14     public static void main(String[] args) {
15         Scanner scanner = new Scanner(System.in);
16         int n = scanner.nextInt(); // broj na glavi
17         int m = scanner.nextInt(); // broj na tomovi
18         int[] x = new int[n]; // stranici po glava
19         for (int i = 0; i < n; i++)
20             x[i] = scanner.nextInt();
21
22         int[][] A = new int[m][n]; // strani
23         int[][] C = new int[m][n]; // tomovi
24
25         // ako imame samo eden tom site glavi vleguvaat vo
nego
26         A[0][0] = x[0];
27         C[0][0] = 1;
28         for (int j = 1; j < n; j++) {
29             A[0][j] = A[0][j - 1] + x[j];
30             C[0][j] = 1;
31         }
32
```

```
33          for (int i = 1; i < m; i++)
34              for (int j = i; j < n; j++) {
35                  int k = j;
36                  int suma = x[k];
37                  A[i][j] = Integer.MAX_VALUE;
38
39                  while (suma <= A[i - 1][k - 1]) {
40                      A[i][j] = A[i - 1][k - 1];
41                      C[i][j] = k;
42                      k--;
43                      suma += x[k];
44                  }
45
46                  if (suma <= A[i][j]) {
47                      A[i][j] = suma;
48                      C[i][j] = k;
49                  }
50              }
51          System.out.println(A[m - 1][n - 1]);
52          pechatiTomovi(C, m - 1, n - 1);
53      }
54}
```

Овој проблем е интересен и од друг аспект. Имено, со него може да се илустрира процедурата раздели па владеј, која може решението да го забрза до $O(mnln(n))$. Нема да го даваме псевдокодот, само ќе покажеме како работи процедурата за последниот чекор. Идејата е во тоа што оптималното решение се наоѓа некаде околу онаа вредност k за која сумата на последните k глави го надминува оптималното решение за еден том и k глави помалку. Со оваа процедура, наместо $O(n)$ чекори за пополнување на едно поле во таблата ќе ни бидат потребни $O(ln(n))$ чекори:

- $\lfloor(10+1):2\rfloor = 5$, па го пресметуваме

  $$\max\{A[2,5], x_6 + x_7 + x_8 + x_9 + x_{10}\} = \max\{63,95\} = 95$$

  и гледаме дека сумата на последните 5 глави има поголема вредност од $A[2,5]$. Пради тоа во оптималното решение почетокот на последната глава е или 6 или за некое k поголемо од 6.

- $\lfloor (10+5):2 \rfloor = 7$, па го пресметуваме

$$\max\{A[2,7], x_8 + x_9 + x_{10}\} = \max\{75, 61\} = 75$$

- и гледаме дека сумата на последните 3 глави има помала вредност од $A[2,7]$ Прави тоа што 75 е подобро од 95, во оптималното решение почетокот на последната глава е во интервалот [6, 8].

- $\lfloor (8+5):2 \rfloor = 6$, па го пресметуваме

$$\max\{A[2,6], x_7 + x_8 + x_9 + x_{10}\} = \max\{75, 71\} = 75$$

и гледаме дека сумата на последните 2 глави има помала вредност од $A[2,6]$ Прави тоа што 75 е исто со 75, во оптималното решение почетокот на последната глава можеме да го земеме било кое од 6 и 7.

Прашања и задачи

1. Разгледај го проблемот објаснет во оваа глава, но така да распределбата не мора да биде во точно m томови, туку во најмногу m томови. Имено, проблемот е да се распределат n глави во барем m томови, така да најголемиот том да има најмал можен број на страници. Покажи дека без разлика дали оптималната распределба ќе биде во точно m или најмногу m томови, главата со најмногу страници ќе има иста големина.

2. Разгледајте модификација на проблемот разгледуван во оваа глава, но така да наместо бараната оптимизација, главата со најголем број на страници да биде најтенка, сакаме главата со најмал број на страници да биде најголема.

   a. Покажи дека, ако сакаме главите да ги распределиме во точно m томови, тогаш овој алгортам може да се модифицира за вака дефинираниот проблем!

   b. Дади го оптималното својство за проблемот под a.!

   c. Докажи го оптималното својство кое го предлагаш под b.!

   d. Покажи дека, ако сакаме главите да ги распределиме во најмногу m томови, тогаш решението секогаш е распределба во еден том!

3. Покажи дека оптималното разместување на j глави во i томови, во последниот том ги вклучува или од $k+1$-та до j-тата глава, или од k-тата до j-тата глава, каде k го задоволува равенството (3.2).

4. Дадена е растечка низа од n позиции на x-оската. Треба да се позиционираат $k < n$ објекти на дадените позиции така да растојанието меѓу било кои два соседни објекти биде најголемо. Поточно, минималното растојание помеѓу било кои два објекти да биде најголемо можно.

   a. Покажи како овој проблем може да се претвори во проблемот од претходната задача 2!

b. Покажи дека секогаш мора да се постави некој и на првата и на последната позиција!

c. Дади го оптималното својство!

Проблем 4. 2. Број на растечки поднизи со дадена должина

Во овој дел презентираме проблем од броење кој имаат поголема од квадратна сложеност, кој слично како и претходниот проблем се решава со пресметување на вредности во дводимензионална низа, но за пресметките за секое поле во таа низа имаат линеарна сложеност. Да се потсетиме на проблемот за наоѓање на најдолгата растечка подниза од дадена низа, кој го имавме како задача во делот 2. 1. Овде треба да се избројат сите такви низи, но со веќе предефинирана дадена должина.

Дефинирање на проблемот

За дадена е низа од n броеви, $x_1, x_2, \dots, x_n$ да 1се пресмета бројот на растечки поднизи со должина k. Подниза на низа не е последователна подниза, туку било која низа која се добива со бришење на дел од елементите на првобитната низа.

Анализа на проблемот

Првото прашање е како да се намали проблемот со големина n на проблем или проблеми со помала големина. Јасно поднизите се карактеризираат со нивниот почеток и крај, но и со нивната должина. Сите овие елементи можат да се земат како алтернатива за изведување на рекурзивната равенка над која ќе се базира решението со динамичко програмирање. Треба да забележиме дека во проблемот од поголема големина единствена алтернатива е да додаваме по еден елемент во поднизата, и бидејќи не интересира бројот на елементи во низата, еден од параметрите кои треба да ги разгледуваме е бројот на елементи во поднизата. Вториот параметар треба да одговара на големината на низата од која ја бираме поднизата, па како параметар ќе го земеме последниот елемент.

Со $A[i, r]$ ќе го обележиме бројот на растечки поднизи со должина r, кои завршуваат со елементот на i-та позиција, т.е. x_i, за

$i \leq n$. Ако x_i е последниот елемент, тогаш тој е елемент додаден на растечка подниза со должина $r-1$, која завршува со елемент кој е помал од x_i и е на позиција помала од i-тата позиција. Според тоа $A[i,r]$ е бројот на низи кои завршуваат на $j < i$ со должина $r-1$, кај кои $x_j < x_i$:

$$A[i,r] = \sum_{j<i, x_j<x_i} A[j, r-1].$$

Почетните вредности се добиваат за низи со должина 1, за кои е јасно дека се состојат со елементот на i-тата позиција., па

$$\forall i, A[i,1] = 1.$$

Исто така, ако $i < k$, не може да постојат такви поднизи, па тогаш $A[i,k] = 0$.

Да видиме како ќе работи овој алгоритам за низата 1, 5, 3, 4, 6 и поднизи со должина $k = 3$.

- За $r = 1$, $A[i,1] = 1$ за секое i.
- За $r = 2$:

  $A[2,2] = A[1,1] = 1$;

  $A[3,2] = A[1,1] = 1$, затоа што само елементот на прва позиција е помал од 3.

  $A[4,2] = A[1,1] + A[3,1] = 2$, затоа што елементите на прва и трета позиција се помали од 4.

  $A[5,2] = A[1,1] + A[2,1] + A[3,1] + A[4,1] = 4$.
- За $r = 3$:

  $A[3,3] = A[1,2] = 0$;

  $A[4,3] = A[1,2] + A[3,1] = 0 + 1 = 1$.

  $A[5,3] = A[1,2] + A[2,2] + A[3,2] + A[4,2] = 0 + 1 + 1 + 2 = 4$.

Псевдокодот на решението е даден со П 4. 3:

П 4. 4. ПСЕВДОКОД ЗА БРОЈ НА РАСТЕЧКИ ПОДНИЗИ

```
1  Внеси ја низата X;
2  за i = 1 до n прави A[i,1] = 1;
3    за r = 2 до k прави
4      {
5          за i = 1 до r – 1 прави A[i,r] = 0;
6          за i = r до n прави
7            {
8             A[i,r] = 0;
9             за j = r – 1 до i – 1 прави
10              ако x_j < x_i тогаш A[i,r] = A[i,r] + A[j,r − 1];
11            }
12         }
13 печати A[n,k].
```

Повторно задачата се сведува на пресметување на елементите во дводимензионална низа, па сложеноста ќе Бидејќи во кодот имаме 3 вгнездени циклуси, еден до k, и два до n, сложеноста на решението е $\Theta(kn^2)$.

Кодот во C++ е даден подолу:

C++ 4. 2 БРОЈ НА РАСТЕЧКИ ПОДНИЗИ СО ФИКСНА ДОЛЖИНА.

```
1  #include<iostream>
2  #include <string.h>
3  using namespace std;
4
5  int main() {
6     int n, k;
7     cin >> n;
8     cin >> k;
9     int x [n];
10    for (int i = 0; i < n; i++)
```

```
11        cin>> x[i];
12
13     int A [n][k];
14
15     for(int i=0;i<n;i++)
16        A[i][0]=1;
17     for(int r=1;r<k;r++)
18     {
19        for(int i=0;i<r;i++)
20           A[i][r] = 0;
21        for(int i=r;i<n;i++)
22        {
23           A[i][r]=0;
24           for(int j=r-1;j<i;j++)
25              if(x[j]<x[i])
26                 A[i][r]= A[i][r]+A[j][r-1];
27        }
28     }
29
30     cout << A[n-1][k-1];
31
32     return 0;
33  }
```

Решението во Јава е дадено со JAVA 4. 1 :

JAVA 4. 2 БРОЈ НА РАСТЕЧКИ ПОДНИЗИ СО ФИКСНА ДОЛЖИНА.

```
1  import java.util.Scanner;
2
3  public class Main {
4     public static void main(String[] args) {
5        Scanner scanner = new Scanner(System.in);
6        int n = scanner.nextInt();
7        int k = scanner.nextInt();
8        int[] x = new int[n];
9        for (int i = 0; i < n; i++)
10          x[i] = scanner.nextInt();
11
12       int[][] A = new int[n][k];
13
14       for(int i=0;i<n;i++)
15          A[i][0]=1;
16       for(int r=1;r<k;r++)
17       {
```

```
18          for(int i=0;i<r;i++)
19             A[i][r] = 0;
20          for(int i=r;i<n;i++)
21          {
22             A[i][r]=0;
23             for(int j=r-1;j<i;j++)
24                if(x[j]<x[i])
25                   A[i][r]= A[i][r]+A[j][r-1];
26          }
27       }
28
29       System.out.println(A[n-1][k-1]);
30    }
31 }
```

Прашања и задачи

1. Јасно е дека вкупниот број на растечки поднизи е сумата на броевите на сите растечки поднизи со должина k, за k од 1 до n, што значи дека тој проблем може да се реши ако во оваа задача на крај се соберат сите $A[n,k]$. Дали е тоа најдоброто решение за ваков проблем? Дади ја рекурзивната равенка која го решава овој проблем!

2. Како може да се модифицира проблемот ако сакаме да го пресметаме бројот на сите опаѓачки поднизи со должина k?

3. Да се модифицира овој алгоритам за да се реши проблемот за број на алтернативни поднизи со должина k?

4. За дадена е низа од n броеви, $x_1, x_2, \ldots, x_n$ да се пресмета бројот на поднизи со должина k кои имаат сума N. Опиши алгоритам кој го решава овој проблем!

5. За дадена е низа од n позитивни и негативни броеви, $x_1, x_2, \ldots, x_n$ да се најде подниза со должина k со максимална сума! Дади ја рекурзивната релација над која може да се конструира решениесо динамичко програмирање!

Проблем 4. 3. Верижно множење на матрици

Проблемот кој ќе го разгледаме во овој дел е проблем чија идеја ја имавме разработено претходно во проблемот за пресметување на бројот на правилно распределени загради. Проблемот е оптимизационен и се базира на број на начини на правилно распределување на загради, но овде треба да дадеме еден распоред, оној кој го дава оптималното решение. Да се потсетиме дека за решавање на проблемот со број на правилно распределени загради дадовме три пристапи, кои се разликуваа во начинот на генерирање на распоредите. Идејата за решавање на овој проблем се базира на вториот од нив.

Дефинирање на проблемот

Дадена е низа $X_1, X_2, \dots, X_n$, од n матрици од ред $p_0 \times p_1$, $p_1 \times p_2$,..., $p_{n-1} \times p_n$, кои треба да се помножат. Поточно, целта е да го пресметаме производот $X_1 \cdot X_2 \cdot \dots \cdot X_n$. Ова може да се пресмета користејќи го стандардниот алгоритам за множење на две матрици, како подрутина. Но, прво треба да се одлучи како да се групираат матриците, односно како да се постават заградите при множењето. Множењето матрици е асоцијативно, па сите распореди на заградите даваат ист производ. На пример, ако се дадени четири матрици е X_1, X_2, X_3, X_4, има пет различни начини за целосно поставување загради

$$\Big(\big((X_1X_2)X_3\big)X_4\Big),$$

$$\Big(\big(X_1(X_2X_3)\big)X_4\Big),$$

$$\big((X_1X_2)(X_3X_4)\big),$$

$$\Big(X_1\big((X_2X_3)X_4\big)\Big),$$

$$\big(X_1\big(X_2(X_3X_4)\big)\big).$$

Начинот на кој ќе се множи верига од матрици може да има драматично влијание врз бројот на множења на реални броеви при определување на производот. Во стандардното множење на матрица од ред $p \times q$ со матрица од ред $q \times r$ потребно е да се направат pqr множења. На пример, ако имаме три матрици од ред 10×100, 100×5 и 5×50 и ако ги групираме првите две, па добиениот производ го помножиме со третата ни требаат $10 \cdot 100 \cdot 5 = 5000$ множења на скалари за множење на првите две, при што ќе се добие матрица од ред 10×5, плус $10 \cdot 5 \cdot 50 = 2500$ множења за множењето на добиената матрица со третата матрица. Значи вкупно 7500 множења. Ако пак ги групираме втората и третата матрица, па после множиме со првата имаме $100 \cdot 5 \cdot 50 + 10 \cdot 100 \cdot 50 = 75000$ множења на скалари. Проблемот не е да се измножат матриците, туку да се најде најсоодветниот редослед кој ќе го минимизира бројот на множења на скалари.

Анализа на проблемот

На почеток да го дадеме стандардниот алгоритам за множење на матрици. Матриците A и B можат да се помножат само ако тие се компатибилни, односно, бројот на колони од A треба да биде еднаков на бројот на редици на матрицата B. Ако матрицата A е од ред $p \times q$, а матрицата B од ред $q \times r$, тогаш нивниот производ AB е матрица од ред $p \times r$. Во следниот псевдокод, П 4. 5, $col[A]$ и $col[\mathrm{B}]$ се бројот на колони на A и B соодветно, додека $rows[A]$ и $rows[\mathrm{B}]$ се бројот на редици на A и B соодветно:

П 4. 5. ПСЕВДОКОД ЗА МНОЖЕЊЕ МАТРИЦИ

```
1  внеси ги матриците A и B
2  ако col[A] ≠ rows[B] тогаш прикажи "некомпатибилни"
3      инаку
4          {
5           за i од 1 до rows[A]
6               за j од 1 до col[B] прави
7               {
```

```
8            C[i,j] = 0;
9            за k = 1 до col[A] прави
10                       C[i,j] = C[i,j] + A[i,k] · B[k,j];
11           }
12 врати C.
```

Јасно е дека едно решение на проблемот на оптимално множење на матрици е алгоритамот со груба сила, во кој би требало да се најдат сите можни распореди на загради помеѓу матриците, и за секој распоред да се пресмета колку операции се потребни. Овој број е ист како бројот на правилно распределени загради, односно е Каталановиот број, па сложеноста на ваквото решение е повеќе од експоненцијална односно, како што дискутиравме во делот за број на правилно распоредени загради, $O\binom{2n}{n}$.

Динамичкото решение на проблемот се води од идејата дека тој е сличен со проблемот на број на правилно распределување на загради. Можеме да се водиме по идејата на два од начините на броење на правилни распореди на загради, но многу поедноставно се доаѓа до решението ако го користиме вториот начин.

Кога имаме само една матрица, немаме никакво множење, па овој случај не е интересен. Во случај на две матрици од ред $p_0 \times p_1$ и $p_1 \times p_2$, тогаш исто така има само еден начин да се измножат и бројот на множење на скалари е производ од нивните редови, т.е. $p_0 p_1 p_2$.

За $n \geq 2$, го гледаме последното множење. Во последниот чекор секогаш ќе се множат само две матрици. Во овој момент не интересира како тие претходно биле измножени, туку само не интересира редот на матриците кои се добиле при тоа множење. Можеме да претпоставиме дека последното множење е помеѓу k-тата и $k+1$-та матрица:

$$(X_1 \dots X_k)(X_{k+1} \dots X_n),$$

односно измножени се од првата до k-тата матрица, па од $k+1$-вата до крај и вака добиените матрици се множат меѓу себе. Тогаш

во претходните чекори сме имале одреден број на множење на броеви, но тоа сметаме дека во овој момент ни е пресметано. Бројот на множења кој ќе биде направен во овој последен чекор зависи од димензијата на матриците $X_1 \cdot \ldots \cdot X_k$, што е $p_0 \times p_k$ и $X_{k+1} \cdot \ldots \cdot X_n$, што е $p_k \times p_n$, па бројот е $p_0 p_k p_n$. Оттука, вкупниот број на множења, под услов последното множење да било на матриците $X_1 \cdot \ldots \cdot X_k$ и $X_{k+1} \cdot \ldots \cdot X_n$ е $p_0 p_k p_n$ плус бројот на множења кои сме ги имале за да го пресметаме производот $X_1 \ldots X_k$ и бројот на множења кои сме ги имале за да го пресметаме производот $X_{k+1} \ldots X_n$. Тука се појавуваат две ставки кои треба да се разгледаат:

- Прво, за кое k се добива оптималниот број на множења. Бидејќи не знаеме треба да провериме за секое k.
- Второ, на кој начин биле измножени матриците $X_1 \cdot \ldots \cdot X_k$ и $X_{k+1} \cdot \ldots \cdot X_n$, и колку операции биле потрошени за тоа. Трикот што овде тереба да се увиди е дека и тие требале да бидат измножени најоптимално.

Од оваа анализа може да се изведе следново оптимално својство:

> *Ако во оптимално множење на n-те матрици, т.е. при добивање на производот $X_1 \cdot X_2 \cdot \ldots \cdot X_n$ прво била добиена матрицата $X_i \cdot X_{i+1} \cdot \ldots \cdot X_j = X^{i,j}$, при што се добила веригата матрици*
>
> $$X_1 \cdot \ldots \cdot X_{i-1} \cdot X^{i,j} \cdot X_{j+1} \ldots \cdot X_n,$$
>
> *А потоа множењето продолжило со оваа верига. Тогаш, за добивање на матрицата $X^{i,j}$ биле употребени најмал број на множења на скалари.*

Доказ на оптималното својство: Стандардно, оптималното својство го докажуваме со контрадикција. Нека не е така, односно нека има друг оптимален начин при кој прво била добиена матрицата $X^{i,j}$, но таа не била добиена со најмал број на множења на скалари. Тоа значи дека во моментот во кога се добила веригата $X_1 \cdot \ldots \cdot X_{i-1} \cdot X^{i,j} \cdot X_{j+1} \ldots \cdot X_n$ биле направени помал број на множења отколку во оптималното добивање на $X^{i,j}$. Да направиме друг распоред во кој повторно стигаме до матрицата $X_1 \cdot \ldots \cdot X_{i-1} \cdot X^{i,j} \cdot X_{j+1} \ldots \cdot X_n$, но

сега матрицата $X^{i,j}$ ја добиваме со најмалиот број на множења, а потоа новата верига од матриците $X_1 \cdot \ldots \cdot X_{i-1} \cdot X^{i,j} \cdot X_{j+1} \ldots \cdot X_n$ ја множиме на ист начин како во стариот оптимален начин. Тоа значи дека за ова користиме помал број на множења на скалари, па вкупниот број множења ќе биде помал отколку во оптималниот, што е контрадикција. Значи за добивање на матрицата $X^{i,j}$ биле употребени најмал број на множења на скалари.

Затоа нека со $A[i,j], i \leq j$ го обележиме најмалиот број на множења на скалари за да се добие матрицата $X^{i,j}$. Кога $i = j$ има само една матрица па имаме 0 множења на скалари. Ако за $i < j$ оптималното групирање се добива кога оваа низа ќе се подели меѓу k и $k+1$, $i \leq k < j$ тогаш

$$A[i,j] = A[i,k] + A[k+1,j] + p_{i-1}p_k p_j,$$

бидејќи за оптималниот производ на $X^{i,j}$ ни требаат онолку множења колку што ни требаат за оптималниот производ на $X^{i,k}$ плус за оптималниот производ на $X^{k+1,j}$ плус за да се помножат матриците $X^{i,k}$ и $X^{k+1,j}$, што е $p_{i-1}p_k p_j$. Оттука,

$$A[i,j] = \begin{cases} 0, & i = j \\ \min\limits_{i \leq k < j} \{A[i,k] + A[k+1,j] + p_{i-1}p_k p_j\}, & i < j \end{cases}$$

Алгоритамот се состои во тоа што за секој пар $i < j$ се пресметува оптималниот број на множења, и се забележува како е поделена низата од матрици при последното множење во тој случај. Гледаме дека во рекурзивниот чекор не се намалува вредноста за i или j, туку се намалува големината на наизата, односно се намалува растојанието меѓу i и j. Ако користиме решеие со рекурзија, тоа воопшто нема да се разликува од алгоритамот со груба сила, бидејќи повторно ќе се навраќаме на сите комбинации на групирање. Но, ако искористиме динамичко програмирање, пресметките претходно ќе се искалкулираат и кога веќе еднаш ќе се добие еден оптимален распоред, истиот ќе се користи при изведување на решението за поголемите низи.

Динамичкото програмирање кое трга од назад на напред се базира на следниве чекори:

- Прво се пресметува оптималното поставување на заградите кога i и j се разликуваат за 1.
- Потоа, имајќи го тоа во предвид се пресметува оптималното поставување на загради, ако i и j се разликуваат за два, па за 3, па така се продолжува се додека нивната разлика не стане $n - 1$, што е всушност целата низа.
- При тоа во секој чекор се памети вредноста k соодветна на оптималната поделба на поднизата, што ќе се користи за реконструкција на едно оптимално решение. Тоа го запомнуваме во друга матрица.

Ќе ја опишеме постапката на еден пример. Нека треба да помножиме 6 матрици со димензии: 30×35, 35×15, 15×5, 5×10, 10×20 и 20×25. Вредностите на $A[i,j]$ за една и две матрици, односно за е дадена на Табела 4. 2., а Вредностите на $C[i,j]$ за една и две матрици, односно за е дадена на Табела 4. 3

Табела 4. 2. Почетните вредности на функцијата $A[i,j]$ кога $i = j$ и $j - i = 1$

$A[i,j]$	1	2	3	4	5	6
1	0	$30 \cdot 35 \cdot 15 = 15750$				
2		0	$35 \cdot 15 \cdot 5 = 2625$			
3			0	$15 \cdot 5 \cdot 10 = 750$		
4				0	$5 \cdot 10 \cdot 20 = 1000$	
5					0	$10 \cdot 20 \cdot 25 = 500$
6						0

Табела 4. 3. Почетните вредности на функцијата $C[i,j]$ кога $i = j$ и $j - i = 1$

$C[i,j]$	2	3	4	5	6
1	1				
2		2			
3			3		
4				4	
5					5

За разлика меѓу i и j еднаква на 2, оптимумот се бара меѓу две вредности и соодветната табела е дадена на Слика 4.1.2. Пресметките во табелата се:

$$A[1,3] = \min\begin{cases} A[1,1] + A[2,3] + p_0p_1p_3 \\ A[1,2] + A[3,3] + p_0p_2p_3 \end{cases}$$

$$= \min\begin{cases} 0 + 2625 + 30 \cdot 35 \cdot 5 = 7875 \\ 15750 + 0 + 30 \cdot 15 \cdot 5 = 18000 \end{cases} = 7875$$

$$A[2,4] = \min\begin{cases} A[2,2] + A[3,4] + p_1p_2p_4 \\ A[2,3] + A[4,4] + p_1p_3p_4 \end{cases}$$

$$= \min\begin{cases} 0 + 750 + 35 \cdot 15 \cdot 10 = 6000 \\ 2625 + 0 + 35 \cdot 5 \cdot 10 = 4375 \end{cases} = 4375,$$

$$A[3,5] = \min\begin{cases} A[3,3] + A[4,5] + p_2p_3p_5 \\ A[3,4] + A[5,5] + p_2p_4p_5 \end{cases}$$

$$= \min\begin{cases} 0 + 1000 + 15 \cdot 5 \cdot 20 = 2500 \\ 750 + 0 + 15 \cdot 10 \cdot 20 = 3750 \end{cases} = 2500,$$

$$A[4,6] = \min\begin{cases} A[4,4] + A[5,6] + p_3p_4p_6 \\ A[4,5] + A[6,6] + p_3p_5p_6 \end{cases}$$

$$= \min \begin{cases} 0 + 5000 + 5 \cdot 10 \cdot 25 = 6250 \\ 1000 + 0 + 5 \cdot 20 \cdot 25 = 3500 \end{cases} = 3500.$$

Во матрицата C се става вредноста за k за која се добива минимумот, Слика 4. 1. На пример за $A[1,3]$ помала е вредноста 7875 и се добива од $A[1,1] + A[2,3] + p_0 p_1 p_3$, па $C[1,3] = 1$.

$A[i,j]$	1	2	3	4	5	6
1	0	15750	7875			
2		0	2625	4375		
3			0	750	2500	
4				0	1000	3500
5					0	5000
6						0

$C[i,j]$	2	3	4	5	6
1	1	1			
2		2	3		
3			3	3	
4				4	5
5					5

Слика 4. 1. Вредностите за $A[i, i+2]$ и $C[i, i+2]$, во проблемот за оптимално множење на матрици.

Ако продолжиме вака, добиените вредности ги користиме за да ги пресметаме вредностите на двете матрици во кога разликата меѓу i и j е поголемо од 3, Слика 4. 2. За разлика 3 се добиваат следниве пресметки:

$$A[1,4] = \min \begin{cases} A[1,1] + A[2,4] + p_0 p_1 p_4 \\ A[1,2] + A[3,4] + p_0 p_2 p_4 \\ A[1,3] + A[4,4] + p_0 p_3 p_4 \end{cases}$$

$$= \min \begin{cases} 0 + 4375 + 300 \cdot 35 = 14875 \\ 15750 + 750 + 300 \cdot 15 = 21000 = 9375 \\ 7875 + 0 + 300 \cdot 5 = 9375 \end{cases}$$

$$A[2,5] = \min \begin{cases} A[2,2] + A[3,5] + p_1 p_2 p_5 \\ A[2,3] + A[4,5] + p_1 p_3 p_5 \\ A[2,4] + A[5,5 + p_1 p_4 p_5 \end{cases}$$

$$= \min \begin{cases} 0 + 2500 + 700 \cdot 15 = 13000 \\ 2625 + 1000 + 700 \cdot 5 = 7125 = 7125, \\ 4375 + 0 + 700 \cdot 10 = 11375 \end{cases}$$

$$A[3,6] = \min \begin{cases} A[3,3] + A[4,6] + p_2 p_3 p_6 \\ A[3,4] + A[5,6] + p_2 p_4 p_6 \\ A[3,5] + A[6,6] + p_2 p_5 p_6 \end{cases}$$

$$= \min \begin{cases} 0 + 3500 + 375 \cdot 5 = 5375 \\ 750 + 5000 + 375 \cdot 10 = 9500 = 5375. \\ 2500 + 0 + 375 \cdot 20 = 10000 \end{cases}$$

Минимумите се јавуваат: за $A[1,4]$ кога ќе се подели во $k = 3$, за $A[2,5]$ кога ќе се подели во $k = 3$ и за $A[3,6]$ кога ќе се подели во $k = 3$. Продолжуваме со вериги со должина 4, каде што добиваме дека оптималните вредности и за $A[1,5]$ и за $A[2,6]$ кога ќе се подели во $k = 3$.

$$A[1,5] = \min \begin{cases} A[1,1] + A[2,5] + p_0 p_1 p_5 \\ A[1,2] + A[3,5] + p_0 p_2 p_5 \\ A[1,3] + A[3,5] + p_0 p_3 p_5 \\ A[1,4] + A[5,5] + p_0 p_3 p_5 \end{cases}$$

$$= \min \begin{cases} 0 + 7125 + 600 \cdot 35 = 28125 \\ 15750 + 2500 + 600 \cdot 15 = 27250 \\ 7875 + 1000 + 600 \cdot 5 = 11875 \\ 9375 + 0 + 600 \cdot 10 = 15375 \end{cases} = 11875,$$

$$A[2,6] = \min \begin{cases} A[2,2] + A[3,6] + p_1 p_2 p_6 \\ A[2,3] + A[4,6] + p_1 p_3 p_6 \\ A[2,4] + A[5,6] + p_1 p_4 p_6 \\ A[2,5] + A[6,6] + p_1 p_5 p_6 \end{cases}$$

$$= \min \begin{cases} 0 + 5375 + 875 \cdot 15 = 18500 \\ 2625 + 3500 + 875 \cdot 5 = 10500 \\ 4375 + 5000 + 875 \cdot 10 = 18125 \\ 7125 + 0 + 875 \cdot 20 = 24625 \end{cases} = 10500.$$

На крај за целата низа имаме:

$$A[1,6] = \min \begin{cases} A[1,1] + A[2,6] + p_0 p_1 p_6 \\ A[1,2] + A[3,6] + p_0 p_2 p_6 \\ A[1,3] + A[4,6] + p_0 p_3 p_6 \\ A[1,4] + A[5,6] + p_0 p_4 p_6 \\ A[1,5] + A[6,6] + p_0 p_5 p_6 \end{cases}$$

$$= \min \begin{cases} 0 + 10500 + 750 \cdot 35 = 36750 \\ 15750 + 5375 + 750 \cdot 15 = 32375 \\ 7875 + 3500 + 750 \cdot 5 = 15125 \\ 9375 + 5000 + 750 \cdot 10 = 21875 \\ 11875 + 0 + 750 \cdot 20 = 26875 \end{cases} = 15125.$$

$A[i,j]$	1	2	3	4	5	6
1	0	15750	7875	9375	11875	15125
2		0	2625	4375	7125	10500
3			0	750	2500	5375
4				0	1000	3500
5					0	5000
6						0

$C[i,j]$	2	3	4	5	6
1	1	1	3	3	3
2		2	3	3	3
3			3	3	3
4				4	5
5					5

Слика 4. 2. Вредностите за $A[i,j]$ и $C[i,j]$, во проблемот за оптимално множење на матрици.

Псевдокодот за алгоритамот е следниов:

П 4. 6. ПСЕВДОКОД ЗА ОПТИМАЛНО МНОЖЕЊЕ НА МАТРИЦИ

```
1  Внеси ги матриците X_1, X_2, ..., X_n;
```

```
2   за i од 1 до n прави A[i, i] = 0;
3   за l од 1 до n прави
4       за i од 1 до n – l + 1 прави
5       {
6         j = i – l + 1;
7         A[i, j ] = ∞;
8         за k од i до j – 1 прави
9         {
10          q = A[i, k] + A[k + 1, j] + p_{i−1} p_k p_j;
11          ако q < A[i, j] тогаш
12              {
13                A[i, j ] = q;
14                C[i, j ] = k;
15              }
16          }
17      }
18  врати A и C.
```

За да го реконструираме решението се повикуваме на втората матрица, C.

- Читаме каде е оптимално да се подели низата $X_1 \cdot X_2 \cdot \ldots \cdot X_n$ и нека тоа е во k.
- Потоа се гледа каде е оптимално да се поделат поднизите $X_1 \cdot X_2 \cdot \ldots \cdot X_k$ и $X_{k+1} \cdot \ldots \cdot X_n$, и се додека не се стигне до низа од една матрица.

При тоа во секој чекор се памети вредноста k и отпималниот број на множења на скалари кои се соодветни за таа вредност.

Реконструкцијата ја правиме така што поставуваме загради кои ни покажуваат кој е редоследот на множење на матриците. Тоа може да се направи со рекурзија со следниов псевдокод:

П 4. 7. РЕКОНСТРУКЦИЈА НА ЕДНО ОПТИМАЛНО МНОЖЕЊЕ НА МАТРИЦИ

```
1   PecOpZagr(i, j)
```

```
2   ако i = j тогаш печати "X"i инаку
3     {
4       печати "(";
5       PecOpZagr(i, C[i, j]);
6       PecOpZagr(C[i, j] + 1, j);
7       печати ")";
8     }
```

На нашиот пример ќе прикажеме како ова работи. Прво се повикува

$$PecOpZagr(1,6)$$

Бидејќи $i < j$, од C се чита дека $C[1,6] = 3$ и се добива:

$$(PecOpZagr(1,3)\ PecOpZagr(4,6))$$

$C[1,3] = 1$ и па се добива

$$((PecOpZagr(1,1)\ PecOpZagr(2,3))\ PecOpZagr(4,6))$$

Во првиот дел 1=1, па во наредниот чекор имаме:

$$((X1\ PecOpZagr(2,3))\ PecOpZagr(4,6))$$

$C[2,3] = 2$ па се добива

$$((X1(PecOpZagr(2,2)\ PecOpZagr(3,3)))\ PecOpZagr(4,6))$$

2=2 и 3=3, па во ова ќе се скрати на:

$$((X1(X2\ X3))\ PecOpZagr(4,6))$$

$C[4,6] = 5$, од каде

$$((X1(X2\ X3))\ (PecOpZagr(4,5)PecOpZagr(6,6)))$$

6=6 и $C[4,5] = 4$, од каде

$$((X1(X2\ X3))\ ((\text{PecOpZagr}(4, 4)\text{PecOpZagr}(5, 5))\ X6))$$

На крај

$$((X1(X2\ X3))((X4X5)X6))$$

На крај да ја пресметаме сложеноста, која овде ќе ја покажеме во детали. Прво имаме само еден циклус до n во кој се пополнува главната дијагонала на матрицата A, и тој има сложеност $\Theta(n)$. Потоа имаме три вгнездени циклуси: во првиот l оди од 2 до n, во вториот i оди од 1 до $n - l + 1$, а во третиот: k оди од i до j $- 1 = i + l - 1 - 1 = i + l - 2$. Оттука вкупниот број на операции одговара на следнава сума:

$$\sum_{l=2}^{n}\sum_{i=1}^{n-l+1}\sum_{k=i}^{i+l-2} 1 = \sum_{l=2}^{n}\sum_{i=1}^{n-l+1}(l-1) = \sum_{l=2}^{n}(l-1)\sum_{i=1}^{n-l+1} 1$$

$$= \sum_{l=2}^{n}(l-1)(n-l+1) = \sum_{l=1}^{n-1} l(n-l)$$

$$= n\sum_{l=1}^{n-1} l - \sum_{l=1}^{n-1} l^2 = \frac{n^2(n-1)}{2} - \frac{n(n-1)(2n-1)}{6}$$

$$= \frac{n(n-1)(n+1)}{6}$$

Од каде се добива дека сложеноста на алгоритамот за пресметка на оптималното решение е $\Theta(n^3)$. Бидејќи пополнуваме квадратна матрица, за зачувување на пресметаните вредности ни треба простор $\Theta(n^2)$. Реконструкцијата оди со рекурзија, и во таа рекурзија најлош случај е кога $k = i$ или $k = j$, но тоа не е многу очигледно. Овде ќе дадеме пресметка за просечниот случај со која ќе покажеме дека таа има линеарна сложеност.

Ако две поставувања на загради ги сметаме дека тежат односно имаат сложеност $\mathrm{O}(1)$, тогаш сложеноста на $PecOpZagr$ за верига со големина 1 е $\mathrm{O}(1)$. Нека бројот на операции за верига со големина n ја обележиме со $S(n)$. Можеме да претпоставиме дека секое k има иста веројатност да се добие. Нека направиме индуктивна претпоставка дека за $m < n, S(m) = m$. Тогаш:

$$S(n) = 1 + \frac{1}{n}\sum_{k=1}^{n-1}\big(S(k) + S(n-k)\big) =$$

$$= 1 + \frac{2}{n}\sum_{k=1}^{n-1} S(k) = 1 + \frac{2}{n}\sum_{k=1}^{n-1} k = n.$$

Прво го даваме решението во c++

C++ 4. 3 ВЕРИЖНО МНОЖЕЊЕ НА МАТРИЦИ.

```
1  #include<iostream>
2  #include<bits/stdc++.h>
3  using namespace std;
4
5  void pechatiZagradi(int *C, int i, int j, int n)
6  {
7     if (i == j)
8     {
9        cout << "X" << (i+1);
10    }
11    else
12    {
13       cout << "(";
14       pechatiZagradi(C, i, C[i*n+j], n);
15       pechatiZagradi(C, C[i*n+j]+1, j, n);
16       cout << ")";
17    }
18 }
19
20 int main()
21 {
22    int n; // broj na matrici
23    cin >> n;
24
25    int p [n+1];
26    for(int i=0;i<=n;i++)
27    {
28       cin >> p[i];
29    }
30
31    int A [n][n];
32    int C [n][n];
33
```

```
34    for(int i=0;i<n;i++)
35       A[i][i]  = 0;
36
37    for(int l=0;l<n;l++)
38    {
39       for(int i=0;i<n-l-1;i++)
40       {
41          int j = i+l+1;
42          A[i][j] = INT_MAX;
43          for(int k=i;k<=j-1;k++)
44          {
45             int q = A[i][k] + A[k+1][j] +
p[i]*p[k+1]*p[j+1];
46             if(q<A[i][j])
47             {
48                A[i][j] = q;
49                C[i][j] = k;
50             }
51          }
52       }
53    }
54    cout << A[0][n-1] << endl;
55    pechatiZagradi((int *)&C, 0, n-1, n);
56    return 0;
57 }
```

Решението во Јава е дадено со JAVA 4. 1 :

JAVA 4. 3 ВЕРИЖНО МНОЖЕЊЕ НА МАТРИЦИ.

```
1 import java.util.*;
2
3 public class Main {
4     public static void pechatiZagradi(int[][] C, int i, int
j) {
5         if (i == j) {
6             System.out.print("X" + (i + 1));
7         } else {
8             System.out.print("(");
```

```
9             pechatiZagradi(C, i, C[i][j]);
10            pechatiZagradi(C, C[i][j] + 1, j);
11            System.out.print(")");
12        }
13    }
14
15    public static void main(String[] args) {
16        Scanner scn = new Scanner(System.in);
17        int n = scn.nextInt(); // broj na matrici
18
19        int[] p = new int[n + 1];
20        for (int i = 0; i <= n; i++) {
21            p[i] = scn.nextInt();
22        }
23
24        int[][] A = new int[n][n];
25        int[][] C = new int[n][n];
26
27        for (int i = 0; i < n; i++)
28            A[i][i] = 0;
29
30        for (int l = 0; l < n; l++) {
31            for (int i = 0; i < n - l - 1; i++) {
32                int j = i + l + 1;
33                A[i][j] = Integer.MAX_VALUE;
34                for (int k = i; k <= j - 1; k++) {
35                    int q = A[i][k] + A[k + 1][j] + p[i] *
p[k + 1] * p[j + 1];
36                    if (q < A[i][j]) {
37                        A[i][j] = q;
38                        C[i][j] = k;
39                    }
40                }
41            }
42        }
43        System.out.println(A[0][n - 1]);
44        pechatiZagradi(C, 0, n - 1);
45    }
46}
```

Прашања и задачи

1. Нека е дадена е низа $X_1, X_2, \ldots, X_n$, од n матрици од ред $p_0 \times p_1$, $p_1 \times p_2$,..., $p_{n-1} \times p_n$, кои треба да се помножат и доволно процесори за паралалено процесирање. Сакаме да ги помножиме матриците за минимално време, при што на располагање ни се сите процесори. Ако на пример имаме 5 матрици A, B, C, D и E од ред 2×5, 5×3, 3×4, 4×6 и 6×3 можеме да ги помножиме на следниов начин $((AB)\,C)(DE)$ за да помножиме DE ни требаат 72 операции, за $(AB)\,C$ 30+24, затоа што за AB требаат 30 а за ова да се помножи со C требаат уште 24 операции. Така за да се помножи целото на овој начин, потребни ни се 24+54=78 операции. Да се даде рекурзивната релација која го решава проблемот!

2. Дадена низа од координати на точки од рамнината, кои претставуваат темиња на конвексен полигон. Треба да се направи триангулација на полигонот таква што вкупната сума на сите страни на триаголниците биде најмала. (ако некоја отсечка се јавува како страна на два триаголници, тогаш таа се брои двапати.

   a. Дади ја рекурзивната равенка што го решава овој проблем.

   b. Колкава ќе биде сложеноста на алгоритамот?

   c. Каква ќе биде равенката ако секоја страна ја броиме само по еднаш?

3. Дадена е целобројната низа A од n елементи. Да се вметнат загради во изразот $a_1 - a_2 \ldots - a_n$, така што вредноста на изразот да биде минимална.

   a. Дади ја рекурзивната равенка што го решава овој проблем.

   b. Колкава ќе биде сложеноста на алгоритамот?

4. Игра слична на играта Зулу се игра на следниов начин: Дадена е низа од топчиња во различна боја. Во секој чекор се избира топче и од низата се отстранува најголемиот можен палиндром од (непарен број) топчиња, на кој што избраното топче е

средина. Со секое отстранување на топчиња во низата остануваат празни простори, односно низата може да се подели на повеќе поднизи. Во секој потез се отстрануваат топчиња само во рамките на една подниза. Не пример ако е дадена низа топчиња aabaca и прво го избереме топчето b, ќе останат две поднизи a и ca. Ако во наредниот потез го избереме топчето c, тогаш ќе се отстрани само тоа топче, а не целиот остаток aca. За дадена низа од n топчиња, $x_1, x_2, \ldots, x_n$ треба се пресмета најмалиот број на потези за да се отстранат сите топчиња од низата.

a. Да се даде алгоритам, кој работи во време $O(n^3)$ сличен на алгоритамот за верижно множење на матрици, кој ќе го реши овој проблем!

b. Дади својство кое го има овој проблем кое ќе ти помогне да ја редуцираш димензијата на решението до $O(n^2)$.

Проблем 4. 4. Оптимално бинарно пребарувачко дрво

Во делот за правилно распределување на загради напоменавме дека Каталанoвиот број се јавува на повеќе места, а едно од тие ситуации е дадена во првата задача од тој дел, број на целосни бинарни дрва. Ива е оптимизационен проблем кој се базира на такви дрва, па идејата за решавање е слична со идејата во претходниот проблем.

Имено проблемот на оптимално пребарувачко бинарно дрво, познат и како тежински - балансирано бинарно дрво кој овде го презентираме е еден од најпознатите проблеми од динамичко програмирање на кој неоптимизираното решение со динамичко програмирање е со сложеност $O(n^3)$. Првично дефиниран од Кнут [13]. Да напоменеме дека Кнут има дадено убрзување на алгоритамот до време $O(n^2)$, а Курт [14] го забрзал и до време $O(n)$, но овде ќе ја дадеме основната форма која работи во време $O(n^3)$. Проблемот е интересен и познат поради тоа што има голема практична примена во многу ситуации каде што се бара пребарување низ множество на зборови. На пример, ако дизајниравме програма за превод од англиски на македонски, тогаш за секој збор треба да го најдеме еквивалентниот во нашиот јазик. Бидејќи ќе го пребаруваме секој збор, сакаме тоа пребарување да трае најкратко, поточно најкратко во просек. Тоа значи ако збор се појавува почесто, да се наоѓа во помал број на чекори, а ако се бара поретко во поголем број на чекори. Еден начин да се направи тоа е зборовите да се зачуваат во бинарно пребарувачко дрво. Тогаш почестите зборови би требало да бидат поблизу до коренот. Можно е некои зборови да ги нема во нашиот јазик и самиот алгоритам треба да работи така што ќе дава одговор дека за дадениот збор кој го немаме соодветен превод.

Дефинирање на проблемот

Дадена е подредена низа $k_1, k_2, \ldots, k_n$ од n различни клуча. За секој клуч k_i ни е дадена веројатност p_i дека тој ќе се пребарува. Дозволено е да се пребаруваат и елементи кои не се во низата, па

така имаме и $n + 1$ фиктивни клуча $d_0, d_1, \ldots, d_n$, кои ги претставуваат вредностите кои не се во низата $k_1, k_2, \ldots, k_n$, односно ги претставуваат интервалите помеѓу два клуча. Така d_0 ги претставува сите вредности помали од k_1, d_n ги претставува сите вредности поголеми од k_n и за *i* = 1, 2, . . . , *n* – 1, d_i ги претставува сите вредности меѓу k_i и k_{i+1}. За секој фиктивен клуч d_i дадена ни е веројатност q_i за пребарување на вредностите меѓу k_i и k_{i+1}. Од овие клучеви треба да се изгради бинарно пребарувачко во кое секој клуч k_i е внатрешен јазол, а секој фиктивен клуч е лист. Секое пребарување е или успешно (наоѓање на клуч k_i) или неуспешно (наоѓање на фиктивен клучн d_i), па сумата од сите веројатности е еднаква на 1, т.е.

$$\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1.$$

Проблемот е да се најде вакво бинарно пребарувачко дрво за кое просечниот број на чекори за пребарување во него биде најмал. Ова ќе го нарекуваме цена на дрвото, па сакаме оваа цена да биде најниска.

Анализа на проблемот

За да го разбереме проблемот ќе разгледаме како изгледа такво дрво и како за дадено дрво се пресметува просечниот број на чекори во него. Тука ќе го разгледаме примерот дадени во [15]. Нека се дадени 5 клуча со соодветни веројатности дадени во Табела 3, за кое едно пребарувачко дрво, кое е балансирано е дадено на Слика 11.

Табела 4. 4. Веројатности за пребарување на клучеви и фиктивни клучеви

i	0	1	2	3	4	5
p_i		0.15	0.1	0.05	0.1	0.2
q_i	0.05	0.1	0.05	0.05	0.05	0.1

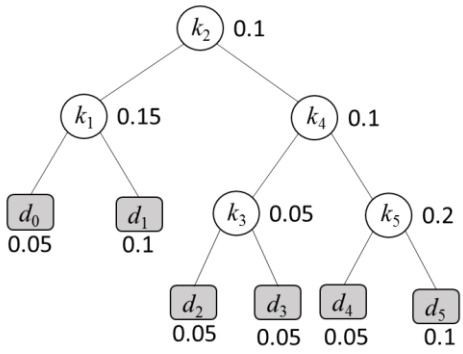
Цената на пребарување на даден збор е бројот на јазли кои се испитуваат додека да се дојде до бараниот збор или да се утврди

дека истиот го нема во дрвото. Тоа е всушност длабочината на соодветниот јазол во дрвото, плус 1. Нека со $D(k_i)$ и $D(d_i)$ ги обележиме длабочините на соодветните јазли во дрвото. Тогаш очекуваната цена за пребарување на одреден клуч е $(D(k_i)+1)p_i$, а фиктивен клуч е $(D(d_i)+1)q_i$. За дадено дрво Т, очекуваната цена може да се пресмета по формулата:

$$E(T) = \sum_{i=1}^{n}(D(k_i)+1)p_i + \sum_{i=0}^{n}(D(d_i)+1)q_i$$

$$= 1 + \sum_{i=1}^{n} D(k_i)p_i + \sum_{i=0}^{n} D(d_i)q_i.$$

Оттука очекуваната цена за дрвото од Слика 4. 3 е во табелата на сликата и таа е еднаква на 2.8. Иако дрвото е балансирано, тоа не значи дека е тоа најдобрата можна цена. Со небалансирано дрво може да се постигне подобра цена, како што е дадено на Слика 4. 4.



теме	p_i	длабочина	цена
k_1	0.15	1	0.3
k_2	0.1	0	0.1
k_3	0.05	2	0.15
k_4	0.1	1	0.2
k_5	0.2	2	0.6
d_0	0.05	2	0.15
d_1	0.1	2	0.3
d_2	0.05	3	0.2
d_3	0.05	3	0.2
d_4	0.05	3	0.2
d_5	0.1	3	0.4
			2.8

Слика 4. 3. Балансирано бинарно пребарувачко дрво и неговата цена.

Анализата на проблемот ќе ја започнеме со набљудување за поддрвата. Може да се забележи дека секое поддрво на од пребарувачкото дрво ги содржи сите клучеви во некој опсег. Тоа значи дека за некои $1 \leq i \leq j \leq n$ дрвото ги содржи сите клучеви во опсегот од k_i до k_j. Поддрвото што ги содржи клучевите $k_i, \dots k_j$ исто така мора како лисја да ги има фиктивните клучеви $d_i, \dots d_j$. На

пример за дрвото од Слика 11, левото поддрво на k_2 ги содржи сите клучеви лево од k_2, а десното сите клучеви десно од k_2. Така, левото поддрво го содржи клучот k_1 и фиктивните клучеви d_1 и d_2, додека десното поддрво ги содржи k_3, k_4 и k_5 и фиктивните клучеви d_2, d_3, d_4 и d_5,

Сега можеме да ја констатираме оптималната потструктура:

> *Ако оптимално бинарно пребарувачко дрво T' на дрвото T ги содржи клучевите $k_i, \dots k_j$, тогаш ова дрво T' мора да биде оптимално за низата клучеви $k_i, \dots k_j$ со веројатности $p_i, \dots p_j$ и низата фиктивни клучеви $d_{i-1}, d_i \ \dots d_j$ со веројатности $q_i, \dots q_j$.*

Доказ на оптималното својство: Ако T' е поддрво на оптималното дрво T', но тоа само по сее не е оптимално, тогаш него можеме да го замениме со оптималното дрво за низата клучеви $k_i, \dots k_j$ со веројатности $p_i, \dots p_j$ и низата фиктивни клучеви $d_{i-1}, d_i \ \dots d_j$ со веројатности $q_i, \dots q_j$. Со тоа ќе добиеме помала цена, што е контрадикторно со претпоставката дека T било оптималното дрво.

□

Нека за дадена подниза од клучеви $k_i, \dots k_j$, еден од нив е коренот на оптималното дрво. Нека тој корен е k_r . Во едното поддрво на k_r ќе бидат клучевите $k_i, \dots k_{r-1}$, а во десното $k_{r+1}, \dots k_j$. Ако како клуч се земе првиот или последниот елемент се добива дрво без клучеви, што е посебен случај. Сепак и ваквите поддрва имаат елементи, затоа што содржат фиктивни клучеви, па во ваквите поддрва без клучеви всушност ќе се најде соодветниот фиктивен клуч.

Треба да се напоменат две работи: прво, дека ние не знаме кое е k_r, па треба да се провери за сите, и второ дека за секое k_r за кое ќе проверуваме левото и десното дрво се со помала должина и тие се оптималните дрва за соодветната подниза од клучеви и фиктивни клучеви. Затоа тие, според оптималното својство можат да бидат пресметани рекурзивно.

Ако со $T_{i,j,r}$ го обележиме оптималното поддрво кое ги содржи клучевите $k_i, \dots k_j$ и фиктивните клучеви $d_{i-1}, d_i \dots d_j$, а коренот му е во $r, r = \overline{i,j}$, тогаш негова оптимална цена $A[i,j]$ е:

$$A[i,j] = \min_{i \le r \le j} \{\text{цената на } T_{i,j,r}\}.$$

Која е цената на $T_{i,j,r}$? Тоа ги содржи поддрвата со клучеви $k_i, \dots k_{r-1}$ и $k_{r+1}, \dots k_j$, на кои оптималните цени им се $A(i, r-1)$ и $A(r+1, j)$. Но тие дрва сега се поддрва на дрво со корен во k_r и секое нивно теме се спуишта за едно ниво подолу. Така ако во случајот кога тоа дрво е изолирано до некој јазол се стигало на пример во 3 чекори, кога ќе го ставиме како поддрво на некој корен, до тој јазол ќе се стига во 4 чекори. Значи цената на секој јазол ќе се зголеми, и тоа точно за веројатноста која ја имаме за тој клуч или фиктивен клуч. Затоа како подрутина треба да ја пресметаме тежината на секое поддрво, односно сумата на веројатностите на сите јазли во тоа поддрво. За ова ќе користиме нова функција дефинирана на следниов начин:

$$w(i,j) = \sum_{t=i}^{j} p_t + \sum_{t=i-1}^{j} q_t.$$

За коренот на дрвото, k_r, ќе ни треба еден чекор за проверка, па тој ќе допринесува во вкупната цена со p_r Оттука,

$$\begin{aligned} A[i,j] &= \min_{i \le r \le j} \{(A[i, r-1] + w[i, r-1]) + (A[r+1, j] + w[r+1, j]) \\ &\quad + p_r\} \\ &= \min_{i \le r \le j} \{A[i, r-1] + A[r+1, j] + w[i,j]\}. \end{aligned}$$

Почетниот случај е всушност случајот кога имаме само фиктивен клуч. Ова може да се направи и поинаку, но наједноставно е да сметаме дека во овој почетен случај $j = i - 1$. Во оваа ситуација во дрвото е само фиктивниот клуч d_{i-1} и очекуваната цена на ваквото дрво е q_{i-1}.

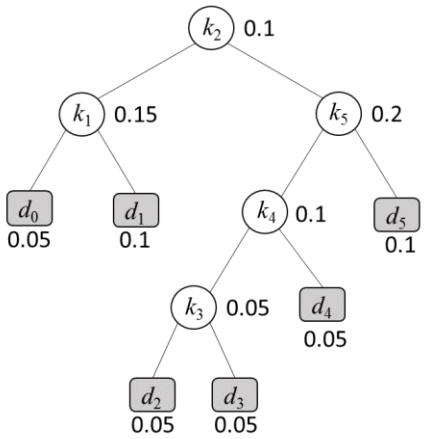
Нашето решение ќе се содржи во вредноста $A[1,n]$. Сега ја добиваме целосната рекурзивна формула:

$$A[i,j]=\begin{cases} q_{i-1}, & j=i-1 \\ \min\limits_{i\le r\le j}\{A[i,r-1]+A[r+1,j]+w[i,j]\}, & i\le j \end{cases}.$$

Да забележиме дека и $w[i,j]$ може да се дефинира рекурзивно со

$$w[i,j]=w[i,j]+p_j+q_j.$$

За да го реконструираме оптималното бинарно пребарувачко дрво дефинираме нова функција $C[i,j]$, за $1 \le i \le j \le n$, која го зачувува индексот r на коренот k_r на оптималното бинарно пребарувачко дрво кое ги содржи клучевите $k_i, \dots k_j$.



теме	p_i	длабочина	цена
k_1	0.15	1	0.3
k_2	0.1	0	0.1
k_3	0.05	3	0.2
k_4	0.1	2	0.3
k_5	0.2	1	0.4
d_0	0.05	2	0.15
d_1	0.1	2	0.3
d_2	0.05	4	0.25
d_3	0.05	4	0.25
d_4	0.05	3	0.2
d_5	0.1	2	0.3
			2.75

Слика 4. 4. Оптимално бинарно пребарувачко дрво и неговата цена.

Пред да го дадеме псевдокодот на алгоритамот да видиме како истиот ќе работи на нашиот пример. Почетните случаи, за дрва кои се состојат од само еден фиктивен клуч, се дадени во Табела 4. 5:

Табела 4. 5. Почетни вредности на функцијата, за дрва кои се состојат од само еден фиктивен клуч

	$i=1$ $j=0$	$i=2$ $j=1$	$i=3$ $j=2$	$i=4$ $j=3$	$i=5$ $j=4$	$i=6$ $j=5$
$w[i,j]$	0.05	0.1	0.05	0.05	0.05	0.1

$A[i,j]$	0.05	0.1	0.05	0.05	0.05	0.1

За $i = j$ имаме, односно кога имаме дрва кои се состојат од еден клуч и соодветните фиктивни клучеви како негово лево и десно дете е дадено во Табела 4. 6:

Табела 4. 6. Вредности на функцијата за дрва кои се состојат од само еден клуч и два фиктивни клучеви.

	$i = j = 1$	$i = j = 2$	$i = j = 3$	$i = j = 4$	$i = j = 5$
$w[i,j]$	.05+.15+.1 = 0.3	.1+.1+.05= 0.25	.05+.05+.0 5= 0.15	.05+.1+.05=0 .2	.05+.2+.1=0. 35
$A[i,j]$	.05+.1+.3= 0.45	.1+.05+.25 = 0.4	.05+.05+.1 5= 0.25	.05+.05+.2= 0.3	.05+.1+.35= 0.5
$C[i,j]$	1	2	3	4	5

За $i = j - 1$ се зема помалата од две можости, која во табелата Табела 4. 7. е напишано здебелено. За $i = 4$ и $j = 5$, па за $C[4,5]$ може да се избере било кое од нив, но во нашиот алгоритам се бира последното до кое ќе дојдеме.

Табела 4. 7. Вредности на функцијата за дрва кои се состојат од два клуча

	$i = 1, j = 2$	$i = 2, j = 3$	$i = 3, j = 4$	$i = 4, j = 5$
$w[i,j]$	.3+.1+.05=0.45	.25+.05+.05=0.35	.15+.1+.05=0.3	.2+.2+.1=0.5
$A[i,j]$	**.05+.4+.45=0.9** .45+.05+.45=.95	**.1+.25+.35=0.7** .4+.05+.35=0.8	.05+.3+.3=0.65 **.25+.05+.3=0.6**	.05+.35+.5=0.9 **.3+.1+.5=0.9**
$C[i,j]$	1	2	4	5

За $i = j - 2$ се зема најмалата од три можности, а за $i = j - 3$ се зема најмалата од четири можости. Минималните вредности во табелите Табела 4. 8. и Табела 4. 9. се напишани здебелено.

Табела 4. 8. Вредности на функцијата за дрва кои се состојат од три клуча

	$i = 1, j = 3$	$i = 2, j = 4$	$i = 3, j = 5$
$w[i,j]$	.45+.05+.05= 0.55	.35+.1+.05=0.5	.3+.2+.1=0.6
$A[i,j]$	.05+.7+.55=1.3 **.45+.25+.55= 1.25** .9+.05+.55=1.5	.1+.6+.5=1.2 **.4+.3+.5=1.2** .7+.05+.5= 1.25	.05+.9+.6= 1.55 .25+.5+.6= 1.35 **.6+.1+.6=1.3**

$C[i,j]$	2	2	5

Табела 4. 9. Вредности на функцијата за дрва кои се состојат од четири клуча

	$i = 1, j = 4$	$i = 2, j = 5$
$w[i,j]$	.55+.1+.05= 0.7	.5+.2+.1= 0.8
$A[i,j]$	.05+1.3+.7= 2.25 **.45+.6+.7=1.75** .9+.3+.7=1.9 1.5+.05+.7= 2.25	.1+1.2+.8=2.1 .4+.9+.8=2.1 **.7+.5+.8=2** 1.25+.1+.8= 2.15
$C[i,j]$	2	4

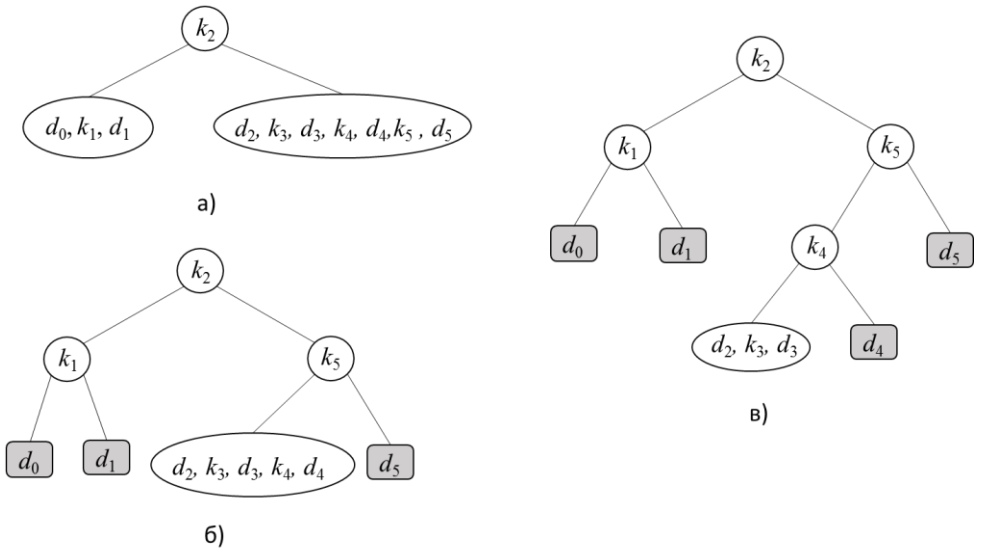
Оптималното дрво е дрвото претставен на Слика 4. 4. Псевдокодот за наоѓање на оптималното решение е следниов:

П 4. 8. РЕКОНСТРУКЦИЈА НА ЕДНО ОПТИМАЛНО МНОЖЕЊЕ НА МАТРИЦИ

```
1   за i од 1 до n + 1 прави A[i, i − 1] = w[i, i − 1] = q_{i−1};
2   за l од 1 до n прави
3     за i од 1 до n − l + 1 прави
4       {
5        j = i + l − 1;
6        A[i, j] = ∞;
7        w[i, j] = w[i, j − 1] + p_i + q_i;
8        за r од i до j прави
9          {
10         t = A[i, r − 1] + A[r + 1, j] + w[i, j];
11         ако t < A[i, j] тогаш
12           {
13            A[i, j] = t;
14            C[i, j] = r;
15           }
16         }
17      }
```

18 врати $A[i,j]$ и $C[i,j]$

Во псевдокодот имаме три вгнездени циклуси, па сложеноста е $O(n^3)$. Пресметките се слични како и во претходниот проблем Роман, панема потреба да ги даваме.



Слика 4. 5. Реконструкција на оптималното решение врз основа на функцијата $C[i,j]$. а) Коренот на дрвото е во k_2. б) Коренот на поддрвото со првиот клуч е во k_1, а на поддрвото составено од третиот до петтиот клуч е во k_5. в) Коренот на поддрвото со третиот и четвртиот клуч е во k_4

Псевдокодот за печатење на оптималното дрво е сличен на тој во проблемот за оптимално множење на матрици. Кодот за печатење на предредоследниот редослед на темињата во дрвото е следниот:

П 4. 9. РЕКОНСТРУКЦИЈА НА ЕДНО ОПТИМАЛНО БИНАРНО ДРВО

```
1  PecatiOpDrvo(i, j)
2  ако i = j – 1 тогаш печати "d"j инаку
3      {
4       печати C[i, j];
5       PecatiOpDrvo (i, C[i, j] – 1);
```

```
6       PecatiOpDrvo (C[i,j] + 1,j);
7     }
```

Во печатењето на оптималното дрво секое теме го печатиме точно по еднаш, па сложеноста е $O(n)$.

Прво го даваме решението во c++

C++ 4. 4 ОПТИМАЛНО БИНАРНО ПРЕБАРУВАЧКО ДРВО.

```
1  #include<iostream>
2  #include <limits>
3  #include <iomanip>
4  #include <cstring>
5  using namespace std;
6
7  void print(float* array, int rows, int cols)
8  {
9     for (int r = 0; r < rows; ++r)
10    {
11       for (int c = 0; c < cols; ++c)
12          cout << fixed << setprecision(2) << array[
(r*rows) + c ] << " ";
13       cout << endl;
14    }
15 }
16
17 int main()
18 {
19    int n;
20    cin >> n;
21    float p [n+1];
22    float q [n+1];
23    for(int i=1;i<=n;i++)
24    {
25       cin >> p[i];
26    }
27    for(int i=0;i<=n;i++)
28    {
29       cin >> q[i];
30    }
31
32    float A [n+2][n+2];
33    float C [n+2][n+2];
34    float w [n+2][n+2];
```

```
35    memset(A, 0, sizeof(float) * (n+2)*(n+2));
36    memset(C, 0, sizeof(float) * (n+2)*(n+2));
37    memset(w, 0, sizeof(float) * (n+2)*(n+2));
38
39    for(int i=1;i<=n+1;i++)
40       A[i][i-1]  = w[i][i-1] = q[i-1];
41
42    for(int l=1;l<=n;l++)
43    {
44       for(int i=1;i<=n-l+1;i++)
45       {
46          int j = i+l-1;
47
48          A[i][j] = numeric_limits<float>::max();
49          w[i][j] = w[i][j-1]+p[j]+q[j];
50          for(int r=i;r<=j;r++)
51          {
52             float t = A[i][r-1]+A[r+1][j]+w[i][j];
53             if(t<A[i][j])
54             {
55                A[i][j] = t;
56                C[i][j] = r;
57             }
58          }
59       }
60    }
61
62    cout << "A" << endl;
63    print(*A, n+2, n+2);
64    cout << "w" << endl;
65    print(*w, n+2, n+2);
66    cout << "C" << endl;
67    print(*C, n+2, n+2);
68
69    return 0;
70 }
```

Решението во Јава е дадено со JAVA 4. 1 :

JAVA 4. 4 ОПТИМАЛНО БИНАРНО ПРЕБАРУВАЧКО ДРВО..

```
1   import java.util.*;
2
3  public class Main {
4     public static void printMatrix(float[][] matrix) {
5        for (int row = 0; row < matrix.length; row++) {
6           for (int col = 0; col < matrix[row].length; col++)
{
7              System.out.printf("%.2f ", matrix[row][col]);
8           }
9           System.out.println();
10       }
11    }
12    public static void main(String[] args) {
13       Scanner scn = new Scanner(System.in);
14       int n = scn.nextInt();
15
16       float[] p = new float[n + 1];
17       float[] q = new float[n + 1];
18       for (int i = 1; i <= n; i++) {
19          p[i] = scn.nextFloat();
20       }
21       for (int i = 0; i <= n; i++) {
22          q[i] = scn.nextFloat();
23       }
24
25       float[][] A = new float[n + 2][n + 2];
26       float[][] C = new float[n + 2][n + 2];
27       float[][] w = new float[n + 2][n + 2];
28
29       for (int i = 1; i <= n + 1; i++)
30          A[i][i - 1] = w[i][i - 1] = q[i - 1];
31
32       for (int l = 1; l <= n; l++) {
33          for (int i = 1; i <= n - l + 1; i++) {
34             int j = i + l - 1;
35             A[i][j] = Float.MAX_VALUE;
36             w[i][j] = w[i][j - 1] + p[j] + q[j];
37             for (int r = i; r <= j; r++) {
38                float t = A[i][r - 1] + A[r + 1][j] +
w[i][j];
39                if (t < A[i][j]) {
40                   A[i][j] = t;
41                   C[i][j] = r;
42                }
```

```
43                  }
44              }
45          }
46          System.out.println("A");
47          printMatrix(A);
48          System.out.println("w");
49          printMatrix(w);
50          System.out.println("C");
51          printMatrix(C);
52      }
53  }
```

Прашања и задачи

1. Колкава ќе биде сложеноста на алгоритамот со груба сила кој го решава овој проблем?
2. Изведи ја сложеноста на алгоритамот ако се направи рекурзивен алгоритам директно врз рекурзивната равенка, без мемоизација?
3. Што е разликата ако овој алгоритам наместо на дрво со n внатрешни темиња се аплицира на дрво со вкупно n темиња?
4. Дадени се n различни природни броеви. Колку различни максимални бинарни купови можат да се направат од нив? Бинарен куп бинарно дрво за кое важи дека секој родител е поголем од неговите деца.
    a. Дади ја рекурзивната равенка!
    b. Дискутирај ја сложеноста на алгоритамот!

5 Проблеми за генерирање на пермутации

Во претходните поглавја разгледувавме комбинаторни проблеми од динамичко програмирање во кои се бројат елементи од некое множество. Во овој дел ќе разгледаме слични проблеми, но од малку поинаков агол. Имено, ќе претпоставиме дека елементите од некое множество се подредени според дадено правило. Тие елементи се обично низи од карактери, односно множеството е множество од стрингови. Правилото по кое тие се подредуваат е најчесто така наречениот лексикографски редослед. Лексикографски редослед (исто така познат како азбучен редослед или редослед на речникот) е начин на кој стринговите се подредени врз основа на азбучен редослед на карактерите од кои се составени. За ова прво ни е потребно да дефинираме подредување на карактерите, односно на азбуката, од кои се составени стринговите.

Дефиниција

> *Нека Σ е дадена подредена азбука. Еден стринг α е пред друг стринг β во лексикографскиот редослед, ако првата буква од α која се разликува од буквата на иста позиција во β е понапред во азбуката. Ако α е префикс на β исто така е пред β. Поформално, ако $\alpha = \gamma\sigma\tau$, $\beta = \gamma\sigma'\tau'$ за некои карактери $\sigma \neq \sigma'$ и стрингови γ, τ и τ', тогаш σ е пред σ' во подредувањето на азбуката.*

При вакво подредување на множество на стрингови можни се две прашања:

- За даден елемент од тоа множество да се определи позицијата во множеството на која се наоѓа тој елемент;
- За дадена позиција да се определи кој елемент стои на таа позиција.

Идејата ќе ја објасниме преку еден познат проблем, проблемот на запишување на целите ненегативни броеви во бинарен запис. Првиот број во ова множество е 0, вториот 1, третиот 10, и така натаму. Како ќе определиме кој по ред е бројот кој со 6 бита се запишува како 001010? Од познатиот алгоритам за претворање на бинарен број во декаден број имаме:

$$001010_2 = 2^3 + 2^1 = 10_{10},$$

што е всушност 11-тиот број во низата.

Проблемите од динамичко програмирање кои овде ќе ги анализираме обично работат на тој начин што бројат колку од елементите се позиционирани пред елементот чија позиција треба да ја определиме. Анализата ја правиме на следниов начин:

- Првата единица во бинарната низа се наоѓа на третата позиција, па тоа значи дека пред бројот кој се претставува со оваа бинарна низа се сите броеви кои започнуваат со 3 нули. Тоа се броевите кои на последните 3 позиции имаат било кој бит, т.е. нивниот број е $2^3 = 8$.
- Сега треба да провериме колку броеви на кои бинарниот запис им започнува со 001 се пред нашиот број. Наредната единица во бинарната низа се наоѓа на првата позиција, што значи дека сите 6-битни броеви кои започнуваат со 00100 се пред нашиот број, а такви се $2^1 = 2$, затоа што има само толку начини да се пополни последната позиција. Оттука следува дека пред нашиот број има 10 броја, т.е. нашиот е на 11-та позиција, па тоа е бројот 10 во декаден запис.

Да го погледнеме обратното! Како ќе определиме која 6-битна бинарна низа е 50-та по ред? Секако, знаеме дека на 50-та позиција е бројот 49 и еден начин да го најдеме е стандардниот начин:

- 49:2=24(остаток 1), па последната цифра е 1
- 24:2=12(остаток 0), па претпоследната цифра е 0;
- 12:2=6(остаток 0), па третата цифра од позади е 0;
- 6:2=3(остаток 0), па четвртата цифра од позади е 0
- 3:2=1(остаток 1), па петтата цифра од позади е 1;
- 1:2=0(остаток 1), и шестата цифра од позади е 1.

Тука ги прекинуваме пресметките и го добиваме бројот 110001. Ако бројот треба да се запише со повеќе цифри, напред додаваме цифри 0.

Како што кажавме претходно, алгоритмите кои овде ќе ги анализираме обично работат на тој начин што прво се генерира првиот карактер со кој се запишува елементот, па затоа ќе илустрираме како бинарниот број може да се генерира во обратна насока:

- Знаеме дека има $2^6 = 64$ различни броеви кои се запишуваат со 6 бита. Половината од нив почнуваат со 1, а другата половина со 0. Тие што почнуваат со 0 се $2^5 = 32$. Бидејќи нам ни треба бројот на 50-та позиција, пред него се сите оние што на шестата позиција имаат 0. Оттука, нашиот број на прва позиција има 1.
- Потоа, можеме да забележиме дека од броевите кои започнуваат со 1-ца, нашиот број е 50 – 32 = 18-тиот по ред. Тоа значи дека ако ја игнорираме таа единица, треба да го најдеме 18-тиот број кој се запишува со низа од битови со должина 5. Од 32-та броја кои се запишуваат со 5 бита, повторно половината, т.е. 16, започнуваат со 0. Бидејќи бројот 18 > 16, пред нашиот број се сите оние броеви што започнуваат со 0, од каде бројот што не интересира има 1 и на втората позиција, односно, на петтата позиција од позади.
- Повторно можеме да забележиме дека од броевите кои започнуваат со 11, тој што го бараме е 18 – 16 = 2-от по ред, поточно, ни треба вториот број по ред од броевите кои се запишуваат со 4 бита. Од 16-те броеви кои се запишуваат со 4

бита, првите 8 започнуваат со 0. Бројот што не интересира е еден од нив, па наредната цифра му е 0.

- Слично, од 8-те 3-битни броеви, 4 започнуваат со 0. Од 2 < 4, следува дека и петтата цифра на бројот што го бараме е 0.
- Од 4-те 2-битни броеви, 2 започнуваат со 0, па нашиот број на петта позиција има 0.
- Од двата броја кои можат да се запишат со еден бит, на нас ни треба вториот, па значи шестата позиција имаме 1.

Да илустрираме како врз основа на горната анализа се изведува рекурзивната релација. Нека $A[i,j]$ е i- тата бинарна низа по ред со должина j. Тогаш

$$A[i,j] = \begin{cases} 1 \circ A[i-2^{j-1}, j-1], & i > 2^{j-1}, j > 1 \\ 0 \circ A[i, j-1] & 1 \le i \le 2^{j-1}, j > 1 \end{cases}.$$

Почетните услови се $A[1,1] = 0$ и $A[2,1] = 1$. Знакот „$\circ$“ означува конкатенација, т.е. спојување. Формулата може да се искористи и за $A[1,1]$ и $A[2,1]$, но тогаш треба да ставиме почетен услов $A[i,0] = \lambda$, каде λ е празниот стринг.

Да илустрираме како оваа рекурзија ќе работи на нашиот пример:

$$A[50,6] = 1 \circ A[18,5] = 11 \circ A[2,4] = 110 \circ A[2,3]$$
$$= 1100 \circ A[2,2] = 11000 \circ A[2,1] = 110001.$$

Повеќето од проблемите кои ќе ги разгледаме во овој дел се модификација на комбинаторните проблеми кои ги спомнавме порано во книгава. Целта е да ја покажеме врската меѓу двата типа на задачи кои се базираат на иста приказна. Имено, ако имаме проблем од типот за даден стринг да го определиме неговиот реден број, тогаш стратегијата е да изброиме колку стрингови има пред него. Ако пак ја имаме опцијата за дадена позиција да се определи кој елемент е позициониран на неа, тогаш множеството го

разделуваме во дисјунктни подредени класи и броиме во која класа припаѓа елементот кој не интересира.

Проблем 5. 1. Распоредување на низи од парички

Првиот проблем кој ќе го разгледаме е проблемот на парички кој го разгледавме во Пример 2.1, со тоа што сега се фокусираме на редоследот на подредување на комбинациите кои претставуваат плаќање на одредена сума со еден и два денари.

Дефинирање на проблемот

Пијалоците кои се порачуваат од автоматот за пијалоци, можат да се платат со железни парички, но само со монети од еден и два денари. Монетите се пуштаат во автоматот една по една. Начините на плаќање на сума од n денари ги подредуваме следејќи го следново правило:

- Од две низи со ист префикс кои прв пат се разликуваат во i-тиот број, низата на која i-тиот број е 2 е пред низата на која i-тиот број е 1. Под префикс се подразбира и празен стринг

На пример за сума од $n = 5$ денари редоследот на низите е следен: 221, 212, 2111, 122, 1211, 1121, 1112, 11111.

ПОТПРОБЛЕМ 1

За дадена пермутација од парички од еден и два денари на кои сумата им е n, да се пресмета нејзината позиција во подредената низа од сите пермутации од монети од еден и два денари со сума n.

ПОТПРОБЛЕМ 2

За дадени n и k, да се најде k-тата пермутација од парички од еден и два денари на кои сумата им е n.

Анализа на проблемот

Да се потсетиме на рекурзивната релација за број на пермутации, (Проблем 2.1):

$$A[n] = A[n-1] + A[n-2], n > 1;$$
$$A[0] = 1;$$
$$A[1] = 1.$$

Според нашето правило на подредување на пермутациите, $A[n-1]$ можеме да го гледаме како број на пермутации кои почнуваат со еден денар, додека $A[n-2]$ можеме да го гледаме како број на пермутации кои почнуваат со два денари. Ако нашата пермутација започнува со 2, тогаш таа е во оние $A[n-2]$ пермутации кои започнуваат со 2, а тоа се пермутациите кои се јавуваат порано од оние кои започнуваат со 1. Од друга страна, ова е бројот на пермутации на кои сумата на елементите е $n-2$. Ако пак нашата пермутација започнува со 1, таа е во оние $A[n-1]$ пермутации кои започнуваат со 1, што е всушност бројот на пермутации на кои сумата на елементите е $n-1$.

ПОТПРОБЛЕМ 1

За да го решиме проблемот на која позиција е дадена пермутација, всушност треба да изброиме колку пермутации се наоѓаат пред неа. На пример нека ни е дадена пермутацијата 1211.

- Бидејќи оваа пермутација започнува со 1, пред неа се сите пермутации кои започнуваат со 2, а тоа се $A[3] = 3$.
- Наредната монета е 2, па немаме низи кои би имале друга монета на ова место преоѓаме на наредната монета.
- Третата цифра е 1, па пред нашата пермутација се сите кои започнуваат со 122, а тоа се $A[0] = 1$

Оттука, пред нашата пермутација има 3 + 1 = 4 пермутации, па нашата е петтата.

Сега можеме да го изведеме правилото:

- Ако првата цифра на пермутацијата од 1-ци и 2-ки на кои сумата е n е 1, т.е. е од облик 1α, тогаш пред неа се сите пермутации на кои сумата е n и започнуваат со 2, што се

$A[n-2]$ пермутации. Останатите пермутации кои се пред неа, се оние кои се пред α, на кои сумата им е $n-1$.

- Ако првата цифра на пермутацијата од 1-ци и 2-ки на кои сумата е n е 2, т.е. е од облик 2α, тогаш пред неа се пермутациите кои се пред α на кои сумата им е $n-2$.

Нека $C[i,\alpha]$ го претставува редниот број на пермутацијата од 1-ци и 2-ки на кои сумата им е i. Секако α е таква пермутација. Тогаш,

$$C[i,\alpha]=\begin{cases}1, & i=1 \text{ или } i=0\\ A[i-2]+C[i-1,\alpha'], & i>1, \alpha=1\alpha'\\ C[i-2,\alpha'] & i>1, \alpha=2\alpha'\end{cases}.$$

Оваа рекурзија за нашиот пример би работела на следниов начин:

$$C[5,1211]=A[3]+C[4,211]=3+C[2,11]$$
$$=3+A[0]+C[1,1]=5.$$

Почетниот дел од псевдокодот П 5. 1. за решавање на овој проблем ги вклучува и пресметките за бројот на низи од единици и двојки на кои сумата на елементите им е n. Редниот број на бараната пермутација се зачувува во рекурзивната функција $C[n,niza]$.

П 5. 1. РЕДЕН БРОЈ НА НИЗА ОД ЕДИНИЦИ И ДВОЈКИ

```
1  Внеси го n;
2  A[0] = A[1] = 1;
3  за i = 2 до n прави A[i] = A[i − 1] + A[i − 2]
4  C[n, niza];
5  функција C[i, α];
6      ако i = 0 или i = 1 тогаш врати 0 инаку
7          {
8            ако α = 1α′ тогаш врати A[i − 2] + C[i − 1, α′]
9                  инаку врати C[i − 2, α′].
```

```
10          }
```

Рекурзивната функција се повикува од сумата на броевите во стрингот и од самиот стринг и при секое повикување стрингот се намалува за еден карактер. Тоа значи дека, при пресметана функција A, $C[i,\alpha]$ ќе се повикува $O(n)$ пати. Сепак, ако стрингот е долг, за негово препишување без првите карактери ќе треба време $O(n)$, па ако истиот се препишува времето може да се зголеми до $O(n^2)$. Имплементацијата може да се направи така да параметарот стринг ќе се замени со позицијата од која истиот го повикуваме. Така, ако веќе сме ги поминале првите два карактери за низата 1211, наместо да се повика $C[2,11]$ може да се повика $C[2,3]$, како што е имплементирано во програмата во C++ дадена подолу. Бидејќи за пресметување на функцијата A е потребно време $O(n)$, целиот алгоритам има линеарна сложеност.

C++ 5. 1 РЕДЕН БРОЈ НА НИЗА ОД ЕДИНИЦИ И ДВОЈКИ

```
1  #include<iostream>
2  #include <string>
3  using namespace std;
4
5  int* A;
7  int C(int i, string niza) {
8     if(i==0 || i==1)
9        return 1;
10
11    if(niza[0] == '1')
12    {
13       int p = C(i-1, niza.substr(1));
14       return A[i-2] + p;
15    }
16    else
17    {
18       int p = C(i-2, niza.substr(1));
19       return p;
20    }
21 }
22
```

```
23 int main()
24 {
25    int n;
26    cin >> n;
27    string niza;
28    cin >> niza;
30    A = new int[n+1];
31    A[0] = A[1] = 1;
32    for(int i=2;i<=n;i++)
33    {
34       A[i] = A[i-1] + A[i-2];
35    }
36
37    cout << C(n, niza);
38    return 0;
39 }
```

Програмата во Јава го препишува целиот стринг и можете да проверете дека ако A е препресметана претходно, тогаш таа ќе работи побавно.

JAVA 5. 1 РЕДЕН БРОЈ НА НИЗА ОД ЕДИНИЦИ И ДВОЈКИ

```
1 import java.util.*;
2
3 public class Main {
4     public static int[] A;
5
6     public static int C(int i, String niza) {
7         if (i == 0 || i == 1)
8             return 1;
9
10        if (niza.charAt(0) == '1') {
11            int p = C(i - 1, niza.substring(1));
12            return A[i - 2] + p;
13        } else {
14            int p = C(i - 2, niza.substring(1));
15            return p;
16        }
17    }
18
19    public static void main(String[] args) {
```

```
20          Scanner scn = new Scanner(System.in);
21          int n = scn.nextInt();
22          String niza = scn.next();
23
24          A = new int[n + 1];
25          A[0] = A[1] = 1;
26          for (int i = 2; i <= n; i++) {
27              A[i] = A[i - 1] + A[i - 2];
28          }
29
30          System.out.println(C(n, niza));
31      }
32}
```

ПОТПРОБЛЕМ 2

За да определиме која пермутација е на дадена позиција, прво треба да видиме со која цифра почнува пермутацијата, затоа што знаеме колку пермутации почнуваат со 2, а колку со 1. Ако редниот број на нашата пермутација е помал од вкупниот број на пермутации кои почнуваат со 2, тогаш таа почнува со 2, во спротивно, почнува со 1. И во двата случаи, откога ќе определиме кој број се наоѓа на првата позиција ќе можеме да го намалиме проблемот за една позиција.

Пред да ја напишеме формалната рекурзија, ќе ја објасниме постапката на пример. Нека сакаме да ја најдеме 5-тата пермутација од парички од еден и два денари на кои сумата им е 5. Знаеме дека $A[0] = A[1] = 1, A[2] = 2, A[3] = 3, A[4] = 5$.

- Прво, пермутации со сума 5 кои почнуваат со 2 има $A[3] = 3$. Ја бараме пермутацијата на петтата позиција, па таа не е во тие што почнуваат со 2, значи почнува со 1. Од пермутациите што почнуваат со 1, оваа е 5 – 3 = 2-ра по ред. Значи, за да ја определиме наредната паричка треба да ја најдеме втората пермутација по ред за парички со вкупна сума 4.
- Од 5-те пермутации со сума 4, со бројот 2 почнуваат првите $A[2] = 2$, и бидејќи нашата е 2-та, таа е во овие две. Значи наредна цифра во неа е 2. Останува да се наредат парички со вкупна сума $4 - 2 = 2$ денари, и ни треба 2-та од нив.

- Пермутации за сума на парички 2 кои почнуваат со 2 има $A[0] = 1$, а нам ни треба втората, значи овие пермутации се пред нашата. Оттука наредната цифра е 1, а останува да ја најдеме 2 – 1 = 1-вата пермутација за сума на парички 1. Тоа е пермутацијата 1. Оттука елементот кој го бараме е 1211.

На крај да ја определиме рекурзивната равенка на која ќе го темелиме алгоритамот. Нека со $B[i,j]$ ја обележиме j -тата пермутација за сума на парички i. Тогаш:

$$B[i,j] = \begin{cases} i & i = 1,2, j = 1 \\ 1 \circ B[i-1, j - A(i-2)], & i > 1, j > A[i-2]. \\ 2 \circ B[i-2, j] & i > 1, j \leq A[i-2] \end{cases}$$

За нашиот пример рекурзијата работи на следниов начин:

$$B[5,5] = 1B\big[4,5 - A[3]\big] = 1B[4,5-3] = 1B[4,2] = 12B[2,2]$$
$$= 121B[1,2 - A[0]] = 121B[1,2-1] = 121B[1,1] = 1211.$$

Псевдокодот е П 5. 2.:

П 5. 2. НИЗА ОД ЕДИНИЦИ И ДВОЈКИ СО ДАДЕН РЕДЕН БРОЈ

```
1   Внеси ги n и k;
2   A[0] = A[1] = 1;
3   за i = 2 до n прави A[i] = A[i − 1] + A[i − 2];
4   B[n, k];
5   функција B[i, j];
6       ако j = 1 тогаш
7           ако i = 1 тогаш врати 1 инаку врати 2;
8               инаку
9                   ако j ≤ A[i − 2] тогаш
10                      {
11                       печати "2";
12                       B[i − 2, j];
```

```
13                  }
14                  инаку
15                      {
16                        печати"1"
17                        B[i − 1, j − A(i − 2)]
18                      }
```

Кодот во Јава е следниов.

ЈАVA 5. 2 НИЗА ОД ЕДИНИЦИ И ДВОЈКИ СО ДАДЕН РЕДЕН БРОЈ

```
1  import java.util.*;
2
3  public class Main {
4      public static int[] A;
5
6      public static void B(int i, int j) {
7          if (j == 1) {
8              if (i == 1)
9                  System.out.print("1");
10             else
11                 System.out.print("2");
12         } else {
13             if (j <= A[i - 2]) {
14                 System.out.print("2");
15                 B(i - 2, j);
16             } else {
17                 System.out.print("1");
18                 B(i - 1, j - A[i - 2]);
19             }
20         }
21     }
22
23     public static void main(String[] args) {
24         Scanner scn = new Scanner(System.in);
25         int n = scn.nextInt();
26         int k = scn.nextInt();
27
28         A = new int[n + 1];
29         A[0] = A[1] = 1;
30         for (int i = 2; i <= n; i++) {
```

```
31              A[i] = A[i - 1] + A[i - 2];
32          }
33
34          B(n, k);
35      }
36}
```

Сложеноста и на овој алгоритам, освен од $B[i,j]$, зависи и од функцијата $A[i]$. Бидејќи, од една страна $A[i]$ се пресметува во време $O(n)$, а од друга страна за пресметка на $B[i,j]$ во лините од 4 до 18 повторно е потребно време $O(n)$, целата сложеност на алгоритамот е $O(n)$. Ова е сложеноста на алгоритмите во Јава, JAVA 5. 2 и C++, C++ 5. 2.

C++ 5. 2 РАСПОРЕДУВАЊЕ НА НИЗИ ОД ПАРИЧКИ

```
1  #include<iostream>
2  #include <string>
3  using namespace std;
4
5  int* A;
6
7  void B(int i, int j)
8  {
9     if(j==1)
10    {
11       if(i==1)
12          cout << "1";
13       else
14          cout << "2";
15    }
16    else
17    {
18       if(j<=A[i-2])
19       {
20          cout << "2";
21          B(i-2,j);
22       }
23       else
24       {
25          cout << "1";
26          B(i-1, j-A[i-2]);
```

```
27        }
28     }
29 }
30
31 int main()
32 {
33     int n;
34     int k;
35
36     cin >> n >> k;
37
38     A = new int[n+1];
39     A[0] = A[1] = 1;
40     for(int i=2;i<=n;i++)
41     {
42        A[i] = A[i-1] + A[i-2];
43     }
44
45     B(n, k);
46     return 0;
47 }
```

Прашања и задачи

1. Како треба да се преправат алгоритмите од овој пример ако редоследот на пермутациите се наредени по лексикографски редослед, при што единицата е пред двојката?

   На пример, за $n = 5$, редоследот е: 11111, 1112, 1121, 1211, 122, 2111, 212, 222. Да се дадат рекурзивните функции за

   a. Наоѓање на k-тиот број во S!

   b. Определување колку броеви во S се помали или еднакви од даден број од S.

2. Нека S е множеството на n-цифрени броеви кои можат да се формираат само од цифрите 0, 1 и 2, такви што сумата на сите цифри е n, а нулите се јавуваат само на крајот од бројот. Тие се подредени со стандардното подредување на природни броеви. Како треба да се преправи решението на претходната задача за да се реши овој проблем?

3. Нека S е множеството на битстрингови со должина n кои немаат две последователни нули. Подредувањето на броевите е по големина на бројот кој го претставуваат. На пример, за $n = 3$ подредувањето е: 010, 011, 101, 110, 111.

   a. Да се најде k-тиот број во S!

   b. За даден број од S да се определи колку броеви во S се помали или еднакви на него.

   c. Што е разликата меѓу решението на овој проблем и проблемот на битстрингови со должина n кои немат две последователни единици?

Проблем 5. 2. Реден број на битстринг со константен број на единици

Вториот проблем кој ќе го разгледаме е верзија на проблемот за број на патишта во матрица која нема препреки, кој овде ќе биде дефиниран со битови. Имено ако движењето надесно го обележиме со 1, а надолу со 0, тогаш ќе добиеме бит низа на која бројот на единици е еднаков на бројот на колони, а бројот на нули е еднаков на бројот на редици. Ваквите бит-низи можеме да ги подредиме врз основа на подредувањето на ненегативните цели броеви.

Дефинирање на проблемот

За дадени природни броеви n и m, $n > m$, да се формираат битови низи со должина n, кои имаат точно m единици.

ПОТПРОБЛЕМ 1

За дадено $k \leq \binom{n}{m}$, да се најде k-тиот елемент од оваа низа.

ПОТПРОБЛЕМ 2

За даден елемент од низата да се пресмета неговата позиција во низата.

На пример, ако $n = 4$, а $m = 2$ низите кои се бараат се подредени по лексикографски редослед се: 0011, 0101, 0110, 1001, 1010, 1100. Значи ако ни се бара 3-тата низа, треба да се отпечати 0110, а ако ни се бара која е позицијата на 1010, треба да се отпечати 5.

Анализа на проблемот

Редниот број k мора да е помал од $\binom{n}{m}$, затоа што вкупниот број на низи со должина n и точно m единици е $\binom{n}{m}$. Во анализата на решението можеме да го искористиме овој факт, но како што напоменавме порано, во првиот дел од првата глава,

пресметувањето на бројот на комбинации е секогаш подобро да се прави користејќи го Паскаловиот триаголник, заради тоа што $n!$ може да биде многу голем број и да не може да се зачува во алоцираната меморија за еден број. Затоа во нашата анализа ќе ја користиме рекурзивната релација за број на комбинации, која овде пак ќе ја изведеме.

Нека со $A[i,j]$ го обележиме бројот на бинарни низи со должина i кои имаат точно j единици. Првиот бит може да биде или 0 или 1. Ако е 0, тогаш во остатокот од низата со должина $i-1$ се наоѓаат сите j единици, а ако е 1, тогаш во остатокот од низата со должина $i-1$ има една единица помалку, односно $j-1$ единици. Оттука

$$A[i,j] = A[i-1,j-1] + A[i-1,j].$$

Почетните вредности се: $A[i,0] = A[i,i] = 1$, затоа што тоа се низите со сите единици или сите нули.

i \ j	0	1	2	3
1	1	1		
2	1	2	1	
3	1	3	3	1
4	1	4	6	4
5	1	5	10	10

Слика 5. 1. Број на бинарни низи со должина i со точно j единици $A[i,j]$.

Во вакви случаи е подобро да се оди со мемоизација за да не се пресметуваат сите вредности, бидејќи некои од нив може да не се појават во пресметките, но ние овде ќе претпоставиме дека $A[i,j]$ ни е пресметана. Алгоритамот ќе го опишеме на примерот со $n=5$ и $m=3$. На Слика 5. 1. се дадени вредностите на $A[i,j]$ потребни во пресметките, кои понатаму ќе ги користиме како готови. Да забележиме од табелата дека вкупниот број на низи кои ги имаме се 10.

ПОТПРОБЛЕМ 1

Ако треба да ја најдеме низата на позиција 6, првото прашање е дали таа почнува со 1 или со 0.

- Низи со должина 5 со 3 единици кои почнуваат со 0 се $A[4,3] = 4$, па сите тие се пред низата која ја бараме, што значи дека таа низа ќе започнува со 1. Уштеповеќе, тоа ќе биде втората низа која започнува со 1, затоа што 6 – 4 = 2. Во овој момент ќе запишеме 1-ца за почетната позиција.

- Сега ни треба втората низа со должина 4 со 2 единици. Низи со должина 4 со 2 единици кои почнуваат со 0 се $A[3,2] = 3$, па низата која ја бараме е една од нив, што значи дека наредниот бит во неа ќе биде 0.

- Продолжуваме да ја бараме втората низа со должина 3 со 2 единици. Имаме $A[2,2] = 1$ низа со должина 3 со 2 единици која започнува со 0, и бидејќи 1 < 2 таа низа е пред низата која треба да ја реконструираме, што значи дека наредниот бит во неа ќе биде 1, и ова го запишуваме. Уштеповеќе, нашата низа ќе биде првата која на оваа позиција има 1, затоа што 2 – 1 = 1.

- Во наредниот чекор ја бараме првата низа со должина 2 со една единица. Вакви низи што започнуваат со 0 се $A[2,1] = 2$, па низата која треба да ја реконструираме е една од нив. Затоа овде запишуваме 0.

- На крај ја бараме првата низа со должина 1 со една единица. Низи со должина 1 со една единица кои почнуваат со 0 нема, па овде запишуваме 1.

Значи низата која ја бараме е 10101.

За да го изведеме рекурзивно решение дефинираме функција $B[i,j,r]$ која ќе го генерира првиот бит од на r-тата низа од битови од множеството низи со должина i кои имаат точно j единици:

$$B[i,j,r] = \begin{cases} 0, & i = 1, j = 0, k = 1 \\ 1, & i = 1, j = 1, k = 1 \\ 1 \circ B[i-1, j-1, r - A[i-1,j]], & r > A[i-1,j] \\ 0 \circ B[i-1,j,r], & r \leq A[i-1,j] \end{cases}.$$

$B[i,j,r]$ не постои за вредности на i,j и k кои не се опфатени со горната рекурзија. За нашиот пример пресметката на $B[n,m,k]$ е:

$$B[5,3,6] = 1B[4,2,2] = 10B[3,2,2]$$
$$= 101B[2,1,1] = 1010B[1,1,1] = 10101.$$

Да видиме дека за пресметките не ни се потребни сите вредности од табелата за А, на пример за да го пресметаме

$$A[4,3] = A[3,3] + A[3,2] = A[3,3] + A[2,2] + A[2,1]$$
$$= A[3,3] + A[2,2] + A[1,1] + A[1,0].$$

При пресметка на оваа вредност ќе ги пресметаме и сите останати $A[i,j]$ кои ќе ни бидат потребни во другите пресметки во рекурзијата, па гледаме дека вредностите $A[3,0]$, $A[2,0]$, $A[4,1]$, $A[4,2]$, $A[4,4]$ не ни се потребни.

Псевдокодот за овој алгоритам е следниот:

П 5. 3. БИТСТРИНГ СО КОНСТАНТЕН БРОЈ НА ЕДИНИЦИ, СО ДАДЕН РЕДЕН БРОЈ

```
1   Внеси n, m, k;
2   B[n, m, k];
3   функција A[i, j];
4       ако i = j или j = 0 тогаш A[i, j] = 1 инаку
5           A[i, j] = A[i − 1, j] + A[i − 1, j − 1];
6   функција B[i, j, r];
7       ако i = j = r = 1 тогаш печати 1 инаку
8           ако i = r = 1 и j = 0 тогаш печати 0 инаку
9               ако r > A[i − 1, j] тогаш
10                  {
11                  печати 1;
12                  B[i − 1, j − 1, r − A[i − 1, j]];
13                  }
14              инаку
15                  {
```

```
16                печати 0;
17                B[i − 1, j, r];
18              }
```

Сложеноста на алгоритамот зависи од пресметувањето на функциите $A[i,j]$ и $B[i,j,k]$. Временската сложеност за пресметување на функцијата $A[i,j]$ е $O(n^2)$. Имено иако во понатамошниот дел од кодот е можено да не ни се потребни сите вредности на оваа функција, бидејќи барем еднаш треба да се пресмета $A[m,n]$, за нејзно пресметување ќе имаме $O(n^2)$ операции. Од друга страна за $B[i,j,k]$ не се повикува за сите вредности на i,j и k и при секое повикување на оваа функција сите овие параметри се намалуваат, па таа ќе се самоповика налмногу n пати. Оттука за делот во линиите 6-18 во псевдокодот е потребно време $O(n)$. Оттука, сложеноста на целиот код зависи од сложеноста за пресметка на $A[m,n]$, а тоа е $O(n^2)$.

Следуваат кодовите во Јава и C++. Во кодот во јава даваме коментари за повеќето делови, додека во кодот во C++ истите се изоставени.

JAVA 5. 3 БИТСТРИНГ СО КОНСТАНТЕН БРОЈ НА ЕДИНИЦИ, СО ДАДЕН РЕДЕН БРОЈ

```
1  import java.util.*;
2
3  public class Main {
4     public static int[][] Avalues;
5
6     public static int A(int i, int j) {
7        if (Avalues[i][j] > 0)
8           return Avalues[i][j];
9
10       if (j == 0 || i == j)
11          Avalues[i][j] = 1;
12       else
13          Avalues[i][j] = A(i - 1, j - 1) + A(i - 1, j);
14
15       return Avalues[i][j];
16    }
17
18    public static void B(int i, int j, int r) {
19       if (i == j && j == r && r == 1)
20          System.out.print("1");
21       else if (i == r && r == 1 && j == 0)
```

```
22            System.out.print("0");
23         else if (r > A(i - 1, j)) {
24            System.out.print("1");
25            B(i - 1, j - 1, r - A(i - 1, j));
26         } else {
27            System.out.print("0");
28            B(i - 1, j, r);
29         }
30      }
31
32      public static void main(String[] args) {
33         Scanner scn = new Scanner(System.in);
34         int n = scn.nextInt();
35         int m = scn.nextInt();
36         int k = scn.nextInt();
37
38         Avalues = new int[n + 1][m + 1];
39         B(n, m, k);
40      }
41   }
```

Кодот во C++, суштински не се разликува од кодот во јава.

C++ 5. 3 БИТСТРИНГ СО КОНСТАНТЕН БРОЈ НА ЕДИНИЦИ, СО ДАДЕН РЕДЕН БРОЈ

```
1  #include<iostream>
2  #include <vector>
3  using namespace std;
4
5  vector<vector<int>> Avalues;
6
7  int A(int i, int j) {
8     if (Avalues[i][j] > 0)
9        return Avalues[i][j];
10
11    if (j == 0 || i == j)
12       Avalues[i][j] = 1;
13    else
14       Avalues[i][j] = A(i - 1, j) + A(i - 1, j - 1);
15
16    return Avalues[i][j];
17 }
18
19 void B(int i, int j, int r) {
20    if (i == j && j == r && r == 1)
```

```
21        cout << "1";
22     else if (i == r && r == 1 && j == 0)
23        cout << "0";
24     else if (r > A(i - 1, j)) {
25        cout << "1";
26        B(i - 1, j - 1, r - A(i - 1, j));
27     } else {
28        cout << "0";
29        B(i - 1, j, r);
30     }
31 }
32
33 int main() {
34     int n, m, k;
35     cin >>n >> m >> k;
36
37     Avalues.resize(n+1, vector<int>(m+1, 0));
38     B(n, m, k);
39
40     return 0;
41 }
```

ПОТПРОБЛЕМ 2

Ако треба да ја најдеме позицијата на низата 10101, треба да изброиме колку од низите се пред неа.

- Бидејќи низата почнува со 1, пред неа се сите низи со должина 5 кои почнуваат со 0, а тоа се $A[4,3] = 4$.
- Пред неа се и сите низи со должина 5 кои почнуваат со 100, што е еднакво на сите низи со должина 2 со две единици, а тоа се $A[2,2] = 1$.
- Пред неа се и сите низи со должина 5 кои почнуваат со 10100, што не е можно затоа што оваа низа е со должина 5, а има само две единици.

Значи пред нашата низа има 5 низи, па таа е 6-тата во редицата. Рекурзијата ќе ја дефинираме на следниов начин. Нека со $C(i,j,\alpha)$ го означиме редниот број на α, каде α е стринг од битови, со должина i во која има точно j единици.

Тогаш:

$$C[i,j,\alpha]=\begin{cases}0, & i=1,j=1,\alpha=0\\ 1, & i=1,j=1,\alpha=1\\ A[i-1,j]+C[i-1,j-1,\alpha'], & \alpha=1\alpha'\\ C[i-1,j,\alpha'], & \alpha=0\alpha'\end{cases}.$$

Од оваа формула $C[5,3,10101]$ е:

$$\begin{aligned}C[5,3,10101]&=A[4,3]+C[4,2,0101]=4+C[3,2,101]\\&=4+A[2,2]+C[2,1,01]=4+1+C[1,1,1]\\&=5+1=6.\end{aligned}$$

Во псевдокодот на овој алгоритам, нема да ја препишеме **функција** $A[i,j]$, затоа што тоа е истата рекурзивна функција од претходно:

П 5. 4. РЕДЕН БРОЈ НА ДАДЕН БИТСТРИНГ СО КОНСТАНТЕН БРОЈ НА ЕДИНИЦИ

```
1  Внеси n, m, bitniza;
2  C[n, m, bitniza];
3  функција C[i, j, α];
4      ако i = j = 1 и α = 0 тогаш врати 0 инаку
5          ако i = j = 1 и α = 1 тогаш врати 1 инаку
6              ако α = 1α' тогаш
7                  врати A[i − 1, j] + C[i − 1, j − 1, α'];
8                      инаку врати C[i − 1, j, α'].
```

Сложеноста и на овој алгоритми не зависи од функцијата $C[i,j,\alpha]$, туку од $A[i,j]$, затоа што ако $A[i,j]$ е прекалкулирана, тогаш

$C[i,j,\alpha]$ може да се направи да работи во линеарно време. Имено, во секој чекор од пресметката на $C[i,j,\alpha]$ првиот параметар се намалува за еден, но повторно треба да се има предвид дека ако стриктно го следиме псевдокодот, може да се случи да го препишуваме стрингот постојано и таа операција препишување да ни заземе линеарно време при секое повикување на функцијата. Затоа, како што коментиравме во претходниот пример, наместо препишување на подстрингот можеме само да ја памтиме неговата почетна позиција во иницијалниот стринг.

Најлошиот случај за $A[i,j]$ е да се пресметаат сите потребни вредности за n и m. Иако со мемоизација може многу да се намали, теориски не може да се покаже дека сложеноста е помала од $O(nm)$. Затоа, сложеноста на целиот алгоритам е $O(nm)$.

□

Кодовите во Јава и C++ кои ги даваме овде се итеративни решенија кои се базираат на третиот пристап.

JAVA 5. 4 РЕДЕН БРОЈ НА ДАДЕН БИТСТРИНГ СО КОНСТАНТЕН БРОЈ НА ЕДИНИЦИ

```
1 import java.util.*;
2
3 public class Main {
4     public static int[][] Avalues;
5
6     public static int A(int i, int j) {
7         if (Avalues[i][j] > 0)
8             return Avalues[i][j];
9
10        if (j == 0 || i == j)
11            Avalues[i][j] = 1;
12        else
13            Avalues[i][j] = A(i - 1, j - 1)+ A(i - 1, j);
14
15        return Avalues[i][j];
16    }
17
18    public static int C(int i, int j, String niza) {
19        if (i == j && j == 1 && niza.compareTo("0") == 0)
20            return 0;
```

```
21          else if (i == j && j == 1 && niza.compareTo("1") ==
0)
22              return 1;
23          else if (niza.charAt(0) == '1')
24          {
25              return A(i - 1, j) + C(i - 1, j - 1,
niza.substring(1));
26          }
27          else
28              return C(i - 1, j, niza.substring(1));
29      }
30
31      public static void main(String[] args) {
32          Scanner scn = new Scanner(System.in);
33          int n = scn.nextInt();
34          int m = scn.nextInt();
35          String bitniza = scn.next();
36
37          Avalues = new int[n + 1][m + 1];
38
39          System.out.println(C(n, m, bitniza));
40      }
41}
```

Во решението во C++ функцијата B не се повикува од стринг, туку од неговата почетна позиција како подстринг на почетниот стринг, но тоа не го менува времето на извршување значајно, затоа што целиот алгоритам има квадратна сложеност заради делот за пресметување на функцијата A.

C++ 5. 4 РЕДЕН БРОЈ НА ДАДЕН БИТСТРИНГ СО КОНСТАНТЕН БРОЈ НА ЕДИНИЦИ

```
1  #include<iostream>
2  #include <vector>
3  #include <string>
4  using namespace std;
5
6  vector<vector<int>> Avalues;
7
8  int A(int i, int j) {
```

```
9     if (Avalues[i][j] > 0)
10       return Avalues[i][j];
11
12    if (j == 0 || i == j)
13       Avalues[i][j] = 1;
14    else
15       Avalues[i][j] = A(i - 1, j - 1)+ A(i - 1, j);
16
17    return Avalues[i][j];
18 }
19
20 int C(int i, int j, string niza) {
21    if (i == j && j == 1 && niza == "0")
22       return 0;
23    else if (i == j && j == 1 && niza == "1")
24       return 1;
25    else if (niza[0] == '1')
26    {
27       return A(i - 1, j) + C(i - 1, j - 1, niza.substr(1));
28    }
29    else
30       return C(i - 1, j, niza.substr(1));
31 }
32
33 int main() {
34    int n, m;
35    cin >> n >> m;
36    string bitniza;
37    cin >> bitniza;
38
39    Avalues.resize(n+1, vector<int>(m+1));
40
41    cout << C(n, m, bitniza);
42
43    return 0;
44 }
```

Прашања и задачи

1. Која ќе биде сложеноста на алгоритамот за решавање на следнава задача:
   За дадена низа $k_1, k_2, \dots, k_r$, $k_i \leq \binom{n}{m}$, да се отпечатат k_i-тите елемент од оваа низата дефинирана во овој проблем.

2. Бидејќи секој бит стринг со должина n кој има точно m единици одговара на една комбинација од m елементи од множество со n елементи, кодот за генерирање на r-тиот битстринг со мала преправка може да се преправи во код за генерирање на r-тата пермутација, ако пермутациите се подредени така што множество со помали елементи се јавува порано во подредувањето. Слично и кодот за определување на редниот број на даден битстринг може да се преправи во код за определување на редниот број на даден пермутација. Дади ги псевдокодовите на овој проблем и решенијата во програмските јазици C++ и Јава!

3. Нека S е множеството на сите n-пермутации со повторување со должина m, кои се подредени со стандардното подредување на природни броеви. Да се дадат рекурзивните равенки за:

   a. Наоѓање на k-тиот број во S!

   b. Редниот број на дадена пермутација од S.

   c. Која ќе биде сложеноста на решението со динамичко програмирање?

3. Нека S_n е множеството n-пермутации со повторување со должина n, $x_n x_{n-1} \dots x_1$ такви што за секое i, $x_i \leq i$, подредено со стандардното лексикографско подредување. На пример за $n = 3, S = \{111,121,211,221,311,321\}$. Да се дадат рекурзивните равенки за:

   a. Бројот на елементи во S_i.

   b. Редниот број на дадена пермутација од S.

c. Наоѓање на k-тиот број во S!

d. Која ќе биде сложеноста на решението со динамичко програмирање за двата алгоритми?!

4. Нека S е множеството n-пермутации без повторување на првите n елементи, со стандардното лексикографско подредување.

a. Да се покаже дека секоја пермутација од задача 3 одговара на една пермутација без повторување!

b. Како може да се адаптира алгоритамот од задача 3 за да го реши овој проблем, за наоѓање на k-тиот број во S и редниот број на дадена пермутација од S?

c. Анализирај ја сложеноста на алгоритмите со динамичко програмирање за двата алгоритми!

Проблем 5. 3. Лексикографско правилно поставување на загради

Наредниот пример е повторно познатиот проблем на правилно поставување на загради, но овде ќе не интересира редниот број на такво поставување на загради. Проблемот е често поставуван како задача на натпревари [16], а исто така можат да се сретнат задачи кои се сведуваат на овој проблем или се слични на него. Можеме да дефинираме повеќе начини на подредување на правилно распределени загради, но во нашиот проблем ќе го разгледаме случајот кога тие се подредени лексикографски.

Дефинирање на проблемот

Стринговите од правилно поставени мали загради се состојат само од два карактери: „(“ и „)“. Сметаме дека подредената азбука е Σ={(,)}. Според тоа лексикографското подредување на 3 пара загради е следново: ((())), (()()), (())(), ()(()), ()()().

ПОТПРОБЛЕМ 1

За дадени природни броеви n и k, да се најде k-тата лексикографски подредена пермутација од n правилно поставени парови на загради.

ПОТПРОБЛЕМ 2

За дадена пермутација од n правилно поствени парови загради, да се определи која по ред е во лексикографското подредување.

Анализа на проблемот

Претходно разгледавме три алгоритми кои го пресметуваат бројот на начини на правилно поставување на загради, но не секој од нив одговара на броењето на бројот на пермутации пред некоја дадена пермутација. Имено, овде е битно дека колку порано се јавува „)“, толку поназад е таквата пермутација или, колку порано ги наредиме

отворените загради, толку таквата пермутација е понапред. Оттука следува дека можеме да го искористиме третиот начин на броење на заградите, затоа што таа рекурзија води сметка за местоположбата на заградите однапред наназад. Вториот начин објаснет во Проблемот 3.3 за правилно распределување парови на загради не може да се искористи, а првиот начин може да се искористи, но модифициран, наместо да се гледа последната отворена заграда, да се води сметка за првата затворена заграда. Анализата околу тоа која рекурзивна равенка може да се искористи и каква рекурзивна равенка треба да бараме ќе зборуваме на крај од анализата на овој проблем.

Нашето решение овде ќе се базира на рекурзијата објаснета во третиот начин на броење на добро поставени парови загради, која беше следнава: Ако $A[i,j]$ е бројот на начини на кои можат да се допишат i целосни парови загради и j затворени загради, тогаш

$$A[i,j] = \begin{cases} 1, & i = 0 \\ A[i-1,1], & j = 0 \\ A[i-1,j+1] + A[i,j-1], & i > 0, j > 0 \end{cases}.$$

Во формулата $A[i-1,j+1]$ е бројот на начини во кои наредна заграда е „(“, додека $A[i,j-1]$ е бројот на начини во кои наредна заграда е „)“. Секое поставување што почнува со „(“ е пред она кое почнува со „)“. Во понатамошната анализа ќе сметаме дека $A[i,j]$ ни е пресметано со динамичко програмирање за сите вредности за кои ќе ни биде потребно.

ПОТПРОБЛЕМ 1

Постапката за определување на пермутацијата која се наоѓа на дадена позиција прво ќе ја објасниме на пример, па потоа ќе ја генерализираме. Нека треба да ја определиме 2-рата пермутација од 3 парови загради. Јасно, таа почнува со „(“.

- Во првиот чекор потребно е да определиме дали следен карактер кој треба да се стави е „(“ или „)“. Битно е да се забележи дека сите пермутации кои почнуваат со „((“ се пред било која што почнува со „()“. Ако со „((“ почнуваат повеќе или

еднакво на 2, тогаш нашата пермутација почнува со „((“, во спротивно нашата пермутација почнува со „()“. Затоа броиме колку пермутации почнуваат со „((“, а тоа се $A[1,2] = 3$. Оттука, пермутацијата која ја бараме почнува со „((“.

- Слично, за да го определиме наредниот карактер треба да го пресметаме бројот на пермутации кои почнуваат со „(((“, а тоа се $A[0,3] = 1$. Нам ни треба 2-та пермутација, па таа не може да започнува вака, т.е. започнува со „(()“. Уште повеќе нам ни треба 2 – 1 =1-тата пермутација која започнува „(()“.

- Сега гледаме колку пермутации се такви кои почнуваат со „(()(“, а тоа се $A[0,2] = 1$, па нашата пермутација започнува вака. Бидејќи до кај нема други опции отколку да се изнаредат затворени загради, пермутацијата која ја бараме е (()()).

Во општ случај, нека до некаде сме ги наредиле заградите да одговараат на пермутацијата која ја бараме и нека знаеме дека таа е k-та по ред со тој почеток. Нека со $B[i,j,k]$ го обележиме стрингот на кој треба да се постават i правилно поставени загради и уште j затворени загради, а таа низа по лексикографски редослед е на позиција k. Тогаш, со горната формула броиме колку пермутации ќе имаме ако како наредна заграда ставиме „(. Такви пермутации се $A[i-1, j+1]$.

- Ако тој број е помал или еднаков на k, нареден карактер е „(“. Бројот на целосни загради сега се намалува за еден, а на загради кои треба да се затворат се зголемува за еден и

$$B[i,j,k] = (B[i-1, j+1, k].$$

- Во спротивно, нареден карактер е „)“ и нашата пермутација ќе биде $(r - A[i-1, j+1])$-та по ред на која нареден карактер и е „)“. Сега бројот на целосни загради не се променува, но бидејќи сме затвориле една заграда, бројот а на загради кои треба да се затворат се намалува за еден и

$$B[i,j,k] =)B[i, j-1, k - A[i-1, j+1]].$$

Конечно можеме да ја дадеме целосната рекурзивна равенка:

$$B[i,j,k] = \begin{cases}), & i = 0, j = 1 \\ )B[0, j-1, 1], & i = 0, j > 1 \\ (B[i-1, 1, k], & j = 0 \\ (B[i-1, j+1, k], & k \le A[i-1, j+1] \\ )B[i, j-1, k - A[i-1, j+1]], & k > A[i-1, j+1] \end{cases}$$

- Работата на алгоритамот за пресметка на $B[3,0,2]$ се одвива на следниов начин:

$$B[3,0,2] = (B[2,1,2] = ((B[1,2,2] =$$
$$= (()B[1,1,1] = (()(B[0,2,1] = (()()).$$

Следуваат псевдокодот и кодовите во Java и C++.

П 5. 5. ОПРЕДЕЛУВАЊЕ НА НИЗА ОД ЗАГРАДИ, АКО Е ДАДЕН НЕЈЗИНИОТ РЕДЕН БРОЈ

```
1.  внеси ги n и k;
2.  за j од 1 до n прави A[0, j] = 1;
3.  за i од 1 до n прави
4.     {
5.      A[i, 0] = A[i − 1,1];
6.      за j од 1 до i прави
7.         A[i, j] = A[i − 1, j + 1] + A[i, j − 1] · A[n − k];
8.     }
9.  B[n, 0, k];
10. функција B[i, j, r];
11.    ако i =  0 тогаш печати „)“ j пати инаку
12.         ако j = 0 тогаш
13.              {
14.                печати „(“ ;
15.                врати B[i − 1,1, r]
16.               }
17.               инаку
```

```
18.          ако r ≤ A[i − 1, j + 1] тогаш
19.              {
20.                печати „(“ ;
21.                врати B[i − 1, j + 1, r]
22.              }
23.            инаку
24.              {
25.                печати „)“ ;
26.                врати B[i, j − 1, r − A[i − 1, j + 1]]
27.               }
```

JAVA 5. 5 ОПРЕДЕЛУВАЊЕ НА К-ТА НИЗА ОД ЗАГРАДИ

```
1 import java.util.*;
2 public class Main {
3     public static int[][] A;
4     public static void B(int i, int j, int r) {
5         if (i == 0) {
6             for (int k = 0; k < j; k++)
7                 System.out.print(")");
8         } else if (j == 0) {
9             System.out.print("(");
10            B(i - 1, 1, r);
11        } else if (r <= A[i - 1][j + 1]) {
12            System.out.print("(");
13            B(i - 1, j + 1, r);
14        } else {
15            System.out.print(")");
16            B(i, j - 1, r - A[i - 1][j + 1]);
17        }
18    }
19
20    public static void main(String[] args) {
21        Scanner scn = new Scanner(System.in);
22        int n = scn.nextInt();
23        int k = scn.nextInt();
24        A = new int[n + 1][n + 1];
25
26        for (int j = 1; j <= n; j++)
27            A[0][j] = 1;
28
29        for (int i = 1; i < n; i++) {
30            A[i][0] = A[i - 1][1];
31            for (int j = 1; j < n; j++) {
32                A[i][j] = A[i - 1][j + 1] + A[i][j - 1];
33            }
34        }
35        B(n, 0, k);
36    }
37}
```

```
1  #include<iostream>
2  #include <vector>
3  using namespace std;
4
5  vector<vector<int>> A;
6
7  void B(int i, int j, int r){
8     if(i == 0){
9        for(int k=0;k<j;k++)
10          cout << ")";
11    }
12    else if(j == 0){
13       cout << "(";
14       B(i-1, 1, r);
15    }
16    else if (r<=A[i-1][j+1]){
17       cout << "(";
18       B(i-1, j+1, r);
19    }
20    else{
21       cout << ")";
22       B(i, j-1, r-A[i-1][j+1]);
23    }
24 }
25
26 int main() {
27    int n, k;
28    cin >> n >> k;
29
30    A.resize(n+1, vector<int>(n+1));
31
32    for (int j=1;j<=n;j++)
33       A[0][j] = 1;
34
35    for(int i=1;i < n;i++){
36       A[i][0] = A[i-1][1];
37       for(int j=1; j< n;j++){
38          A[i][j] = A[i-1][j+1]+A[i][j-1];
39       }
40    }
```

```
41
42      B(n, 0 ,k);
43
44      return 0;
45  }
```

Сложеноста на алгоритамот повторно зависи од пресметката на функцијата $A[i,j]$ и функцијата $B[i,j,k]$. Во дел 3 каде што го разгледувавме проблемот за број на паравилно распределени загради видовме дека сложеноста на решението за пресметка на $A[i,j]$ е $O(n^2)$. Функцијата за пресметка на $B[i,j,k]$ е рекурзивна, но можеме да забележиме дека откога ќе се пресметаат вредностите $A[i,j]$, таа се рекурзивно се повикува себе си само по еднаш, па делот од кодот кој одговара на генерирањето на загради, во линиите од 9 до 27 има линеарна сложеност. Оттука, целата сложеност зависи само од функцијата $A[i,j]$ и е $O(n^2)$. Претходно кажавме дека каталановиот број може да се пресмета и линеарно, но ова не можеме да го искористиме за да ја подобриме ефикасноста на овој алгоритам, затоа што овде ни е потребен точно бројот $A[i,j]$.

ПОТПРОБЛЕМ 2

Процесот на определување на редниот број на дадена пермутација, да ја обележиме со α, е обратен, т.е. броиме колку пермутации има пред пермутацијата α. Тоа го правиме секогаш кога ќе дојдеме до „)“, затоа што сите пермутации со исти префикс како α кај кои првиот карактер во кој се разликуваат е таков што во α е „)“, а во нив е „(“, се наоѓаат пред α. На пример пред пермутацијата „(()())“ се сите пермутации со префикси „(((“ и „(()((“. Всушност, можеме да сметаме дека пред нашата пермутација е и пермутацијата „(())((“, но таква пермутација со 4 пара загради нема. Оттука, пред „(()())“ има $A[1,3] + A[0,3] = 4 + 1 = 5$ пермутации, па нејзиниот реден број е 6. Врз основа на гореизнесеното, алгоритамот треба да работи на тој начин што го прегледуваме стрингот кој го имаме карактер по карактер, и секогаш кога ќе наидеме на затворена заграда ја заменуваме со отворена и броиме колку начини на правилно распределени загради со тој префикс има. Сите овие броеви се собираат и редниот број на нашата пермутација е за еден поголем од

тој збир. Но со функцијата која ние ја дефинираме таа единица автоматски ќе се додава со последната затворена заграда.

Дефинираме $C[i,j,\alpha]$ да биде редниот број на суфиксот α кој е дел од правилни распореди на загради и во кој има i цели парови загради и j затворени загради. Тогаш:

$$C[i,j,\alpha] = \begin{cases} 1, & i = 0 \\ C[i-1, j+1, \alpha'], & \alpha = (\alpha'. \\ A[i-1, j+1] + C[i, j-1, \alpha'], & \alpha =)\alpha' \end{cases}$$

На почеток треба да го повикаме $C[n, 0, string]$, каде $string$ е стрингот на кој треба да му го определиме редниот број, а тој се состои од n парови загради. Псевдокодот е следниов:

П 5. 6. ОПРЕДЕЛУВАЊЕ РЕДЕН БРОЈ ЗА ДАДЕНА НИЗА ОД ЗАГРАДИ

1. Внеси n, $bitniza$;
2. $C[n, 0, string]$;
3. функција $C[i,j,\alpha]$;
4. ако $i = 0$ тогаш врати 1 инаку
5. ако $\alpha =)\alpha'$ тогаш врати $A(i-1, j+1) + C(i, j-1, \alpha')$;
6. инаку врати $C(i-1, j+1, \alpha')$.

Јасно е дека и овде сложеноста на алгоритамот пред се зависи од $A[i,j]$, затоа што ако $A[i,j]$ е прекалкулирана, тогаш $C[i,j,\alpha]$ ќе работи во линеарно време. Бидејќи за пресметка на $A[i,j]$ ни треба квадратно време, сложеноста на целиот алгоритам е $O(n^2)$.

Ако имаме проблем во кој треба да се одговори на повеќе прашања од типовите од проблем 1 или проблем 2 или комбинација од нив, на пример r прашања во кои се бара да се определи редниот број на различни r стрингови, тогаш $A[i,j]$ ќе се пресмета само еднаш, и сложеноста на алгоритамот ќе биде $O(nr + n^2)$ и ако $n > r$ ќе биде $O(nr)$.

На крај да анализираме какво својство треба да има една рекурзивна равенка која можеме да се искористиме за решавање на еден проблем од ваков тип, но посебно овој проблем, затоа што тој се решава со повеќе рекурзивни равенки. Имено, секоја рекурзивна равенка ги дели елементите на дисјунктни класи и го брои бројот на елементи во секоја од тие класи. За да некоја рекурзивна равенка може да се искористи за наоѓање на реден број на елементот, треба за секои две класи од елементи да важи сите елементи од едната во нив да бидат пред сите елементи од другата од нив во така дефинираниот редослед, бидејќи само така можеме а избираме колку елементи има пред или после некоја пермутација. Така, во претходното решение класите се сите пермутации на кои нареден елемент им е „(“ и сите пермутации на кои нареден елемент им е „)“. Секој елемент од првата класа е пред секој елемент од втората класа, па затоа можевме да ја искористиме соодветната равенка. Но, втората предложена рекурзивна равенка од Проблем 3.3 не може да се искористи затоа што класите таму не го задоволуваат ова својство. Да ја погледнеме ситуацијата кога имаме 4 пара загради. Во оваа ситуација втората рекурзивна равенка ги дели сите правилни распореди на загради на 4 класи според позицијата на затворената заграда соодветна на првата отворена заграда (која може да се најде на втората, четвртата, шестата и осмата позиција во стрингот). Да забележиме дека во класата каде затворената заграда соодветна на првата отворена заграда е на последна позиција, еден од стринговите е „((()(())))“, а во класата каде затворената заграда која ја затвара првата отворена заграда е на 6-та позиција се наоѓаат стринговите „((()))()“ и „(((())))()“. Бидејќи „((()(())))“ е пред „((()))()“ и после „(((())))()“, рекурзијата (3. 7) не може да се искористи за решавање на овој проблем. Рекурзијата (3. 6) може да се искористи за решавање на проблемот на реден број на правилно поставени загради, но со мала модификација. Имено, наместо со $A[m,k]$ да го бележиме бројот на стрингови кои се состојат од m правилно распоредени парови на загради, на кои последната отворена заграда е на позицијата k, ќе го бележиме бројот на стрингови кои се состојат од m правилно распоредени парови на загради, на кои првата затворена заграда е на позицијата k. Тогаш, колку поголемо е k, толку редниот број на таа пермутација е помал. Ова ќе биде оставено за вежба.

На крај ги даваме кодовите во Јава и C++:

JAVA 5. 6 ОПРЕДЕЛУВАЊЕ РЕДЕН БРОЈ ЗА ДАДЕНА НИЗА ОД ЗАГРАДИ

```
1  import java.util.*;
2
3  public class Main {
4     public static int[][] Avalues;
5
6     public static int A(int i, int j) {
7        if (Avalues[i][j] > 0)
8           return Avalues[i][j];
9
10       if (i == 0 || i == j)
11          Avalues[i][j] = 1;
12       else
13          Avalues[i][j] = A(i - 1, j - 1) + A(i - 1, j);
14       return Avalues[i][j];
15    }
16
17    public static int C(int i, int j, String niza) {
18       if (i == 0)
19          return 1;
20       if (niza.charAt(0) == ')')
21          return A(i - 1, j + 1) + C(i, j - 1,
niza.substring(1));
22       else
23          return C(i - 1, j + 1, niza.substring(1));
24    }
25
26    public static void main(String[] args) {
27       Scanner scn = new Scanner(System.in);
28       int n = scn.nextInt();
29       String bitniza = scn.next();
30
31       Avalues = new int[n + 1][n + 1];
32
33       System.out.println(C(n, 0, bitniza)+1);
34    }
35 }
```

Следи кодот во C++:

C++ 5. 6 Определување реден број за дадена низа од загради

```
#include<iostream>
#include <vector>
#include <string>
using namespace std;

vector<vector<int>> Avalues;

int A(int i, int j) {
   if (Avalues[i][j] > 0)
      return Avalues[i][j];

   if (i == 0 || i == j)
      Avalues[i][j] = 1;
   else
      Avalues[i][j] = A(i - 1, j - 1)+ A(i - 1, j);

   return Avalues[i][j];
}

int C(int i, int j, string niza) {
   if (i==0)
      return 1;
   if (niza[0]==')')
      return A(i-1, j+1) + C(i, j-1, niza.substr(1));
   else
      return C(i-1, j+1, niza.substr(1));
}

int main() {
   int n;
   cin >> n;
   string bitniza;
   cin >> bitniza;

   Avalues.resize(n+1, vector<int>(n+1));

   cout << C(n, 0, bitniza) + 1;

   return 0;
}
```

Прашања и задачи

1. Нека со $A[m,k]$ е бројот на стрингови кои се состојат од m правилно распоредени парови на загради, на кои првата затворена заграда е на позицијата k. Користејќи рекурзивна равенка за A да се даде рекурзивна равенка која ќе пресмета која по ред е во лексикографското подредување на загради, за дадена пермутација од n правилно поствени парови на загради.

2. Дефинираме двојна пермутација од ред n како низа која се состои од првите $2n$ природни броеви: $(a_1, a_2, \dots, a_n, a_{n+1}, a_{n+2}, \dots, a_{2n})$. За неа важи:

   - првите n елементи се во растечки редослед: $a_1 < a_2 < \cdots < a_n$;
   - последните n елементи се во растечки редослед: $a_{n+1} < a_{n+2} < \cdots < a_{2n}$;
   - за секое i, $a_i < a_{n+i}$: $a_1 < a_{n+1}, a_1 < a_{n+1}, \dots, a_n < a_{2n}$.

   Пример (1, 2, 4, 3, 5, 6) е двојна пермутација од ред 3 затоа што (1, 2, 3) и (4, 5, 6) се растечки и (1,4), (3,5) и (2,6) се растечки. Од друга страна двојни пермутации не се: (1, 4, 3, 2, 5, 6) – бидејќи (1, 2, 3) не е растечка, (1, 2, 4, 3, 6, 5) – бидејќи (4, 6, 5) не е растечка, (1, 4, 5, 2, 3, 6) – бидејќи (5, 2) не е растечка.
   Ваквите пермутации можеме да ги подредиме по лексикографски редослед, како во табелата за $n = 3$:

позиција	пермутација
1	1 2 3 4 5 6
2	1 2 4 3 5 6
3	1 2 4 5 3 6
4	1 4 2 3 5 6
5	1 4 2 5 3 6

   Треба да се одговори на два вида на прашања:

   a. Која е пермутацијата на дадена позиција?

b. На која позиција е дадена пермутација?

3. Дефинираме дупла пермутација од ред n како низа која се состои од првите n природни броеви, така што секој од нив се појавува по двапати. За неа важи дека секој подстринг кој што почнува и завршува со еден ист број е парен палиндром, и низата од први појавувања на броените од 1 до n е растечка. Пермутациите се подредени по лексикографски редослед.

   Пример (1, 1, 2, 3, 3, 2) е двојна пермутација од ред 3 затоа што (1, 1), (2, 3, 3, 2) и (3, 3) се парни палиндроми, а пермутациите (1, 2, 1, 3, 2, 3) и (2,2,1,3,3,1) не се, првата бидејќи (1, 2, 1) не е парен палиндром, а втората бидејќи првото појавување на 2 е пред 1. Ваквите пермутации можеме да ги подредиме по лексикографски редослед, како во табелата за $n = 3$:

позиција	пермутација
1	1 1 2 2 3 3
2	1 1 2 3 3 2
3	1 2 2 1 3 3
4	1 2 2 3 3 1
5	1 2 3 3 2 1

   Треба да се одговори на два вида на прашања:

   a. Која е пермутацијата на дадена позиција?

   b. На која позиција е дадена пермутација?

   Дади ги рекурзивните равенки кои го решаваат проблемот!

6 Алчни алгоритми

Во анализата на 0-1 проблемот на ранец разгледавме неколку стратегии кои не го решаваа проблемот, првата, ранецот да се наполни до крај, втората, во секој момент да се зема највредниот предмет и третата, во секој момент да се зема највредниот предмет по единица волумен, и видовме дека секоја од овие стратегии нема секогаш да го најде оптималното решение на проблемот. Но од друга страна, да го разгледаме така нареченият фракцион проблем на ранец, каде што подесувањето е исто, односно крадецот повторно со ранец од n волуменски единици ограбува продавница во која има m типови на предмети, каде i-тиот предмет има вредност од v_i и тежина t_i, а тој сака да земе што е можно повреден товар, но сега крадецот може да земе колку сака мал дел од предметите. На пример 0-1 проблемот би бил да зема конзерви грав, додека фракциониот проблем би бил да зема грав од вреќа. И овој проблем на ранец го има својството за оптимална потструктура, но во фракциониот проблем може да земеме колку сакаме од даден производ, па стратегијата ќе биде да се земе што е можно повеќе од производот кој има најголема цена по единица тежина. Според ова, иако проблемите се слични, фракциониот проблем на ранец е решлив со стратегија која во дадениот момент го земаме производот кој ни изгледа најдобро можно решение по тактиката прво да се пресмета просечната вредност на секој производ во однос на волуменот, а потоа, крадецот почнува со земање на колку што е можно повеќе од овој производ. Кога производот ќе се исцрпи, а во ранецот собира уште, ќе почне да зема колку што е можно повеќе од производот со следната најголемата вредност по волумен, и така натаму се додека собира во ранецот. Така, со сортирање на производите во зависност од нивната вредност по единица тежина, ваквиот алгоритам ќе работи во време $O(n\lg n)$, што е времето за сортирање. Времето за решавање на проблемот кога низата е веќе

сортирана зазема време $O(n)$. Ваквата стратегија се нарекува алчна стратегија, а ваквите алгоритми се нарекуваат алчни алгоритми.

Всушност голем број проблеми за оптимизација може да се решат со вакви алчни алгоритми, кои за разлика од алгоритмите со динамичко програмирање, од една страна се полесни за имплементација и се базираат на полесна идеја, а од друга страна се поефективни, односно побрзи. При алчниот алгоритам секогаш се прави изборот кој во даденият момент изгледа најдобар можен. Всушност се прави локален оптимален избор, со надеж дека тој избор ќе даде оптимално решение и за глобалниот проблем. Сепак, мора да знаеме дека иако алчните алгоритми се прилично моќни, сепак не го даваат секогаш оптималното решение и работат за ограничен број на проблеми. Секако, заради помалата сложеност, често пати ваквата стратегија се користи за да се дојде до приближно најоптималното решение, бидејќи во многу ситуации во пракса е подобро да имаме брз одговор, иако тоа не е најдобриот можен, отколку истиот да го пресметуваме долго време.

Идејата и видовите на проблеми кои се решаваат со алчни алгоритми, како и својствата кои треба да ги поседуваат проблемите кои можат да се решат со оваа стратегија ќе ги објасниме со еден проблем за кој алчната стратегија е многу интуитивна.

Проблем 6. 1. Роман во најмал број на томови

Да се навратиме на проблемот 4.1 за оптимално делење на главите од еден роман по томови. Во проблемот кој го разгледувавме, требаше, при даден број на томови и страници главите на романот да се поделат по томови, така да максималниот број на страници во некоја глава биде најмал можен. Проблемот кој го разгледуваме овде е на некој начин спротивен, затоа што наместо фиксен број на томови имаме фиксен број на страни кои може да се стават во еден том, додека треба да се оптимизира бројот на томови.

Дефинирање на проблемот

Имаме книга за која се дадени бројот на страници на секоја глава, $x_j, j = \overline{1,n}$. Треба главите да ги наредиме во томови, така да секој том има најмногу m страници, при што главите не можат да се делат во два тома.

Анализа на проблемот

Оптималната потструктура за овој проблем има поинаква природа од проблемот за распределба на главите во томови кој го разгледувавме во четвртата глава. Сега во првиот том од книгата можеме да ставаме нова глава од книгата се додека тоа е можно, односно, се додека бројот на страници во тој том да стане поголем од m. Тогаш таа глава ќе ја префрлиме во наредниот том и него ќе го пополнуваме се додека со додавање на нова глава бројот на страници останува помал од m и се така додека не ги наредиме сите глави во томовите. Овој алгоритам е прилично интуитивен и донекаде е јасно дека точно ќе го реши проблемот, но сепак треба да се докаже дека ова би важело, пред се затоа што видовме дека ваквата стратегија многу лесно може да не излаже. Затоа ќе пристапиме кон подетална анализа на проблемот.

Оптималната потструктура која ја поседува овој проблем е следнава:

> *Нека имаме роман кој има n глави и i-тата глава има a_i страници. Ако првите j глави од книгата се поделени во најмал број на томови, така што во секоја глава има најмногу m страници, тогаш во првите k глави, $k < j$ може да се поделат во најмал број на томови.*

Доказ на оптималното својство: Постапката за докажување оди на сличен начин како што тоа го правевме кај проблемите со динамичко програмирање. Нека првите j глави од книгата се поделени во најмал број на томови, така што во секоја глава има најмногу m страници, но првите k глави, за некое $k < j$ не се поделени оптимално. Нека тие се поделени во r_1 томови, но бидејќи тоа не е оптимална поделба, постои начин на кој тие можат да се поделат во помалку томови со најмногу m страници, да речеме $r_2 < r_1$. Тогаш, ако тие k глави ги поделиме во r_2 глави, главите кои во првата поделба биле во ист том со k–тата глава, а се со поголем број од неа да ги ставиме во $r_2 + 1$-та глава, а останатите глави да ги наредиме на ист начин како претходно. Тогаш со оваа распределба се добиваат најмногу онолку глави колку што биле претходно, па ова е или подобро решение од претходното, ако $r_2 + 1 < r_1$, што е контрадикција, или, ако важи $r_2 + 1 = r_1$, е друго оптимално решение. □

Сега кога докажавме дека проблемот има оптимална потструктура, јасно е дека врз база на неа може да се надеваме дека може да се изгради алгоритам со динамичко програмирање. Но дали постои и алчна стратегија што го решава проблемот? За да одговориме на ова прашање ќе го анализираме проблемот од позади наназад, слично како анализата на останатите проблеми за оптимизација кои ги решававме со динамичко програмирање. Последната глава секако се наоѓа во последниот том. Ако во оптималното решение во последниот том се наоѓа само последната глава, тогаш останатите $n - 1$ -ва глава може оптимално да се распределат во претходните томови. Ако во последниот том се наоѓаат само последните две глави, тогаш останатите $n - 2$- глави може оптимално да се распределат во претходните томови. Вака можеме да продолжиме и во општ случај јасно е дека важи дека ако

во последниот том се наоѓаат само последните k глави, тогаш останатите $n - k$ - глави може оптимално да се распределат во претходните томови.

Да забележиме дека ова не можеме да го направиме за секое k, бидејќи сумата на страниците на последните k глави мора да биде помала од m. Нека со $A[j]$ го обележиме минималниот број на томови во кои можат да се разместат првите j глави од книгата. Тогаш важи:

$$A[j] = 1 + \min_{k<j} \left\{ A[j-k] \left| \sum_{r>j-k} x_r \leq m \right. \right\}. \qquad (3.1)$$

Од друга страна е јасно дека ако во последниот том ставиме повеќе глави, тогаш претходните глави ќе можеме да ги распределиме пооптимално. Според тоа го имаме следново алчно својство:

Ако $j_1 < j_2$ тогаш важи и дека $A[j_1] \leq A[j_2]$.

Доказ на алчното својство: Ако првите j_2 глави оптимално сме ги наредиле во одреден број на томови, $A[j_2]$, тогаш од таквиот распоред можеме да ги извадиме сите глави после j_1-та, па првите j_1 глави ќе бидат наредени во помалку или во најлош случај исто толку томови. Оттука следува дека $A[j_1] \leq A[j_2]$. □

Врз основа на ова алчно својство следува дека во последниот том треба да ставиме што е можно повеќе глави. Така, нека k'_j е максималниот број кој го задоволува својството последните k_j глави имаат помалку од m страници, тогаш имаме дека:

$$A[j] > A[j-1] > \cdots > A\left[j - k_j\right].$$

Оттука равенството за $A[j]$ се поедноставува на следниов начин:

$$A[j] = 1 + \min \left\{ A[j-k] \left| \sum_{r>j-k} x_r \leq m \right. \right\} = 1 + A\left[j - k_j\right]. \quad (3.2)$$

Ако го искористиме овој факт, веќе нема да ни треба да го наоѓаме минимумот по сите можности за ставање на главите во последниот том, туку проблемот се сведува само на една можност, да се стават што е можно повеќе глави во тој том, а останатите да се распределат во претходните томови.

Врз основа на оваа рекурзивна врска може да се изгради рекурзивен алгоритам. Псевдокодот на тој алгоритам е следниов:

П 6. 1. ОПРЕДЕЛУВАЊЕ РЕДЕН БРОЈ ЗА ДАДЕНА НИЗА ОД ЗАГРАДИ

```
1   RekRoman2(x_k, k = 1,i, m)
2   b = x_i;
3   додека b ≤ m и i > 0 прави
4         {
5          i = i – 1;
6          b = b + x_i;
7         }
8   ако i = 0 тогаш врати 1 инаку
9         {
10         печати j + 1;
11         врати 1 + RekRoman2(x_k, k = 1,i, m);
12        }
```

Иако е ова рекурзивно решение, сепак има сложеност O(*n*), затоа што функцијата рекурзивно се повикува само еднаш и во сите тие рекурзивни повикувања вредноста на *j* се намалува за 1. Иако овој алгоритам нема голема сложеност, сепак рекурзивниот алгоритам не е добро решение кога е можно да се дизајнира итеративен алгоритам со истата сложеност. Псевдокодот на итеративниот алгоритам е:

П 6. 2. ОПРЕДЕЛУВАЊЕ РЕДЕН БРОЈ ЗА ДАДЕНА НИЗА ОД ЗАГРАДИ

```
1   внеси x_k, k = 1,i, m
2   i = 0;
3   j = 1;
```

```
4   додека j ≤ n прави
5       {
6        i = i + 1;
7        печати i"-тиот том почнува со главата" j;
8        b = x_k;
9       додека b ≤ m и j ≤ n прави
10          {
11           j = j + 1;
12           b = b + x_k
13          }
14      печати i;
15.  }
```

Сложеноста на алгоритамот лесно може да се пресмета врз основа на овој псевдокод. Во секое влегување на внатрешниот циклус вредноста на j се зголемува за 1, и таа вредност никогаш не се намалува. Надворешниот циклус се извршува се додека j не стане еднакво на n, па бројот на операции е $O(n)$.

Без разлика дали овој проблем ќе го решаваме со рекурзија или со итеративен алгоритам, сложеноста ќе биде иста, $O(n)$.

На крај ги даваме кодовите во Јава и C++. Решението во јава е со помош на рекурзија:

ЈАVA 6. 1 РОМАН

```
1 import java.util.*;
2
3 public class Main {
4
5     public static int RekRoman2(int[] x, int i, int m) {
6         int b = 0;
7
8         for (int j = i; i >= 0 && b + x[i] <= m; i--)
9             b += x[i];
10
11        if (i == -1)
12            return 1;
13
14        return 1 + RekRoman2(x, i, m);
15    }
16
17    public static void main(String[] args) {
18        Scanner scn = new Scanner(System.in);
19        int n = scn.nextInt();
20        int m = scn.nextInt();
21        int[] x = new int[n];
22
23        for (int i = 0; i < n; i++) {
24            x[i] = scn.nextInt();
25            if (x[i] > m)
26                System.out.println("Brojot na strani vo " +
i + "-tata galva e pogolem od maksimalniot broj na strani vo
tom.");
27        }
28
29        System.out.println(RekRoman2(x, n - 1, m));
30    }
31}
```

Решението во C++ кое следи е итеративно и ја избегнува рекурзијата:

C++ 6. 1 РОМАН

```
#include<iostream>
using namespace std;

int main() {
   int n, m;
   cin >> n >> m;
   int x [n];

   for(int i=0;i<n;i++)
   {
      cin >> x[i];
      if (x[i] > m)
         return -1;
   }

   int i = 0;
   int j = 0;

   while(j<n)
   {
      i++;
      cout << i << "-tiot tom pochnuva so glavata " << j+1
<< endl;
      int b = 0;
      for(; j < n && b + x[j] <= m; j++)
         b += x[j];
   }

   return 0;
}
```

Прашања и задачи

1. Во една галерија се наредени n слики на еден ѕид, во една линија. Дадена е далечината на секоја од сликите од почетокот на ѕидот, $x_i, i = \overline{1,n}$. За да се зачуваат сликите потребно е да се стават одреден број на чувари. Секој од нив може да пази на далечина x лево од него и десно од него. Проблемот кој што треба да се реши е да се распоредат најмал број на чувари во галеријата.

   a. Да се даде оптималното својство кое го поседува проблемот!

   b. Да се покаже дека проблемот поседува алчно својство!

   c. Да се објасни како ќе се реши проблемот!

   d. Која ќе биде сложеноста на алгоритамот?

2. Дадена е низа со големина n во која секој елемент е или полицаец или крадец. Секој полицаец може да фати само еден крадец, но само крадец кој е на далечина најмногу k од него. Треба да го најдеме максималниот број крадци што можат да бидат фатени.

   a. Да се даде оптималното својство кое го поседува проблемот!

   b. Дади го алчно својство кое го поседува проблемот!

   c. Да се објасни како ќе се реши проблемот!

   d. Која ќе биде сложеноста на алгоритамот?

Проблем 6. 2. Алчна стратегија

Од дискусијата во претходната глава видовме дека за голем број од проблемите за оптимизација многу лесно можат да направиме погрешен заклучок дека може да се користи алчна стратегија, всушност тоа не може да биде решение. Затоа е потребна голема претпазливост кога истата се користи и несомнено доказ дека оптималното решение може да се избере со некоја алчна стратегија. Затоа уште малку да продискутираме за оваа техника, со цел полесно да се дојде до заклучок дали за некој проблем може да се користи алчен алгоритам или не.

Алчната стратегија обезбедува оптимално решение на даден проблем со тоа што прави низа од избори. Во секој момент кога треба да се направи избор, во алчната стратегија се избира оној избор кој изгледа најдобар во тој момент. Оваа стратегија е хевристичка и не секогаш обезбедува оптимално решение, но некогаш истата работи. Според тоа, кога имаме индиции дека некој проблем може да се реши со алчна стратегија, за да видиме дали тој навистина и го дава оптималното решение во секој случај, треба малку подетално да го анализираме проблемот. На пример, при изградбата на алчниот алгоритам за решавање на проблемот за распределба на главите од еден роман во минимален број на томови, така да ниеден том не надмине m страници се помина низ следниве чекори:

1. Наоѓање на оптималната структура на проблемот.
2. Дизајнирање на рекурзивно решение.
3. Докажување дека во секоја фаза на рекурзијата, едно од оптималните избори е алчниот избор, па секогаш можеме да го избереме.
4. Покажување дека после алчниот избор проблемот се сведува на само еден потпроблем.
5. Развој на рекурзивен алгоритам во кој се имплементира алчната стратегија

6. Конвертирање на рекурзивниот алгоритам во итеративен алгоритам.

При анализата на овој алгоритам поминавме низ скоро сите фази на динамичкото програмирање, освен фазата на дизајнирање на алгоритам со динамичко програмирање. Но, во пракса обично не е така, т.е. обично, кога развиваме алчен алгоритам скокаме голем дел од чекорите. Секако не мора да развиваме рекурзивен алгоритам, но многу често кога за некој проблем може да се искористи алчна стратегија, обично тоа го имаме во ум и правиме оптимизација на динамичкиот алгоритам. Но ако сакаме истиот да покажеме дека работи, или ако сакаме да се осигураме дека е нашиот алгоритам е добар, тогаш треба да ги поминеме чекорите 1, 2 и 3. Во општ случај чекорите при дизајнирање на алчен алгоритам се следниве:

1. Да се претвори проблемот за оптимизација во проблем во кој се прави избор и при тоа останува да се реши само еден потпроблем.

2. Да се докаже дека со правење на алчен избор секогаш постои оптимално решение на оригиналниот проблем, т.е. да се докаже дека алчниот избор е безбеден.

3. Да се демонстрира дека со правење на алчен избор, она што останува е потпроблем, со својството дека ако се комбинира оптимално решение на потпроблемот со алчниот избор што сме го направиле, ќе се дојде до оптимално решение на оригиналниот проблем.

Нема генерален начин како да се користат алчните алгоритми, ни некој генерален начин дали за некој проблем може да се искористи ваков алгоритам. За да може да се реши некој проблем со алчно програмирање, мора прво да има оптимална потструктура, која морање да ја има и за да се реши со динамичко програмирање, но дополнително треба да може да се направи алчен избор. Ако можеме да покажеме дека проблемот ги има овие својства, тогаш сме на добар пат кон развивање на алчен алгоритам.

Клучната работа која проблемите кои можат да се решат со алчен алгоритам ги карактеризира е својството на алчен избор: глобалното оптимално решение може да се добие со правење на

локално оптимален (алчен) избор. Со други зборови, кога ќе се размислува кој избор да се направи, се прави изборот што изгледа најдобар во тековниот проблем, без оглед на резултатите од потпроблемите. Ова е тоа во што алчните алгоритми се разликуваат од динамичкото програмирање. Во динамичкото програмирање, правиме избор во секој чекор, но изборот обично зависи од решенијата на потпроблемите. Потоа, обично проблемот го решаваме со така наречениот одоздола-нагоре принцип, тргнувајќи од помалите потпроблеми кон поголемите. Во алчниот алгоритам се избира она што изгледа најдобро во определениот момент и потоа проблемот се решава без разлика на тоа што произлегува од направениот избор. Изборот може да зависи од другите избори до тој момент, но не може да зависи од било какви идни избори или решенија на потпроблеми. Така, алчната стратегија обично напредува во врвот надолу, правејќи еден алчен избор по друг, при што проблемот се намалува за еден.

Во примерот кој го разгледавме претходно, алчниот избор беше многу очигледен, но не сите проблеми кои се решаваат со оваа техника не се толку лесно видливи. Понекогаш е потребна подлабока анализа за да се извлечат својството на оптималност и алчното својство. Во примерите кои понатаму ќе ги разгледаме ќе обратиме внимание и на аспектот како да се разграничи и осети дали одредена алчна стратегија е добра за решавање на проблемот.

Прашања и задачи

1. Да се навратиме на проблемот за најкраток пат во матрица во која во секое поле има запишано број. Една интуитивна алчна стратегија која во тој проблем може да се употреби е секогаш да се движиме од полето во кое сме, кон полето кое има помала вредност.

    a. Дадете пример дека оваа алчна стратегија нема да го даде оптималното решение!

    b. Дали можете да дадете алчен пристап кој секогаш го дава оптималното решение?

    c. Која ќе биде временската сложеност на вашето предложено решение под b. во најлош случај?

2. Да се навратиме на проблемот за ред на билети. Една интуитивна алчна стратегија е да се групираат двајца за кои времето тие двајцата да купуваат посебно минус времето тие двајцата да купуваат посебно има најголема вредност. Дали оваа алчна стратегија секогаш го дава оптималното решение?

Проблем 6.3 Проблем за интервално распоредување

Постојат поголем број оптимизациони проблеми за распоредување или закажување на активности кои се одвиваат во текот на некој временски интервал. Проблемите од овој тип можат да се јават и во друга форма, како распоредување на предмети со различна должина кои не смеат да стојат на исто место и слично. Ние овде ќе разгледаме еден таков проблем, а како вежба се дадени уште некои слични проблеми.

Дефинирање на проблемот

Еден професор учествува на конференција на која има паралелни предавања. Во програмата на конференцијата е дадено кое предавање кога почнува и кога завршува. Професорот си обележал кои предавања го интересираат, но увидел дека не може да стигне да ги посети сите. Тој не сака да влегува на веќе почнато предавање, ниту пак да излезе од предавањето пред тоа да заврши. Затоа решил да го пресмета најголемиот број на предавања на кои може да присуствува. Предавањата се одвиваат во простории кои се доволно близу едно до друго, па професорот не троши време за преместување од една во друга просторија. Во општ случај овој проблем може да се јави во различна форма, но основата е да се распоредат одреден број на активности кои бараат некој ист ресурс, како што е овде професорот, што значи дека може да се прави само една активност во дадено време и имаат иста цел, да се најде максимално множество на активности кои се меѓусебно не се поклопуваат, како овде предавањата. Ваквите активности, кои не се преклопуваат се нарекуваат меѓусебно компатибилни. Нека претпоставиме дека имаме n дадени предавања $x_i, i = \overline{1,n}$ и секое предавање x_i има време кога почнува s_i и време кога завршува f_i, каде $0 \leq 0 \leq s_i < f_i < \infty$. Секое предавање x_i се одвива во текот на интервалот $[s_i, f_i)$. За две предавања a_i и a_j велиме дека се компатибилни ако интервалите $[s_i, f_i)$ и $[s_j, f_j)$ немаат дел кој се

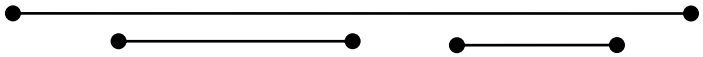
преклопува. Тогаш ќе важи дека $s_i \geq f_j$ или $s_j \geq f_i$. Во Табела 6. 1 е дадено множество од 10 предавања кои се интересни за професорот. Множеството предавања $\{x_1, x_3, x_8, x_{10}\}$ е компатибилно множество, но не е најголемото такво, затоа што множеството $\{x_2, x_9, x_6, x_8, x_{10}\}$ е поголемо такво множество и воедно најголемото со тоа својство.

Табела 6. 1. Распоред на предавања. Првата редица го дава редниот број на предавањето, втората редица го дава почетокот на предавањето, а третата редица е времето на завршување на секое предавање.

x_i	1	2	3	4	5	6	7	8	9	10
s_i	3	1	5	5	2	7	9	8	4	11
f_i	5	4	8	10	9	8	12	10	6	13

Анализа на проблемот

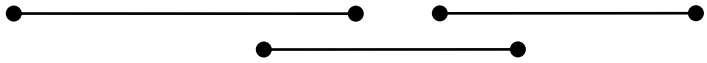
Многу често најпрва идеја за решавање на овој проблем е да се земаат активностите кои почнуваат порано, затоа што тоа асоцира дека тие би завршиле порано. Ова би можело да не води до решение ако сите интервали имаат еднаква должина, но не и во случај кога должината на активностите е произволна, можеби првото предавање е толку долго што нема да остане место за ниту едно друго предавање, како што е прикажано на Слика 6. 1. На пример ако имаме три предавања, такви што првото трае од време 1 до време 8, второто од време 2 до време 4, а третото од време 5 до време 7, тогаш подобро е професорот да отиде на вторите две предавања наместо на првото. Оваа стратегија јасно нема да го реши проблемот, па мора да продолжиме со анализата.



Слика 6. 2. Ако го избереме предавањето кое почнува прво, ќе избереме само едно предавање. Изборот на останатите две предавања е подобар избор

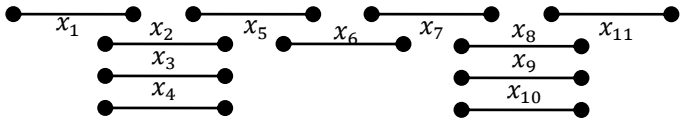
Бидејќи не интересира професорот да седи подолго на предавања, туку само да посети поголем број од нив, една алчна идеја која може да ни текне е во распоредот на професорот да ги ставаме предавањата кои траат пократко. Оваа стратегија јасно би не

водела кон тоа професорот да има повеќе слободно време за други предавања. Многу лесно можеме да видиме дека ова паѓа во вода, како што е прикажано на Слика 6. 3. Да претпоставиме дека имаме три предавања, такви што првото трае од време 1 до време 5, второто од време 6 до време 10, а третото од време 4 до време 7. Според оваа алчна стратегија, прво би го зеле последното предавање кое трае најкратко, но тогаш професорот нема да може да посети ниту едно од другите две предавања. Од друга страна, тој може без проблем да отиде на првото и второто предавање, што е оптималното решение во овој случај. Значи и оваа алчна стратегија пропаѓа.



Слика 6. 3. Ако го избереме предавањето кое трае најкратко прво, ќе избереме само едно предавање. Изборот на останатите две предавања е подобар избор.

Последната анализа може да не води и кон друго размислување, ако имаме поголем избор на активности кои може да ги селектираме, тогаш можеме да избереме поголем број од нив. За оваа идеја треба да ги селектираме оние предавања кои се преклопуваат со најмал број од другите предавања. Значи треба да ги најдеме оние предавања кои имаат најголем број взаемно компатибилни предавања. Но дали и ова ќе "упали"? На Слика 6. 4 се дадени 11 предавања на кои временските интервали во кои ќе се одвиваат се: [0, 3), [2, 5), [2, 5), [2, 5), [4, 7), [6, 9), [8, 11), [10, 12), [10, 12), [10, 12), [11, 14). Ако прво го избереме 6-тото предавање, потоа ќе можеме да избереме само уште 2 предавања, додека со избор на првото, петтото, седмото и единаесеттото предавање ќе имаме пооптимално распоредување на предавањата. Нити оваа идеја не го решава проблемот



Слика 6. 4. Ако прво го избереме x_6, потоа ќе можеме да избереме само уште 2 предавања, додека со избор на x_1, x_5, x_7 и x_{11} даваат подобар избор.

Три идеи за алчни стратегии за решавање на овој проблем ни пропаднаа, па може да загубиме надеж дека овој проблем може да се реши на ваков брз начин. Но сепак, проблемот е во тоа што не го избравме вистинскиот алчен избор.

За да дојдеме до правилната алчна стратегија, прво ќе почнеме со карактеризирање на својството на оптималност. Да претпоставиме дека како прво предавање кое треба да го посети професорот е i -тото предавање, кое почнува во времето s_i , а завршува во времето f_i. Тогаш, после тоа предавање тој може да посетува само предавања кои почнуваат по времето f_i . Тие предавања мора да бидат оптимално распоредени, затоа што ако не се оптимално распоредени, можеме да најдеме друг избор со кој професорот може да отиде на поголем број предавања, па заедно со ова прво предавање би добиле подобар распоред од оној за кој што тврдевме дека е оптимален. Ова веќе води кон општото својство за оптималност:

> *Ако во оптималното распоредување на предавањата, првите k предавања се $a_{j_1}, a_{j_2}, \dots, a_{j_k}$, тогаш останатите предавања оптимално се распоредени во времето после f_{j_k}.*

Доказ на оптималното својство: Да претпоставиме дека својството не важи, т.е. ако предавањата кои почнуваат после времето f_{j_k} не се оптимално распоредени, можеме да најдеме друг избор со кој професорот може да отиде на поголем број предавања, па заедно со овие први r предавања би добиле подобар распоред од оној за кој што тврдевме дека е оптимален. □

Нека со S_i го обележиме множеството на предавања кои завршуваат по i-тото предавање, т.е. после времето $f_i, i = \overline{1,n}$, со S_0

го обележиме целото множество на активности, а со $A[i]$ го обележиме максималниот број на предавања кои професорот може да ги слуша во интервалот од $[f_i, \infty)$, односно максималниот број предавања после првото предавање. Тогаш ако како прво се избере i-тото предавање, ќе имаме дека

$$A[0] = 1 + A[i].$$

Ако пак со $C[i]$ го обележиме оптималното множество на компатибилни предавања од множеството S_i. Тогаш ако прво се избере k-тото предавање,

$$C[0] = \{x_i\} + C[i].$$

Но, бидејќи ние не знаеме кое е прво предавање, тогаш треба да провериме по сите можни предавања, па рекурзивната равенка е:

$$A[0] = 1 + \max_{1 \le i \le n} A[i]. \qquad (3.3)$$

Ако ја користиме оваа равенка за да направиме рекурзивно решение, ќе треба да проверуваме за сите можни подмножества од активности и таквото решение ќе има сложеност $O(2^n)$, бидејќи имаме толку подмножества. Секако, јасно е дека ако i-тото предавање во оптималното распоредување е избрано како прво предавање, тогаш во S_i нема предавање кое завршува пред тоа да почне, па оттука можеме да ги намалиме можностите за S_i. Од друга страна, ако некое предавање почнува после i-тото предавање, тогаш мора и да заврши после i-тото предавање, па S_i се состои од дел од предавањата кои завршуваат после i-тото предавање. Оттука некако природно е да ги подредиме активностите се по растечки редослед по нивното време на завршување, па понатаму ќе сметаме дека: $f_0 \le f_1 \le \cdots \le f_n \le f_{n+1}$. Вака, $S_i \subseteq \{x_{i+1}, \dots, x_n\}$, што значително ја намалува сложеноста на решението претставено со рекурзивната равенка (3. 3). Имено, ако градиме стандарден алгоритам со динамичко програмирање, тогаш треба да за секое множество S_i, почнувајќи од $i = n$ до $i = 1$ да го пресметаме оптималното

распоредување над активностите во него по следнава рекурзивна равенка:

$$A[i] = \begin{cases} 0, & S_i = \emptyset \\ 1 + \max\limits_{j \in S_i} A[j], & \text{инаку} \end{cases}. \qquad (3.4)$$

Со користење на оваа рекурзија, сложеноста за пресметување на $A[i]$ е $O(n)$, па вкупната сложеност би била $O(n^2)$.

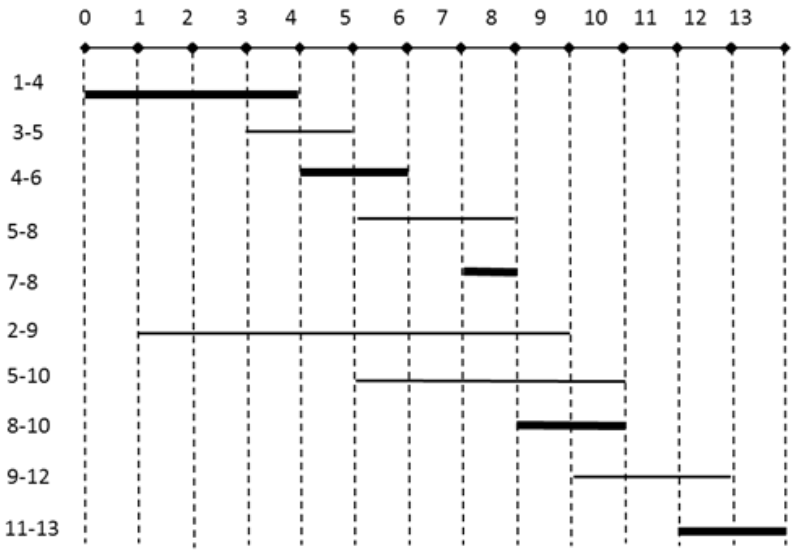
Сепак, може да се забележи дека секогаш ќе можеме да го земаме првиот елемент од S_i, што ќе биде алчното својство кое го поседува овој проблем, изразена во следново тврдење:

> *Нека S_i е непразно множество и нека x_k е првата активност во S_i, односно активноста со најрано време на завршување: $f_k = \min\limits_j S_i$. Тогаш постои оптимално множество од компатибилни активности над S_i во кое е активноста x_k.*

Доказ на алчното својство: Прво ќе докажеме дека нема активности кои почнуваат после завршувањето на i-тата и завршуваат пред почетокот на k-тата активност. Ако постои таква активност x_r тогаш $f_i \leq s_r < f_r \leq s_k < f_k$, па x_r е исто така во S_i и тоа има порано време на завршување од x_k, што е во контрадикција со нашиот избор на x_k. Сега нека претпоставиме дека постои оптимално множество на компатибилни активности од S_i во кое што не е активноста x_k. Тогаш во тоа множество прва завршува некоја друга активност x_r. Секако, таа мора и прва да почнала. Но времето на завршување на x_r е после времето на завршување на x_k, па на местото на активноста x_r може да се стави активноста x_k и со тоа да се добие друго оптимално множество на компатибилен активности од S_i. □

Користејќи го ова алчно својство, наоѓањето на оптималното решение се сведува само на една можност. Уште повеќе таа активност може многу лесно да се најде. Така решението многу се убрзува. За да го решиме потпроблемот S_i треба само да ја избереме активноста x_k со најраното време на завршување и да ја додадеме

на множеството на активности, а потоа на ова множество да го додадеме оптималното решение на потпроблемот S_k.



Слика 6. 5. Пример за подредени активности по растечки редослед на времето на завршување. Здебелените активности се активностите кои влегуваат во оптималното множество.

На примерот на Слика 6. 5 алгоритамот работи на следниов така што прво ја зема првата активност која трае од време 1 до време 4. Во множеството S_1 прва активност е активноста која е во временскиот интервал [4,6). Во множеството S_3 прва активност е активноста која е во временскиот интервал [7,8), во множеството S_5 прва активност е активноста која е во временскиот интервал [8,10) и на крај во множеството S_8 прва активност е активноста која е во временскиот интервал [11,13).

Рекурзивното решение е дадено со процедурата $RAktivnost$. Тоа го зема стартното и завршното време на активностите, дадени како низи s_i и f_i, како и индексите k и n кои го дефинираат потпроблемот S_k. Ако n-те влезни активности се подредени по растечки редослед во однос на времето на завршување, тогаш сложеноста е $O(n)$, а ако не е така, тогаш повеќе време зазема сортирањето, $O(n\ln(n))$.

П 6. 3. РАСПРЕДЕЛУВАЊЕ НА АКТИВНОСТИ, РЕКУРЗИВНО РЕШЕНИЕ

```
1  RAktivnost (s_i, f_i, j, n)
2  k = j + 1;
3  додека k ≤ n и s_k < f_j прави k = k + 1;
4        ако k ≤ n тогаш врати {x_k} ∪ RAktivnost (s_k, f_k, k, n);
5                          инаку
6                              врати ∅
```

Рекурзивното решение лесно може да се конвертира во итеративно во кое активностите кои влегуваат во оптималното множество веднаш се зачувуваат или печатат:

П 6. 4. РАСПРЕДЕЛУВАЊЕ НА АКТИВНОСТИ

```
1  Внеси s_i, f_i;
2  n = length[s];
3  печати x_1;
4  j = 1;
5  за k од 2 до n прави
6     ако s_k ≥ f_j тогаш
7          {
8            печати x_k;
9            j = k.
10         }
```

И двете процедури работат во време $\Theta(n)$, под претпоставка дека времињата на завршување се подредени.

Кодовите во Јава и C++ кои ги даваме овде се итеративни решенија кои се базираат на третиот пристап.

```java
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner scn = new Scanner(System.in);
        int n = scn.nextInt();
        int[] s = new int[n];
        int[] f = new int[n];

        for (int i = 0; i < n; i++)
            s[i] = scn.nextInt();

        for (int i = 0; i < n; i++)
            f[i] = scn.nextInt();

        System.out.print("1 ");

        int j = 0;
        for (int k = 1; k < n; k++) {
            if (s[k] >= f[j]) {
                System.out.print((k + 1) + " ");
                j = k;
            }
        }
    }
}
```

```
1  #include<iostream>
2  using namespace std;
3
4  int main() {
5     int n;
6     cin >> n;
7     int s [n];
8     int f [n];
9
10    for(int i=0;i<n;i++)
11       cin >> s[i];
12
13    for(int i=0;i<n;i++)
14       cin >> f[i];
15
16    cout << "1 ";
17
18    int j = 0;
19    for(int k=1;k<n;k++)
20    {
21       if (s[k] >= f[j])
22       {
23          cout << (k+1) << " ";
24          j=k;
25       }
26    }
27 }
```

Прашања и задачи

1. На едно пристаниште пристап имаат n бродови. На пристаништето се наредени k алки за врзување на бродови, и секој брод има точно една алка на која смее да се врзе, односно тој може да го врзе својот брод само на алката која му е доделена. Дадени се должините на секој од бродовите. Бродот се врзува паралелно на пристаништето и мора да биде врзан така што алката што му е доделена е помеѓу почетокот и крајот на бродот, вклучувајќи ги неговите крајни точки. Краевите на бродовите може да се допираат меѓусебе, но бродовите не смеат да се преклопуваат. Поради ова ограничување сите бродови не можат да бидат врзани во исто време. Проблемот е да се најде максималниот број на бродови кои можат да бидат врзани во исто време на означените алки.

   a. Дади го и докажи го оптималното својство на овој проблем.

   b. Покажи дека проблемот поседува алчно својство.

   c. Опиши како ќе го решиш проблемот!

   d. Реши ја задачата во C++ и Јава и провери го твоето решение на задачата Маѓепсници на системот МЕНДО, [17].

2. Дадена е низа од n активности $x_i, i = \overline{1,n}$ и за секоја активност е дадено времето кога почнува s_i и времето кога завршува f_i, каде $0 \leq 0 \leq s_i < f_i < \infty$. Две активности a_i и a_j се компатибилни ако интервалите $[s_i, f_i)$ и$[s_j, f_j)$ немаат дел кој се преклопува. За секоја активност е дадена заработка y_i. Да се изберат оние активности кои ја максимизираат вкупната заработка. Како ќе го модифицирате решението на задачата од овој дел за да го решите овој проблем?

3. Дадена е низа од n активности $x_i, i = \overline{1,n}$ и за секоја активност е дадено времето кога мора да заврши f_i и заработката од неа y_i . Секоја работа трае време 1. Треба да се максимизира вкупната заработка. Опиши алгоритам со кој ќе го решиш овој проблем!

Проблем 6. 4. Хафманов код

Проблемот кој овде ќе го разгледаме решава едно од основните прашања во областа на компресија на податоци, област која е дел од основите на дигиталната комуникација. Компјутерите во суштина работат над низа од битови (односно, низи кои се состојат само од симболите 0 и 1), па затоа се потребни начини со кои текст напишан во побогати азбуки, како азбуки со кои се служат луѓето, да го претворат овој текст во долга низа од битови. Ваквите постапки се наречени шеми за кодирање. Наједноставен начин да се направи ова е со користење на фиксен број на битови, каде што за секој симбол во азбуката ќе се резервира различна низа, а потоа овие низи само ќе се спојат и ќе го формираат текстот. Битно е после да има лесен начин на декодирање на така запишаниот текст, што не е проблем со кодови со фиксна должина. На пример ако го земеме кодот со фиксна должина од Табела 6. 2., тогаш зборот „*aba*“ ќе се кодира со шемата „000001000“. При декодирање, првите три бита се декодираат во „*a*“, вторите три во „*b*“ и последните 3 во „*a*“. Ако претпоставиме дека во дадена датотека од 100.000 знаци која треба да се кодира фреквенциите со кои се појавува секој од знаците е како што се дадени во Табела 6. 2, тогаш со код со фиксна должина ќе треба да се искористат барем по 3 бита за секоја буква, па би ни биле потребни 300.000 битови за да се кодира целата датотека. Прашањето е дали тоа може да се направи подобро, односно со помал број на битови.

Ова прашање за намалување на просечниот број на битови по буква е основен проблем во областа на компресија на податоци. Кога големи датотеки од податоци треба да се испратат преку комуникациските мрежи или да се складираат на тврди дискови, важно е истите да се претстават колку што е можно покомпактно, но сепак мора да се запази условот дека подоцна читачот на датотеката треба да биде способен тој код правилно да го реконструира. Многу истражувања се посветени на овој проблем познат како дизајн на алгоритми за компресија, кои работат на тој начин што ја преземаат датотеката како влез и да го намалат просторот за нејзино запишување со ефикасно шеми за кодирање.

Во овој дел ќе опишеме еден од основните начини на решавање на овој проблем, познат како Хафманов код. Хафмановите кодови се широко користени и многу ефективни техники за компресирање на податоци; обично со нив се добива заштеда од 20% до 90%, во зависност од карактеристиките на податоците кои се компресираат.

Дефинирање на проблемот

Хафмановите кодови се тип на кодови со променлива должина, што значи дека за претставување на некои карактери се користат повеќе, а за некои помалку битови. Природно е колку почесто се јавува некој знак, толку помала бит низа да се користи за негова репрезентација, па поради тоа, Хафмановиот алчен алгоритам, за оптимална репрезентација на знаците како бинарни стрингови, работи над дадена табела од фреквенции на појавување на знаците. Стратегијата е знаците со поголема фреквенција да се кодираат со кратки бит-низи, а карактерите кои имаат мала фреквенција со долги бит-низи.

Табела 6. 2. Пример за фреквенција на карактерите во дадена датотека и код со фиксна и променлива должина.

Карактери	a	b	c	d	e	f
Фреквенција во илјади	40	20	13	12	10	5
Код со фиксната должина	000	001	010	011	100	101
Код со променлива должина	0	100	101	111	1100	1101

Во Табела 6. 2. е прикажан еден таков код. Тука, најфреквентниот знак, „*a*“ се кодира со кодирачки збор со еден бит, т.е. „0“, додека „*f*“ кој ретко се појавува се претставува со 4-битниот стринг „1100“. Овој код побарува (40 · 1 + (20 + 13 + 12) ·3 + (10+5)·4) · 1000 = 235000 битови за претставување на целата датотека, што претставува заштеда од 25%. Воедно ова е и едно од оптималните дрва.

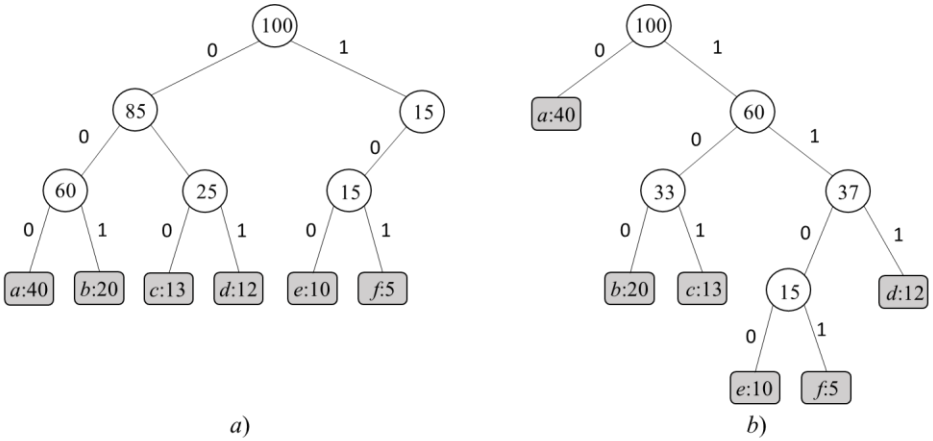
Јасно е дека дадена низа од знаци лесно може да се кодира во ваква низа: само се спојуваат кодовите на стрингот. На пример зборот abc се кодира со $0 \circ 101 \circ 100 = 0101100$, каде “$\circ$” се користи за да се претстави конкатенација. Но прашањето е дали дадена низа од битови ќе може еднозначно да се декодира во низа од знаци. Може да се забележи дека ниту еден кодиран збор не е префикс на друг

кодиран збор, па ова ќе ни ја обезбеди таа еднозначност. На пример да видиме како ќе се декодира бит-низата „0101100“:

- Низата почнува со „0“, а во шемата имаме само една буква чиј код почнува со „0“, па тој бит ќе се замени со „*a*“ и ќе остане да се декодира бит-низата „101100“
- Низата „101100“почнува со „1“, а во шемата нема буква која се кодира само со „1“, а сите останати букви се на почеток на кодот имаат „1“. Затоа за декодирање мора да се земе и наредниот бит, т.е. да се разгледа „10“. Со ова исто така не се декодира ниедна буква, а само „*b*“ и „*c*“го имаат како префикс, па мора да го земеме третиот бит за да одлучиме која од овие две букви е наредна. Нареден бит е пак „1“, и гледаме дека со „101“ почнува само „*b*“, од каде заклучуваме дека таа е наредната буква. Не може да настане забуна, затоа што нема друга буква која на почеток го има „101“.
- На крај останува да се декодира бит-низата „100“, што е кодот на „*c*“. Слично како претходно не може да настане забуна и некој дел од оваа бит-низа да се декодира поинаку, затоа што ниту со „1“, ниту со „10“ не се кодира ниту еден друг карактер.

Ваквите кодови се нарекуваат префиксни кодови. Може да се покаже дека било кој код може да се претвори во префиксен код, па не се губи од општоста ако вниманието го насочиме само на вакви кодови. Префиксните кодови се добри затоа што тие се лесни и за декодирање, бидејќи ниту еден збор не е префикс на друг збор, па ако го читаме почетокот на кодот, точно ќе знаеме како да се декодира, затоа што тоа е еднозначно дадено.

За да можеме да декодираме потребна ни е соодветна структура за репрезентација. За ова најдобро е да се искористи бинарно дрво, во кое ако читаме 0 ќе значи да одиме кон левото дете, а 1 кон десното, се додека не дојдеме до лист кој ни кажува со кој карактер треба да се декодира прочитаниот подстринг од низата. Структурата е илустрирана на Слика 6.4.1, за првиот и вториот тип на кодирање.



Слика 6. 6. Бинарни дрва за репрезентација на кодови. а) Структура која одговара за кодот со фиксна должина од Табела 6. 2. б) Структура која одговара за префиксниот код од Табела 6. 2.

За дадено дрво T соодветно за некој код, лесно може да се пресмета бројот на битови кои се потребни да се декодира датотеката. Нека со $f[\sigma]$ ја обележиме фреквенцијата на знакот σ во азбуката Σ, а со $d_T[\sigma]$ длабочината на листот σ во дрвото. Да забележиме дека $d_T[\sigma]$ е исто така должината на кодираниот збор за знакот σ. Сега бројот на битови потребни да се кодира дадена текст е:

$$A[T] = \sum_{\sigma \in \Sigma} f[\sigma] \cdot d_T[\sigma], \qquad (3.5)$$

што се дефинира како цена на дрвото T. Целта е да се конструира дрво кое има најмала цена.

Анализа на проблемот

За да има смисла целиот овој алгоритам за наоѓање на оптимален префиксен код битно е следново својство:

Својство 6. 1

Секој префиксен код може да се претстави со целосно бинарно дрво и обратно, секое целосно бинарно дрво со точно толку

листови колку што има знаци во азбуката може да се дефинира префиксен код за таа азбука.

Доказ: Ако ни е дадено целосно бинарно дрво со $|\Sigma|$ листови, тогаш секој лист може да се обележи со некој знак од азбуката и секое ребро кое оди кон лево дете со „0“, а секое ребро кое оди кон десно дете со „1“. Со тоа ќе добиеме целосно бинарно дрво кое одговара на шема за еднозначно декодирање на секој знак од азбуката.

Од друга страна, за даден префиксен код, можеме да изградиме бинарно дрво рекурзивно, со тоа што ќе почнеме од коренот и сите знаци чии енкодиции започнуваат со „0“ ќе се стават во левото поддрво, додека сите знаци чии енкодиции започнуваат со „1“ ќе се стават во десното поддрво. Понатаму истата постапка рекурзивно се применува на секое од поддрвата се додека не дојдеме до азбука со само еден знак. □

Врз основа на ова својство, потрагата по оптимален префиксен код може да се гледа како на наоѓање на целосно бинарно дрво T со минимална цена. Од друга страна треба да се забележи уште еден едноставен факт за бинарно дрво кое одговара на оптимален код:

Својство 6. 2

Ако префиксниот кодот е оптимален, истиот секогаш ќе може да се претстави со целосно бинарно дрво, односно секое внатрешно теме на дрвото има точно две деца.

Доказ: Нека T е бинарно дрво кое одговара на некој оптимален префиксен код и кој има јазол u со точно едно дете v. Тогаш u можеме да го извадиме од T и на негово место како лево дете да го закачиме целото поддрво со корен од v, а u да го ставиме како десно дете на v. Така се добива ново дрво во кое бројот на битови потребни за кодирање на било кој лист е помал или еднаков отколку во T, поточно стриктно помал за сите листови во поддрвото на v, и ист на сите останати. Оттука префиксниот код кој одговара на T' е пооптимален од тој кој одговара на префиксниот код на T, што е контрадикција. □

Сега вниманието може да го насочиме само на целосни бинарни дрва, и ако Σ е азбуката од која се земаат знаците, тогаш дрвото има точно $|\Sigma|$ листови, по еден за секој знак од азбуката, и точно $|\Sigma|-1$ внатрешни јазли. Често користена техника кога сме во потрага по ефикасен алгоритам е да претпоставиме дека имаме некое делумно знаење за оптималното решение и истото да го искористиме за наоѓање на комплетно решение. Овде можеме да се запрашаме што ако некој ни го даде оптималното целосно бинарно дрво, но не ни е дадено кој лист за кој знак одговара. За да го комплетираме решението, ќе треба да ги обележиме лисјата со знаците од азбуката. Многу е интуитивно дека знаците со помала фреквенција треба да бидат на подлабоките лисја. Оттука, знакот со најмала фреквенција е на еден од листовите кои се најдлабоко во дрвото. Од друга страна, затоа што дрвото е целосно, неговиот родител мора да има уште едно дете, па тој лист има сестринско теме, на кое природно е да го обележиме со знакот со втора по ред најмала фреквенција. Ова е всушност алчното својство кое го поседува проблемот:

> *Нека Σ е азбука во која фреквенцијата на знакот σ од Σ е $f[\sigma]$. Ако x и y се двата знака во Σ со најмала фреквенција, тогаш постои оптимален префиксен код во кој кодовите за x и y имаат иста должина и се разликуваат само во последниот бит.*

Доказ на алчното својство: Нека T е дрво соодветно на некој оптимален префиксен код и нека a и b се двата знака кои соодветствуваат на листови кои се најдлабоко во дрвото T. Без губење на општост може да се претпостави дека $f[a] \leq f[b]$ и $f[x] \leq f[y]$. Уште повеќе, $f[x] \leq f[a]$ и $f[y] \leq f[b]$, па ако ги смениме позициите на a и x ќе се добие ново дрво T', такво што разликата на цените меѓу дрвата T и T' е:

$$A[T] - A[T'] = \sum_{\sigma \in \Sigma} f[\sigma] \cdot d_T[\sigma] - \sum_{\sigma \in \Sigma} f[\sigma] \cdot d_{T'}[\sigma]$$

$$= f[x] \cdot d_T[x] + f[a] \cdot d_T[a] - f[x] \cdot d_{T'}[x] - f[a] \cdot d_{T'}[a]$$

$$= f[x] \cdot d_T[x] + f[a] \cdot d_T[a] - f[x] \cdot d_T[a] - f[a] \cdot d_T[x]$$

$$= (f[a] - f[x]) \cdot d_T[a] - (f[a] - f[x]) \cdot d_T[x]$$

$$= (f[a] - f[x])(d_T[a] - d_T[x]) \geq 0$$

Од претпоставката дека T е дрво за код со најмала цена, нееднаквоста мора да биде еднаквост, од каде следува дека новото дрво T' има иста цена со T, т.е. е друг код кој има најмала цена. Слично, ако ги смениме позициите на b и y во T' ќе се добие ново дрво T'', такво што разликата на цените меѓу дрвата T' и T'' е исто така поголема или еднаква на 0, со што и T'' е код со најмала цена, со што лемата е докажана. □

Според ова својство, при градењето на дрвото од одоздола па нагоре, комотно може да се избере алчниот избор и да се почне со знаците со најмала фреквенција. Она што се забележува дека до родителот на тие два знака во дрвото се стига секогаш кога ќе се стигне до некој од листовите кои ги репрезентираат тие два знака, па фреквенцијата на родителот е иста на фреквенцијата на тие два знака, а неговата длабочина е за еден помала. Оттука го имаме следново својство:

Нека Σ е азбука во која фреквенцијата на знакот σ од Σ е $f[\sigma]$, во која x и y се двата знака во Σ со најмала фреквенција. Нека Σ' се добива од Σ со замена на знаците x и y со нов знак z, кој претходно не е во Σ, т.е. $\Sigma' = (\Sigma \setminus \{x, y\}) \cup \{z\}$. Во Σ' се дефинира функција на фреквенција f' со:

$$f'[\sigma] = \begin{cases} f[\sigma], & \sigma \neq z \\ f[x] + f[y], & \sigma = z \end{cases}.$$

Тогаш

$$A[T'] = A[T] - f[z].$$

Имаме:

$$(\forall \sigma \in \Sigma\{x, y\}) d_T[\sigma] = d_{T'}[\sigma]$$

$$\Rightarrow (\forall \sigma \in \Sigma \setminus \{x, y\}) f[\sigma] \cdot d_T[\sigma] = f[\sigma] \cdot d_{T'}[\sigma].$$

Користејќи $d_T[x] = d_T[y] = d_{T'}[z] + 1$ се добива:

$$f[x] \cdot d_T[x] + f[y] \cdot d_T[y] = (f[x] + f[y])(d_{T'}[z] + 1)$$
$$= f[z] \cdot d_{T'}[z] + f[z],$$

од каде следува:

$$A[T] = A[T'] + f[x] + f[y],$$

со што својството е докажано. □

Останува уште да се покаже оптималната подструктура која ја поседува проблемот:

> *Нека Σ е азбука во која фреквенцијата на знакот σ од Σ е $f[\sigma]$, во која x и y се двата знака во Σ со најмала фреквенција. Нека Σ' се добива од Σ со замена на знаците x и y со нов знак z, кој претходно не е во Σ, т.е. $\Sigma' = (\Sigma \setminus \{x, y\}) \cup \{z\}$. Во Σ' се дефинира функција на фреквенција f' со*

$$f'[\sigma] = \begin{cases} f[\sigma], & \sigma \neq z \\ f[x] + f[y], & \sigma = z \end{cases}.$$

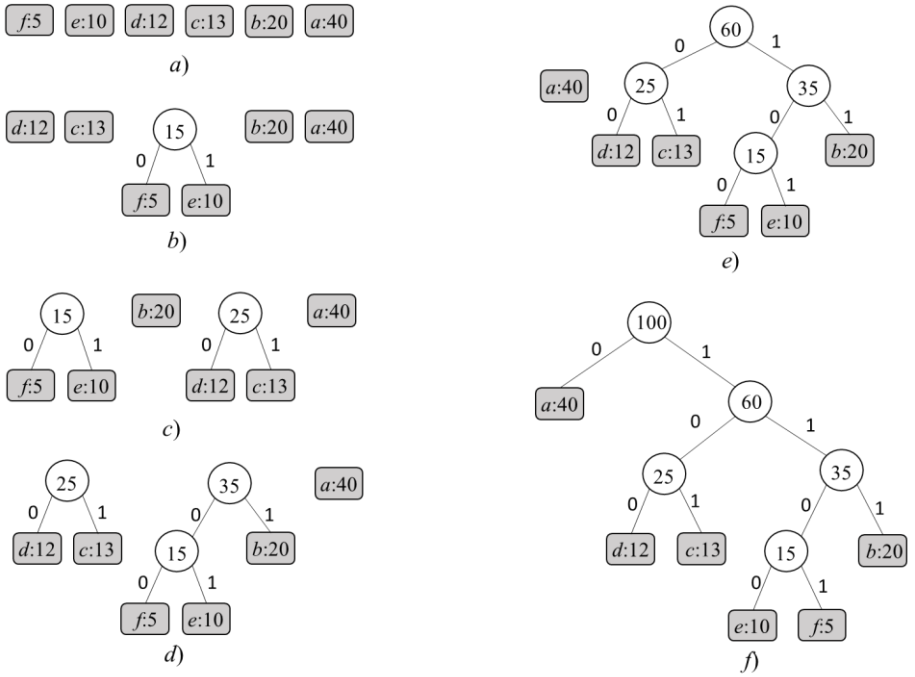
Доказ на оптималното својство: Ако T' е некое кое дрво кое репрезентира оптимален префиксен код за азбуката Σ' со функција на фреквенција f', тогаш дрвото T што се добива од T' со заменување на листот z со внатрешен јазол кој ги има како деца x и y, е дрво соодветно за оптимален префиксен код за азбуката Σ со функција на фреквенција f.

Ако T не е дрво кое репрезентира некој оптимален префиксен код за Σ, тогаш постои дрво T'' такво што $A[T''] < A[T]$. Од претходното својство, без губење на општост, може да претпоставиме дека листови со најголема длабочина во T'' се x и y. Нека T''' е дрвото кое се добива од T'' во кое заедничкиот родител на x и y се заменува со z со фреквенција $f[z] = f[x] + f[y]$. Тогаш

$$A[T'''] = A[T''] - f[x] - f[y] < A[T] = A[T] - f[x] - f[y] = A[T'],$$

што е контрадикција дека T'е соодветно за оптимален префиксен код за Σ'. Оттука, T мора да е соодветно дрво за оптимален префиксен код за Σ. □

Горното својство ни кажува дека ако го најдеме оптималното целосно бинарно дрво за Σ' со f', тогаш сме го нашле и за Σ со f. Треба само од темето z да формираме два листа за x и y. Основниот случај е кога имаме само два знака, од кои формираме дрво со висина 1, во кое тие две се сестрински темиња. Како и секој алгоритам од динамичко програмирање и овде дрвото го градиме одоздола нагоре.



Слика 6. 7. a) Почетно подредување на темињата по растечки редослед на фреквенција. b) Спојување на двата карактери со најмала фреквенција во едно родителско теме со фреквенција 15 и поместување на трета позиција. c) Буквите d и c добиваат заеднички родител со фреквенција 25 и новото теме се поместува на трета позиција. d) Буквите b, d и c се спојуваат во дрво со фреквенција 35. e) Дрвата

со фреквенција 25 и 35 добиваат заеднички родител со фреквенција 60. f) Оптималното дрво се добива со спојување на преостанатите две дрва.

Претпоставуваме дека Σ е множество од n знаци и секој знак σ од Σ е објект со дефинирана фреквенција $f(\sigma)$. Со алгоритамот се гради дрво T соодветно на оптималниот код, и дрвото се гради од оздола нагоре. Се започнува со множество од $|\Sigma|$ листови и за да се креира целосното дрво се прават $|\Sigma| - 1$ операции "спојување". Како резултат на спојување на два објекта се добива нов објект чија фреквенција е збир од фреквенциите на двата објекта кои се спојуваат. Ова чекор по чекор е претставено на Слика 6. 7. Во првиот чекор сите карактери се подредуваат во растечки редослед. Потоа во секој чекор првите два јазли се спојуваат во еден родителски јазол, кој како фреквенција ја добива сумата на фреквенциите на овие два јазли. Така добиеното теме се преместува во низата, така да низата остане подредена по растечки редослед. Ова се прави се додека сите темиња не се спојат во едно дрво. На крај се добива дрвото на Слика 6. 7., што е оптималното дрво кое има тежина 235000. Можеме да забележиме дека дрвото кое го добивме со алгоритамот не е истото од Слика 6. 6, но и двете се оптимални дрва. Имено структурата на дрвата е иста, само се променети местата на дел од темињата, но само на темињата на иста висина. Тоа покажува дека не мора да имаме единствено оптимално дрво, односно постојат и оптимални дрва кои не се добиваат со овој алгоритам.

Во алгоритамот во секој чекор се прави модификација на низата со која таа треба да остане подредена по растечки редослед. Имено, постојано треба да се наоѓаат двата објекта со минимална фреквенција, или да се најде местото на новодобиеното теме. Операциите од овој тип заземаат линеарно време ако се работи со наивен алгоритам, затоа за да се забрза овој дел треба да се користи мин-приоритетна верига со клуч f. Ваквата структура троши константно време за да го најде минимумот во некоја низа, но за додавање на нов елемент и ажурирање на низата е потребно логаритамско време. Во нашиот алгоритам ние во секој чекор ги наоѓаме двата најмали елементи, за што во веќе ажурирана верига троши константно време. За ова користиме функција $IzvadiMin(Q)$, која во скоро сите јазици може да се користи како готова функција. Потоа новодобиеното теме го вметнуваме во веригата, со функција

$Vnesi(Q, z)$, после што таа треба да се ажурира. За ова не треба да се грижиме затоа што во скоро секој јазик постои функција за внесување на елемент во верига, и самата функција автоматски ја ажурира веригата. Но треба да се води сметка дека таа една линија код не зазема константно, туку логаритамско време.

Псевдокодот е следниов:

П 6. 5. ХАФМАНОВ КОД

```
1  Внеси ја Σ и f[σ].
2  n = |Σ|
3  Q = Σ;
4  за i од 1 до n − 1 прави
5     {
6       levo[z] = x = IzvadiMin(Q);
7       desno [z] = y = IzvadiMin(Q\{x});
8       f [z] = f [x] + f [y];
9       Vnesi(Q, z, f [z]).
10    }
11 врати IzvadiMin(Q)
```

Со горниот псевдокод се гради бинарно дрво, а со последната наредба се враќа дрвото. Врз основа на дрвото може да се изгради префиксниот код.

Сложеноста на овој алгоритам зависи од структурата која ќе ја користиме за зачувување на минимумот во даден момент. Подобра имплементација на веригата е со бинарен куп, кој зазема време 1 за барање на минималниот елемент и време $O(\ln n)$ за ажурирање на купот. За множество C од n знаци, за иницијализацијата на Q се троши време $O(n)$. При секое влегување во циклусот кој се извршува $n - 1$ пати, за ажурирање на веригата се троши време $O(\ln n)$. Обично ова се случува во чекорот каде се внесува елемент во веригата. Оттука, времето на работа на алгоритамот е $O(n \ln n)$.

Кодовите во Јава и C++ кои ги даваме овде се итеративни решенија кои се базираат на третиот пристап.

ЈАVA 6. 3 ХАФМАНОВ КОД

```
1 import java.util.*;
2 import java.util.PriorityQueue;
3
4 public class Main {
5
6     public static class Elem implements Comparable<Elem> {
7         public String c;
8         public int f;
9         public Elem levo;
10        public Elem desno;
11
12        public Elem(String c, int f) {
13            this.c = c;
14            this.f = f;
15        }
16
17        public Elem(String c, int f, Elem levo, Elem desno)
{
18            this.c = c;
19            this.f = f;
20            this.levo = levo;
21            this.desno = desno;
22        }
23
24        @Override
25        public int compareTo(Elem elem) {
26            if (this.f > elem.f) {
27                return 1;
28            } else if (this.f < elem.f) {
29                return -1;
30            } else {
31                return 0;
32            }
33        }
34
35        @Override
36        public String toString() {
```

```
37              return "(" + this.c + ", " + this.f + ")";
38          }
39      }
40
41      public static void main(String[] args) {
42          Scanner scn = new Scanner(System.in);
43          int n = scn.nextInt();
44          String[] E = new String[n];
45          int[] f = new int[n];
46
47          for (int i = 0; i < n; i++)
48              E[i] = scn.next();
49
50          for (int i = 0; i < n; i++)
51              f[i] = scn.nextInt();
52
53          PriorityQueue<Elem> Q = new PriorityQueue<>();
54          for (int i = 0; i < n; i++)
55              Q.add(new Elem(E[i], f[i]));
56
57          for (int i = 0; i < n; i++) {
58              Elem x = Q.remove();
59              Elem y = Q.remove();
60
61              System.out.println(x + " " + y);
62
63              int fz = x.f + y.f;
64              String Ez = x.c + y.c;
65              Q.add(new Elem(Ez, fz, x, y));
66          }
67      }
68}
```

Во двата кода се користи мин-верига како структура за зачувување на низата од фреквенции. Кодот во C++ е:

C++ 6. 3 ХАФМАНОВ КОД

```
#include<iostream>
#include <queue>
#include <vector>
using namespace std;

struct Elem
{
   string c;
   int f;
   Elem* levo;
   Elem* desno;
};

struct Comp{
   bool operator()(const Elem& a, const Elem& b){
      return a.f > b.f;
   }
};

int main() {
   int n;
   cin >> n;
   string E [n];
   int f [n];

   for(int i=0;i<n;i++)
      cin >> E[i];

   for(int i=0;i<n;i++)
      cin >> f[i];

   priority_queue<Elem, vector<Elem>, Comp> Q;

   for(int i=0;i<n;i++)
   {
      Elem el = {E[i], f[i]};
      Q.push(el);
```

```
38     }
39
40     for(int i=0;i<n;i++){
41        Elem x = Q.top(); Q.pop();
42        Elem y = Q.top(); Q.pop();
43
44        cout << "(" << x.c << ", " << x.f << ")";
45        cout << "(" << y.c << ", " << y.f << ")" << endl;
46        int fz = x.f + y.f;
47        string Ez = x.c+y.c;
48        Elem el = {Ez, fz, &x, &y};
49        Q.push(el);
50     }
51     return 0;
52 }
```

Прашања и задачи

1. Дадете ги чекорите со кои можете да ја пресметате големината на кодираниот текст за дадена порака!
2. Како ќе пресметаш колку битови ќе се заштедат ако за дадена порака се искористи Хафманов код наместо обичен код во кој секој карактер ќе се кодира со битстринг со еднаква должина?
3. Колкава е висината на дрво кое одговара на низа на Фибоначи со должина n?
4. Направи алгоритам кој даден текст ќе го кодира според даден префиксен код.
5. Дади псевдокод за алгоритам кој дадена кодирана низа според некој даден префиксен код ќе го дешифрира текстот.
6. Дизајнирај алгоритам кој даден за даден префиксен код ќе дешифрира даден текст.

7 Алгоритми од графови

Многу проблеми од различни дисциплини можат да се претстават со помош на графови. Еден од најочигледните проблеми кои се моделираат со графови се секако патните карти. Градовите, селата и раскрсниците на картата може да се прикажат како точки, што претставуваат темиња на графот, додека патиштата може да се прикажат со линии кои ги поврзуваат точките. Ако сакаме да прикажеме колкава е должината на еден така прикажан сегмент пат од едно до друго место, тоа можеме да го прикажеме со запишување на број кој над секој сегмент од пат кој ќе соодветствува на должината на тој дел. Така, ако сакаме да патуваме од едно до друго место, тоа најчесто сакаме да го направиме по најкраткиот можен пат. Проблемот на наоѓање на најкраткиот можен пат е еден од најважните проблеми од оптимизација и неговото решение се базира динамичко програмирање, затоа тоа е еден од проблемите за кои ќе се зборува во оваа глава. Но, освен за прикажување на карти, со графови можат да се визуелизираат и проблеми од друга природа. Да го разгледаме проблемот на поставување на линкови на интернет. Од една веб страна обично се поставуваат повеќе врски кон надворешни или внатрешни страници, па и овој проблем може да се моделира со помош на графови на тој начин што секоја веб страна ќе претставува теме, а секој линк од да речеме страната А до страната В ќе биде ребро. Потоа можеме да испитуваме кои се оние страници кои се повикуваат една со друга, колкав е најмалиот број на последователни страници кои е потребно да се посетат за да од одредена страна стигнеме до некоја друга и слично.

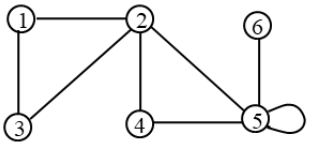
Во првата глава ги дадовме основните термини и својства кај графови и дрва, а во оваа глава на почеток ќе се осврнеме на начините на претставување на графовите и структурите на податоци кои за тоа можеме да ги користиме. Потоа ќе дадеме неколку

познати алгоритми од динамичко програмирање и алчни алгоритми за најкраток пат и минимално скелетно дрво.

7.1 Репрезентација на графови

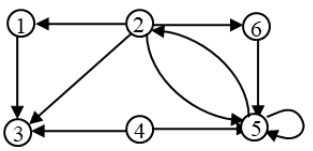
За воопшто да можеме да работиме со графови, потребно е да најдеме начин како истите ќе ги претставиме преку код. Има два основни начини на презентација на графови, со помош на **листи на соседство** и со помош на **матрици на соседство**. Која репрезентација ќе се користи зависи најмногу од видот на проблемот, но и од самата природа на графот над кој што работи алгоритамот. Секако целта е секогаш да се избере најдобрата можна репрезентација за да се добие што е можно поефикасен алгоритам. Различна репрезентација може да даде различна ефикасност на алгоритамот, како во брзина, така и во мемориски простор, и многу често зависи од бројот на ребра во зависност од бројот на темиња кои ги има графот. Затоа, во зависност од бројот на ребра во графот, графовите ги делиме на густи и ретки графови. За еден граф велиме дека е **густ** ако бројот на ребра е близу до квадратот на бројот на темиња, односно $|E|$ е близу до $|V|^2$. **Ретки** графови се оние графови за кои $|E|$ е многу помало од $|V|^2$.

И двата стандардни начини на репрезентирање на графови, листи и матрици на соседство, можат да се користат и за ориентирани и за неориентирани графови.



1	2, 3
2	1, 3, 4, 5
3	1, 2
4	2, 5
5	2, 4, 5, 6
6	5

a)



1	3
2	1, 3, 5, 6
3	
4	3, 5
5	2, 5
6	5

b)

Слика 7. 1. a) Листи на соседство на неориентиран граф. b) Листи на соседство на ориентиран граф.

Презентацијата со помош на ***листа на соседство*** на графот $G = (V, E)$ се состои од една низа која можеме да ја наречеме Adj, од листи од $|V|$, по една за секое теме во V. За секој $u \in V$, листата на соседство, $Adj[u]$ се состои од сите темиња *v* такви што постои ребро (u, v) во E, односно

$$Adj[u] = \{v \mid (u, v) \in E\}.$$

На Слика 7. 1. се дадени листи на соседство на еден ориентиран и еден неориентиран граф.

При репрезентацијата на графот $G = (V, E)$ со помош на ***матрица на соседство***, претпоставуваме дека темињата се нумерирани со $1, 2, \ldots, |V|$ во произволен редослед. При овој начин на репрезентација се прави од $|V| \times |V|$ матрица $M = [m_{ij}]$ таква што

$$m_{ij} = \begin{cases} 1, & \text{ако } (i,j) \in E \\ 0, & \text{ако } (i,j) \notin E \end{cases}$$

Во неориентиран граф $M = M^T$. Матриците на соседство за графовите на Слика 7. 1. се дадени на Слика 7. 2.

	1	2	3	4	5	6
1	0	1	1	0	0	0
2	1	0	1	1	1	0
3	1	1	0	0	0	0
4	0	1	0	0	1	0
5	0	1	0	1	1	1
6	0	0	0	0	0	1

a)

	1	2	3	4	5	6
1	0	0	1	0	0	0
2	1	0	1	0	1	1
3	0	0	0	0	0	0
4	0	0	1	0	1	0
5	0	1	0	0	1	0
6	0	0	0	0	0	1

b)

Слика 7. 2. а) Листи на соседство на неориентираниот граф од Слика 7.1.1 а). b) Листи на соседство на ориентираниот граф од Слика 7.1.1 b)

Прашањето е кој начин на репрезентација на влезните графови е подобар? Вообичаено се преферира претставување со листа на соседство, бидејќи обезбедува покомпактен начин за презентирање на ретки графови, односно за неа ни е потребна помала меморија. За примерот претставен на Слика 7. 1 *а*), за претставувањето на ориентираниот граф, за листата на соседство ни требаше да мемориграме 20 податоци, додека за матрицата на соседство ни беше потребно да мемориграме 36 податоци. Во секоја листа на соседство темињата обично се зачувуваат по случаен редослед. Така, во општ случај, за да еден неориентиран граф $G = (V, E)$ да се претстави со листи на соседство ни треба по еден податок за обележување на секоја листа, а тоа е бројот на темиња и по два за секое ребро, значи вкупно $|V| + 2|E|$ податоци. Ако $G = (V, E)$ е ориентиран граф, тогаш повторно ни треба по еден податок за обележување на секоја листа, а тоа е бројот на темиња, додека за сумата од должините на сите листи на соседство е $|E|$. И во двата случаи, преставувањето со помош на листа на соседство го има саканото својство дека потребната меморија е $\Theta(|V| + |E|)$. Од друга страна за матрицата на соседство и во двата случаи ни требаат $|V|^2$ податоци. Заради ова матрици на соседство се користат кога графот е густ. Но, од друга страна во матрицата на соседство се стават само 1-ци и 0-ли, па тоа можеме да го гледаме и како дополнителна

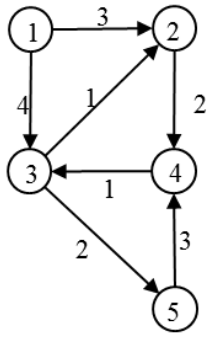
предност, затоа што наместо за секој влез на матрицата да се користи еден збор во меморијата на компјутерот, се користи еден бит по влез.

Секако ние начинот го одбираме не само според меморијата која и е потребна за да ги зачуваме податоците, туку и брзината со која можеме да достапиме до податоците. Затоа, на прв поглед матрицата на соседство ни изгледа дека за оваа намена ни е подобра варијанта, бидејќи бргу можеме да кажеме дали две темиња се поврзани со ребро или не. За оваа операција, кога сакаме да најдеме дали во графот постои ребро (u, v) , ако користиме листи на соседство ќе треба да ја пребараме целата листа на соседство на u, се додека не го најдеме v. Ако го најдеме, тогаш можеме да престанеме со пребарување и да дадеме одговор дека реброто постои, но ако не го најдеме, тогаш треба да ја пребараме целата листа. Така, за оваа операција има сложеност $O(Adj[u]) \leq O(|V|)$. Но, како што ќе видиме понатаму, во алгоритмите кои решаваат проблеми со помош на графови претежно се користи операцијата да се најде дали постои ребро кое излегува од одредено теме или да се земе едно теме од листата на соседство на определено теме, а не да се најде дали постои ребро меѓу две темиња. Овие операции е повторно се побрзи ако се користи листи на соседство, затоа што во тој случај само треба да видиме дали листата на соседство на некој елемент е празна или не или само да го земеме првиот елемент од таа листа. За разлика од ова, ваквата операција при работа со матрици на соседство мора да се изведе со пребарување на целата редица соодветна на даденото теме, што сега ќе има сложеност $\Theta(|V|)$.

И листите и матриците на соседство можат да се искористат и за претставување на мултиграфови. Ако во графот имаме повеќе ребра кои поврзуваат исти темиња, тогаш тоа со матрица на соседство можеме да го прикажеме на тој начин што во матрицата наместо 1–ца ќе го ставиме бројот на такви ребра. За да прикажеме мултиграф со листи на соседство ќе треба на секој елемент во листата на соседство да му се придружи бројот на такви ребра. Во секој случај сложеноста на прикажување повторно останува иста.

Често во реални проблеми се сретнуваат графови кај кои на секое ребро му е придружен реален број што може да претставува:

растојание, цена, проток, профит итн. Таквите графови ги нарекуваме ***тежински графови***. Всушност тежински граф е граф на кој му се придружува така наречена функција на тежина w: $E \rightarrow \boldsymbol{R}$. Тежината може да репрезентира повеќе атрибути, должина реброто, цена на врската, време, капацитет и слично.



a)

1	(2, 3), (3, 4)
2	(4, 2)
3	(2, 1), (5, 2)
4	(3, 1)
5	(4, 3)

b)

	1	2	3	4	5
1	0	2	4	0	0
2	0	0	0	2	0
3	0	1	0	0	2
4	0	0	1	0	0
5	0	0	0	3	0

c)

Слика 7. 3. a) Тежински граф. b) Претставување на тежински граф со листи на соседство. c)Претставување на тежински граф со матрица на соседство.

И листите и матриците на соседство лесно можат да бидат адаптирани за претставување на тежински графови, слично како што можат да се употребат за мулти – графови, Слика 7. 3. При претставувањето со помош на листи на соседство, тежината на реброто (u, v) се зачувува заедно со темето *v* во листата на соседство на u, додека при претставувањето со помош на матрица на соседство, тежината на реброто се зачувува како влез во u-тата редица и v-тата колона од матрицата на соседство. Ако некое ребро не постои, на соодветниот влез во матрицата, може да се зачува вредност NIL, иако во некои проблеми е погодно да се користи вредност како 0 или ∞.

Прашања и задачи

1. Да се напише псевдокод кој од листа на соседство прави матрица на соседство. Која е сложеноста на овој алгоритам?
2. Да се напише псевдокод кој од матрица на соседство прави листа на соседство. Која е сложеноста на овој алгоритам?
3. Да се напише псевдокод кој од низа ребра прави листа на соседство. Која е сложеноста на овој алгоритам?
4. Да се напише код кој од листа на ребра прави матрица на соседство. Која е сложеноста на овој алгоритам?
5. Да се напише код кој од низа насочени ребра која претставува кореново дрво, ја прави функцијата родител на кореново дрво! Која е сложеноста на овој алгоритам?
6. Колкава е сложеноста на алгоритам кој го пресметува бројот на ребра во граф даден со листа на соседство?
7. Колкава е сложеноста на алгоритам кој го пресметува степенот на секое теме во граф зададен со листа на соседство?
8. Колкава е сложеноста на алгоритам кој го пресметува бројот на ребра во граф даден со матрица на соседство?
9. Колкава е сложеноста на алгоритам кој го пресметува степенот на секое теме во граф зададен со матрица на соседство?

7.2 Најкраток пат во граф

Како што кажавме претходно, графовите често се користат за со нив да се претстават патни карти, па основен проблем поврзан со графови е проблемот за наоѓање на најкраток пат. Еден начин е да се пресмета должината на сите можни патишта и да се најде кој од нив е најкраток, но тоа зафаќа големо време, а ако постојат циклуси, ќе имаме бесконечно многу можни рути.

Во ваков проблем за наоѓање на најкраток пат, секое ребро во графот има своја должина, па затоа истиот може да се визуализира со тежински граф, во кој што тежината на реброто претставува должина на истото. Графот може да биде и ориентиран и неориентиран во зависност од проблемот.

Формално, за даден граф $G = (V, E)$, со функција на тежина $w: E \to \boldsymbol{R}$, должина на патот $p = \langle v_0, v_1, v_2, \ldots, v_k \rangle$ се дефинира како сума од должините на нивните темиња:

$$l(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i).$$

Дефинираме должина на најкраткиот пат од u до v со:

$$\delta(u, v) = \begin{cases} \min\left\{ l(p) \middle| u \xrightarrow{p} v \right\}, & \text{постои пат од } u \text{ до } v \\ \infty, & \text{инаку} \end{cases}.$$

Најкраток пат

Најкраток пат од темето u до темето v е било кој пат p со тежина $l(p) = w(u, v)$.

Има повеќе варијанти на овој проблем:

Најкратки патишта од еден извор или до еден сливник:

Да се најдат должините на сите најкратки патишта од дадено теме u до сите останати темиња, или да се најдат должините на сите најкратки патишта од секое теме до дадено теме u. Дополнително да се реконструира еден таков пат.

Најкраток пат за даден пар темиња:

Да се најде должината на најкраткиот пат од дадено теме u до дадено теме v и да се најде еден таков пат.

Иако ова изгледа дека е полесен проблем, нема алгоритам кој работи асимптотски побргу од проблемот за најкраток пат до една дестинација.

Најкраток пат меѓу сите парови:

Да се најдат должините на најкратките патишта за секој пар темиња u и v.

Оптималното својство кое го има овој проблем е дека најкраткиот пат меѓу две темиња се состои од најкратки патишта меѓу темињата кои лежат на тој пат, односно дека потпатиштата на најкраток пат се најкратки патишта. Според ова тука може да се примени динамичко програмирање. Формално искажано својството е следново:

Нека за даден тежински ориентиран граф $G = (V, E)$ над кој е дефинирана функција на тежина $w: E \to \boldsymbol{R}$, патот $p = \langle v_0, v_1, v_2, \dots, v_k \rangle$ е најкраток пат од темето v_1 до темето v_k. Нека за сите i и j такви што $1 \leq i \leq j \leq k$, $p_{ij} = \langle v_i, v_{i+1}, \dots, v_j \rangle$ се потпатишта на p од темето v_i до темето v_j. Тогаш, p_{ij} е најкраток пат од v_i до v_j.

Доказ на оптималното својство: Да го декомпонираме патот p во

$$v_1 \xrightarrow{p_{1i}} v_i \xrightarrow{p_{ij}} v_j \xrightarrow{p_{jk}} v_k.$$

Сега неговата должина може да се запише како:

$$l(p) = l(p_{1i}) + l(p_{ij}) + l(p_{jk}).$$

Да претпоставиме дека постои пократок пат p'_{ij} од v_i до v_j.од патот p_{ij} со тежина $l\left(p'_{ij}\right) < l(p_{ij})$. Тогаш, патот

$$v_1 \xrightarrow{p_{1i}} v_i \xrightarrow{p'_{ij}} v_j \xrightarrow{p_{jk}} v_k$$

има должина

$$l(p_{1i}) + l(p'_{ij}) + l(p_{jk}) < l(p),$$

што е пат со помала должина од најкраткиот пат од v_1 до v_k. Со ова се добива контрадикција со претпоставката дека p е најкраткиот пат од v_1 до v_k.

Прашања и задачи

1. Дали е можно најкраткиот пат во граф со ненегативна функција на тежина да содржи циклус?

2. Дали е можно во графот да постои негативен циклус?

3. Ако во графот постои циклус со тежина -1 на кој припаѓа темето u. Кој е најкраткиот пат од темето u до темето u?

4. Ако не постои пат од едно до друго теме, тогаш ќе сметаме дека патот меѓу нив е со должина ∞. Како можеме да ја модифицираме функцијата на тежина за да секој пат меѓу темиња меѓу кои не постои пат има тежина ∞?

5. Нека е даден тежински граф $G = (V, E)$ со функција на тежина $w: E \rightarrow \boldsymbol{R}$, нека $\delta(i, k)$ е најкраткиот пат од темето i до темето j, а $\delta(j, k)$ е најкраткиот пат од темето j до темето k. Да се докаже неравенството на триаголник:

   $$\delta(i, k) \leq \delta(i, j) + w(j, k),$$

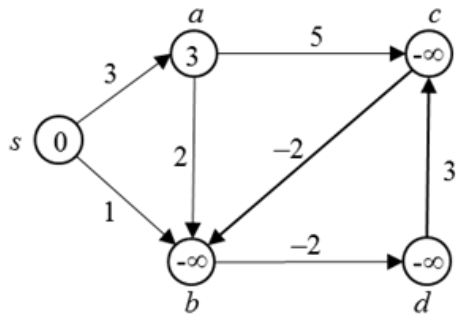
   каде $\delta(u, v) = \infty$ ако од u до v не постои пат.

7.3. Најкраток пат во ориентиран граф со негативни ребра и алгоритам на Белман-Форд

Некои реални ситуации на кои решението е наоѓање на најкраток пат, може да се моделираат и со графови со негативни ребра. Стандарден алгоритам кој го решава овој проблем е алгоритамот познат како алгоритам на Белман-Форд кој го даваме во ова поглавје. Алгоритамот првично бил предложен од Алфонско [18], но именуван е по Ричард Белман и Лестер Форд кои го имаат публикувано нешто подоцна, [19].

Дефинирање на проблемот

За даден ориентиран тежински граф *G*(*V, E*) кај кој ребрата можат да имаат негативни тежини, треба да се најде најкраток пат од едно теме s. Во ваков граф може да има проблем на некој пат од u до v има негатевен циклус, бидејќи тогаш со движење по тој циклус патот може да станува се помал и помал, па дефинираме дека најкраткиот пат меѓу тие две темиња е $\delta(u, v) = -\infty$. На пример графот на Слика 7.2.1. содржи негативен циклус $\langle b, d, c, b\rangle$ со должина – 1, па до темето b можеме да стигнеме преку едно ребро (s, b) кое има должина 1, потоа ако го свртиме циклусот ќе добиеме пат со должина 0, ако пак го свртиме должината ќе се намали на – 1 и со секое наредно поминување на циклусот добиваме се помала и помала вредност. Ако свртуваме по циклусот бесконечно многу пати, таа вредност ќе стане $-\infty$. Во ваква ситуација ќе сметаме дека до темето b нема најкраток пат. Но истиот ефект не се јавува за темето a, затоа што нема пат од s до a кој содржи негативен циклус, па најкраткиот пат до a е 3.



Слика 7. 4. Граф во кој има негативен циклус.

Целта е да се креира алгоритам кој ќе го даде најкраткиот пат од изворот до сите останати темиња, ако тој пат не е $-\infty$, а ако има циклус дофатлив од изворот, истиот да го детектира.

Анализа на проблемот

Ако графот има само ребра со позитивни тежини, тогаш е јасно дека секој најкраток пат меѓу две темиња не може да содржи циклус, бидејќи со отстранување на истиот ќе се добие пократок пат. Од друга страна ако во графот има ребра со негативни тежини, но нема ниту еден циклус со негативна должина, тогаш сигурно ќе постои најкраток пат кој не содржи циклус до секое од темињата. Имено ако на најкраткиот пат има циклус, тогаш него можеме да го отстраниме, и бидејќи тој нема негативна должина, мора неговата должина да е 0, бидејќи инаку би добиле пократок пат што е контрадикција. Затоа можеме да претпоставиме дека ако постои најкраток пат, тогаш постои ацикличен најкраток пат, т.е. најкраток пат кој не содржи циклус. Тоа пак значи дека таквиот ацикличен најкраток пат во најлош случај може да помине низ сите темиња точно по еднаш, или истиот ќе има најмногу $|V| - 1$ ребро. Според тоа го имаме следново својство:

Својство

Ако од u до v постои најкраток пат, тогаш постои и најкраток пат со најмногу $|V| - 1$ ребро.

Врз основа на ова својство и својството на оптималност, можеме да изведеме рекурзивна формула по бројот на ребра кои ги

има во најкраткиот пат. Нека $A[u, i]$ е најкраткиот пат од изворот s до темето u со најмногу i ребра. Тогаш:

- Ако најкраткиот пат има помалку од i ребра, тогаш најкраткиот пат со најмногу i ребра е ист со најкраткиот пат со најмногу со $i - 1$ –но ребро, т.е.

$$A[u, i] = A[u, i - 1].$$

- Ако пак најкраткиот пат има точно i ребра и претпоследното теме на најкраткиот пат е v, тогаш до темето v има најкраток пат со $i - 1$ –но ребро и важи:

$$A[u, i] = A[v, i - 1] + w(v, u).$$

Почетните услови се во случај кога имаме пат со должина 0. Во таква ситуација единствено најкраткиот пат до изворот е 0, додека до сите останати темиња е ∞. Решението кое го бараме е кога $i = |V| - 1$, бидејќи ако постои најкраток пат не може да има повеќе од $|V| - 1$-но ребро. Целата рекурзија е следнава:

$$A[u, i] = \begin{cases} 0, & i = 0 \text{ и } u = s \\ \infty, & i = 0 \text{ и } u \neq s \\ \min\left\{A[u, i-1], \min\limits_{(v,u)\in E}\{A[v, i-1] + w(v, u)\}\right\}, & \text{инаку} \end{cases}.$$

Врз основа на оваа рекурзивна равенка, може да се направи следниов псевдокод кој го дава точното решение во случај кога во графот нема негативен циклус:

П 7. 1. НАЈКРАТОК ПАТ ВО ГРАФ СО НЕГАТИВНИ РЕБРА ВО КОЈ НЕМА НЕГАТИВЕН ЦИКЛУС

```
1  Внеси ги V и E;
2  за секое u ∈ V прави A[u, 0] = ∞;
3  A[s, 0] = 0;
4  за i од 1 до |V| − 1 прави
5      за секое u ∈ V прави
```

```
6       {
7        A[u, i] = A[u, i − 1];
8       за секое ребро (v, u) ∈ E;
9       ако A[u, i] > A[v, i − 1] + w(v, u) тогаш
10                              A[u, i] = A[v, i − 1] + w(v, u).
11      }
```

Што ќе се случи кога во графот ќе има негативен циклус? Со оваа процедура ќе се пресметаат одредени вредности за $A[u, |V| - 1]$. Ако нема негативни циклуси тогаш ова би бил најкраткиот пат, па ако го пресметаме $A[u, |V|]$ за секое теме тогаш ниту една од вредностите нема да се промени. Но ако има негативен циклус, тогаш ќе постои барем едно теме на кое патот со должина $|V|$ ќе му биде пократок од патот со должина $|V| - 1$. Ова е сумирано во следново својство:

Својство

Во графот постои негативен циклус дофатлив од изворот s ако и само ако постои теме u такво што

$$A[u, |V|] < A[u, |V| - 1].$$

Доказ: Прво да ја докажеме насоката дека ако постои теме u такво што $A[u, |V|] < A[u, |V| - 1]$, тогаш во графот има негативен циклус. Да увидиме дека најкраткиот пат со најмногу $|V|$ ребра мора да има точно $|V|$ ребра, зошто во спротивно тој би имал иста должина со најкраткиот пат со најмногу $|V| - 1$-но ребро, односно не би се разликувал од него. Поради тоа овој пат има циклус. Ќе покажеме дека тој циклус мора да биде негативен. Да претпоставиме дека циклусот $\langle u = v_0, v_1, v_2, \dots, v_{k-1}, v_k = u \rangle$ има должина k. Тогаш до u има и потпат од циклусот со должина $|V| - k$, и тој потпат е најкраток пат до u со должина $|V| - k$ и уште повеќе $A[u, |V| - k] > A[u, |V|]$. Тогаш:

$$A[u,|V|] = A[u,|V|-k] + \sum_{j=1}^{k} w(v_{j-1}, v_j)$$

$$> A[u,|V|] + \sum_{j=1}^{k} w(v_{j-1}, v_j),$$

Од каде следува дека

$$0 > \sum_{j=1}^{k} w(v_{j-1}, v_j),$$

што значи дека циклусот е негативен.

Обратно, да претпоставиме дека за секое теме u, $A[u,|V|] \geq A[u,|V|-1]$ и дека во графот постои негативен циклус. Нека негативниот циклус е $\langle v_0, v_1, v_2, \dots, v_{k-1}, v_k = v_0 \rangle$. Тогаш:

$$\sum_{j=1}^{k} A\big[v_j, |V|-1\big] \leq \sum_{j=1}^{k} A\big[v_j, |V|\big]$$

$$= \sum_{j=1}^{k} \Big(A\big[v_{j-1}, |V|-1\big] + w\big(v_{j-1}, v_j\big)\Big)$$

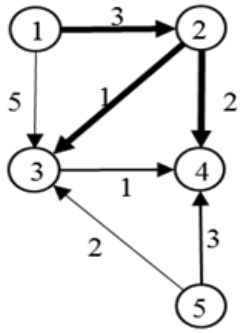
$$= \sum_{j=0}^{k-1} A\big[v_j, |V|-1\big] + \sum_{j=1}^{k} w(v_{j-1}, v_j)$$

Од каде следува дека

$$0 \leq \sum_{j=1}^{k} w(v_{j-1}, v_j),$$

што е контрадикција дека циклусот е негативен. □

Оттука, за да провериме дали во графот има негативен циклус дофатлив од изворот, треба да го пресметаме $A[u, |V|]$ за секое теме и ако има барем една промена на вредностите тогаш ќе заклучиме дека има негативен циклус. На ова се базира алгоритамот на Белман-Форд.



$\pi[1] = \text{NIL}$
$\pi[2] = 1$
$\pi[3] = 2$
$\pi[4] = 2$
$\pi[5] = \text{NIL}$

Слика 7. 5. Граф од претходници и функцијата π во која се зачувува истиот.

Во алгоритамот на Белман-Форд важна е уште една опсервација. Имено може да се забележи дека проверката дали $A[u, i] > A[v, i-1] + w(v, u)$ не мора да се прави по ред за секое теме, туку само е битно во секое поминување на циклусот по i да се направи по еднаш. Затоа внатрешните два циклуси во алгоритмот може да се заменат со еден циклус кој ќе оди по секое ребро. Уште повеќе, нема потреба функцијата *A* да зависи од параметарот i Ова нема да ја намали сложеноста на алгоритамот, туку само ќе го поедностави кодот.

Пред да го дадеме кодот на алгоритамот на Белман-Форд, да видиме како може да се реконструира еден најкраток пат. Да претпоставиме дека сме го нашле најкраткиот пат меѓу две темиња и можеме на рака да го нацртаме на графот. На пример на Слика 7. 5. со здебелени ребра се прикажани најкратките патишта од темето 1 до темињата 2, 3 и 4. До темето 5 нема воопшто пат, па најкраткиот пат можеме да сметаме дека е ∞. На цртежот можеме да оставиме трага со здебелување на ребрата и со движење по нив да го најдеме патот што ни треба, но истиот тоа треба да се направи и во нашата програма, односно за да реконструираме еден најкраток пат, во текот

на алгоритамот треба да оставиме трага која ќе ни помогне околу реконструкцијата. Оттука се поставува прашањето каква техника да се искористи за ова. Можеме да забележиме дека од здебелените ребра може да се формира дрво и ако по тоа дрво се движиме од изворот до некое достигливо теме, тогаш тоа е најкраткиот пат до тоа теме. Да видиме како би го нашле најкраткиот пат од 1 до 4. Прво, јасно треба да одиме по реброто (1, 2), но понатаму не знаеме дали треба да се придвижиме по реброто (2, 3) или по (2, 4). При вакви разгранувања, ако тргаме од изворот треба да пребаруваме по секоја гранка се додека не стигнеме до бараното теме. Но што ако тргнеме наопаку, односно да видиме кое теме е претходник на 4, тоа е темето 2, а претходник на 2 е 1. Така лесно и веднаш ќе го реконструираме патот но во обратна насока, па на ова се темели идејата за реконструкција. Од друга страна, ако не постои најкраток пат како за темето 5, тогаш тоа веднаш може да биде увидено или од фактот дека најкраткиот пат до него е бескрај или затоа што нема ребро од ова дрво кое влегува во него.

За реконструкција на најкраток пат најдобро е да се искористи една структура наречена граф од претходници. Графот од претходници за даден извор s се состои од едно дрво од најкратки патишта, во кое се сите темиња достигливи од изворот и изолирани темиња, кои се темињата кои не се достигливи од изворот. Дрвото од најкратки патишта е кореново дрво со корен во изворот s. Графот од претходници се зачувува на тој начин што на секое теме достигливо од изворот се памти родителот, а за изворот и темињата кои не се достигливи се чува NIL, во функција $\pi: V \rightarrow V \cup \{\text{NIL}\}$. Графот од претходници постепено се ажурира во текот на работата на алгоритамот. Кога алгоритамот ќе заврши со работа, за сите темиња достигливи од изворот во $\pi[v]$ се чува претходникот на v во најкраткиот пат од изворот, додека за самиот извор и темињата кои не се достигливи од изворот во $\pi[v]$ се чува NIL. Ова е илустрирано на Слика 7. 5.

Формално, за даден граф $G = (V, E)$, за секое теме v дефинираме претходник $\pi[v]$ кој е или некое друго теме, или NIL. Графот од претходници е $G_\pi = (V, E_\pi)$, каде

$$E_\pi = \{(\pi[v], v) \in E | \pi[v] \neq \text{NIL}\}.$$

Уште повеќе, ако постои пат од s до u во G_π, тогаш тоа е најкраткиот пат s до u во G, а ако не постои, тогаш $\pi[u] = \text{NIL}$. За графот на Слика 7. 5 множеството ребра во графот од претходници е: $E_\pi = \{(1,2),\ (2,3),\ (2,4)\}$.

Дрвото од најкратки патишта е подграфот од G_π над множеството темиња

$$V_\pi = \{v \in V | \pi[v] \neq \text{NIL}\} \cup \{s\}.$$

Овие атрибути се зачувуваат и се користат за печатење на едно оптимално решение. Така, најкраткиот пат од темето 1 до темето 4 на графот од Слика 7. 5, го печатиме во обратен редослед на следниов начин:

- печатиме 4;
- пресметуваме $\pi[4] = 2 \neq \text{NIL}$ и печатиме 2;
- пресметуваме $\pi[2] = 1 \neq \text{NIL}$ и печатиме 1;
- пресметуваме $\pi[1] = \text{NIL}$ и престануваме.

Псевдокодот на алгоритамот на Белман-Форд, заедно со графот со градбата на дрвото од претходници е следниов:

П 7. 2. БЕЛМАН ФОРД

```
1   Внеси ги V и E;
2   за секое u ∈ E прави
3     {
4       A[u] = ∞;
5       π[u] = NIL;
6     }
7     A[s] = 0;
8   за i од 1 до |V| − 1 прави
9     за секое ребро (v, u) ∈ E
10      ако A[u] > A[v] + w(v, u) тогаш
11            {
```

```
12              A[u,i] = A[v,i − 1] + w(v,u).
13              π[u] = v.
14            }
15  за секое ребро (v,u) ∈ E
16    ако A[u] > A[v] + w(v,u) тогаш печати T инаку печати F.
```

Ако алгоритамот отпечати F, тоа ќе значи дека во графот постои циклус со негативна тежина, а ако отпечати T, таков циклус не постои и добиените вредности $A[u]$ ќе ја даваат вредноста на најкраткиот пат, додека $\pi[u]$ ќе го дава претходникот на секое теме во графот од претходници. Оваа функција може да се искористи за да се реконструира еден најкраток пат или да се најде еден негативен циклус. Печатењето на најкраткиот пат може да се направи со следниов псевдокод:

П 7. 3. ПЕЧАТЕЊЕ НА НАЈКРАТКИОТ ПАТ

```
1  Внеси го u;
2  ако u = s тогаш печати u инаку
3      ако π[u] = NIL тогаш печати „нема пат“ инаку
4          додека π[u] ≠ NIL прави печати u = π[u].
```

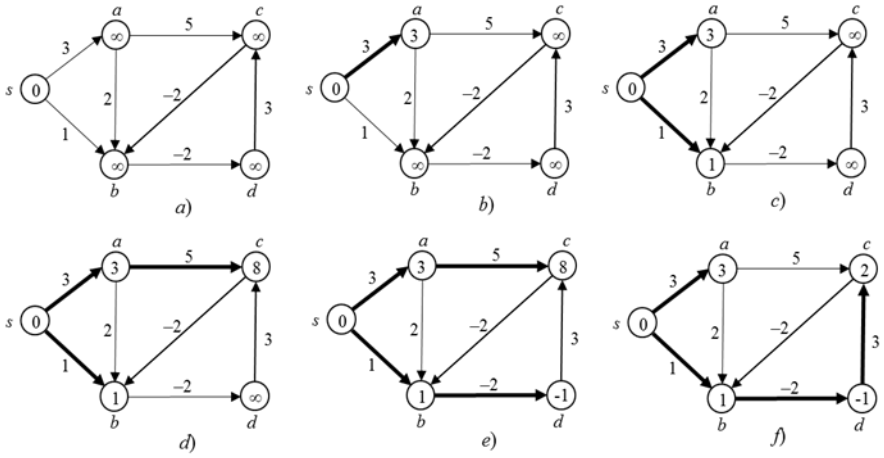
Во ситуација кога има негативен циклус, печатењето на еден таков циклус може да се направи со следниов псевдокод:

П 7. 4. ПЕЧАТЕЊЕ НА НЕГАТИВЕН ЦИКЛУС

```
1 најди u за кое A[u] ја сменило неговата вредност во последниот
    циклус;
2 печати v = u;
3 додека π[v] ≠ u ∈ E прави печати v = π[v].
```

За функцијата На Слика 7. 6 е дадена првата итерација на алгоритамот на Белман-Форд. Редоследот по кој се поминуваат

ребрата е следниов: (s,a), (s,b), (a,b), (a,c), (b,d), (s,a), (c,b), (d,c). Останатите 4 итерации се дадени на Слика 7. 7. Се гледа дека вредноста во темето b се променува во последната итерција, па значи дека има негативен циклус.

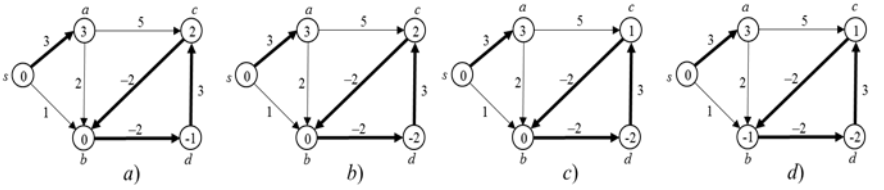


Слика 7. 6. Првата итерација на алгоритамот на Белман-Форд. Редоследот на ребрата е: $(s,a),(s,b),(a,b),(a,c),(b,d),(s,a),(c,b),(d,c)$.

Печатењето на негативниот циклус ќе се одвива на следниов начин:

$$b \ldots \pi[b] = c \ldots \pi[c] = d.$$

Овде ќе престане, затоа што $\pi[d] = b$.



Слика 7. 7. Од втората до петтата итерација на алгоритамот на Белман-Форд. Вредноста во темето *b* е променета, па има негативен циклус.

Алгоритамот на Белман-Форд работи во време $O(|V||E|)$, бидејќи иницијализацијата зазема време $\Theta(|V|)$ и секое од $|V|-1$-те поминувања во линиите $\Theta(|E|)$. Кодовите во Јава и C++ се дадени со JAVA 7.1 и C++ 7.1.

JAVA 7. 1 АЛГОРИТАМ НА БЕЛМАН ФОРД

```
1 import java.util.*;
2 import java.lang.*;
3 import java.io.*;
4
5 class Graph {
7     class Edge {
8         int src, dest, weight;
9
10        Edge() {
11            src = dest = weight = 0;
12        }
13    }
14
15    int V, E;
16    Edge edge[];
17
18    Graph(int v, int e) {
19        V = v;
20        E = e;
21        edge = new Edge[e];
22        for (int i = 0; i < e; ++i) edge[i] = new Edge();
23    }
24
25    void BellmanFord(Graph graph, int src) {
26        int V = graph.V, E = graph.E;
27        int dist[] = new int[V];
28        for (int i = 0; i < V; ++i) dist[i] =
Integer.MAX_VALUE;
29        dist[src] = 0;
30        for (int i = 1; i < V; ++i) {
31            for (int j = 0; j < E; ++j) {
32                int u = graph.edge[j].src;
33                int v = graph.edge[j].dest;
34                int weight = graph.edge[j].weight;
35                if (dist[u] != Integer.MAX_VALUE && dist[u]
```

```
+ weight < dist[v]) dist[v] = dist[u] + weight;
            }
        }
        for (int j = 0; j < E; ++j) {
            int u = graph.edge[j].src;
            int v = graph.edge[j].dest;
            int weight = graph.edge[j].weight;
            if (dist[u] != Integer.MAX_VALUE && dist[u] +
weight < dist[v]) {
                System.out.println("Grafot sodrzhi negativen
ciklus.");
                return;
            }
        }
        printArr(dist, V);
    }

    void printArr(int dist[], int V) {
        System.out.println("Rastojanie od izvorot");
        for (int i = 0; i < V; ++i) System.out.println(i +
"\t\t" + dist[i]);
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int V;
        int E;
        V = sc.nextInt();
        E = sc.nextInt();
        Graph graph = new Graph(V, E);
        for (int i = 0; i < E; i++) {
            int s = sc.nextInt();
            int d = sc.nextInt();
            int w = sc.nextInt();
            graph.edge[i].src = s;
            graph.edge[i].dest = d;
            graph.edge[i].weight = w;
        }
        graph.BellmanFord(graph, 0);
    }
}
```

Во двете програми се печати негативен циклус, ако тој постои, или најмалото растојание од дадено теме.

C++ 7. 1 Алгоритам на Белман-Форд

```
#include <bits/stdc++.h>
#include <iostream>
using namespace std;
struct Edge {
   int src, dest, weight;
};
struct Graph {
   int V, E;
   struct Edge* edge;
};

struct Graph* createGraph(int V, int E)
{
   struct Graph* graph = new Graph;
   graph->V = V;
   graph->E = E;
   graph->edge = new Edge[E];
   return graph;
}

void printArr(int dist[], int n)
{
   printf("Rastojanie od izvorot\n");
   for (int i = 0; i < n; ++i)
      printf("%d \t\t %d\n", i, dist[i]);
}

void BellmanFord(struct Graph* graph, int src)
{
   int V = graph->V;
   int E = graph->E;
   int dist[V];

   for (int i = 0; i < V; i++)
      dist[i] = INT_MAX;
   dist[src] = 0;

```

```
38    for (int i = 1; i <= V - 1; i++) {
39       for (int j = 0; j < E; j++) {
40          int u = graph->edge[j].src;
41          int v = graph->edge[j].dest;
42          int weight = graph->edge[j].weight;
43          if (dist[u] != INT_MAX && dist[u] + weight <
dist[v])
44             dist[v] = dist[u] + weight;
45       }
46    }
47
48    for (int i = 0; i < E; i++) {
49       int u = graph->edge[i].src;
50       int v = graph->edge[i].dest;
51       int weight = graph->edge[i].weight;
52       if (dist[u] != INT_MAX && dist[u] + weight < dist[v])
{
53          printf("Grafot ima negativen ciklus.");
54          return;
55       }
56    }
57
58    printArr(dist, V);
59
60    return;
61 }
62
63 int main()
64 {
65    int V, E;
66    cin >> V >> E;
67
68    struct Graph* graph = createGraph(V, E);
69
70    for(int i=0;i<E;i++)
71    {
72       int s, d, w;
73       cin >> s >> d >> w;
74       graph->edge[i].src = s;
75       graph->edge[i].dest = d;
76       graph->edge[i].weight = w;
```

```
77      }
78
79      BellmanFord(graph, 0);
80
81      return 0;
82  }
```

Прашања и задачи

1. Покажи како работи алгоритамот на Белман-Форд на графот на Слика 7. 5 од темето 5.

2. Направи модификација на алгоритамот на Белман – Форд, така да во случај кога постои најкраток пат до сите темиња, алгоритамот не завршува во $|V|$ итерации, туку бројот на итерации да зависи од бројот на ребра во најдолгиот циклус!

3. Нека е даден ацикличен ориентиран граф и нека сметаме дека темињата на графот се тополошки сортирани. Дали можеш да дадеш модификација на алгоритамот на Белман –Форд за ваков граф, така да истиот работи во време $O(|E|)$?

4. За да се произведе некој производ потребно е да се завршат неколку поединечни задачи. Нека x_i е времето во кое ќе се изврши i-тата задача. Времињата на извршување на задачите подлежат на одредени ограничувања, во смисла дека секое ограничување значи дека мора да помине најмалку одреден временски период, или најмногу одреден временски период, меѓу две задачи. На пример, ако во времето x_1 применуваме лепило за кое треба да почека 2 часа пред да го залепиме делот до времето x_2, тогаш го имаме ограничувањето $x_2 \geq x_1 + 2$ или, еквивалентно, $x_1 - x_2 \leq -2$. За даден ваков систем од ограничувања да се даде алгоритам кој ќе пресмета барем едно можно решение.

7.4 Најкраток пат во граф со ненегативни тежини на ребрата и Алгоритам на Дикстра

Алгоритамот кој го разгледуваме во овој дел е многу едноставен алчен алгоритам за наоѓање на најкраток пат од еден извор, предложен од Едсгер Дикстра 1959 [20].

Дефинирање на проблемот

За даден ориентиран или неориентиран тежински граф $G = (V, E)$ во кој сите ребра се со ненегативна тежина, треба да се најде најкраток пат од едно теме, извор, s до сите останати темиња.

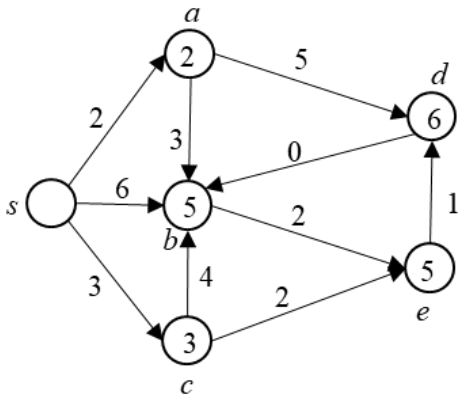
Анализа на проблемот

Алгоритамот работи така што патиштата ги гради ребро по ребро почнувајќи од изворот, но не го гради само патот кон определено теме, туку кон сите темиња заедно, градејќи го дрвото од најкратки патишта кое е исто како во алгоритамот на Белман-Форд. При тоа се води по алчниот принцип, земајќи го во секој чекор реброто кое најмногу одговара. Но при тоа не оди во права линија, туку сепак испробува повеќе варијанти, користејќи го оптималното својство, дека најкраткиот пат до секое теме се состои од најкратки патишта до темињата кои лежат на тој пат. Главното својство кое се користи во овој случај кога секое ребро $(u, v) \in E, w(u, v) \geq 0$, е неравенството на триаголник:

> *Ако $A[u]$ и $A[v]$ се најкратките патишта до темињата u и v соодветно, тогаш за секое ребро $(v, u) \in E \in E$, важи*
>
> $$A[u] \leq A[v] + w(v, u).$$

Дополнително важи и следново својство:

Ако темето u лежи на најкраткиот пат од изворот до темето v, тогаш $A[u] \leq A[v]$.



Слика 7. 8. Ориентиран граф со ребра со позитивна тежина.

Да ја опишеме идејата на алгоритамот графот претставен на Слика 7. 8 на кој сакаме да го определиме најкраткиот пат од изворот s до темето d. Јасно е дека дека најкраткиот пат до изворот s е 0. Најчеста алчна стратегија е да се движиме по најлесното ребро, затоа што така ни изгледа дека најкраткиот пат би имал најмала должина, но ако ја користиме таквата стратегија би добиле пат $\langle s, a, b, e, d \rangle$, кој има должина 8, а нашиот најкраток пат како што ќе определиме подоцна има должина 6. Не работи ни стратегијата со најмал број на ребра, да речеме еден таков пат е $\langle s, a, d \rangle$, со должина 6. Па каква стратегија ќе употребиме? Имено наместо да се сконцентрираме на темето d или било кое конкретно теме, треба да одиме чекор по чекор и да ги определуваме темињата до кои веќе сме сигурни дека сме го пресметале најкраткиот пат, и да гледаме каде можеме да стигнеме од нив. За дадениот граф на Слика 7. 8 можеме да почнеме да ја оценуваме далечината до темињата кои се негови соседи: a, b и c, затоа што барем едно од нив мора да биде првото теме на патот до било кое теме.

- Далечината до темето a ако се употреби реброто (s, a) е $l[a] = 2$, далечината до темето b, ако се употреби реброто

(s, b) е $l[b] = 6$, а до темето c ако се употреби реброто (s, c) е $l[c] = 3$, Слика 7. 9 *a*). Но ова не значи дека тоа се најкратките патишта. Имено, до темето b добиваме должина 6, а до него можеме да стигнеме по патот $\langle s, a, b\rangle$ кој има должина 5. Но, со сигурност знаеме дека сме го добиле најкраткиот пат до темето a, затоа што ако на најкраткиот пат до a прво е темето b, должината ќе биде поголема или еднаква на 6, а ако е прво темето c, должината ќе биде барем 3. Во никој случај не е помала од 2, па можеме да заклучиме дека најкраткиот пат до a е $A[a] = 2$.

- Сега проверуваме колкава би била должината на најкратките патишта до соседите на a, ако на тие најкратки патишта лежи темето a. Патот до темето b би бил

$$l[b] = A[a] + w(a, b) = 2 + 3 = 5,$$

што е подобро од претходно, а до темето d, Слика 7. 9 b) би бил

$$l[d] = A[a] + w(a, d) = 2 + 5 = 7.$$

Во овој момент заклучуваме дека должината на најкраткиот пат до темето c не може да биде помала од оваа што ја имаме во моментов, бидејќи ако на тој пат е темето a, тогаш наредно теме е или b или d, а патиштата преку тие две темиња се подолги од 3. Оттука, $A[c] = 3$.

- Понатаму продолжуваме да ја оценуваме должината на најкратките патишта до соседите на c, ако на тие најкратки патишта лежи c, Слика 7. 9 *c*). Патот до темето b би бил

$$A[c] + w(c, b) = 3 + 4 = 7,$$

што е полошо од претходната оценка, па останува $l[b] = 5$. Патот до темето e е

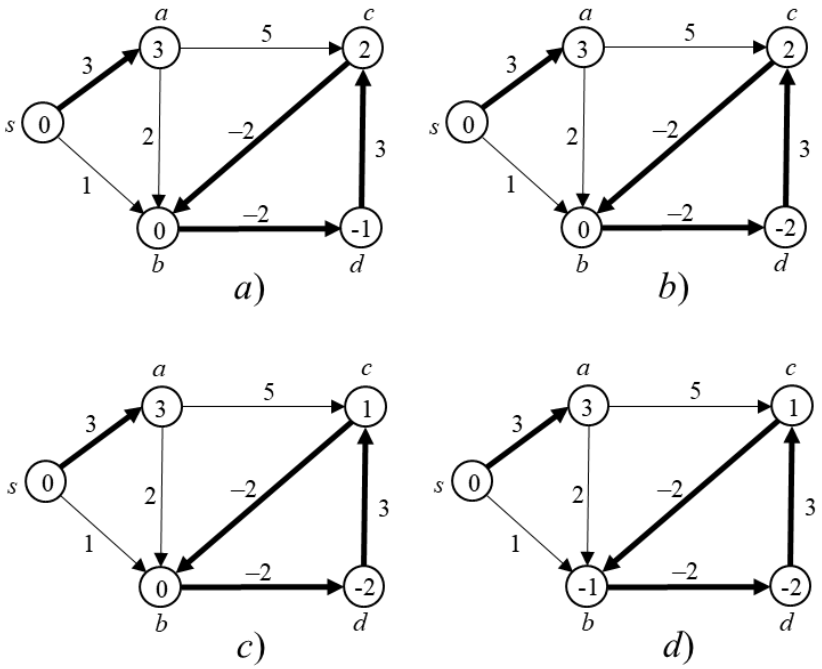
$$l[e] = A[c] + w(c, e) = 3 + 2 = 5.$$

До e до сега немавме оценето најкраток пат, па ова е најдобрата оценка во овој момент. Може да забележиме дека сега имаме две темиња, b и e, со најмала оценета вредност, $A[b] = A[e] =$5. И до двете од нив најкраткиот пат е оној што е оценет во моментот, па во наредниот чекор не е битно кое од нив ќе го земе прво за ажурирање на вредностите на останатите темиња. Само за илустрација, можеме да го земеме прво темето b, преку кое не се добива подобрување на ниту една оценета вредност.

- На крај ги ажурираме оценетите вредности на најкраткиот пат до темињата кои се соседи на e, а за кои сеуште не сме ја оцениле најоптималната вредност. Тоа е само темето d до кое најкраткиот пат преку темето e е

$$l[d] = A[e] + w(e, d) = 5 + 1 = 6,$$

што е подобро од претходно и воедно е најкраткиот пат $A[d] = 6$.



Слика 7. 9. а) Ажурирање на вредностите од темето *a*. с) Ажурирање на вредностите од темето *c*. d) Ажурирање на вредностите од темето *b*. с) Ажурирање на вредностите од темето *e*.

Во алгоритамот постепено се подобрува оценетата вредност на најкраткиот пат до секое од темињата различни од изворот, се додека не се дојде до вистинската вредност на најкраткиот пат. Во секоја итерација алгоритамот определува по едно теме за кое со сигурност се знае дека оценката добиена за него е најкраткиот пат, и потоа ги ажурира најкратките патишта до неговите соседи. Ова теме се става во множество S, кое се одржува како множество од темиња чиј најкраток пат од изворот s е определен. Прво, бидејќи секое ребро $(u, v) \in E, w(u, v) \geq 0$, нема негативни циклус, јасно е дека најкраткиот пат до изворот s мора да биде 0, па $S = \{s\}$, линија 7 од П 7.5. Потоа во секоја итерација се определува теме u од $V - S$ кое што има најмала оценета вредност од сите темиња во $V - S$, линија 12 од П 7.5. и истото се додава во S, линија 13, при што за него се заклучува дека оценетата вредност во тој момент е точната вредност на најкраткиот пат до него, линија 14. Потоа се ажурира најкраткиот

пат до соседите на u кои сеуште не се во S, линии 15-20. Псевдокодот е следниов:

П 7. 5. Алгоритам на Дикстра

```
1  Внеси ги V и E;
2  за секое u ∈ E прави
3     {
4       l[u] = A[u] = ∞;
5       π[u] = NIL;
6     }
7  l[s] = 0;
8  S = ∅;
9  Q = V;
10 додека Q ≠ ∅ прави
11    {
12      u = izvadi_min(Q);
13      S = S ∪ {u}
14      A[u] = l[u];
15      за секое v од листата на соседи на u,  v ∈ Adj[u];
16        ако l[v] > A[u] + w(u, v) тогаш
17           {
18             l[v] = A[u] + w(u, v).
19             π[v] = u.
20           }
21    }
```

За да ја генерализираме постапката за првиот чекор при најкраткиот пат, на почеток иницијализираме дека должината на најкраткиот пат до секое теме е бескрај. Со тоа, на крај за темињата до кои не може да се стигне ќе остане оваа вредност, што по дефиниција ни значеше дека темето не е достигливо од изворот.

Дополнително, во секој чекор се ажурира и дрвото од најкратки патишта. Имено, секогаш кога некоја вредноста $l[u]$ ќе се

ажурира од темето v, темето v ќе се постави за родител на u во дрвото од најкратки патишта, односно ќе поставиме $\pi[u] = v$. На почеток за секое теме ќе се постави $\pi[u] = \text{NIL}$, а кога алгоритамот ќе заврши вредноста NIL ќе ја има само изворот и темињата кои не се достигливи од изворот.

Може да се увиди дека во горната имплементација функцијата l не мора воопшто да фигурира и може да се замени со A. Таа овде е ставена само за појасно објаснување на постапката.

Бидејќи алгоритамот на Дикстра секогаш во S го додава темето за кое функцијата l има најмала вредност, т.е. темето од $V - S$ што е најблизу до S, тој се базира на алчна стратегија. Знаеме дека алчната стратегија не секогаш го дава точното оптимално решение, но во случајот на овој алгоритам клучот на точноста на алгоритамот е во тоа што во времето кога темето u се додава во S, со сигурност знаеме дека $l[u] = A[u]$. Доказот е даден во следнава Теорема:

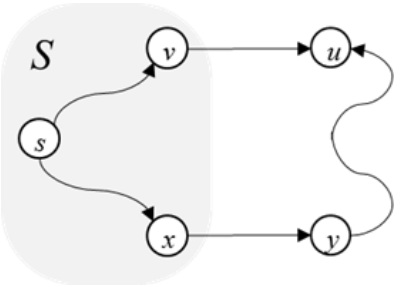
ТЕОРЕМА (ТОЧНИОТ НА АЛГОРИТАМОТ НА ДИКСТРА)

Ако алгоритамот на Дикстра работи над тежински, ориентиран граф $G = (V, E)$ со ненегативна функција на тежини w и извор s, тогаш $A[u]$ е најкраткиот пат од s до u за секое теме $u \in V$.

Доказ: За да го докажеме алгоритамот треба да ја докажеме следната инваријанта на циклусот **додека** во линиите 10-21 во алгоритамот П 7. 5.

Ако пред почетокот на циклусот за секое теме $u \in S$, $A[u]$ е најкраткиот пат од s до u, тогаш тоа важи и по завршувањето на циклусот.

Првото теме кое се става во S е s, и S и во тоа време $A[s] = 0$. Ова е најмалата должина што воопшто можеме да ја имаме, па јасно во овој случај инваријантата е точна.



Слика 7. 10. Најкраткиот пат до u и алтернативниот пат преку темето y.

Од друга страна, ако во моментот кога u се додава S, $l[u] = \infty$, тогаш за сите други v од $V - S$ ќе важи $l[v] = \infty$. Тоа би значело дека сите темиња кои останале во $V - S$ не се достигливи од изворот, па за нив $A[u] = \infty$, што сметавме дека е најкраток пат во случај кога нема пат. Останува да се докаже дека инваријантата е точна за сите темиња достигливи од изворот.

Да претпоставиме дека инваријантата не е точна за некое теме достигливо од изворот, односно дека во некој момент во S се додава теме u за кое $l[u] < \infty$ не е најкраткиот пат од s до u. Нека u е првото такво теме кое се додава во S. Мора $u \neq s$, па во моментот кога u се додава во S, $S \neq \emptyset$. Според тоа во тој момент има некаков пат од s до u, бидејќи инаку $l[u] = \infty$, но тој не може да е најкраткиот пат, па најкраткиот пат е некој друг пат p од s до u. Во моментот пред да се додаде u на S, дел од темињата од патот p се во S, (барем s е такво теме), а дел не се (барем u е такво теме). Нека првото теме од p кое што не е во S е темето y и нека $x \in S$ е претходникот на y на тој пат p. Патот p може да се декомпонира на: патот од 1 до x, патот од x до y и на крај патот од y до u, како што е прикажано на Слика 7. 10. Во моментот пред да се додаде u на S,

$$A[y] \leq l[y] \leq A[x] + w(x, y) = A[y].$$

Последното равенство е точно, од својството на оптималност на најкраткиот пат, односно затоа што патот од s до y е потпат од најкраткиот пат од s до u. Оттука во овој момент $l[y]$ е должината на најкраткиот пат од s до y. Но најкраткит пат до y е помал или еднаков на најкраткиот пат до u , па се добива следнава контрадикција:

$$l[y] = A[y] \leq A[u] < l[u] \leq l[y].$$

Значи претпоставката дека $l[u]$ не е најкраткиот пат од s до u, не е точна.

Бидејќи на почеток $S = \emptyset$, пред првото влегување во циклусот ќе важи дека $A[u]$ е најкраткиот пат од s до u за секое теме $u \in S$. На крај, кога ќе се излезе од циклусот, односно кога Q ќе се испразни, ќе имаме дека $S = V$. Па за сите темиња $u \in V$, $A[u]$ ќе ја има вредноста на најкраткиот пат. □

Од оваа теорема следува дека G_π ќе биде кореновото дрво од најкратки патишта со извор s.

Сложеноста на алгоритамот се анализира со техника наречена амортизирачка анализа. Имено во **за** циклусот, кој оди по сите соседи на некое теме, во најлош случај може да се случи бројот на соседи на некое теме да биде $O(|V|)$. Овој циклус е вгнезден во **додека** циклусот во кој се влегува точно $|V|$ пати, па сложеноста е $O(|E||V|)$, што воопшто не е подобро од алгоритамот на Белман-Форд, и секако е многу лоша оцена за брзината на работа на овој алгоритам.

Да забележиме дека секое теме $u \in V$ се додава на S точно еднаш, и неговата листа на соседи, $Adj[u]$ се разгледува точно еднаш во **за** циклусот, линија 15 во алгоритамот П 7. 5. Бидејќи вкупниот број на ребра е $|E|$, вкупниот број итерации во овој циклус е $|E|$, и вкупниот број на операции на промена на вредноста на функцијата $l[u]$ е исто така $|E|$. Оттука, времето на работа на алгоритамот на Дикстра зависи од имплементацијата на веригата. Но секоја операција во верига базирана на бинарен куп за која зборувавме претходно, работи во време $O(\ln |Q|)$, а во Q не можеме да имаме повеќе од $|V|$ темиња, следува дека сложеноста на алгоритамот е $O(|E| \ln |V|)$. Може да се искористи и структурата Фибоначиев куп, но истиот ќе мора да се испрограмира, затоа што не е стандардна структура во програмските јазици. Со тоа сложеноста на алгоритамот ќе се намали на $O(|V| \ln|V| + |E|)$.

```
1 import java.util.*;
2 import java.lang.*;
3 import java.io.*;
4
5 class Main {
6     static int V;
7
8     int minDistance(int dist[], Boolean sptSet[]) {
9         int min = Integer.MAX_VALUE, min_index = -1;
10
11        for (int v = 0; v < V; v++)
12            if (sptSet[v] == false && dist[v] <= min) {
13                min = dist[v];
14                min_index = v;
15            }
16
17        return min_index;
18    }
19
20    void printSolution(int dist[], int n) {
21        System.out.println("Rastojanie od izvorot");
22        for (int i = 0; i < V; i++)
23            System.out.println(i + " >> " + dist[i]);
24    }
25
26    void dijkstra(int graph[][], int src) {
27        int dist[] = new int[V];
28
29        Boolean sptSet[] = new Boolean[V];
30
31        for (int i = 0; i < V; i++) {
32            dist[i] = Integer.MAX_VALUE;
33            sptSet[i] = false;
34        }
35
36        dist[src] = 0;
37
38        for (int count = 0; count < V - 1; count++) {
39            int u = minDistance(dist, sptSet);
40
```

```
41              sptSet[u] = true;
42
43              for (int v = 0; v < V; v++)
44                  if (!sptSet[v] && graph[u][v] != 0 &&
45                          dist[u] != Integer.MAX_VALUE &&
46                          dist[u] + graph[u][v] < dist[v])
47                      dist[v] = dist[u] + graph[u][v];
48          }
49
50          printSolution(dist, V);
51      }
52
53      public static void main(String[] args) {
54          Scanner sc = new Scanner(System.in);
55
56          int E;
57          Main.V = sc.nextInt();
58          E = sc.nextInt();
59          int graph[][] = new int[V][V];
60          for (int i = 0; i < E; i++) {
61              int s = sc.nextInt();
62              int d = sc.nextInt();
63              int w = sc.nextInt();
64              graph[s][d] = w;
65          }
66
67          Main t = new Main();
68          t.dijkstra(graph, 0);
69      }
70}
```

Програмите во јава и C++ дадени овде ги печатат должините на најкртките патишта од едно теме до сите останати темиња.

C++ 7. 2 АЛГОРИТАМ НА ДИКСТРА

```
1   #include <stdio.h>
2  #include <limits.h>
3  #include<iostream>
4  #include<vector>
5  using namespace std;
6
7  static int V;
8
9  int minDistance(int dist[], bool sptSet[])
10 {
11    int min = INT_MAX, min_index;
12
13    for (int v = 0; v < V; v++)
14     if (sptSet[v] == false && dist[v] <= min)
15        min = dist[v], min_index = v;
16
17    return min_index;
18 }
19
20 int printSolution(int dist[], int n)
21 {
22    printf("Rastojanie od izvorot\n");
23    for (int i = 0; i < V; i++)
24      printf("%d >> %d\n", i, dist[i]);
25 }
26
27 void dijkstra(vector<vector<int>> &graph, int src)
28 {
29     int dist[V];
30     bool sptSet[V];
31
32     for (int i = 0; i < V; i++)
33       dist[i] = INT_MAX, sptSet[i] = false;
34
35     dist[src] = 0;
36
37     for (int count = 0; count < V-1; count++)
38     {
39       int u = minDistance(dist, sptSet);
40
41       sptSet[u] = true;
42
```

```
43        for (int v = 0; v < V; v++)
44         if (!sptSet[v] && graph[u][v] && dist[u] != INT_MAX
45                                    && dist[u]+graph[u][v] <
dist[v])
46            dist[v] = dist[u] + graph[u][v];
47      }
48
49      printSolution(dist, V);
50  }
51
52  int main()
53  {
54     int E;
55     cin >> V >> E;
56     vector<vector<int>> graph(V, vector<int>(V, 0));
57
58     for(int i=0;i<E;i++)
59     {
60        int s, d, w;
61        cin >> s >> d >> w;
62        graph[s][d] = w;
63     }
64
65     dijkstra(graph, 0);
66
67     return 0;
68  }
```

Прашања и задачи

1. Дади едноставен ориентиран граф со негативна тежина на некои ребра за кои не работи алгоритамот на Дикстра.

2. Нека е даден тежински граф во кој тежините означуваат веројатност да се помине по дадениот директен пат. Да се даде псевдокод за алгоритам која пресметува веројатноста на најверојатниот пат од еден извор до сите станати темиња.

3. Дади идеја како може да се искористи алгоритамот на Дикстра за граф во кој наместо тежини на ребрата се дадени тежини на темињата.

4. Алгоритамот на Дикстра во секој чекор бара теме со минимална вредност, што рековме дека е најдобро да се имплементира со мин-приоритетен куп во кој секое ажурирање зафаќа време $O(lnn)$. Колкава ќе биде сложеноста на алгоритамот ако наместо ваква верига искористиме

   a. Низа?

   b. Листа?

   Објасни зошто!

7.5 Најкраток пат меѓу секој пар темиња Алгоритам на Флојд-Варшал

Во претходното поглавје беа разгледани алгоритми за наоѓање на најкратките патишта од дадено теме до сите други темиња во графот. Со ваквите алгоритми можевме да го најдеме најкраткиот пат од еден град од кој тргаме до сите градови на патната карта. Но како да направиме табела на растојанија меѓу сите парови на градови за патната карта? Ваквиот проблем е генерализација на проблемот за најкраток пат од секој можен извор до секоја можна дестинација.

Јасно е дека проблемот за најкраток пат меѓу секој пар може да се реши со извршување на алгоритамот за најкраток пат со еден извор $|V|$ пати, по еднаш за секој теме како извор. Ако сите тежини на ребрата се ненегативни, можеме да го користиме алгоритамот Дикстра. Во овој случај, ако мин-приоритетната верига ја имплементираме со бинарен куп времето на работа ќе биде $O(|V||E|\ln|V|)$, а ако се искористи фибоначиев куп, сложеноста ќе биде $O(|V|^2\ln|V| + |V||E|)$. Ако пак имаме ребра со негативни тежини, можеме да го користиме алгоритамот на Белман-Форд. Комплексноста во овој случај ќе биде $O(|V|^2|E|)$, што за густ граф би значело $O(|V|^4)$.

Има неколку пристапи за решение на овој проблем, и сите се базираат на оптималното својство кое го поседува проблемот за најкраток пат, кој го дадовме во претходниот дел. Секако, сложеноста се очекува да биде поголема од сложеноста на алгоритмите за најкраток пат од еден извор и да напоменеме дека ако знаеме дека графот има само негативни тежини на ребрата, тогаш најдобар е пристапот да се искористи алгоритамот на дикстра за секое теме. Но во случај на граф со негативни тежини на ребрата, пристапот со повеќекратно користење на алгоритамот на Белнман-Форд ќе има многу голема сложеност. Во овој дел ќе дадеме еден ефективен алгоритам за решавање на овој проблем во случај на граф кој има и ребра со негативна тежина. Алгоритамот е познат како

алгоритам на Флојд-Варшал и се базира на динамичко програмирање, но идејата за рекурзијата е малку поинаква од идејата во алгоритамот на Белман-Форд. За вежба ќе биде дадена и идеја за алгоритам кој во случај на ретки графови има помала сложеност од алгоритамот на Флојд-Варшал.

Дефинирање на проблемот

Даден е ориентиран тежински граф $G = (V, E)$ со функција на тежина w: $E \rightarrow \boldsymbol{R}$ која ги пресликува темињата во реални вредности кои одговараат на тежините. За секој пар на темиња u и $v \in V$, треба да се најде најкраткиот пат од u до v, каде што тежината на патот е збир на тежините на ребрата долж него.

Анализа на проблемот

Во проблемот за наоѓање на најкратки патишта од еден извор се пресметуваа две битни работи, должината на најкраткиот пат и дрвото од најкратки патишта, со корен во изворот. Аналогно, за овој проблем за секој пар темиња треба да се пресмета најкраткиот пат од u до v, кој овде ќе го обележиме со $A[u, v]$. Во оваа ситуација не можеме да дефинираме дрво од најкратки патишта, затоа што за различно теме u дрвото би било различно. Да се потсетиме дека ова дрво од најкратки патишта го зачувувавме со зачувување на претходникот на секое теме на најкраткиот пат, во функцијата π. И во алгоритмите за најкраток пат меѓу секој пар повторно за најкраткиот пат од u до v ќе го пресметуваме претпоследното теме (ако такво теме постои) на тој пат (ако таков пат постои), односно темето кое е непосредно пред v, но за различни темиња u ова теме ќе биде различно. Затоа сега наместо $\pi[u]$, ќе имаме функција $\pi[u, v]$, која ќе ја наречеме матрица од претходници. Во основните случаи ќе имаме $A[u, u] = 0$ и $\pi[u, u] = \text{NIL}$. Ако најкраткиот пат од u до v има само едно ребро, тогаш $A[u, v] = w(u, v)$ и $\pi[u, v] = u$. Ако пак не постои најкраток пат од u до v или ако постои негативен циклус, тогаш повторно ќе ја користиме истата нотација како во случајот со еден извор, односно $A[u, v] = \infty$ и $\pi[u, v] = \text{NIL}$.

Оттука, излезот од алгоритамот за најкраток пат меѓу секој пар темиња ќе претставува една $|V| \times |V|$ матрица во која ќе се запишуваат сите $|V|^2$ растојанија и претходници. Во многу од мапите

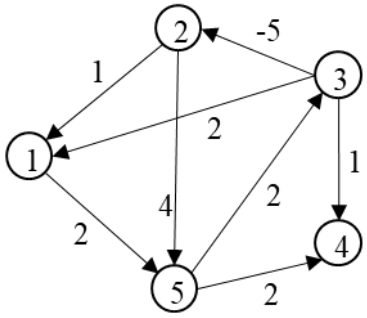
на најкратки патишта од една до друга дестинација треба во табелата само да се погледне кое е најкраткото растојание. За да истото се реконструира, треба да се направи мало надополнување на алгоритмите кое ќе ја користи низата од претходници на сличен начин како што тоа се правеше во случај на најкраток пат со еден извор.

Алгоритам на Флојд-Варшал

Алгоритамот на Флојд Варшал работи на ориентирани графови, пред се затоа што претпоставува дека во графот има ребра со негативна тежина, па ако графот е неориентиран, тогаш едно негативно ребро $\{u, v\}$ ќе прави негативен циклус$\langle u, v, u\rangle$. Времето на работа на алгоритамот е $O(|V|^3)$.

Алгоритамот на Флојд-Варшал е алгоритам кој користи динамичко програмирање, но врз основа на поинаква рекурентна релација од претходните два алгоритми. И за овој алгоритам сметаме дека темињата во графот се подредени од v_1 до $v_{|V|}$ и понатаму темето v_i ќе го бележиме само со i, т.е. сметаме дека множеството $V = \{1, 2, \ldots, n\}$. За разлика од алгоритамите за најкраток пат од еден извор, каде рекурзивната релација зависеше од бројот на ребра на патот, овде рекурзивната релација ќе зависи од најголемиот реден број на некое теме кое е внатрешно теме на патот.

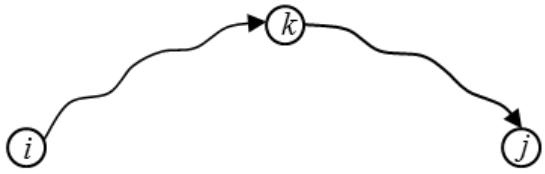
Под внатрешно теме на патот $p = \langle v_1, v_2, \ldots, v_m \rangle$ го сметаме секое теме теме различно од v_1 и v_m, т.е. секое теме од $\{v_2, v_3, \ldots, v_{m-1}\}$. За секој пар темиња $i, j \in V$, ги разгледуваме сите патишта од i до j, на кои сите внатрешни темиња имаат реден број помал или еднаков на k, односно припаѓаат на множеството $\{1, 2, \ldots, k\}$. Алгоритамот го користи својството за едноставен пат, односно дека постои најкраток пат кој нема циклус.



Слика 7. 11. Ориентиран граф во кој нема негативен циклус, но има циклус со тежина 0.

Нека со $A[i,j,k]$ ја обележиме должината на најкраткиот пат од темето i до темето j, на кој сите внатрешни темиња се во множеството $\{1,2,\ldots,k\}$. На пример, за графот на Слика 7. 11, најкраткиот пат од темето 3 до темето 5 на кој сите внатрешни темиња се во множеството $\{1\}$ има должина 4, додека најкраткиот пат меѓу истите две темиња на кој сите внатрешни темиња се во множеството {1, 2} има должина -1.

Кога $k = 0$ се работи за пат од i до j кој нема внатрешни темиња кои се нумерирани со број поголем од 0, од каде следува дека таквиот пат нема воопшто внатрешни темиња. Значи ваквиот пат има најмногу едно ребро, па $A[i,j,0] = w(i,j)$. За графот на Слика 7. 11, најкраткиот пат од темето 2 до темето 5 на кој сите внатрешни темиња се стриктно помали од 1, т.е. во множеството $\emptyset$ е 4. Почетните вредности се $w[i,i] = 0$ и $w(i,j) = \infty$ кога $(i,j) \notin E$. Рекурзивната равенка за $A[i,j,k]$ зависи од третиот параметар, k, па во секој чекор се ажурираните вредности ќе ги зачувуваме во матрица од најкратки патишта $D_k = [A[i,j,k]]_{n\times n}$ за $k = \overline{0,n}$. Вистинските најкратки тежински патишта ќе се содржат во матрица D_n.



Слика 7. 12. Декомпозиција на патот од i до j преку темето k.

За да дефинираме рекурзивна релација, треба да ја препознаеме врската помеѓу најкраткиот пат од i до j во кој внатрешните темиња се обележани со редни броеви помали или еднакви од k и најкратките патишта во кој внатрешните темиња се обележани со редни броеви помали или еднакви од $k-1$. Имаме два случаи во однос на тоа дали темето k е внатрешно теме на најкраткиот пат од i до j во кој внатрешните темиња се обележани со редни броеви помали или еднакви од k или не е.

- Ако тој пат не го содржи темето k, тогаш е јасно дека овој пат е еднаков на најкраткиот пат од i до j во кој внатрешните темиња се обележани со редни броеви помали или еднакви од $k-1$, па во овој случај $A[i,j,k] = A[i,j,k-1]$.
- Во спротивно, ако темето k се наоѓа на тој пат, тогаш се јавува само еднаш, бидејќи во спротивно би имале циклус. Во ваква ситуација па патот може да се декомпонира на пат од i до k и пат од k до j, како што е претставено на Слика 7. 12. И двата потпатишта не содржат темиња со редни броеви поголеми или еднакви на k, т.е. редните броеви на тие патишта се од множеството $\{1,2,\ldots,k-1\}$. Уште повеќе, тоа се најкратките патишта од i до k и од k до j, кои содржат темиња од $\{1,2,\ldots,k-1\}$, па нивната должина е $A[i,k,k-1]$ и $A[k,j,k-1]$ соодветно.

Сето ова е сумирано во следнава рекурзивна равенка:

$$A[i,j,k] = \begin{cases} w(i,j), & k = 0 \\ \min\{A[i,j,k-1], A[i,k,k-1] + A[k,j,k-1]\}, & k > 0 \end{cases}.$$

Бидејќи за секој пат сите внатрешни темиња се од $\{1,2,\ldots,n\}$, конечниот одговор е $A[i,j,n]$ за сите $i,j \in V$.

За реконструкција на најкратките патишта потребно ни е да ја пресметаме и матрицата од претходници. Таа може да се пресмета откога ќе се пресмета матрицата од најкратки патишта, во време $O(n^3)$, но истата може да се пресмета и во текот на работата на самиот алгоритам, паралелно со пресметување на матрицата од најкратки патишта. Слично како и за матрицата од најкратки патишта

и оваа матрица ажурира во секој чекор. Матрицата која се добива во k-тиот чекор ќе ја обележиме со $\Pi_k = [\pi[i,j,k]]_{n\times n}, k = \overline{0,n}\ \Pi_m$, каде $\pi[i,j,k]$ се дефинира како претходник на темето j на најкраткиот пат од темето i во кој пат сите внатрешни темиња се од множеството $\{1,2,\ldots,k\}$.

Во иницијалниот случај, за $k = 0$, најкраткиот пат од i до j нема внатрешни темиња, па

$$\pi[i,j,0] = \begin{cases} \text{NIL}, & i = j \text{ или } w(i,j) = \infty \\ i, & i \neq j \text{ и } w(i,j) < \infty \end{cases}.$$

За да ја изведеме оваа функција за $k \geq 1$, повторно ќе го разгледаме декомпонираниот пат од i до k и од k до j. Можни се два случаи, $k \neq j$ и $k = j$. Во случај кога $k \neq j$ претходникот кој се наоѓа на патот од i до j е истиот како претходникот за j кој сме го избрале на најкраткиот пат од k до j, на кој сите внатрешни темиња му се во множеството $\{1,2,\ldots,k-1\}$. Во спротивно, кога $k = j$, претходник на j на патот од i до j е точно она теме кое е претходник на k на најкраткиот пат од i до k на кој сите измеѓу темиња му се во множеството $\{1,2,\ldots,k-1\}$. Формално, за $k \geq 1$ рекурзивната релација е

$$\pi[i,j,k] =$$

$$= \begin{cases} \pi[i,j,k-1], & A[i,j,k-1] \leq A[i,k,k-1] + A[k,j,k-1] \\ \pi[k,j,k-1], & A[i,j,k-1] > A[i,k,k-1] + A[k,j,k-1] \end{cases}.$$

Врз основа на оваа рекурзивна формула можеме да го дадеме алгоритамот со динамичко програмирање.

П 7. 6. АЛГОРИТАМ НА ФЛОЈД ВАРШАЛ

```
1  Внеси го графот G = (V = {1, 2, … , n}, E, w)
2  за i = 1 до n прави
3     за j = 1 до n прави
4        ако i ≠ j & (i, j) ∈ E прави π[i, j, 0] = i инаку
5           {
6              w[i, j]= ∞;
```

```
7              π[i, j, 0] = NIL
8            }
9  за i = 1 до n прави
10   за j = 1 до n прави A[i, j, 0] = w(i, j);
11 за k = 1 до n прави
12   за i = 1 до n прави
13     за j = 1 до n прави
14       ако A[i, j, k − 1] ≤ A[i, k, k − 1] + A[k, j, k − 1] тогаш
15         {
16           A[i, j, k] = A[i, j, k − 1];
17           π[i, j, k] = π[i, j, k − 1];
18         }
19         инаку
20           {
21             A[i, j, k] = A[i, k, k − 1] + A[k, j, k − 1];
22             π[i, j, k] = π[k, j, k − 1];
23           }
24 врати D и Π
```

Откога се пресметани најкратките патишта меѓу сите темиња и матрицата од претходници, печатењето на најкратките патишта може да се направи на сличен начин како што тоа се прави во останатите алгоритми со следниов псевдокод:

П 7. 7. РЕКОНСТРУКЦИЈА НА НАЈКРАТОК ПАТ ОД МАТРИЦА ОД ПРЕТХОДНИЦИ

```
1  внеси ги i и j;
2    ако i = j тогаш печати i инаку
3      ако π[i, j, n] = NIL тогаш печати „нема пат“ инаку
4        {
5          печати j;
6          додека π[i, j, n] ≠ NIL ∈ E прави печати j = π[i, j, n].
```

```
7           }
```

Матриците кои се добиваат кога алгоритамот работи над графот од Слика 7. 11 се следниве:

$$D_0 = \begin{bmatrix} 0 & \infty & \infty & \infty & \infty \\ \infty & 0 & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \\ \infty & \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & \infty & 0 \end{bmatrix}, \quad \Pi_0 = \begin{bmatrix} \text{NIL} & \text{NIL} & \text{NIL} & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & \text{NIL} & \text{NIL} \end{bmatrix},$$

$$D_1 = \begin{bmatrix} 0 & \infty & \infty & \infty & 2 \\ 1 & 0 & \infty & \infty & 4 \\ 2 & -5 & 0 & 1 & \infty \\ \infty & \infty & \infty & 0 & \infty \\ \infty & \infty & 2 & 2 & 0 \end{bmatrix}, \quad \Pi_1 = \begin{bmatrix} \text{NIL} & \text{NIL} & \text{NIL} & \text{NIL} & 1 \\ 2 & \text{NIL} & \text{NIL} & \text{NIL} & 2 \\ 3 & 3 & \text{NIL} & 3 & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & 5 & 5 & \text{NIL} \end{bmatrix},$$

$$D_2 = \begin{bmatrix} 0 & \infty & \infty & \infty & 2 \\ 1 & 0 & \infty & \infty & 3 \\ 2 & -5 & 0 & 1 & 4 \\ \infty & \infty & \infty & 0 & \infty \\ \infty & \infty & 2 & 2 & 0 \end{bmatrix}, \quad \Pi_2 = \begin{bmatrix} \text{NIL} & \text{NIL} & \text{NIL} & \text{NIL} & 1 \\ 2 & \text{NIL} & \text{NIL} & \text{NIL} & 1 \\ 3 & 3 & \text{NIL} & 3 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & 5 & 5 & \text{NIL} \end{bmatrix},$$

$$D_3 = \begin{bmatrix} 0 & \infty & \infty & \infty & 2 \\ 1 & 0 & \infty & \infty & 3 \\ -4 & -5 & 0 & 1 & -2 \\ \infty & \infty & \infty & 0 & \infty \\ \infty & \infty & 2 & 2 & 0 \end{bmatrix}, \Pi_3 = \begin{bmatrix} \text{NIL} & \text{NIL} & \text{NIL} & \text{NIL} & 1 \\ 2 & \text{NIL} & \text{NIL} & \text{NIL} & 1 \\ 2 & 3 & \text{NIL} & 3 & 2 \\ \text{NIL} & \text{NIL} & \text{NIL} & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & 5 & 5 & \text{NIL} \end{bmatrix},$$

$$D_4 = \begin{bmatrix} 0 & \infty & \infty & \infty & 2 \\ 1 & 0 & \infty & \infty & 3 \\ -4 & -5 & 0 & 1 & -2 \\ \infty & \infty & \infty & 0 & \infty \\ -2 & -3 & 2 & 2 & 0 \end{bmatrix}, \Pi_4 = \begin{bmatrix} \text{NIL} & \text{NIL} & \text{NIL} & \text{NIL} & 1 \\ 2 & \text{NIL} & \text{NIL} & \text{NIL} & 1 \\ 2 & 3 & \text{NIL} & 3 & 2 \\ \text{NIL} & \text{NIL} & \text{NIL} & \text{NIL} & \text{NIL} \\ 3 & 3 & 5 & 5 & \text{NIL} \end{bmatrix},$$

$$D_5 = \begin{bmatrix} 0 & -1 & 4 & 4 & 2 \\ 1 & 0 & 5 & 5 & 3 \\ -4 & -5 & 0 & 0 & -2 \\ \infty & \infty & \infty & 0 & \infty \\ -2 & -3 & 2 & 2 & 0 \end{bmatrix}, \Pi_5 = \begin{bmatrix} \text{NIL} & 5 & 5 & 5 & 1 \\ 2 & \text{NIL} & 5 & 5 & 1 \\ 2 & 3 & \text{NIL} & 5 & 2 \\ \text{NIL} & \text{NIL} & \text{NIL} & \text{NIL} & \text{NIL} \\ 3 & 3 & 5 & 5 & \text{NIL} \end{bmatrix},$$

Последната D матрица, D_5 ни ја дава должината на најкратките патишта. На пример, најкраткиот пат од 2 до 4 има должина 5, а најкраткиот пат од 3 до 2 има должина -5. Последната матрица Π_5 може да се искористи за реконструкција на најкратките патишта. Ако сакале да го реконструираме најкраткиот пат од 2 до 4:

- прво ја гледаме вредноста $\pi[2,4,5]$=5,
- па вредноста $\pi[2,5,5] = 1$,
- па вредноста $\pi[2,1,5] = 2$
- и на крај $\pi[2,2,5] = \text{NIL}$.

Патот ќе се отпечати од назад на напред како: 4, 5, 1, 2.

Времето на работа на алгоритамот на Флојд-Варшал е определен со вгнездените циклуси по k, i и j. Делот од кодот во третиот вгнезден циклус зазема време $O(1)$, па вкупното време на работа е $\mathrm{O}(n^3)$.

Кодовите во Јава и C++ кои ги даваме овде се итеративни решенија кои се базираат на третиот пристап.

JAVA 7. 3 АЛГОРИТАМ НА ФЛОЈД-ВАРШАЛ

```
1   import java.util.*;
2  import java.lang.*;
3  import java.io.*;
4
5  class Main {
6     final static int INF = Integer.MAX_VALUE;
7     static int V;
8
9     void floydWarshall(int graph[][]) {
10       int dist[][] = new int[V][V];
11       int i, j, k;
12
13       for (i = 0; i < V; i++)
14          for (j = 0; j < V; j++)
15             dist[i][j] = graph[i][j];
16
17
18       for (k = 0; k < V; k++) {
19          for (i = 0; i < V; i++) {
20             for (j = 0; j < V; j++) {
21                if (dist[i][k] == INF || dist[k][j] == INF)
```

```
22                     continue;
23                  if (dist[i][k] + dist[k][j] < dist[i][j])
24                     dist[i][j] = dist[i][k] + dist[k][j];
25               }
26            }
27         }
28
29         printSolution(dist);
30      }
31
32      void printSolution(int dist[][]) {
33         System.out.println("Matrica na rastojanija.");
34         for (int i = 0; i < V; ++i) {
35            for (int j = 0; j < V; ++j) {
36               if (dist[i][j] == INF)
37                  System.out.print("INF ");
38               else
39                  System.out.print(dist[i][j] + "   ");
40            }
41            System.out.println();
42         }
43      }
44
45      public static void main(String[] args) {
46         Scanner sc = new Scanner(System.in);
47         int E;
48         Main.V = sc.nextInt();
49         E = sc.nextInt();
50         int graph[][] = new int[V][V];
51         for (int[] row: graph)
52            Arrays.fill(row, INF);
53         for (int i = 0; i < V; i++)
54            graph[i][i] = 0;
55         for (int i = 0; i < E; i++) {
56            int s = sc.nextInt();
57            int d = sc.nextInt();
58            int w = sc.nextInt();
59            graph[s][d] = w;
60         }
61
62         Main a = new Main();
63
64         a.floydWarshall(graph);
65      }
66   }
```

```
#include <bits/stdc++.h>
#include <vector>
#include <iostream>
#include <limits.h>
using namespace std;

#define INF INT_MAX

static int V;

void printSolution(vector<vector<int>> &graph);

void floydWarshall(vector<vector<int>> &graph)
{
   int i, j, k;

   for (k = 0; k < V; k++)
   {
      for (i = 0; i < V; i++)
      {
         for (j = 0; j < V; j++)
         {
            if (graph[i][k] == INF || graph[k][j] == INF)
                  continue;
            if (graph[i][k] + graph[k][j] < graph[i][j])
               graph[i][j] = graph[i][k] + graph[k][j];
         }
      }
   }

   printSolution(graph);
}

void printSolution(vector<vector<int>> &dist)
{
   cout<<"Matrica na najkratki rastojanija\n";
   for (int i = 0; i < V; i++)
   {
      for (int j = 0; j < V; j++)
      {
         if (dist[i][j] == INF)
            cout<<"INF"<<"     ";
         else
            cout<<dist[i][j]<<"     ";
      }
      cout<<endl;
```

```
47    }
48 }
49
50 int main()
51 {
52    int E;
53    cin >> V >> E;
54    vector<vector<int>> graph(V, vector<int>(V, INF));
55    for (int i = 0; i < V; i++)
56          graph[i][i] = 0;
57
58    for(int i=0;i<E;i++)
59    {
60       int s, d, w;
61       cin >> s >> d >> w;
62       graph[s][d] = w;
63    }
64
65    floydWarshall(graph);
66    return 0;
67 }
```

Прашања и задачи

1. Матрицата од претходници на најкратките патишта може да се пресмета откога ќе се пресмета матрицата од најлесни патишта, со користење на матрицата од тежини на ребрата. Дадете алгоритам за пресметување на оваа матрица кој работи во време $O(n^3)$!

2. Како со алгоритамот на Флојд-Варшал, може да се детектира циклус со негативна тежина?

3. Транзитивен затворач на ориентиран граф $G = (V, E)$ се дефинира како граф $G^* = (V, E^*)$, каде

$$E^* = \{(i, j) | \text{ од } i \text{ до } j \text{ постои пат во } G\}$$

   Дади алгоритам за пресметување на транзитивниот затворач на даден граф.

4. Нека графот G е тежински ориентиран граф добиен од кореново дрво на кое за секој лист е додадено тежинско ребро од листот назад кон коренот. Нека тежините на ребрата во графот се ненегативни. Дади ефикасен алгоритам за пресметување на најкратките патишта меѓу сите темиња за ваков граф.

5. Детерминистички конечен автомат се дефинира како петорка $M = (Q, \Sigma, \delta, s, F)$, каде Q е конечно множество на состојби, Σ е конечна влезна азбука, s е почетната состојба, F е множество крајни состојби, кое е подмножество од Q и δ е функција на премин која претставува пресликување од $Q \times \Sigma \to Q$. Тој може да се претстави со сврзан граф, каде состојбите од множеството Q се темиња, s е почетното теме, додека ребрата се означени со буквите од азбуката и тоа ако имаме премин од облик $\delta(q, a) = p$, каде $q, p \in Q$ и $a \in A$. Секој стринг кој го препознава детерминистичкиот конечен автомат претставува пат (во кој може да има и циклус) од почетното теме до некое од темињата во F.

   Теоремата на Клини кажува дека јазикот претставен со конечен автомат е ист со некој јазик кој се опишува со регуларен израз.

Направи варијанта на алгоритамот на Флојд Варшал кој за даден конечен автомат го пресметува соодветен регуларен.

6. (Алгоритам на Џонсон) Нека е даден тежински, ориентиран граф $G = (V, E)$ со функција на тежина w: $E \rightarrow \boldsymbol{R}$. Нека h: $V \rightarrow \boldsymbol{R}$ е било која функција која ги пресликува темињата во реални броеви. За секое ребро $(u, v) \in E$, дефинираме нова функција

$$w'(u, v) = w(u, v) + h(u) - h(v)$$

Нека $p = \langle v_0, v_2, \ldots, v_k \rangle$ е пат од темето v_0 до темето v_k. Покажи дека важи

a. p е најкраткиот пат од v_0 до v_k со функција на тежина w акко тој е најкраткиот пат со функција на тежина w', односно $w(p) = A(v_0, v_k)$ акко $w'(p) = A'(v_0, v_k)$.

b. G има циклус со негативна тежина, со користење на функцијата на тежина w акко G има циклус со негативна тежина, со користење на функцијата на тежина w'.

c. Дади алгоритам кој ќе го искористи репондерирањето, односно премерувањето од претходната задача, за да пресмета дали има негативен циклус во графот и ако нема ќе го искористи за наоѓање на најкратки патишта од секое до секое теме со помош на алгоритамот на Дикстра. Ова се нарекува алгоритам на Џонсон.

d. Која е комплексноста на алгоритамот на Џонсон?

7.6 Алгоритми за најлесно дрво

Моделирањето на патишта е само една од практичните ситуации во кои се користат тежински графови. Да го разгледаме проблемот на расчистување на снег по улиците. Кога снегот нагло ќе заврне, ги затрупува улиците и го пореметува сообраќајот. Претпријатијата кои се треба да го расчистат патот се среќаваат со проблемот на кои улици да им дадат предност при расчистувањето. Да претпоставиме дека се дадени населбите и множеството двонасочни улици меѓу нив. Знаеме дека претходно постоел барем еден начин да се стигне од една населба во друга, но по снежното невреме сите патишта се затрупани. Претпријатието треба да расчисти доволно улици, така да повторно граѓаните можат да комуницираат, т.е. да постои расчистен пат помеѓу било кој пар населби.

Проблемот може да се моделира со помош на граф $G(V, E)$ во кој локациите ќе се обележат со темиња, а патиштата кои можат да се воспостават да се обележат со ребра. Јасно е дека било кој сврзан подграф од G ќе биде решение на проблемот. Но, да претпоставиме дека претпријатието сака што е можно побргу да им излезе во пресрет на граѓаните, а знае колку време му е потребно за расчистување на секој од патиштата. Тогаш на градот не му треба било кое решение, туку она во кое претпријатието би потрошило најмалку време. Овој проблем е познат како проблем на наоѓање на минимално скелетно дрво и во општ случај треба да се најде подграф од даден тежински граф кој ќе го има својството да постои пат меѓу било кои две темиња во графот и дополнително тој пат да биде оптимален, со најмала или најголема тежина. Оваа оптималност може да се разгледува од повеќе аспекти, на пример, минимална цена, минимална вкупна должина, максимална добивка и слично. Ако графот не е тежински, или пак сите ребра имаат иста тежина, оптималниот подграф е било кое дрво што ги содржи сите темиња од графот. Ваквите дрва се нарекуваат **скелетни дрва**, и некое такво дрво може да се определи со некој од алгоритмите за пребарување BFS или DFS [15].

Ако се работи за тежински граф, тогаш не интересира да го најдеме оној сврзан подграф, кој има најмала тежина. Ако ребрата имаат позитивни тежини, тогаш тој подграф е дрво кое се нарекува **минимално скелетно дрво**.

Дефинирање на проблемот

Даден е сврзан неориентиран тежински граф $G(V, E)$ и функција на тежина $w: E \to \mathbb{R}^+$ ω, со која за секое ребро $\{u, v\} \in E$, имаме дадено тежина $w(\{u, v\}) > 0$ која ја специфицира цената која ни е потребна за да ги поврземе u и v. Проблемот е да се одбере подмножество од ребра од $E' \subseteq E$, такво што *вкупната тежина*

$$W(E') = \sum_{e \in E'} w(e),$$

да биде минимална.

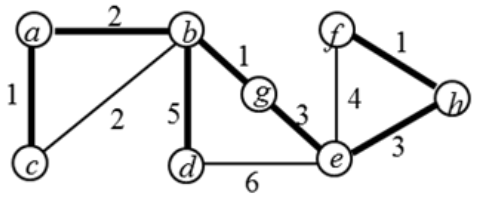
Анализа на проблемот

Прво да докажеме дека графот кој го бараме е дрво т.е. дека важи следново својство:

Својство

Ако $G'(V, E')$ е сврзан подграф од $G(V, E)$ со минимална вкупна тежина, тогаш G' е дрво.

Доказ: Да претпоставиме дека постои граф со минимална вкупна тежина кој не е дрво. Тогаш во тој граф мора да постои барем еден циклус. Ако извадиме едно од ребрата во тој циклус, графот ќе остане сврзан, а бидејќи сите тежини на ребрата се позитивни, вкупната тежина ќе се намали за некоја позитивна вредност и ќе стане помала, што е контрадикција со тврдењето дека таквиот граф има минимална тежина. □



Слика 7. 13. Минимално скелетно дрво на даден граф.

Користењето на ова својство проблемот го сведува на наоѓање на дрво $G'(V,T)$, каде $T \subseteq E$, кое ќе ги поврзе сите темиња за најмала можна вкупна цена. На Слика 7. 13 е дадено едно такво дрво кое има тежина 16. Може да се забележи дека за даден граф не мора да постои само едно вакво дрво, на пример за графот на сликата реброто {1, 2} може да се замени со реброто {2, 3} и да се добие друго минимално скелетно дрво.

Ако се дозволи ребрата да имаат тежина 0, тогаш е можно да постои сврзан подграф кој не е дрво со својството да биде најлесен подграф кој ги опфаќа сите темиња од графот, но тогаш со тргање на ребра со тежина 0 ќе се добие и дрво со ваквото својство. Така да решението на проблемот повторно може да се ограничи на наоѓање на најлесно опфаќачко дрво.

Во овој дел ќе се задржиме на два алчни алгоритми, алгоритамот на Крушкал и алгоритамот на Прајм. И двата алгоритми имаат иста сложеност ако се користи бинарен куп, $O(|E|\ln|V|)$, но алгоритамот на Крушкал, со користење на Фибоначиев куп може да се забрза до време $O(|V|\ln|V|)$, што би било побрзо ако V е помало од $O(|E|)$. На почеток кратко ќе опишеме уште еден алгоритам кој е обратна верзија на алгоритамот на Крушкал.

За да добиеме појасна слика за тоа кои својства ќе ни помогнат за да ја анализираме точноста на алгоритмите кои го решаваат проблемот за наоѓање на најоптимално скелетно дрво, накратко ќе ја објасниме идејата за секој од нив.

<u>Алгоритам на Прајм</u>:

Во овој алгоритам се бира произволно теме од кое ќе почне да се гради дрвото, кое се става во дрвото. Потоа во секој чекор се додава

ново теме од дрвото на тој начин што се додава најлесното ребро кое сврзува теме од дрвото со теме кое сеуште не е ставено во дрвото.

Алгоритам на Крушкал:

Алгоритамот на Крушкал гради шума која е подграф од почетниот граф. Имено, на почеток графот нема ребра, а секое теме претставува дрво во графот. Потоа во секој наредeн чекор се сврзуваат две дрва од графот така што се додава најлесното ребро кое сврзува две различни дрва од шумата. Тоа се прави се додека не се спојат сите дрва од шумата во едно единствено дрво. Можеме да забележиме дека едно ребро ќе спојува две различни дрва, ако со негово додавање не се јавува циклус. Оттука, овој алгоритам прво ги подредува ребрата во растечки редослед, а потоа во секој чекор го додава наредното ребро од подредената низа, но само под услов тоа да не прави циклус со веќе додадените ребра.

Обратен алгоритам на Крушкал:

Обратниот алгоритам почнува спротивно, со целосниот граф и во секој чекор се вади по едно ребро од него, она ребро кое е најисплатливо да се извади во тој момент, но кое нема да ја наруши сврзаноста на графот. Всушност, тоа е реброто кое од ребрата кои не се мостови, т.е. ребрата со својство кога ќе ги отстраниме графот да остане сврзан, има најголема тежина. Поточно, во секој момент се вади најтешкото ребро кое е дел од некој циклус.

Од овој краток опис е јасно дека сите овие алгоритми се алчни алгоритми. Во првите два кон графот се додава реброто кое во тој момент е најисплатливо да се додаде, додека во последниот алгоритам се одзема реброто кое во тој момент е најисплатливо да се извади од графот. Првите два алгоритми имаат ист пристап, односно во секој момент се додава по едно ребро од дрвото. Тие ја одржуваат следнава општа инваријанта на циклус:

> *Во секој чекор од алгоритамот се гради подграф* $G_A = (V, A)$ *од графот* $G(V, E)$ *, таков што* A *е подмножество од ребра на некое минимално скелетно дрво на* G.

Не баш секое ребро може да се додаде на графот, поточно на графот може да се додаваат само ребра кои во тој момент го задоволуваат условот да не прават циклус. Всушност во секој чекор се определува едно ребро $\{u, v\}$ кое може да се додаде на A без да се прекрши инваријантата. Понатаму ќе ги специфицираме и анализираме овие два алгоритми.

Алгоритам на Крушкал

Алгоритамот на Крушкал работи на тој начин што во секој чекор додава по едно ребро во растечката шума. Иницијално шумата нема ниту едно ребро и секое теме од графот претставува едно дрво во шумата. Тоа значи дека на почеток шумата е графот $(V, \emptyset)$. Додавањето се врши на тој начин што од сите ребра кои поврзуваат две дрва во шумата го наоѓа најлесното ребро кое ако се додаде во графот, во графот нема да се добие циклус. Алгоритамот терминира кога ќе се добие сврзан граф, односно шума со само едно дрво.

Алгоритамот на Крушкал е алчен алгоритам, бидејќи со него во секој чекор кон шумата се додава едно ребро со најмала можна тежина. Работата на алгоритамот е прикажана на Слика 7.6.2.

Прво во кратки црти да го дадеме псевдокодот на алгоритамот, а потоа ќе ја докажеме неговата точност и ќе се задржиме на неговата имплементација со акцент на сложеноста во зависност од имплементацијата.

П 7. 8. ОСНОВЕН АЛГОРИТАМ НА КРУШКАЛ

```
1  Внеси го графот G(V, E) и функција на тежина w.
2  KrushkalPoceten(G(V, E), w)
3  A = ∅;
4  Сортирај ги ребрата во неопаѓачки редослед по нивната
   тежина
5  за секое ребро {u, v} ∈ E земено по неопаѓачки редослед
6      ако реброто сврзува две различни сврзани компоненти
7          тогаш
8              A = A ∪{(u, v)};
9  врати A.
```

За да се покаже точноста на овој алгоритам треба да се покажат две работи, прво, дека алгоритамот продуцира скелетно дрво и второ дека тоа дрво е со најмала тежина. За да покажеме дека

алгоритамот продуцира скелетно дрво треба да покажеме дека важи следново својство:

Својство

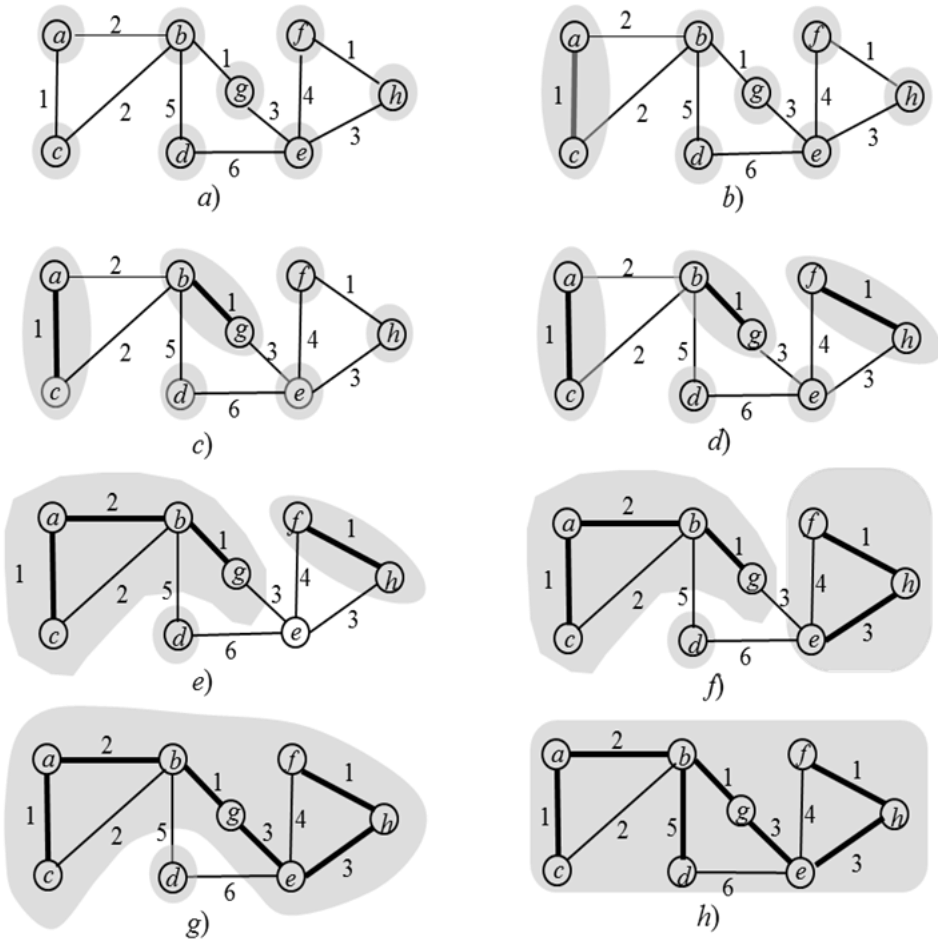
Ако $G(V,E)$ е сврзан тежински неориентиран граф и E' е множеството ребра продуцирани од алгоритамот на Крушкал. Тогаш графот $G'(V,E')$ е ацикличен сврзан граф.

Доказ: Ако графот $G'(V,E')$ има циклус, тогаш некое од додадените ребра ќе биде последното ребро кое ќе го затвара циклусот, па тоа ребро нема да биде ребро кое поврзува две различни дрва, туку ребро кое поврзува темиња од едно исто дрво. Од друга страна $G'(V,E')$ не може да биде несврзан, бидејќи првото теме кое доаѓа на ред и кое сврзува две дрва од графот ќе биде додадено на графот. □

За да се покаже дека дрвото кое се продуцира со алгоритамот на Крушкал е најлесното дрво треба да се покаже општата инваријанта на циклус:

> *Нека $G(V,E)$ е сврзан тежински неориентиран граф со реална функција на тежина w дефинирана над E. Нека A е подмножеството од E кое е се добива во некој чекор при работата на алгоритамот на Крушкал. Тогаш постои минимално скелетно дрво кое ги содржи сите ребра од множеството A.*

Доказ на инваријантата: На почеток $A = \emptyset$, па е подмножество од множеството ребра на било кое минимално скелетно дрво на G. Нека пред влегување во циклусот знаеме дека множеството A е подмножество од некое минимално скелетно дрво. Тогаш кога ќе влеземе во циклусот, земаме едно ребро $\{u,v\}$ и прво испитуваме дали реброто сврзува две различни дрва во шумата. Ако сврзува две ралични дрва, тогаш нема да го додадеме, па A останува исто, а со тоа по излегувањето од циклусот повторно A е подмножество од множеството на ребра на некое минимално скелетно дрво на G.



Слика 7. 14. Работа на алгоритамот на Крушкал. Во секој чекор се додава ново ребро и едно од дрвата расте за едно теме. Во првиот чекор секое теме претставува едно дрво. Во последниот чекор, сите темиња се во едно дрво.

Останува да се разгледа случајот кога $\{u, v\}$ поврзува две различни дрва од шумата. Нека A е подмножество од множеството ребра на минималното скелетно дрво $G_T(V, E_T)$ и нека претпоставиме дека $A \cup \{\{u, v\}\}$ не е подмножество од множеството ребра на G_T. Тоа значи дека ако на G_T го додадеме реброто $\{u, v\}$ ќе добиеме циклус. Мора да постои некое ребро e во E_T кое не се наоѓа во A, бидејќи ако се овде сите ребра од тој циклус освен $\{u, v\}$, тогаш $\{u, v\}$ не би сврзувал две различни сврзани компоненти. Јасно е дека e не може да има помала тежина отколку $\{u, v\}$, затоа што инаку тоа било избрано пред $\{u, v\}$, а во таков случај би било додадено на

растечкото дрво. Сега графот $G'_T(V, E_T - \{e\} \cup \{\{u, v\}\})$ е повторно дрво со иста или помала тежина од G_T и ги содржи сите ребра од A. Бидејќи претпоставивме дека G_T е минимално скелетно дрво, не може да постои скелетно дрво со помала тежина од него, па следува дека G_T и G'_T имаат иста тежина, т.е. G'_T е минимално скелетно дрво кое го содржи $A \cup \{\{u, v\}\}$. □

Во секој чекор во основната верзија на алгоритамот на Крушкал треба да се пресмета дали даденото ребро сврзува две различни дрва или не, т.е. треба да се увиди дали темињата кои се сврзуваат со даденото ребро во тој момент се во исто дрво или не. Имено тоа би значело дека во моментот кога се проверува дали во дрвото да се додаде реброто $\{u, v\}$, треба да се види дали тие две темиња се претходно сврзани. Всушност ефективноста на алгоритамот зависи токму од овој чекор. Наивен, т.е. наједноставен начин да се направи оваа проверка е со некој од алгоритмите за пребарување, BFS или DFS, но за ова да можеме да го имплементираме, во секој чекор треба да градиме нов граф со додавање новите ребра. Ако за презентација на графот користиме листа на соседство, тогаш при додавањето на реброто $\{u, v\}$, во листата на соседство на u ќе се додаде v и обратно, во листата на соседство на v ќе се додаде u.

Постапката за додавање на елементи во листите на соседство зафаќа константно време, но од друга страна за алгоритамот за пребарување со кој треба да се определи дали реброто кое кандидат за да се стави во дрвото прави циклус или не се троши време $O(|E|)$. Тоа би значело дека целиот алгоритам би имал сложеност $O(|E|(|V| + |E|))$.

Сето ова може да се изведе значително побргу со користење на структурата *дисјунктни множества*. Растечката шума која се гради во текот на работата на алгоритамот на Крушкал може да се зачувува во коренови дрва. Секое кореново дрво ќе биде едно од дрвата во шумата. Припадноста на едно теме во едно дрво може ќе се бележи со коренот на дрвото. Проверката дали две темиња се во исто дрво или не ќе се направи со споредба на корените на дрвата во кои тие припаѓаат. Ако двете темиња имаат ист корен, значи

припаѓаат во исто дрво, па реброто не треба да се додаде, ако пак немаат ист корен, тогаш припаѓаат во различно дрво и реброто треба да се додаде. Дрвата стандардно можат да се зачувуваат преку функцијата π, со која на секое теме му се зачувува неговиот родител. Како и во секоја структура за зачувување на дрва, ако темето е коренот, тогаш нема родител, па зачувуваме дека неговиот родител е NIL.

Додавањето може да се направи на тој начин што на коренот на едното од дрвата ќе се закачи коренот на другото дрво. За да го најдеме коренот на дрвото на некое теме u, треба да се искачуваме до коренот на дрвото во кое тоа теме се наоѓа. Тоа го правиме со пресметување на неговиот родител, па родителот на родителот итн., што е најдобро да се имплементира рекурзивно, како што е дадено со процедурата $NajdiKoren$:

П 7. 9. НАОЃАЊЕ НА КОРЕН НА ДРВО

```
1  NajdiKoren(v)
2    ако π[v] == NIL тогаш врати v инаку врати NajdiKoren(π[v])
```

За да добиеме што е можно поголем бенефит од користењето на ваквите дрва, истите треба да тежнееме да бидат што е можно пониски, во спротивно пребарувањето по нив би имало иста сложеност како пребарувањето со алгоритмите за пребарување. Затоа воведуваме нова функција h со која ќе ја бележиме висината на дрвото. На почеток висината на секое дрво е 0. Стратегијата за добивање на што е можно пониско дрво се базира на тоа пониското дрво да се закачува на коренот на повисокото, т.е. за родител на коренот на пониското дрво да се става коренот на повисокото дрво. Ако дрвата се со еднаква висина, тогаш не е битно кое дрво ќе се закачи на коренот на другото, но она што треба да се забележи дека во секој случај новодобиеното дрво ќе ја зголеми својата висина за 1. Во ваков случај треба да се обнови висината на дрвото, како што е дадено во функцијата $Unija$. Ако едно од дрвата е пониско од другото, тогаш добиеното дрво не може да биде повисоко отколку повисокото од претходните дрва. Во ваква реализација, ако

темињата веќе се во иста сврзана компонента, т.е. исто дрво, со процедурата $Unija$ нема да се промени ништо.

П 7. 10. АЛГОРИТАМ ЗА НАОЃАЊЕ НА УНИЈА

```
1   Unija(u, v)
2   uKoren = NajdiKoren(u);
3   vKoren = NajdiKoren(v);
4   ако uKoren ≠ vKoren тогаш
5       ако h(uKoren) < h(vKoren) тогаш
6           π(uKoren) = vKoren
7       инаку ако h(uKoren) > h(vKoren) тогаш
8               π(vKoren) = uKoren инаку
9                   {
10                    π(vKoren) = uKoren;
11                    h(uKoren) = h(uKoren) + 1.
12                  }
```

Конечно можеме да го дадеме комплетниот алгоритам на Крушкал:

П 7. 11. АЛГОРИТАМ НА КРУШКАЛ

```
1   внеси го графот G(V, E) и тежинската функција w
2   A = ∅;
3   за секое теме v ∈ V прави
4     {
5       π(v) = NIL
6       h(v) = 0;
7     }
8   Сортирај ги ребрата во неопаѓачки редослед по нивната
    тежина
9   за секое ребро {u, v} ∈ E по неопаѓачки редослед
10    ако NajdiKoren(u) ≠ NajdiKoren(v) тогаш
```

```
11        {
12          Unija(u, v);
13          A = A ∪{{u, v}};
14        }
15  врати A.
```

Да го анализираме времето на работа на опишаната имплементација на алгоритамот на Крушкал. Јасно е дека сортирањето на ребрата по тежина зазема време $O(|E| \ln(|E|))$. Останува да се пресмета времето кое се троши за функциите $NajdiKoren$ и $Unija$. Во функцијата $Unija$ време зафаќа само функцијата $NajdiKoren$, сите други операции се прават во константно време. Функција $NajdiKoren$ троши време колку што е висината на дрвото на кое го бара коренот, а тоа е најмногу $\log_2|E|$ пати. Оттука, бидејќи оваа функција се повикува за секое ребро, вкупната сложеност е $O(|E| \ln(|E|))$.

Кодовите во Јава и C++ кои ги даваме овде се итеративни решенија кои се базираат на третиот пристап.

JAVA 7. 4 АЛГОРИТАМ НА КРУШКАЛ

```java
import java.util.*;
import java.lang.*;
import java.io.*;

class Graph {
    class Edge implements Comparable<Edge> {
        int src, dest, weight;

        public int compareTo(Edge compareEdge) {
            return this.weight - compareEdge.weight;
        }
    }

    ;

    class subset {
        int parent, rank;
    }
```

```
19
20      ;
21
22      int V, E;
23      Edge edge[];
24
25      Graph(int v, int e) {
26          V = v;
27          E = e;
28          edge = new Edge[E];
29          for (int i = 0; i < e; ++i)
30              edge[i] = new Edge();
31      }
32
33      int find(subset subsets[], int i) {
34          if (subsets[i].parent != i)
35              subsets[i].parent = find(subsets,
subsets[i].parent);
36
37          return subsets[i].parent;
38      }
39
40      void Union(subset subsets[], int x, int y) {
41          int xroot = find(subsets, x);
42          int yroot = find(subsets, y);
43
44          if (subsets[xroot].rank < subsets[yroot].rank)
45              subsets[xroot].parent = yroot;
46          else if (subsets[xroot].rank > subsets[yroot].rank)
47              subsets[yroot].parent = xroot;
48
49          else {
50              subsets[yroot].parent = xroot;
51              subsets[xroot].rank++;
52          }
53      }
54
55      void KruskalMST() {
56          Edge result[] = new Edge[V];
57          int e = 0;
58          int i = 0;
```

```
59          for (i = 0; i < V; ++i)
60              result[i] = new Edge();
61
62          Arrays.sort(edge);
63
64          subset subsets[] = new subset[V];
65          for (i = 0; i < V; ++i)
66              subsets[i] = new subset();
67
68          for (int v = 0; v < V; ++v) {
69              subsets[v].parent = v;
70              subsets[v].rank = 0;
71          }
72
73          i = 0;
74
75          while (e < V - 1) {
76              Edge next_edge = new Edge();
77              next_edge = edge[i++];
78
79              int x = find(subsets, next_edge.src);
80              int y = find(subsets, next_edge.dest);
81
82              if (x != y) {
83                  result[e++] = next_edge;
84                  Union(subsets, x, y);
85              }
86          }
87
88          System.out.println("Slednive rebra vleguvaat vo
minimalnoto skeletno drvo.");
89          for (i = 0; i < e; ++i)
90              System.out.println(result[i].src + " -- " +
91                      result[i].dest + " == " +
result[i].weight);
92      }
93
94      public static void main(String[] args) {
95          Scanner sc = new Scanner(System.in);
96          int V;
97          int E;
```

```
        V = sc.nextInt();
        E = sc.nextInt();
        Graph graph = new Graph(V, E);
        for (int i = 0; i < E; i++) {
            int s = sc.nextInt();
            int d = sc.nextInt();
            int w = sc.nextInt();
            graph.edge[i].src = s;
            graph.edge[i].dest = d;
            graph.edge[i].weight = w;
        }
        graph.KruskalMST();
    }
}
```

```
1   #include <bits/stdc++.h>
2      using namespace std;
3
4      class Edge
5      {
6         public:
7         int src, dest, weight;
8      };
9
10     class Graph
11     {
12        public:
13        int V, E;
14        Edge* edge;
15     };
16
17     Graph* createGraph(int V, int E)
18     {
19        Graph* graph = new Graph;
20        graph->V = V;
21        graph->E = E;
22
23        graph->edge = new Edge[E];
24
25        return graph;
26     }
27
28     class subset
29     {
30        public:
31        int parent;
32        int rank;
33     };
34
35     int find(subset subsets[], int i)
36     {
37        if (subsets[i].parent != i)
38           subsets[i].parent = find(subsets,
subsets[i].parent);
39
```

```
40          return subsets[i].parent;
41      }
42
43      void Union(subset subsets[], int x, int y)
44      {
45          int xroot = find(subsets, x);
46          int yroot = find(subsets, y);
47
48          if (subsets[xroot].rank < subsets[yroot].rank)
49              subsets[xroot].parent = yroot;
50          else if (subsets[xroot].rank > subsets[yroot].rank)
51              subsets[yroot].parent = xroot;
52
53          else
54          {
55              subsets[yroot].parent = xroot;
56              subsets[xroot].rank++;
57          }
58      }
59
60      int myComp(const void* a, const void* b)
61      {
62          Edge* a1 = (Edge*)a;
63          Edge* b1 = (Edge*)b;
64          return a1->weight > b1->weight;
65      }
66
67      void KruskalMST(Graph* graph)
68      {
69          int V = graph->V;
70          Edge result[V];
71          int e = 0;
72          int i = 0;
73
74          qsort(graph->edge, graph->E, sizeof(graph->edge[0]),
myComp);
75
76          subset *subsets = new subset[( V * sizeof(subset)
)];
77
78          for (int v = 0; v < V; ++v)
```

```
79          {
80             subsets[v].parent = v;
81             subsets[v].rank = 0;
82          }
83
84          while (e < V - 1 && i < graph->E)
85          {
86             Edge next_edge = graph->edge[i++];
87
88             int x = find(subsets, next_edge.src);
89             int y = find(subsets, next_edge.dest);
90
91             if (x != y)
92             {
93                result[e++] = next_edge;
94                Union(subsets, x, y);
95             }
96          }
97
98          cout<<"Slednite rebra vleguvaat vo minimalnoto
skeletno drvo\n";
99          for (i = 0; i < e; ++i)
100            cout<<result[i].src<<" -- "<<result[i].dest<<" ==
"<<result[i].weight<<endl;
101         return;
102      }
103
104      int main()
105      {
106         int V, E;
107         cin >> V >> E;
108         Graph* graph = createGraph(V, E);
109
110         for(int i=0;i<E;i++)
111         {
112            int s, d, w;
113            cin >> s >> d >> w;
114            graph->edge[i].src = s;
115            graph->edge[i].dest = d;
116            graph->edge[i].weight = w;
117         }
```

```
118         KruskalMST(graph);
119         return 0;
120     }
```

Алгоритам на Прајм

Во алгоритамот на Прајм, слично како и во алгоритамот на Крушкал, во секој чекор се додава по едно ребро од растечкото минимално скелетно дрво, но за разлика од него, овој алгоритамот го има својството ребрата кои се додаваат секогаш да формираат само едно дрво. Како корен од кој дрвото почнува да се гради можеме да го избереме било кое произволно теме. Дрвото расте се додека не ги спои сите темиња од V.

На почеток дрвото нема ниту едно ребро и се бираме едно теме кое се става во дрвото. Додавањето на ребра се врши на тој начин што се бира најлесното ребро кое кон дрвото приврзува теме кое сеуште не е додадено на дрвото. Ако има повеќе такви ребра, може да се избере било кое од нив. Алгоритамот терминира кога кон дрвото ќе се припојат сите темиња од V.

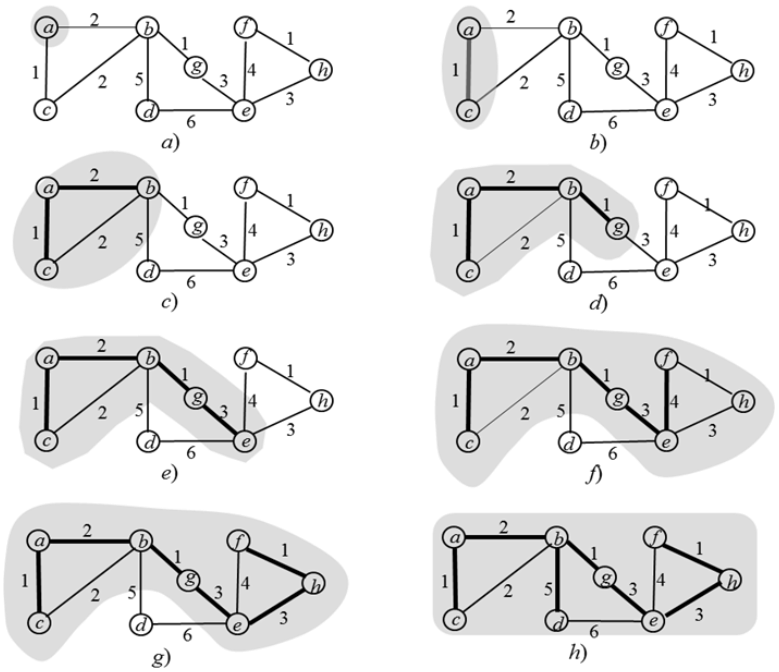
И овој алгоритам е алчен алгоритам, бидејќи со него во секој чекор во растечкото дрво се додава реброто кое најмногу одговара во тој момент. Работата на алгоритамот е прикажана на Слика 7. 15.

Во кратки црти да го дадеме псевдокодот на алгоритамот, а потоа ќе се задржиме на анализа како е најдобро истиот да се имплементира.

П 7. 12. Алгоритам на Прајм

```
1  внеси го графот G(V, E) и тежинската функција w
2  A = ∅;
3  V' = {x};
4  додека V'≠V прави
5      {
6       избери ребро со минимална тежина {u, v} ∈ E,
7            такво што u ∈ V', а v ∉ V';
8       A = A∪{{u, v}};
9       V' = V'∪{v};
10     }
```

11 врати A.



Слика 7. 15. Алгоритам на Прајм во кој дрвото почнува да се формира од темето a. Во секој чекор во скелетното дрво се додава по едно теме.

Да се докаже точноста на овој алгоритам повторно треба да се покаже дека алгоритамот продуцира скелетно дрво и дека тоа дрво е со најмала тежина. За тоа ни е потребно следново својство:

Својство

Нека е даден сврзан тежински неориентиран граф $G(V,E)$ Тогаш графот $G'(V',A)$ кој се продуцира во секој чекор при алгоритамот на Прајм е дрво и кога алгоритамот терминира добиениот граф е скелетно дрво за G.

Доказ: Јасно дека почетниот граф со едно теме без ребра е дрво, и да претпоставиме дека до моментот додека да се добие графот $G'(V',A)$ ова својство е запазено. Значи помеѓу било кои две темиња од графот G' постои единствен пат и истиот не содржи циклус. Јасно, додавањето на новото теме v и новото ребро $\{u,v\}$ не го нарушува

постоењето на пат меѓу темињата кои претходно биле додадени во дрвото. Затоа доволно е само да се покаже дека постои единствен пат од v до било кое друго теме. Пат постои бидејќи постои пат од v до u и од u до било кое друго теме од графот. Овој пат е единствен, бидејќи единствен пат од v до u е реброто $\{u, v\}$, а од v до секое друго теме постои само еден пат. □

За да се покаже дека алгоритамот на Прајм го продуцира најлесното дрво, повторно ни треба општата инваријанта а циклус.

> *Нека $G(V, E)$ е сврзан тежински неориентиран граф со реална функција на тежина w дефинирана над E. Нека $G' = (V', A)$ е графот кој се добива во некој чекор при работата на алгоритамот на Прајм. Тогаш постои минимално скелетно дрво кое ги содржи сите ребра од множеството A.*

Доказ на инваријантата: На почеток $A = \emptyset$, па е подмножество од множествота ребра на било кое минимално скелетно дрво на G. Нека пред да се влезе во циклусот знаеме дека A е подмножество од множеството ребра на минималното скелетно дрво $G_T(V, E_T)$. Ако по додавањето на реброто $\{u, v\}$, $A \cup \{\{u, v\}\}$ е подмножество од E_T, доказот е во ред. Сега да претпоставиме дека со додавањето на реброто $\{u, v\}$, $A \cup \{\{u, v\}\}$ не е подмножество од E_T. Тогаш бидејќи $G_T(V, E_T)$ е дрво, мора во него да постои единствен пат од u до v. Да го земеме првото ребро по тој пат кое во моментот поврзува теме кое е во V' со теме кое не е во V'. Нека тоа ребро е реброто e. Ова ребро e сеуште не е додадено во G' и не може да има помала тежина од $\{u, v\}$, затоа што инаку би било најлесното ребро кое поврзува теме од V' со теме надвор од V'. Значи e има поголема или иста тежина со $\{u, v\}$. Да го разгледаме графот што се добива од G_T со отстранување на реброто e и додавање на реброто $\{u, v\}$, т.е. графот дефиниран со $G'_T(V, E_T - \{\{u, v\}\}) \cup \{e\})$. Овој граф е повторно дрво со иста или помала тежина од G_T, ги содржи сите ребра од A и плус реброто $\{u, v\}$. Бидејќи претпоставивме дека G_T е минимално скелетно дрво, не може да постои скелетно дрво со помала тежина од него, па следува дека G_T и G'_T имаат иста тежина (па и e и $\{u, v\}$ имаат иста тежина). Така ново-конструираното дрво G'_T е минимално скелетно дрво кое го содржи $A \cup \{\{u, v\}\}$. □

Алгоритамот на Прајм во секој чекор го избира реброто со минимална тежина кое има едно теме во дрвото, а друго надвор од дрвото, а ова е најдобро да се имплементира со истата структура која ја користевме за имплементација на алгоритамот на Дикстра, односно со мин-бинарен куп во кој се чуваат темињата од графот кои сеуште не се ставени во дрвото. Оттука, имплементацијата на алгоритмите на Прајм и Дикстра е многу слична. Разликата е во тоа што клучот на темињата во бинарниот куп во алгоритамот на Дикстра е најкраткото растојание од изворот до соодветното теме, додека овде е најкраткото ребро од веќе формираниот дел од дрвото до соодветното теме. На почеток, стандардно, во Q се стават сите темиња од графот и иницијално клучот за секое теме кој ќе го зачувуваме во функцијата l се подесува на вредност ∞. Функцијата π ќе биде NIL за секое теме, затоа што сеуште немаме ставено ниту едно теме во дрвото, па нема ребро кое поврзува теме од дрвото со теме надвор од него. Во првиот чекор бираме произволно теме кое ќе го ставиме во дрвото, едноставно може да го избереме првото теме од V. Нека ова теме го обележиме со x. Во веригата Q поставуваме $l(x) = 0$ и за сите темиња v од Q кои се поврзани со x со ребро ја ажурираме вредноста на функциите l и π така што $l(v) = w(x, v)$, а $\pi(v) = x$. Во секоја понатамошна итерација кон дрвото се приклучуваме темето од Q кое има најмала вредност за функцијата l, заедно со реброто $\{v, \pi(v)\}$. Сега алгоритамот можеме да го претставиме со следниов псевдокод:

П 7. 13. АЛГОРИТАМ НА ПРАЈМ

```
1   Внеси го графот G(V, E) и тежинската функција w;
2   за секое v ∈ V прави
3     {
4       l(v) = ∞;
5       π(v) = NIL;
6     }
7   избери x прв елемент од V и стави l(x) = 0;
8   Q = V;
9   A = ∅;
10  V' = {x};
```

11 **додека** $Q \neq \emptyset$ **прави**

12 {

13 извади теме $v \in Q$ за кое $l(v) = \min_{u \in Q} l(u)$;

14 $A = A \cup \{\{u, v\}\}$;

15 $V' = V' \cup \{v\}$;

16 **за** секое u од листата на соседство на v **прави**

17 **ако** $u \notin Q$ и $w(\{u, v\}) < l(u)$ **тогаш**

18 {

19 $\pi(u) = v$;

20 $l(u) = w(\{u, v\})$;

21 }

22 }

23 **врати** A.

Кодовите во Јава и C++ кои ги даваме овде се итеративни решенија кои се базираат на третиот пристап.

JAVA 7. 5 АЛГОРИТАМ НА ПРАЈМ

```
1   import java.util.*;
2  import java.lang.*;
3  import java.io.*;
4
5  class Main {
6     private static int V;
7
8     static int minKey(int key[], Boolean mstSet[]) {
9        int min = Integer.MAX_VALUE, min_index = -1;
10
11       for (int v = 0; v < V; v++)
12          if (mstSet[v] == false && key[v] < min) {
13             min = key[v];
14             min_index = v;
15          }
16
17       return min_index;
18    }
19
20    static void printMST(int parent[], int graph[][]) {
21       System.out.println("Rebro \tTezhina");
22       for (int i = 1; i < V; i++)
```

```
23          System.out.println(parent[i] + " - " + i + "\t" +
graph[i][parent[i]]);
24      }
25
26      static void primMST(int graph[][]) {
27          int parent[] = new int[V];
28          int key[] = new int[V];
29          Boolean mstSet[] = new Boolean[V];
30
31          for (int i = 0; i < V; i++) {
32              key[i] = Integer.MAX_VALUE;
33              mstSet[i] = false;
34          }
35
36          key[0] = 0;
37          parent[0] = -1;
38
39          for (int count = 0; count < V - 1; count++) {
40              int u = minKey(key, mstSet);
41
42              mstSet[u] = true;
43
44              for (int v = 0; v < V; v++)
45                  if (graph[u][v] != 0 && mstSet[v] == false &&
graph[u][v] < key[v]) {
46                      parent[v] = u;
47                      key[v] = graph[u][v];
48                  }
49          }
50
51          printMST(parent, graph);
52      }
53
54      public static void main(String[] args) {
55          Scanner sc = new Scanner(System.in);
56          int E;
57          V = sc.nextInt();
58          E = sc.nextInt();
59          int graph[][] = new int[V][V];
60          for (int i = 0; i < E; i++) {
61              int s = sc.nextInt();
62              int d = sc.nextInt();
63              int w = sc.nextInt();
64              graph[s][d] = w;
65              graph[d][s] = w;
66          }
67          primMST(graph);
```

```
68    }
69 }
```

C++ 2. 6 Алгоритам на Прајм

```
1   #include <bits/stdc++.h>
2  using namespace std;
3
4  #define INF INT_MAX
5  static int V;
6
7  int minKey(int key[], bool mstSet[])
8  {
9     int min = INT_MAX, min_index;
10
11    for (int v = 0; v < V; v++)
12       if (mstSet[v] == false && key[v] < min)
13          min = key[v], min_index = v;
14
15    return min_index;
16 }
17
18 void printMST(int parent[], vector<vector<int>> &graph)
19 {
20    cout<<"Rebro \tTezhina\n";
21    for (int i = 1; i < V; i++)
22       cout<<parent[i]<<" - "<<i<<"
\t"<<graph[i][parent[i]]<<" \n";
23 }
24
25 void primMST(vector<vector<int>> &graph)
26 {
27    int parent[V];
28    int key[V];
29    bool mstSet[V];
30
31    for (int i = 0; i < V; i++)
32       key[i] = INT_MAX, mstSet[i] = false;
33
34    key[0] = 0;
35    parent[0] = -1;
36
37    for (int count = 0; count < V - 1; count++)
38    {
39       int u = minKey(key, mstSet);
40       mstSet[u] = true;
41
```

```
42        for (int v = 0; v < V; v++)
43           if (graph[u][v] && mstSet[v] == false &&
graph[u][v] < key[v])
44              parent[v] = u, key[v] = graph[u][v];
45     }
46
47     printMST(parent, graph);
48  }
49
50  int main()
51  {
52     int E;
53     cin >> V >> E;
54     vector<vector<int>> graph(V, vector<int>(V, INF));
55
56     for(int i=0;i<E;i++)
57     {
58        int s, d, w;
59        cin >> s >> d >> w;
60        graph[s][d] = w;
61        graph[d][s] = w;
62     }
63
64     primMST(graph);
65
66     return 0;
67  }
```

Да ја разгледаме сложеноста на алгоритамот. На почеток да забележиме дека за иницијализацијата на сите елементи се троши време $O(|V|)$, па целата сложеност ќе зависи од начинот како ќе се имплементиа мин-приоритетната верига. Наједноставно е да се искористи бинарен куп, за кој како што напоменавме претходно вадењето на минималниот елемент и ажурирањето на купот зазема време $\mathrm{O}(ln|V|)$. Можеме да забележиме дека овие операции би се правеле најмногу за секое ребро по еднаш, затоа што само еднаш секое ребро може да стане ребро кое излегува од новододаденото теме кон растечкото дрво и теме надвор од тоа дрво. Така вкупното време кое ќе се троши за оваа операција е $\mathrm{O}(|\mathrm{E}|ln|V|)$. Сумарно, комплексноста на алгоритамот е $\mathrm{O}(|E|ln|V|)$. Секако, наместо бинарен куп може да се искористи и Фибоначиев куп, што ќе биде побрза, но посложено за имплементација.

Прашања и задачи

1. Максимално скелетно дрво за даден сврзан неориентиран тежински граф $G(V, E)$ со функција на тежина w е скелетно дрво со максимална можна тежина. Направи модификација на алгоритмите на Крушкал и Прајм за наоѓање на максимално скелетно дрво.

2. Покажи дека во сврзан неориентиран тежински граф $G(V, E)$ со функција на тежина w ребрата кои не лежат на циклус сигурно се ребра од било кое минимално скелетно дрво.

3. Нека e е ребро со минимална тежина во сврзан неориентиран тежински граф $G(V, E)$ со тежинска функција w. Покажи дека постои скелетно дрво во кое се наоѓа e.

4. Нека e е ребро со максимална тежина во сврзан неориентиран тежински граф $G(V, E)$ со тежинска функција w. Покажи дека постои скелетно дрво во кое не се наоѓа e.

5. Покажи дека ако во графот $G(V, E)$ со функција на тежина w е дозволено да има и ребра со негативна тежина, тогаш неговиот подграф со минимална тежина не мора да биде дрво.

6. Кабелската мрежа во која имало n корисници била купена од две фирми. Претходно кабелската имала поставено комуникациски линии кои дозволувале да постои комуникација меѓу секои двајца корисници. Двете фирми си ги поделиле линиите така да за секоја линија била одговорна или првата или втората фирма. Бидејќи веќе постоечката мрежа била застарена новите сопственици сакале да ги заменат, така да заменат најмал број на линии и секои два корисници да можат да комуницираат по новите линии. За да не прават дополнителни трошоци за копање, тие сакале да ги искористат веќе постоечките канали за жиците. Договорот бил секој да ги заменува жиците за кои е одговорен, но така да првата фирма треба да замени точно $k, k < n$ од линиите, а втората да замени точно $n - k - 1$ линии. Дадете алгоритам кој определува дали ова е можно и ако е можно дава дава едно такво дрво.

7. Администраторите на една компјутерска мрежа во која модемите се поврзани со кабли установиле дека ако каблите се подолги, тогаш брзината на пренос на податоците се намалува. Множеството модеми е V, а множеството кабли е E и се смета дека е позната должината на секој кабел.

8. Дади алгоритам кој работи во време $O(|E|)$ со кој администраторите можат да проверат дали постои подмрежа од дадената мрежа во која нема ниту еден кабел подолг од дадена должина b.

9. Дади алгоритам со кој може да се изгради подмрежа од дадената мрежа со која секој пар од темиња ќе биде поврзан, со најмала можна должина на најдолгиот кабел кој ќе биде вклучен во неа.

8 Динамичко програмирање со бит-маски

Техниката со бит-маски најчесто се користи за решавање на проблеми кои се во класата на НП комплексни проблеми, но и такви за кои е покажано дека не можат да се решат во полиномно време. Под бит-маска се подразбира подмножество од множеството $\{1,2,\dots,n\}$ кое е претставено со низа од битови во која 1 на позиција i означува дека i-тиот елемент се наоѓа во подмножеството, а 0 на позиција i означува дека i -тиот елемент не се наоѓа во подмножеството. Оптималното својство кое е карактеристично за овие проблеми е такво што проблемот над дадено множество го намалува на потпроблеми над подмножества од тоа множество.

Со оваа техника се решаваат и комбинаторни и за оптимизациони проблеми. Комбинаторните проблеми најчесто се поврзани со број на пермутации со одредено својство, додека оптимизационите проблеми во основа имаат некоја тежинска функција за секоја пермутација, па треба да се добие пермутацијата за која таа тежинска функција е оптимална. Наивниот пристап е да се генерираат сите можности еден по еден и да се избројат или за секој од нив да се пресмета тежинската функција, што би довело до решение со комплексност $O(n!)$, каде n е големината на влезот. Динамичкото програмирање помага сложеноста на решението на да се намали до експоненцијална временска и просторна сложеност.

Натпреварувачките задачите кои се решаваат со бит-маски се препознаваат по тоа што немаат големи ограничувања, па n е најчесто во дијапазон од 15 до 20, иако во некои ситуации можат да се искористат дополнителни техники со кои може да се реши проблем со големина до 50. Во овој дел ќе објасниме неколку карактеристични проблеми кои ја илустрираат оваа техника.

8.1 Избор на броеви од матрица

Првиот проблем што го разгледуваме е едноставен пример за илустрација на техниката со бит-маски.

Дефинирање на проблемот

За дадена матрица X од реални броеви, од ред $n \times n$ треба да се избере точно по еден број од секоја редица и секоја колона,така да нивниот збир биде минимален.

Анализа на проблемот

Да разгледаме како и го решиле проблемот на пример. Нека е дадена матрицата од ред 4×4:

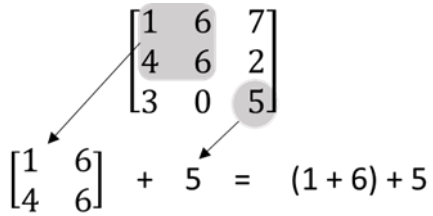
$$X = \begin{bmatrix} 1 & 6 & 7 & 2 \\ 4 & 6 & 2 & 5 \\ 3 & 0 & 5 & 8 \\ 2 & 1 & 6 & 4 \end{bmatrix}.$$

Од првата редица можеме да избереме број на 4 начини, првиот, вториот, третиот или четвртиот. Ако сме избрале некој број, тогаш од втората редица веќе не можеме да го избереме истиот број, па ни остануваат три можности. За третата редица ќе имаме две можности за избор, а за четвртата останува една. Така имаме 4! можности, па ако работиме со наивниот пристап би требало да ја пресметаме сумата за секоја од овие можности и да ја најдеме најмалата. Со тоа би имале 4! проверки. Тоа значи дека во ситуација на матрица од ред $n \times n$ сложеноста на наивниот алгоритам би била $O(n!)$. Пристапот за тоа како се дизајнира вакво решение со алгоритам на груба сила може да се разгледа во [21].

Да забележиме дека секој избор може да се престави со една пермутација. На пример изборот $x_{12} + x_{21} + x_{34} + x_{43}$, можеме да го

претставиме со пермутацијата 2134, каде i-тиот број ја дава колоната од која се избира елементот од i-тата редица.

За да го изведеме решението со динамичко програмирање, да разгледаме како би ги бирале елементите и кои пермутации не мора да ги разгледуваме. Ако од првата и втората редица ги избереме првите два броја, тогаш повеќе се исплати од првата редица да се избере првиот, а од втората вториот број, затоа што така се добива збир $x_{12} + x_{21} = 7$, а ако избереме обратно добиваме збир $x_{21} + x_{12} = 10$. Значи оптималниот збир за подматрицата составена од првите две редици и првите две колони се добива за пермутацијата 12. Сега да претпоставиме дека за подматрицата составена од првите три редици и првите три колони се добива кога од третата редица се бира третиот број.



Слика 8. 1. Најмал збир за матрица 3×3 ако од третата редица се избере третиот број.

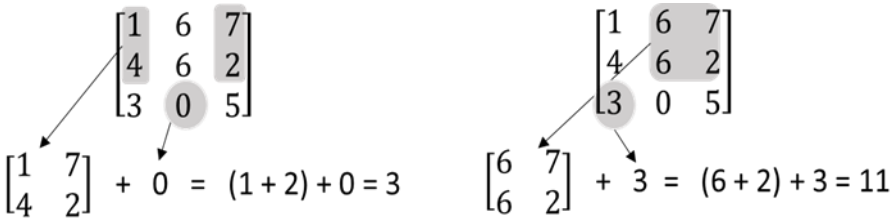
Може да забележиме дека не мора да ги разгледуваме и двете пермутации 123 и 213, затоа што и за едната и за другата пермутација треба да се изберат двата броја од подматрицата од првите две редици и колони и на тоа да се додаде третиот број од третата редица, 5. Оттука, јасно дека 123 е подобра опција, затоа што 12 беше подобра опција, Слика 8. 1.

Секако, ова не е оптималниот избор за подматрицата составена од првите три редици и првите три колони, затоа што како елементот од третата редица може да се избере и првиот и вториот:

- Кога од третата редица се бира првиот број, тогаш на првите две позиции треба да се изберат броеви од колоните 2 и 3, Слика 8.1.2 *а*). Подобар избор е пермутацијата 23, со збир 8. Со додавање на првиот елемент од третата редица вкупниот збир во оваа ситуација е 8+3=11.

- Кога од третата редица се бира вториот број, тогаш на првите две позиции треба да се изберат броеви од колоните 1 и 3, Слика 8.1.2 *b*). Во оваа ситуација подобар избор е пермутацијата 13, со збир 3. Со додавање на вториот елемент од третата редица вкупниот збир во оваа ситуација е 3+0=3.

Оттука, оптималниот избор за подматрицата составена од првите три редици и првите три колони е пермутацијата 132, со збир 3.



Слика 8. 2. a) Најмал збир за матрица 3×3 ако од третата редица се избере првиот број. b) Најмал збир за матрица 3×3 ако од третата редица се избере вториот број.

Веќе доаѓаме до идејата за оптималното својство кое го поседува проблемот:

> *Ако во оптималниот избор на точно еден број од секоја редица и колона од матрица Y од ред $i \times i$ од последната редица се избере j-тиот елемент, тогаш останатите броеви треба да се изберат на оптимален начин од матрицата од ред $(i-1) \times (i-1)$ која се добива со вадење на елементите од последната редица и j-тата колона.*

Оптималното својство можеме да го искажеме и погенерално:

> *Ако во оптималниот избор на точно еден број од секоја редица и колона од матрица Y od red $n \times n$ се избере елементот на позиција (i, j), тогаш останатите броеви треба да се изберат на оптимален начин од матрицата од ред $(i-1) \times (i-1)$ која се добива со вадење на елементите од i-тата редица и j-тата колона.*

Ако горното резонирање и оптималното својство ги искористиме во наредниот чекор имаме:

- Ако од последната редица го земеме последниот број, 4, од првите три редици ги бираме елементите од позициите 1, 2 и 3, за што веќе добивме оптимален збир 3. Вкупниот збир во оваа ситуација е 3 + 4 = 7.

- Ако од последната редица го земеме третиот број, 6, од првите три редици ги избираме елементите од позициите 1, 2 и 4. Најмал број што се добива вака е 6, кој се добива на два начини: 1 4 2 и 4 1 2. Вкупната сума во оваа ситуација е 12, па не е оптимална затоа што е поголема од 7.

- Ако пак од последната редица го земеме вториот елемент, 1, тогаш од првите три редици мора оптимално да се земат елементи од првата, третата и четвртата редица. Тоа оптимално се прави со изборот 4 3 1 2, кога се добива сумата 8 > 7.

- И последна опција е од последната редица да се избере првиот елемент, 2. Тогаш од првите три редици оптимално треба да се изберат елементи од втората, третата и четвртата колона. Најмалиот збир што при тоа се добива е за пермутацијата 4 3 2 1, за која се добива збир 6, што е всушност и оптималното решение.

Да размислиме како оваа врска може да се искористи за формално запишување на рекурентната релација. Изборот на оптималното решение за матрицата од ред 4×4 е најдоброто решение за избор на броеви од првите 4 редици, од колоните од множеството {1,2,3,4}. Во однос на избор на членот од четвртата редица имаме една од следниве можности:

- Да се избере членот од 4-тата колона, кога од првите три редици треба оптимално да се изберат членови од колоните {1,2,3},

- Да се избере членот од 3-тата колона, кога од првите 3 редици треба оптимално да се изберат членови од колоните {1,2,4}.

- Да се избере членот од 2-та колона, кога од првите три редици треба оптимално да се изберат членови од колоните {1,3,4} и

- Да се избере членот од 3-тата колона, кога од првите 3 редици треба оптимално да се изберат членови од колоните {2,3,4}.

Веќе се наѕира рекурзивната релација, која зависи од множествата на дадено множество. Прво, секое k -елементно подмножество од n-елементно множество ги означува елементите кои се избрани од првите k редици. За секој елемент во множеството ја пресметуваме оптималната вредност кога тој ќе се избере од последната, т.е. $|X|$ -тата позиција.

$$A[X] = \min_{i \in X} A[X - \{i\}] + x_{i,|X|}.$$

За да не работиме со стандардни множества, нив можеме да ги претставиме со бит низи, кои се познати како бит маски. Да објасниме што значи бит маска. Маска во битмаска значи да се сокрие нешто, а бит маска значи да се искористат бинарни броеви за репрезентација на одредени состојби. На пример, нека е дадено множество од 4 елементи во кое елементите се подредени, како множеството {1, 2, 3, 4}. Секое од подмножествата на ова множество може да се престави со бит низа со должина 4. Да речеме множеството {2, 4} може да се престави како 0101, а {1, 2, 4} со 1101. Празното множество се претставува со 0000, додека целото множество со 1111. Имено, за секое подмножество од дадено множество со n елементи, дефинираме n-димензионален бит таков што ако j-тиот елемент е во подмножеството, на j-та позиција се става битот 1, а ако не е 0.

Бит маските се погодни за извршување на основните операции со множества. На пример

- Ако имаме две бит низи y и z, одговараат на подмножествата Y и Z соодветно, тогаш нивниот пресек е битот $y\&z$, додека нивната унија е битот $y|z$, каде & е операцијата „и“, а | е операцијата „или“.

- Ако сакаме да додадеме елемент во некое множество, тогаш треба да го креираме битот соодветен за едно-елементното множество со тој елемент и да ја искористиме операцијата &. На пример, ако на множеството {2,4} сакаме да го додадеме

елементот 3, тој бит го креираме како 2^2, и му правиме операција | со бит-низата 1010. Се добива:

$$1010|0100 = 1110.$$

- Ако сакаме да извадиме елемент од некое множество, тогаш откога ќе ја креираме бит маската за тој елемент, ја земаме нејзината негација и ја правиме операција & со бит маската за соодветното множество. Пример, ако од множеството {2, 3, 4} сакаме да го извадиме четвртиот елемент, тоа можеме да го направиме на следниов начин:

$$1110 \,\&\, \overline{2^3} = 1110 \,\&\, \overline{1000} = 1110 \,\&\, 0111 = 0110.$$

- За да провериме дали некој елемент се наоѓа во множеството, треба да ја формираме бит маската соодветна за него и да направиме & операција со бит маската за множеството. Ако при тоа се добие 0, тогаш јасно, елементот не е во множеството, а ако се добие било кој број различен од 0, елементот е во множеството. На пример ако сакаме да провериме дали елементот 2 е во множеството {2,4}, на бит маската соодветна на ова множество 1010 и правиме операција & со 2^1:

$$1010\&0010 = 0010 > 0.$$

Бидејќи не се доби 0, елементот е во множеството. Елементот 1 не е во множеството бидејќи:

$$1010\&(20) = 1010\&0001 = 0000 = 0,$$

Алгоритамот со рекурзија и мемоизација на нашиот пример би ги пресметувал вредностите по следниов редослед:

- Прво рекурзијата се повикува за 1111:

$$A[1111] = \min \begin{cases} A[1110] + x_{41} \\ A[1101] + x_{42} \\ A[1011] + x_{43} \\ A[0111] + x_{44} \end{cases}.$$

- Сеуште не ни е позната вредноста на ниту еден елемент од кој се повикува рекурзивната релација, па прво рекурзивно ја пресметуваме вредноста на A за првата бит маска:

$$A[1110] = \min\begin{cases} A[1100] + x_{32} \\ A[1010] + x_{33}. \\ A[0110] + x_{34} \end{cases}$$

Веќе можеме да ги пресметаме вредностите од кои се повикува оваа функција, но и да ја пресметаме функцијата C која се користи за реконструкција:

$$\begin{aligned} A[1100] &= \min\{A[1000] + x_{23}, A[0100] + x_{24}\} \\ &= \min\{2 + 2{,}7 + 5\} = 4 \\ C[1100] &= C[1000]3 = 43 \\ A[1010] &= \min\{A[1000] + x_{22}, A[0010] + x_{24}\} \\ &= \min\{2 + 6{,}6 + 5\} = 8 \\ C[1010] &= C[1000]2 = 42 \\ A[0110] &= \min\{A[0100] + x_{22}, A[0010] + x_{23}\} \\ &= \min\{7 + 6{,}6 + 2\} = 8 \\ C[0110] &= C[0010]3 = 23 \end{aligned}$$

Веќе можеме да ја пресметаме $A[1110]$ и $C[1110]$:

$$\begin{aligned} A[1110] &= \min\{4 + 0{,}8 + 5{,}8 + 8\} = 4; \\ C[1110] &= C[1100]2 = 432. \end{aligned}$$

- Рекурзивно ја повикуваме функцијата за втората бит маска:

$$A[1101] = \min\begin{cases} A[1100] + x_{31} \\ A[1001] + x_{33} \\ A[0101] + x_{34} \end{cases} = \min\begin{cases} 4 + 3 \\ A[1001] + 5. \\ A[0101] + 8 \end{cases}$$

и ги пресметуваме непознатите вредностите на функцијата:

$$A[1001] = \min\{A[1000] + x_{21}, A[0001] + x_{24}\}$$
$$= \min\{2 + 4{,}1 + 5\} = 6$$
$$C[1001] = C[1000]1 = 41$$
$$A[0101] = \min\{A[0100] + x_{21}, A[0001] + x_{23}\}$$
$$= \min\{7 + 4{,}1 + 2\} = 3;$$
$$C[0101] = C[0001]3 = 13$$

Оттука,

$$A[1101] = \min\begin{cases} 4+3 \\ A[1001]+5 \\ A[0101]+8 \end{cases} = \min\begin{cases} 4+3 \\ 6+5=7. \\ 3+8 \end{cases}$$
$$C[1101] = C[1100]1 = 431.$$

- Рекурзивно ја повикуваме функцијата за третата бит маска:

$$A[1011] = \min\begin{cases} A[1010]+x_{31} \\ A[1001]+x_{32} \\ A[0011]+x_{34} \end{cases} = \min\begin{cases} 8+3 \\ 6+0 \\ A[0011]+x_{34} \end{cases}.$$

Па останува да се пресмета:

$$A[0011] = \min\{A[0010] + x_{21}, A[0001] + x_{22}\}$$
$$= \min\{6 + 4{,}1 + 6\} = 7$$
$$C[0011] = C[0001]2 = 12$$

Оттука,

$$A[1011] = \min\{8 + 3{,}6 + 0{,}7 + 8\} = 6;$$
$$C[1011] = C[1001]2 = 412.$$

- Рекурзивната функција за четвртата бит маска веднаш може да се пресмета:

$$A[0111] = \min\begin{cases} A[0110] + x_{31} \\ A[0101] + x_{32} \\ A[0011] + x_{33} \end{cases} = \min\begin{cases} 8+3 \\ 3+0=3. \\ 7+8 \end{cases}$$

$$C[0111] = C[0101]2 = 132.$$

На крај

$$A[1111] = \min\{4+2{,}7+1{,}6+6{,}3+4\} = 6;$$
$$C[1111] = C[1110]1 = 4321$$

Сега можеме да ја изведеме рекурзивната релација:

$$A[m] = \min_{m,m\&2^i>0} A[\, m\&\overline{2^i}] + x_{|m|,i},$$

каде $|m|$ е бројот на единици во маската m.

Псевдокодот со кој изборот на броеви од матрица се пресметува од горе надолу, и во кој во текот на постапката се пресметува и бројот на единици во маската е следниов:

П 8. 1. ИЗБОР НА БРОЕВИ ОД МАТРИЦА

```
1   внеси n и матрицата од цени X;
2   за mask од 1 до 2^n − 1 прави B[mask] = −1;
3   B[0] = 0;
4   broj_elem[2^n − 1] = n;
5   врати A[2^n − 1];
6   функција A[mask]
7       ако B[mask] ≠ −1 врати B[mask] инаку
8        {
9        B[mask] = ∞;
10       за j од 0 до n − 1 прави
```

```
11          ако mask & 2^j > 0 тогаш
12            {
13            N_elem[mask & ! (2^j)] = i;
14            broj _elem[mask & ! (2^j)] – 1 = i
15            B[mask] = min(B[mask], A[mask & ! (2^j)] + x_ij)
16            }
17        врати B[mask]
18        }
```

Повеќето програмски јазици, како C++ и Java содржат готови функции за некои операции со маски. На пример постои операција која брои колку единици има во една маска, што е всушност бројот на елементи во соодветното множество. Овде тоа ни е потребно за да определиме од која редица треба да го земеме соодветниот елемент. Функцијата за броење на единици ќе ја наречеме $broj\_elem$. Готови функции има и за додавање или вадење на елемент или за проверка дали даден елемент е во множеството, но во нашиот код тоа ќе го направиме со горните операции. Како и за другите типови на решение, ако решението го конструираме со рекурзија, мора да употребиме мемоизација. Најчесто полесно е да се користат итерации, но мора да се пази на тоа елементите од низата од кои зависи формулата секогаш да бидат пресметани. Во нашиот пример секогаш ни требаат маските со точно една единица помалку, но многу е потешко без рекурзија да се генерираат прво маските со помал, па потоа маските со поголем број на единици. Да забележиме дека не е битно тие да се генерираат по растечки редослед на бројот на единици и дека можеме да ги генерираме по растечки редослед на природниот број што го претставуваат. Имено, секогаш важи дека ако бинарната репрезентација на y се добива со замена на една единица од бинарната репрезентација на x со нула, тогаш $y < x$, па пресметките може да се прават по растечки редослед на бројот кој го претставуваат од 0 до $2^n - 1$, со следниов псевдокод:

П 8. 2. ИЗБОР НА БРОЕВИ ОД МАТРИЦА

```
1  Внеси n и матрицата од цени X;
2  За m = 0 до 2^n − 1 прави A[m] = ∞;
```

3 $A[0] = 0;$

4 за $mask$ од 0 до $2^n - 1$ прави

5 {

6 $i = broj\_elem[mask];$

7 за j од 0 до n прави

8 ако $mask \,\&\, !\,(2^{j-1}) = 0$ прави

9 $A[mask|2^{j-1}] = min\{A[mask|2^{j-1}], A[mask] + x_{ij}\};$

10 }

11 врати $A[2^{n-1}]$.

Во продолжение го даваме и кодот во Јава:

JAVA 8. 1 ИЗБОР НА БРОЕВИ ОД МАТРИЦА

```
1  import java.util.*;
2
3  public class Main {
4     public static int bitcount(int n) {
5        int count = 0;
6        while (n != 0) {
7           count += n & 1;
8           n >>= 1;
9        }
10       return count;
11    }
12
13    public static void main(String[] args) {
14       Scanner sc = new Scanner(System.in);
15       int n;
16       n = sc.nextInt();
17       int[][] X = new int[n][n];
18
19       for (int i = 0; i < n; i++)
20          for (int j = 0; j < n; j++)
21             X[i][j] = sc.nextInt();
22
23       int[] A = new int[1<<n];
24       Arrays.fill(A, Integer.MAX_VALUE);
25       A[0] = 0;
26
27       for (int mask = 0; mask < (1<<n) - 1; mask++) {
28          int i = bitcount(mask);
29          for (int j = 1; j <= n; j++)
30             A[mask|(1<<(j-1))] = Math.min(A[mask|(1<<(j-
```

```
1))], A[mask] + X[i][j-1]);
31        }
32
33        System.out.println(A[(1 << n) - 1]);
34     }
35 }
```

Истото решение во C++ изгледа вака:

C++ 8. 1 Избор на броеви од матрица

```
1  #include<iostream>
2  #include<cstring>
3  #include<bits/stdc++.h>
4  using namespace std;
5  int bitcount(unsigned int n)
6  {
7    unsigned int count = 0;
8    while(n)
9    {
10    count += n & 1;
11    n >>= 1;
12   }
13   return count;
14 }
15 int main()
16 {
17    int n;
18    cin >> n;
19    int X[n][n];
20    int i,j,k,mask;
21    for(i=0;i<n;i++)
22       for(j=0;j<n;j++)
23          cin>>X[i][j];
24
25    int A[1<<n];
26    for (i=0;i<1<<n;i++)
27          A[i] = INT_MAX;
28
29    A[0] = 0;
30
31    for (int mask = 0; mask < (1<<n) - 1; mask++) {
32       int i = bitcount(mask);
33       for (int j = 1; j <= n; j++)
34          A[mask|(1<<(j-1))] = min(A[mask|(1<<(j-1))],
A[mask] + X[i][j-1]);
35    }
```

```
36
37      cout << A[(1 << n) - 1];
38      return 0;
39  }
```

Мемориската комплексност на алгоритамот е $O(2^n)$, затоа што за секое множество, односно за секоја бит маска, се памети по една вредност. Циклусот од 4 до 10 се движи по сите n елементни множества, додека внатре има вгнезден циклус кој се движи побројот на елементи. Оттука, временската сложеност на алгоритамот е $O(2^n n)$.

Прашања и задачи

1. Дади псевдокод за алгоритам со груба сила кој го решава проблемот даден во ова поглавје.

2. Анализирај ја сложеноста на решението со груба сила на проблемот даден во ова поглавје.

3. Дадени се матрици X и Y позитивни реални броеви, со димензија $n \times n$. Треба да се избере точно по еден број од секоја редица и секоја колона од матрицата X, така да нивниот збир биде минимален, но збирот на броевите на истите позиции од матрицата Y да не надминува некоја вредност N.

   a. Да се даде рекурзивната релација над која може да се конструира решение со динамичко програмирање

   b. Да се даде псевдокод на решение кое работи во време $O(2^n nN)$.

   c. За колку различни вредности на суми од матрицата Y може да има вредности за функцијата која го пресметува оптималното решение за дадена маска m?

4. Даден е број n и m подмножества $S_i, i = \overline{1,m}$ од множеството $\{1, 2, \dots, n\}$.

   a. Да се даде рекурентна равенка која го пресметува најмалиот број на множества S_i такви што $\bigcup S_i = \{1, 2, \dots, n\}$.

   b. Која е сложеноста на предложениот алгоритам?

5. Даден е број n и m подмножества $S_i, i = \overline{1,m}$ од множеството $\{1, 2, \dots, n\}$.

   a. Да се даде рекурентна равенка која го пресметува најголемиот број на множества S_i такви што $S_i \bigcap S_j = \emptyset$, за $i \neq j$.

   b. Која е сложеноста на предложениот алгоритам?

6. Дадени се позициите на n војници и позициите на n бази, со нивните X и Y координати. Војниците треба да се распоредат на базите, така што вкупното растојание што треба да го изминат сите војници да биде најмало. Растојанието што го минува војникот кој е на позиција (x_1, y_1) до базата која е на позиција (x_2, y_2) е $|x_1 - x_2| + |y_1 - y_2|$. Опиши како ќе го решиш проблемот!

8.2. Различни капи

Следниов проблем е повторно проблем на преместувања, [22], но можеме да го гледаме како класичен комбинаторен проблем во кој треба да се избројат бројот на преместувања.

Дефинирање на проблемот

Нека има n различни видови на капи и m луѓе. Секој човек поседува дел од видовите на капи, и секој има барем еден тип на капи, но не мора да има од сите видови. Тие сакаат да отидат на забава, но за да забавата биде интересна решиле секој од нив да носи различна капа. На колку начини може да се направи тоа, ако за секој човек е дадена листата од капи кои тој ја има во својата колекција.

Јасно е во проблемот дека бројот на различни видови на капи мора да биде поголем од бројот на луѓе, и во задачата бројот на капи може да биде многу голем, но за да точно се избројат сите можности бројот на луѓе мора да е мал, најмногу до 30.

Анализа на проблемот

Иако на прв поглед претходниот и овој проблем изгледаат дека се сосема различни, затоа што претходниот проблем е оптимизационен, а овој комбинаторен, тие суштински се исти, и во двата проблеми треба да се разгледаат сите можни пермутации; во овој проблем да се избројат, а во претходниот да се најде најдобрата пермутација. За да дојдеме до решението, ќе ги разгледаме сличностите и различностите меѓу овие два проблеми.

Во претходниот пример бројот на редици и колони беше еднаков, додека во овој проблем бројот на капи и луѓе може да се разликува. Имено, во претходниот проблем беше исто дали маската ќе одговара на редиците или колоните од кои се избира бројот, затоа што изборот требаше да се направи така да и од секоја редица и од секоја колона ќе се избере по точно еден број. За разлика од тоа, во овој проблем имаме ситуација кога за секој човек треба да се одбере капа, но не секоја капа мора да има човек кој ќе ја носи, па битно е да се

размисли што ќе претставува маската. Од друга страна, во претходниот проблем самата маска не само што кажуваше од кои колони се избрани броевите, туку даваше и информација за тоа од колку редици е направен изборот, затоа што тој број беше еднаков на бројот на единици во мапата. Во оваа ситуација, дел од видовите капи може воопшто да не се изберат, па потребна ни е дополнителна информација за тоа од колку видови на капи сме го направиле изборот до некој даден момент.

Да се потсетиме дека претходниот проблем го решававме така што се движевме редица по редица избирајќи колона од која ќе го земеме наредниот број, а маската одговараше на колоните кои што се избрани (единиците во маската одговараа на колоните на избраниот број). Едно од главните прашања кое треба да си го поставиме е дали маската треба да одговара на капите или на луѓето. Бидејќи секој човек треба да се појави на забавата, а секој тип на капа не мора, тогаш маската треба да одговара на луѓето. Дополнително треба да се стави бројач кој ќе води сметка кои капи се распределени.

Да разгледаме пример во кој на забавата има тројца луѓе. Нека првиот ги поседува капите 1, 2 и 3, вториот капите 2, 4 и 5, а третиот капите 1 и 5. За секој тип на капа ќе направиме листа од луѓе кои го поседуваат тој тип на капа. Нека $Kapa[i]$ е листата на i-тата капа. Тогаш $Kapa[1] = \{1,3\}$, $Kapa[2] = \{1,2\}$, $Kapa[3] = \{1\}$, $Kapa[4] = \{2\}$ и $Kapa[5] = \{2,3\}$. Ова може да се претстави со табелата која е во коренот на кореновото дрво на Слика 8. 3.

Да забележиме дека петтата капа може или да биде или да не биде во распределбата.

- Ако не ја ставиме во распределбата, капите 1, 2, 3 и 4 треба да ги распределиме на сите 3-ца, што е илустрирано со првото дете на коренот на Слика 8. 3.
- Ако пак биде во распределбата, неа може да ја добие еден од луѓето во множеството $Kapa[5]$, а останатите 4 капи ќе бидат распределени на останатите двајца, што е илустрирано со второто и третото дете на коренот на Слика 8. 3.

Сега, ако со $A[1111,5]$ го обележиме бројот на начини да се распределат 5-те капи на сите 4-ца, тогаш

$$A[1111,5] = A[1111,4] + (A[1101,4] + A[1011,4] + A[0111,4]).$$

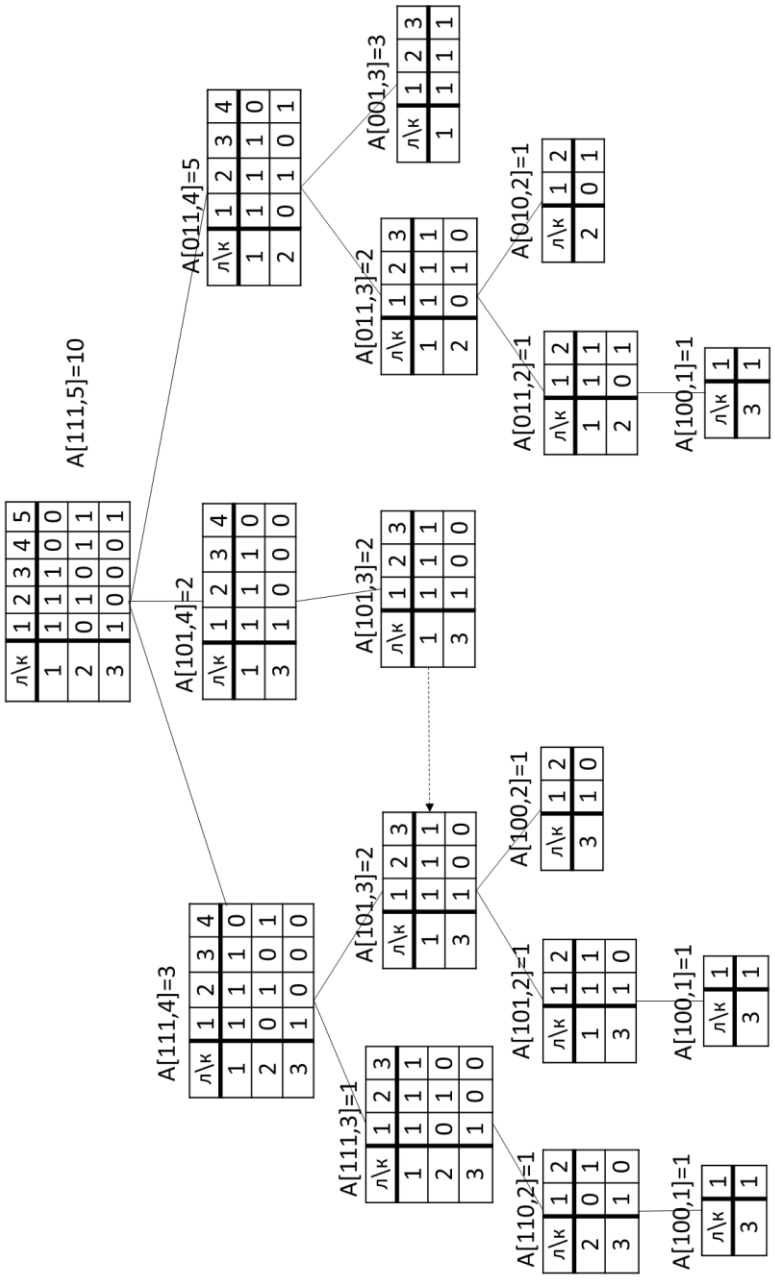
Првата компонента во функцијата *A* е маска која одговара на луѓето на кои им се распределени капи, а вториот број, 4, кажува дека се распределуваат првите 4 капи. Со ова проблемот се намалува од проблем за распределување на 5 капи, во проблеми за распределување на 4 капи. На сличен начин секој проблем за распределување на 4 капи може да се раздели на потпроблеми за распределување на 3 капи, како што е илустрирано втората генерација деца од дрвото прикажано на Слика 8. 3.

Врз основа на горната анализа, можеме да ја изведеме генералната рекурзивна врска:

$$A[mask, j] = A[mask, j-1] + \sum_{i \in Kapa[j]} A\left[mask - 2^i, j-1\right].$$

Останува да се определат почетните услови. Јасно е дека ако во маската има повеќе од j единици, тогаш не може да се направи распределување и во таков случај вредноста на функцијата е 0. Од друга страна, ако $mask = 0$, тогаш $A[0, j] = 1$, бидејќи бројот на начини на j капи се распределат на 0 луѓе е 1. Целата рекурзивна функција е следнава:

$$A[mask, j] = \begin{cases} 0, & broj\_elem[mask] > j \\ 1, & mask = 0 \\ A[mask, j-1] + \sum_{i \in Kapa[j]} A\left[mask - 2^i, j-1\right], & \text{инаку} \end{cases}.$$



Слика 8. 3. Илустрација за алгоритамот за распоредување различни капи на различни луѓе.

Во псевдокодот П 8. 3. паралелно се пресметува и бројот на елементи во множеството на кое одговара дадената маска.

П 8. 3. БРОЈ НА ПРЕМЕСТУВАЊА

```
1   внеси го бројот на видови капи n и бројот на луѓе m;
2   за j од 1 до n
3     {
4      внеси ја листата луѓе кои ја поседуваат j-тата капа, Kapi[j];
5      за  i од 1 до 2^m − 1 прави B[mask, i] = −1;
6      B[0, j] = 1;
7     }
8     broj_el[2^n − 1] = n;
9     врати A[2^n − 1, m];
10    функција A[mask, j]
11       ако B[mask, j] ≠ −1 врати B[mask, j] инаку
12           {
13             ако broj_el[mask] > n тогаш
14              B[mask, n] = 0 инаку
15                {
16                 B[mask, j] = A[mask, j − 1];
17                 за i во Kapa[j] прави
18                    ако mask & 2^i > 0 тогаш
19                        {
20                         mask1 = mask & ! (2^i);
21                         broj_el[mask1] = broj_el[mask] − 1;
22                         B[mask, j] = B[mask, j] + A[mask1, j − 1]
23                        }
24           }
25         врати B[mask, j];
26  }
```

Псевдокодот П 8. 3. работи со мемоизација, што значи дека рекурзивно ја повикуваме функцијата $A[mask, i]$. За да избегнеме повеќекратно пресметување на $A[mask, i]$ користиме помошна функција $B[mask, i]$ која ја прма вредноста на $A[mask, i]$ ако таа е

веќе пресметана, а ако не е тогаш рекурзивно ја пресметуваме нејзината вредност.

Следуваат кодовите во јава и C++, прво кодот во C++:

C++ 8. 2 БРОЈ НА ПРЕМЕСТУВАЊА

```
#include<bits/stdc++.h>
#include <math.h>
using namespace std;

int n;
int m;
vector<vector<int>> B;
vector<int> broj_el;
vector<vector<int>> kapa;


int A(int mask, int j)
{
   if (mask<0 || j<0)
      return 0;
   if(B[mask][j] != -1)
   {
      return B[mask][j];
   }
   if(broj_el[mask] > j + 1)
   {
      B[mask][j] = 0;
   }
   else{
      B[mask][j] = A(mask,j-1);
      for(int i:kapa[j])
      {
         if((mask&(1<<(i-1)))>0)
         {
            int mask1=mask & ~(1<<(i-1));
            broj_el[mask1] = broj_el[mask] - 1;
            B[mask][j] = B[mask][j]+A(mask1,j - 1);
         }
      }
   }
   return B[mask][j];
}


int main()
{
```

```
42    cin >> n;
43    cin >> m;
44    B = vector<vector<int>>(1<<m, vector<int>(n,-1));
45    broj_el = vector<int>(1<<m);
46    kapa = vector<vector<int>>(n);
47
48    for(int i=1; i<B.size(); i++)
49       broj_el[i] = log2(i)+1;
50
51    for(int j=0;j<n;j++)
52    {
53       vector<int> k;
54       int iii;
55       cin >> iii;
56       for(int ii = 0;ii<iii;ii++) {
57          int kk;
58          cin >> kk;
59          k.push_back(kk);
60          for(int jj=j;jj<n;jj++) {
61             if (B[1 << (kk - 1)][jj] == -1)
62                B[1 << (kk - 1)][jj] = 0;
63             B[1 << (kk - 1)][jj]++;
64          }
65       }
66       kapa[j] = k;
67    }
68    broj_el[(1<<m)-1] = m;
69    cout << A((1<<m)-1, n-1);
70 }
```

Истиот код во јазикот Јава изгеда вака:

ЈАVA 8. 2 БРОЈ НА ПРЕМЕСТУВАЊА

```
1  import java.util.*;
2
3  public class Main {
4     public static int n;
5     public static int m;
6     public static int [][] B;
7     public static int [] broj_el;
8     public static ArrayList<Integer> [] kapa;
9
10    public static int A(int mask, int j)
11    {
12       if (mask<0 || j<0)
13          return 0;
14       if(B[mask][j] != -1)
15       {
16          return B[mask][j];
17       }
18       if(broj_el[mask] > j + 1)
19       {
20          B[mask][j] = 0;
21       }
22       else{
23          B[mask][j] = A(mask,j-1);
24          for(int i:kapa[j])
25          {
26             if((mask&(1<<(i-1)))>0)
27             {
28                int mask1=mask & ~(1<<(i-1));
29                broj_el[mask1] = broj_el[mask] - 1;
30                B[mask][j] = B[mask][j]+A(mask1,j - 1);
31             }
32          }
33       }
34       return B[mask][j];
35    }
36
37    public static void main(String[] args) {
38       Scanner sc = new Scanner(System.in);
39       n = sc.nextInt();
40       m = sc.nextInt();
41       B = new int[1<<m][n];
42       broj_el = new int[(1<<m)];
43       kapa = new ArrayList [n];
44
```

```
45          for (int[] row: B)
46             Arrays.fill(row, -1);
47
48          for(int i=1; i<B.length; i++)
49             broj_el[i] = Integer.bitCount(i);
50
51          for(int j=0;j<n;j++)
52          {
53             ArrayList k = new ArrayList();
54             int iii = sc.nextInt();
55             for(int ii = 0;ii<iii;ii++) {
56                int kk = sc.nextInt();
57                k.add(kk);
58                for(int jj=j;jj<n;jj++) {
59                   if (B[1 << (kk - 1)][jj] == -1)
60                      B[1 << (kk - 1)][jj] = 0;
61                   B[1 << (kk - 1)][jj]++;
62                }
63             }
64             kapa[j] = k;
65          }
66          broj_el[(1<<m)-1] = m;
67          System.out.println(A((1<<m)-1, n-1));
68       }
69    }
```

Мемориската комплексност на алгоритамот е $O(2^n n)$, затоа што за секој пар множество-елемент од множеството, се запаметува по една вредност и за функцијата A и за функцијата B.

За да ја пресметаме временската комплексност, повторно ќе користиме амортизациона анализа за да определиме колку пати се пресметува секоја вредност на функцијата $B[mask, i]$ и колку операции користиме при тоа. Имено, бидејќи работиме со мемоизација $B[mask, i]$ се пресметува најмногу еднаш, и во секоја пресметка можеме да сметаме дека вредноста за $A[mask, i]$ е веќе пресметана. Бидејќи во циклусот од линија 17 до линија 23 имаме n операции следува дека временската сложеност на алгоритамот е $O(2^n n)$.

Прашања и задачи

1. Од примерот на Слика 8. 3 даден за илустрација на алгоритамот, може да се забележи дека рекурзивната релација во секој чекор намалува по една колона од почетната матрица, при што проблемот го намалува на проблем во кој се отфрла последната колона на матрицата и матрици во кои се отфрла последната колона и редиците на кои вредноста во последната колона им е различна од 0.

   a. Илустрирај на истиот пример дека наместо за колоните од матрицата слична постапка може да се примени и за редиците на матрицата и решението ќе биде исто. Имено покажи дека рекурзивната равенка во секој чекор може да ја брише последната редица од матрицата и колоните во кои вредноста на елементот во последната редица е различна од 0.

   b. Дади рекурзивна равенка на која ќе се темели решението чија идеја е дадена под а.

   c. Размисли зошто динамичкото програмирање е подобро да се темели на множеството луѓе кои присуствуваат на забавата, наместо на множеството на видови капи кои некој ќе ги носи на забавата!

2. Разгледај ја следнава модификација на проблемот на капи: наместо по најмногу една капа од секој вид, луѓето можат да поседуваат и по повеќе капи од тој вид, (на пример во различна боја). На забавата секој треба да дојде со капа од различен вид, и не смее никои двајца да носат капи од ист вид, дури и тие да се со различна боја. Повторно треба да се најде бројот на начини на кои луѓето можат да се појават на забавата.

   a. Како може да се модифицира решението на проблемот со број на различни капи за да се реши овој модифициран проблем.

   b. Дади го псевдокодот на решението.

   c. Дали ќе има промена во временската сложеност?

d. Дали очекуваш разлика во мемориската сложеност?

3. Дадени се n коцки со различни големини на кои секоја од шесте страни е во различна боја. На секоја страна е запишана по една буква. Даден е и стринг со $m \leq n$. Треба да се најде бројот на начини на кои може да се испише дадениот стринг со помош на дадените коцки. Да опише алгоритам кој го пресметува бројот на начини на кои може да се испише дадениот стринг, ако сите карактери се различни. Дали ќе има разлика во алгоритамот кој го решава проблемот, ако во стрингот има исти карактери?

8.3. Проблем на трговски патник

Проблемот на трговски патник е можеби најпознат проблем кој се решава со бит маски и еден од најпознатите проблеми кои спаѓаат во класата на НП комплетни проблеми.

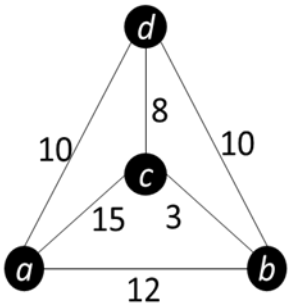
Дефинирање на проблемот

Дадено е множество градови и најкраткото растојание помеѓу било кои два града. Трговецот трга од еден од градовите и треба да ги посети сите градови и пак да се врати на местото од каде тргнал, а при тоа да помине најмал можен пат.

Анализа на проблемот

Проблемот е всушност во даден комплетен тежински граф, $G(V, E, l)$, каде $l: E \rightarrow \boldsymbol{R}^+$ е функција која ги претставува најкратките патишта, да се најде најкраткиот Хамилтонов циклус. Хамилтонов циклус сигурно постои, затоа што графот е комплетен. На пример да го разгледаме графот на Слика 8. 4., најкраткиот хамилтонов циклус има должина 33 и е $< a, b, c, d, a >$. Да напоменеме дека ако графот е граф од најкратки патишта, тогаш не може да се случи во пат од теме x до теме y да биде помал од збирот на патиштата од x до z и од z до y. Ова важи заради својството на триаголник за даден граф. Имено, не значи дека во реалноста патникот нема да помине два пати во ист град, но во графот тој нема да помине два пати во исто теме.

Прво да увидиме дека воопшто не е важно кое теме ќе го земеме за почетно во циклусот. На пример циклусот $< a, b, c, d, a >$ е ист со циклусот $< b, c, d, a, b >$.



Слика 8. 4. Граф од најкратки патишта од секое до секое теме.

Да претпоставиме дека ги знаеме должините на сите патишта кои почнуваат од темето a и минуваат низ сите темиња. Секој од тие патишта со додавање на ребро од последното теме до темето a ќе направи хамилтонов циклус. Да се сконцентрираме на ова последното ребро за најкраткиот циклус. Тоа ребро е од некое од темињата b, c или d до темето a. Според тоа, најкраткиот циклус треба да се бара во еден од следниве циклуси:

- најкраткиот пат на кој последно теме му е темето b плус тежината на реброто (b, a), $l(b, a)$.
- најкраткиот пат на кој последно теме му е темето c плус тежината на реброто (c, a), $l(c, a)$.
- најкраткиот пат на кој последно теме му е темето d плус тежината на реброто (d, a), $l(d, a)$.

Поточно, нашето решение е најдобрата од овие можности. Ако множеството од сите патишта со почеток во a, кои поминуваат низ сите темиња, а завршуваат со темето $x \in \{b, c, d\}$ ги обележиме со $P(\{b, c, d\}, x)$. Тогаш должината на најкраткиот циклус е:

$$\begin{aligned} Najkratok = \min\{&\min\{l(p) | p \in P(\{b, c, d\}, b)\} + l(b, a), \\ &\min\{l(p) | p \in P(\{b, c, d\}, c)\} + l(c, a), \\ &\min\{l(p) | p \in P(\{b, c, d\}, d)\} + l(d, a)\}. \end{aligned}$$

Да забележиме дека не мора претходно да ги имаме пресметано должините на сите патиштата кои завршуваат на b, затоа што нам ни е потребен само $\min\{w(p) | p \in P(\{b, c, d\}, b)\}$, односно

најкраткиот од нив. Слично е и за патиштата со крајно теме c или d. Сега можеме слично да размислуваме и за постапката за пресметување на $\min\{l(p)|p \in P(\{b,c,d\},b)\}$. Имено, претпоследно теме на патот кој што го дава овој минимум е или темето c или темето d, па

$$\begin{aligned}\min\{l(p)|p \in P(\{b,c,d\},b)\} \\ &= \min\{\min\{l(p)|p \in P(\{c,d\},c)\} + l(c,b), \\ &= \min\{l(p)|p \in P(\{c,d\},d)\} + l(d,b)\},\end{aligned}$$

каде $P(\{c,d\},c)$ е множеството патишта со почеток во a, кои поминуваат низ c и d и завршуваат со c, додека $P(\{c,d\},d)$ е множеството патишта со почеток во a, кои поминуваат низ c и d и завршуваат со d.

Веќе се наѕира следново оптимално својство:

Ако на оптималниот пат од темето и до темето v на кој помеѓу темињата u и v стојат темиња од множеството U претпоследно е темето $w \in U$, тогаш тој пат го содржи најкраткиот пат од темето и до w на кој помеѓу овие две темиња стојат темињата од $U/\{w\}$ и како последно ребро реброто (w,v).

Оттука, рекурзивната релација би била:

$$A[U,i] = \min_{j \in U, j \neq i}\{A[U-\{i\},j] + l[j,i]\}.$$

Истата релација претставена со битмаски е:

$$A[mask,i] = \min\{A[mask - 2^i, j] + l[j,i]|2^j \& (mask - 2^i) > 0\},$$

каде i и j се редните броеви на темињата во маската.

Останува уште да се определат почетните услови. Јасно е дека множеството од темиња помеѓу почетното и крајното теме може да биде најмалку празно множество, па во таква ситуација

$$A[2^i, i] = l[pocetno, i].$$

Во нашиот алгоритам ова почетно теме ќе го земеме како последно теме во листата, односно темето обележано со $|V| - 1$.

Да го дадеме псевдокодот што го решава проблемот:

П 8. 4. Трговски патник

```
1  Внеси го G(V, E, l) и за почетното теме одбери го |V| – 1;
2  за mask = 0 до 2^(|V|−1) − 1
3  за j од 0 до |V| прави A[mask, j] = ∞;
4     за mask = 1 до 2^(|V|−1) − 1
5  за i од 0 до |V| – 2 прави
6     ако mask & !(2^i) = 0 прави A[mask, i] = l[|V| − 1, i]
7        инаку ако mask & 2^i > 0 прави
8              за j од 0 до |V| – 2 прави
9                 ако (mask − 2^i) & 2^j > 0 прави
10                   A[mask, i] = min{A[mask, i], A[mask − 2^i, j] +
   l[j, i]};
11 B = ∞;
12 за j од 0 до |V| – 2 прави
13       B = min{B, A[2^(|V|−1) − 1, j] + l[j, |V| − 1]};
14 врати B.
```

Мемориската сложеност на ова решение е $O(2^{|V|-1}|V|)$, затоа што тоа е бројот на елементи за кои се иницира вредност уште во првите два циклуси. Временската сложеност е поголема, $O(2^{|V|-1}|V|)$, затоа што за секое поле кое се пополнува е потребно да се испроцесираат сите помали маски.

За рекунструкција на едно оптимално решение потребно е да воведеме нова функција која ќе го зачувува претходното теме на секое теме во циклусот, односно ќе го чува графот на претходници, исто како во алгоритмите за најкратки патишта. Тоа може да се направи со функција $\pi[mask, i]$ која ќе се ажурира секогаш кога $A[mask, i\ ]$ = $A[mask - 2^i, j] + l[j, i]$ и ќе ја прими вредноста $\pi[mask, i] = j$. Потоа од назад на напред ќе се печатат вредностите

на функцијата π, слично како во алгоритамот за печатење на пат од делот за најкратки патишта. Истото тоа може да го направиме и дополнително со следниов код:

П 8. 5. РЕКОНСТРУКЦИЈА НА ТРГОВСКИ ПАТНИК

```
1  печати |V| – 1
2  mask = 2^(|V|-1);
3  j = 0;
4  додека B < A[mask, j] + l[j, |V| − 1 ] прави j++;
5      печати j;
6      mask = mask – 2^j;
7      додека mask > 0 прави
8          за i = 0 до |V| – 1 прави
9            {
10           ако mask & 2^i > 0 прави
11           ако A[mask + 2^j, j] = A[mask, i] + l[j, i] прави
12         {
13         печати i;
14           mask = mask − 2^i;
15           j = i;
16       }
17   }
18 печати |V| – 1.
```

За нашиот пример алгоритамот би ги пресметувал вредностите по следниов редослед:

$$A[1,0] = 12;$$

$$A[10,1] = 15;$$

$$A[11,0] = 18;$$

$$A[11,1] = 15;$$

$$A[100,2] = 10;$$

$$A[101,0] = 20;$$
$$A[101,2] = 22;$$
$$A[110,1] = 18;$$
$$A[110,2] = 23;$$
$$A[111,0] = min\{A[110,1] + l[0,1], A[110,2] + l[0,2]\}$$
$$= min\{18 + 3, 23 + 10\} = 21;$$
$$A[111,1] = min\{A[101,0] + l[0,1], A[101,2] + l[1,2]\}$$
$$= min\{20 + 3, 22 + 8\} = 23;$$
$$A[111,2] = min\{A[011,0] + l[0,2], A[011,1] + l[1,2]\}$$
$$= min\{18 + 10, 15 + 8\} = 23;$$

Крајната вредност е:

$$B = min\{A[111,0] + l[0,3], A[111,1] + l[1,3], A[111,2] + l[2,3]\}$$
$$= min\{21 + 12, 23 + 5, 23 + 10\} = 33.$$

Кодовите во Јава и C++ се следниве:

JAVA 8. 3 ТРГОВСКИ ПАТНИК

```
1  import java.util.Arrays;
2  import java.util.Scanner;
3
4  public class Main {
5     public static int[][] dist;
6     public static int[][] dp;
7     public static int N;
8     public static final int INF = 1000000000;
9
10    public static int getShortestPath(int mask, int visited)
{
11
12       if (visited == (1 << N) - 1)
13          return dist[mask][1];
14
15       if (dp[mask][visited] >= 0)
16          return dp[mask][visited];
17
18       int ret = INF;
19
20       for (int i = 1; i <= N; i++) {
21
22          if ((visited & (1 << (i - 1))) != 0)
23             continue;
24
25          if (dist[mask][i] == 0)
26             continue;
27
28          int temp = dist[mask][i] + getShortestPath(i,
visited + (1 << (i - 1)));
29          ret = Math.min(ret, temp);
30       }
31
32       return dp[mask][visited] = ret;
33    }
34
35    public static void main(String[] args) {
36       Scanner sc = new Scanner(System.in);
37       N = sc.nextInt();
38       dist = new int[N + 1][N + 1];
39       dp = new int[N + 1][1 << N];
40       for (int i = 1; i <= N; i++) {
41          for (int j = 1; j <= N; j++) {
42             dist[i][j] = sc.nextInt();
```

```
43            }
44        }
45
46        for (int i = 1; i <= N; i++) {
47            Arrays.fill(dp[i], -1);
48        }
49
50        System.out.println(getShortestPath(1, 1));
51    }
52
53 }
```

Истото решение препишано во C++:

C++ 8. 3 Трговски патник

```
#include<iostream>
#include<vector>
using namespace std;

#define INF 999999

int N;
vector<vector<int>> dp;
vector<vector<int>> dist;


int  getShortestPath(int mask,int visited){

   if (visited == (1 << N) - 1)
         return dist[mask][1];

   if (dp[mask][visited] >= 0)
      return dp[mask][visited];

   int ret = INF;

   for (int i = 1; i <= N; i++) {

      if ((visited & (1 << (i - 1))) != 0)
         continue;

      if (dist[mask][i] == 0)
         continue;

      int temp = dist[mask][i] + getShortestPath(i, visited
+ (1 << (i - 1)));
      ret = min(ret, temp);
   }

   dp[mask][visited] = ret;
   return dp[mask][visited];
}

int main(){
   cin >> N;
   dist = vector<vector<int>>(N+1, vector<int>(N+1, 0));
   dp = vector<vector<int>>(N+1, vector<int>(1 << N, -1));

   for(int i=1;i<=N;i++)
```

```
44          for(int j=1;j<=N;j++)
45              cin >> dist[i][j];
46
47      cout<<getShortestPath(1,1);
48
49      return 0;
50  }
```

Прашања и задачи

1. Даден е неориентиран тежински граф G(*V*, *E*), каде *V* го претставува множеството градови, а *E* директните патишта меѓу градовите меѓу кои има директни патишта. Функцијата на тежина ги претставува должините на тие патишта.

   a. Како ќе се модифицира проблемот за трговски патник во ваков случај?

   b. Каков треба да биде графот за да постои решение?

   c. Дали оптималното решение се добива за некој Хамилтонов циклус во графот?

   d. Дали графот мора да содржи Хамилтонов циклус за да може да се најде решение за овој проблем?

2. Даден е неориентиран комплетен тежински граф, со ненегативни тежини на ребрата.

   a. Како ќе го пресметаш најлесниот Хамилтонов пат во графот?

   b. Дали алгоритамот за најлесен Хамилтонов пат е побрз од алгоритамот за најлесен хамилтонов циклус?

3. Дали алгоритамот за трговски патник би работел за најлесен Хамилтонов циклус во било каков граф. Што ќе се отпечати ако нема Хамилтонов циклус?

4. Како ќе се модифицира решението на задачата ако сакаме само да провериме дали има Хамилтонов циклус во графот? Дали ќе има забрзување на решението?

5. Модифицирај го алгоритамот за Хамилтонов пат на следниов проблем: Дадена е низа од природни броеви. За дадено подредување на низата $a_1, a_2, \dots, a_n$ се дефинира тежина на следниов начин

$$S(a_1, a_2, \dots, a_n) = \sum_{i=1}^{n-1} a_i \oplus a_{i+1},$$

каде $\oplus$ е операцијата XOR, а сумата е обична сума од броеви. За дадена низа на броеви да се најде подредувањето кое има минимална тежина!

Упатства и решенија

Прва глава

1.1

1. Одговор: 2000.

2. Одговор: 25. Од дома може да тргне или со автобус или со воз, на 3 + 2 = 5 различни начини и да стигне во градот *C*. Од *A* до *C*, може повторно да избере или автобус или воз на 2 + 3 = 5 начини. Треба да се поминат и двете дистанци, а тоа е можно на $5 \cdot 5 = 25$ начини.

3. Одговор: 40. Со 2 се деливи 60, а со 3 се деливи 40 од броевите. Со 2 и 3 се деливи 120:6=20 броеви, затоа што тие треба да се деливи со 6. Од принцип на вклучување и исклучување, 60 + 40 – 20 = 80 броеви се деливи со некој од броевите 2 и 3, па броеви кои не се деливи ни о 2 ни со 3 се 120 – 80 = 40.

4. Одговор: 36. Користиме обопштен принцип на вклучување и исклучување со тоа што пресметуваме колку од броевите се деливи со 2, 3 и 5 поодделно. Потоа од овој број ги одземаме бројот на броеви кои се деливи со 2 и 3, т.е. со 6, 2 и 5 т.е. со 10 и 3 и 5, т.е. со 15, затоа што секој од нив го броевме по двапати. Но гледаме дека сите броеви кои се деливи и со 2 и со 3 и со 5 ги броевме по еднаш за секој од овие броеви, но потоа ги извадивме по еднаш за секој пар од овие броеви, и сега не се пребројани ниту еднаш. Затоа на крај го додаваме и бројот на броеви деливи со 30. Оттука, бројот на броеви деливи со некој од 2, 3 или 5 е

   $$60 + 40 + 24 – 20 – 12 – 8 + 4 = 84,$$

   па бројот на оние кои не се деливи со ниту еден од нив е 36.

5. Одговор: 6!. Стрингот ABC мора да се јавува како блок, па бараниот број на пермутации ќе го најдеме со пресметување на бројот на пермутации со 6 елементи: блокот ABC како еден елемент и буквите D, E, F, G и H.

6. Одговор: C(5, 3).

7. Одговор: C(10,4)=$\frac{10!}{4!6!}$ и 4!.

8. Одговор: $\binom{n}{2}m+\binom{m}{2}n$. Секој триаголник има едно теме од едната и две темиња од другата права. Има два пристапи, едното теме да се одбере од првата или втората права. Ако се одбере од првата, тоа ќе се направи на m начини, а двете темиња од другата права ќе се одберат на $\binom{n}{2}$ начини, па со овој пристап имаме $\binom{n}{2}m$начини. Слично со другиот пристап че имаме $\binom{m}{2}n$ начини.

9. Одговор: $\frac{10!}{4!3!3!}$. Децата во црвени маички можеме да ги одбереме на $\binom{10}{4}$, потоа тие во сини на $\binom{6}{3}$, затоа што ќе останат уште 10 – 4 = 6 деца и на крај останатите 3 ќе се облечат во зелени маички. Вкупниот број е $\frac{10!}{4!6!}\frac{6!}{3!3!}=\frac{10!}{4!3!3!}$.

10. Одговор: $\frac{10!}{2!3!5!}$.

11. Одговор: $\frac{6!}{2!2!2!}$. Имаме 6 непарни и 3 парни броеви, па за да го направиме овој распоред треба во секоја група да има по еден парен и два непарни броеви. Во групата на двојката можеме да избереме 2 непарни броја на $\binom{6}{2}$ начини. Отога ќе ги избереме, во групата на 4-ката можеме да избереме други два непарни броеви на $\binom{4}{2}$ начини и останатите елементи ќе бидат во групата на 6-ката.

12. Одговор: $6\binom{5}{3}+\binom{6}{3}\frac{1}{2!}$. Овој распоред може да го направиме со два пристапи: првиот, ако во едната група ставиме два парни и еден непарен број, во втората три непарни броеви, а во третата еден парен и два непарни броеви, а вториот пристап ако во

едната група ги ставиме сите парни броеви, а во другите две ставиме по 3 непарни броја. Со првиот пристап двата парни броеви за првата група можеме да ги избереме на $\binom{3}{2} = 3$ начини и непарниот број кој ќе го ставиме во таа група на 6 начини, па таа група можеме да ја формираме на 6 начини. Од останатите 5 непарни броја, групата со 3 непарни броеви можеме да ја избереме на $\binom{5}{3}$ начини и останатите броеви ќе ги ставиме во третата група. Според ова со овој пристап имаме $6\binom{5}{3}$ начини на избор. Со вториот пристап групата со 3 парни броја е само една, првата група со непарни броеви можеме да ја избереме на $\binom{6}{3}$ начини, а останатите ќе одат во третата група. Но бидејќи двете групи со непарни броеви не се разликуваат во суштина, овој број треба да го поделиме со 2!, што е број на начини да ги разместиме овие две групи.

13. Одговор: $\binom{17}{2}$.

14. Одговор: $\binom{14}{2}$. Прво ги бираме трите сиви плочки и останува да се изберат уште 12 плочки во некоја од трите бои.

15. Одговор: $\binom{30}{2}$.

16. Одговор: $\binom{12}{2}^3$. Прво ќе се распределат 10-те црвени ранци, на $\binom{12}{2}$ начини, потоа на исто толку начини ќе се распределат сините и на крај на исто толку начини ќе се распределат зелените ранци.

17. Одговор: $\binom{2n}{n}$. Половина од чекорите треба да бидат во лево, а другата половина во десно. Во низа од $2n$ чекори треба да ги одбереме позициите на чекорите во лево.

18. Одговор:

a. $\binom{m+n}{m}$. Треба да се направат вкупно $m+n$ чекори, од кои m се во лево, па од $m+n$- те чекори треба да ги одбереме позициите на оние кои се во лево.

b. $\binom{r+s}{r}\binom{m+n-(r+s)}{m-r}$. Прво треба да направат вкупно $r+s$ чекори, од кои r се во лево, па потоа $m+n-(r+s)$ чекори од кои $m-r$ се во лево.

19. Одговор: 1.

1.2

1. Одговор: $A[n]=2^n$. Карактеристичната равенка е $x=2$, од каде решението е од облик $C\cdot 2^n$. Константата C се наоѓа од почетниот услов $C\cdot 2^0=1$.

2. Одговор: $A[n]=2^{n+1}-1$. Ја решаваме равенката

$$x^2-3x+2=0,$$

на која решенија се 1 и 2. Сега решение на целата рекурзија е:

$$A[n]=a\cdot 1^n+b\cdot 2^n.$$

Уште треба да се најдат константите од почетните услови, т.е. да се реши системот

$$\begin{cases}1=a+b\\3=a+2b\end{cases},$$

од каде $b=2$ и $a=-1$.

3. Одговор: $A[n]=2\cdot 2^n-(-2)^n$: Ја решаваме равенката

$$x^2-4=0,$$

на која решение се 2 и -2. Сега решение на целата рекурзија е

$$A[n]=a\cdot 2^n+b(-2)^n.$$

Уште треба да се најдат константите од почетните услови, т.е. да се реши системот

$$\begin{cases} 1 = a + b \\ 2 = 2a + 2b \end{cases},$$

од каде $b = -1$ и $a = 2$.

4. Одговор: $A[n] = 2^n(n+1)$: Ја решаваме равенката

$$x^2 - 4x + 4 = 0,$$

на која дупло решение е 2. Сега решение на целата рекурзија е

$$A[n] = 2^n(bn + a).$$

Уште треба да се најдат константите од почетните услови, т.е. да се реши системот

$$\begin{cases} 1 = a \\ 4 = 2a + 2b \end{cases},$$

од каде $b = 1$ и $a = 1$.

5. Одговор: $A[n] = 2(-1)^n - (-2)^n - 1$.

6. Одговор: $A[n] = (n+2) - (n+2)(-2)^n$.

7. Одговор: $A[n] = 3n + 1$. Карактеристичната равенка е $x = 1$, од каде решението на хомогената равенка е од облик

$$a_n^h = C \cdot 1^n = C.$$

Партикуларно решение бараме од облик cn, од каде $c = 3$. Константата C се наоѓа од почетниот услов $C + 3 \cdot 0 = 1$.

8. Одговор: $A[n] = 2^{n-1}(n^2 + n + 2)$. Карактеристичната равенка е $x = 2$, од каде решението на хомогената равенка е од облик:

$$a_n^h = C \cdot 2^n.$$

Партикуларно решение бараме од облик

$$n(cn + d)2^n,$$

од каде $c = d = \frac{1}{2}$ и решението е од облик

$$A[n] = 2^n C + 2^{n-1} n(n+1).$$

Константата C се наоѓа од почетниот услов и $C = 1$.

9. Одговор: $A[n] = 3 + 2^{n+1}(n-1)$. Карактеристичната равенка е $x = 1$, од каде решението на хомогената равенка е од облик

$$a_n^h = C.$$

Партикуларно решение бараме од облик

$$(cn + d)2^n,$$

од каде $c = 2 = -d$ и решението е од облик

$$A[n] = C + 2^n(2n - 2).$$

Константата C се наоѓа од почетниот услов и $C = 3$.

1.3

1. Одговор: a. точно, b. неточно, c. точно, d. точно, e. неточно, f. точно, g. точно, h. неточно, i. неточно, j. неточно, k. неточно, l. точно, m. неточно, n. точно, o. точно.
2. Одговор: a. неточно, b. неточно, c. неточно, d. точно, e. точно, f. неточно, g. точно, h. точно.
3. Одговор: a. неточно, b. неточно, c. точно, d. точно.
4. Одговор: a. неточно, b. точно, c. неточно, d. неточно.
5. Одговор: a. точно, b. неточно, c. неточно, d. точно.
6. Одговор: a. точно, b. точно, c. точно, d. точно.
7. Одговор: a. точно, b. точно, c. точно, d. неточно.
8. Одговор: a. неточно, b. Точно.
9. Одговор: $A[n] = C2^n$.
   a. $\theta(2^n)$, бидејќи $A[n] = 2^n$.
   b. $\theta(0)$, бидејќи C од каде ќе следува и дека $A[n] = 0$.

c. $O(2^n)$, бидејќи тоа е најголема функција која може да се добие за било кои константи.

10. Одговор: $\theta(2^n)$, бидејќи $A[n] = 2^{n+1} - 1$.

11. Одговор: $\theta(2^n n)$, бидејќи $A[n] = 2^n(n+1)$.

12. Одговор: $O(2^n)$, бидејќи $A[n] = a2^n + b(-2)^n$. Не можеме да ставиме $\theta(2^n)$ затоа што за некои почетни услови може да се добие дека функцијата е константна, т.е. 0.

13. Одговор: $\theta(2^n)$, бидејќи $A[n] = 2(-1)^n - (-2)^n - 1$, каде најголем раст по апсолутна вредност има членот $(-2)^n$, а $|(-2)^n| = 2^n$.

14. Одговор: $\theta(n)$, бидејќи општиот облик на решението е

$$A[n] = C + 3n.$$

15. Одговор: Од $A[n] = (an + b) + (cn + d)(-2)^n$.

    a. $O(2^n n)$.

    b. Ако почетните услови се такви да $c = 0$, растот ќе биде $O(2^n)$, ако $c = d = 0$, растот ќе биде $O(n)$, ако $a = c = d = 0$, растот ќе биде $O(1)$

16. Одговор: Бидејќи $A[n] = 2^n C + 2^{n-1} n(n+1)$.

    a. $\theta(2^n n^2)$

    b. Не зависи од почетните услови, бидејќи решението на нехомогената равенка ја има оваа сложеност.

17. Одговор: Бидејќи $A[n] = C + 2^n(2n - 2)$.

    a. $\theta(2^n n)$

    b. Не зависи од почетните услови, бидејќи решението на нехомогената равенка ја има оваа сложеност.

18. Одговор: Бидејќи $A[n] = C2^n - 1$.

    a. $O(2^n)$

b. Зависи од почетните услови, бидејќи решението на нехомогената равенка има поголема сложеност од растот на хомогената равенка. Ако почетните услови се такви да константата $C = 0$, растот на функцијата ќе биде $O(1)$.

1.4

1. Одговор:
   a. $T[n] = \Theta\left(n\sqrt{n}\right)$
   b. $T[n] = \Theta\left(n^{\log_3 2}\right)$
   c. $T[n] = \Theta(n^2)$
2. Одговор:
   a. $T[n] = \Theta(n \log n)$
   b. $T[n] = \Theta\left(n^{\log_4 3}\right)$
3. Одговор:
   a. $T[n] = 1 + T\left[\frac{n}{2}\right]$, од каде следува дека $T[n] = \Theta(\log n)$
   b. $T[n] = 1 + T\left[\frac{n}{3}\right]$, од каде следува дека $T[n] = \Theta(\log n)$
4. Одговор: Иако при вториот алгоритам имаме значително помалку чекори, асимптотскиот раст на двете функции е ист, па не е исплатливо да се дели на три дела, затоа што таа постапка е секогаш покомплицирана.

1.5

1. Одговор: Секое ребро се брои како степен на двете темиња кои ги поврзува. Оттука, вкупниот степен е еднаков на двапати по бројот на ребра.

2. Одговор: $\frac{n(n-1)}{2}$. Секое од n-те темиња може да се поврзе со сите останати темиња, односно неговиот степен е најмногу $n-1$. Оттука, вкупниот степен на сите темиња е најмногу $n(n-1)$. Тврдењето следи веднаш од теоремата за ракување.

3. Одговор: Нека V_1 се темињата со парен степен, а V_2 се темињата со непарен степен. Тогаш

$$2|E| = \sum_{v \in V} \deg(v) = \sum_{v \in V_1} \deg(v) + \sum_{v \in V_2} \deg(v).$$

Јасно е дека сумата по V_1 е парен број. За да биде парна и сумата по V_2 треба да има парен број членови во сумата.

4. Одговор:
   a. не (има непарен број темиња со непарен степен),
   b. да (составен од K_4 и изолирано теме);
   c. не (не може да има теме со степен 5)
   d. да (составен од C_3 и K_2).

5. Одговор:
   a. да (K_5),
   b. не (има изолирано теме);
   c. не (не може биде сврзан, мора сите темиња со степен 1 да се сврзат со петтото теме, па тоа би имало степен 4)
   d. да (C_3).

6. Одговор: Секое ребро се брои еднаш како влезен и еднаш како излезен степен.

7. Одговор: $n(n-1)$. Од секое од n-те темиња може да излегува теме до сите останати темиња. Оттука, вкупниот излезен степен на сите темиња е најмногу $n(n-1)$, што од претходната задача е всушност $|E|$.

8. Помош: Ако граф има циклус, значи дека постојат темиња u и v ткави што постои и пат од u до v и пат од v до u.

9. Помош: Почнуваме да патуваме од темето u, и секогаш кога се влегува во теме и се излегува од него се трошат два степени од темето. Се додека по тој пат се влегува во теме со парен степен, од него ќе може и да се излезе (зошто?). Патот ќе заврши во теме од кое не може да се излезе (зошто ова теме има непарен степен?)

10. Помош: Започнуваме пат од произволно теме. Степенот на темето од кое ќе започнеме , откога ќе го тргнеме првото ребро од патот ќе стане непарно. Натаму треба да се искористи претходната задача за да се докаже дека постои алгоритам кој го формира циклусот.

11. Помош: Искористи ги задача 9 и 10.

12. Одговор: Комплементарен граф на K_n е граф со n темиња и без ребра. Комплементарен граф на C_n е граф со n темиња е граф изоморфен со граф на кој на кружница се наредени n темиња и секое теме е поврзано со сите освен неговите соседи.

13. Одговор: 8. Унијата на двата графа е комплетен граф. Комплетниот граф има $\frac{n(n-1)}{2}$ ребра, додека нашите два графа имаат вкупно 28 ребра.

14. Одговор: Нека u е теме на мост (u, v). Ако има степен 1, тогаш со негово отстранување ќе се отстрани тоа теме и мостот, па остатокот од графот ќе остане сврзан, затоа што всушност мостот го одделувал тоа теме со остатокот од графот. Ако пак нема степен 1, тогаш тоа е сврзано со друго теме, w, освен со v, па со негово отстранување веже нема да постои пат од v до w, што значи дека е точка на артикулација.

15. Одговор: Ако е е мост, тогаш со негово отстранување некои две темиња u и v веќе нема да бидат сврзани со пат, значи не може да постои пат меѓу тие две темиња кој не минува низ реброто e.

16. Одговор: На тој циклус во графот има барем две темиња u и v па бараните два патишта се меѓу нив.

17. Помош: Види што се случува кога прв пат двата пата се разидинуваат, и потоа кога прв пат пак ќе имаат заедничка точка.

1.6

1. Одговор: Не. Може да биде несврзан граф со циклус.
2. Одговор: Да.
3. Одговор: 6.
4. Одговор: 6.
5. Одговор: 9.
6. Одговор: 9.
7. Одговор: 10. Ако шумата се состои од 5 дрва со по две темиња.
8. Одговор: Тернарно дрво со висина 2.
9. Помош: Покажи дека во најлош случај тоа е комплетно дрво.
10. Доказ: Од теорема 1.9. комплетно m-арно дрво има m^h листови, од каде $h = \log_m l$.
11. Помош: Покажи дека најдобар случај е кога дрвото е комплетно.
12. Одговор: $\left\lfloor \frac{n-1}{2} \right\rfloor$. Колку што има внатрешни темиња, ако во секое ниво има само по едно внатрешно теме.
13. Одговор: Внатрешни темиња 3. Листови 7.
14. Одговор: Внатрешни темиња 3. Вкупно темиња 10.
15. Одговор: Не. Бројот на листови мора да е цел број и да важи $10 = 2i + 1$.

Втора глава

2.1

1. Одговор: ≈86. Се добива од $\left(\frac{1+\sqrt{5}}{2}\right)^n = 10^{18} \Longrightarrow n = 18\frac{\ln 10}{\ln\frac{1+\sqrt{5}}{2}}$.
2. Одговор: $\approx 10^6$. Точната вредност не може да се отпечати.
3. Одговор: $\approx 2^{15625}$. Се добива од $4^3 \ln n = 10^6$.
4. Одговор: $A[n] = A[n-1] + A[n-2] + A[n-5]$.
5. Одговор: $A[n] = A[n-1] + A[n-2]$. Се разгледуваат два случаи: кога низата завршува со 1, такви низи се $A[n-1]$ и кога завршува на 0. Кога завршува на 0, претпоследната цифра мора да биде 1-ца, па такви низи се $A[n-2]$.
6. Одговор:
   a. $A[n] = A[n-1] + A[n-3]$. Се разгледуваат два случаи: кога во последната колона се става една плочка вертикално, што се $A[n-1]$ начини да се наредат претходните плочки, и кога последните плочки се хоризонтално, кога има е $A[n-3]$ начини да се наредат претходните плочки.
   b. $A[0] = A[1] = A[2] = 1$.
7. Одговор:
   a. $A[n] = 2A[n-1] + 4A[n-2]$. Се разгледуваат два случаи: кога во последната колона се става една плочка вертикално, која може да се намести со црната половина горе или долу, што се два начини, па со такво редење имаме вкупно $2A[n-1]$ начини да се наредат плочките, и кога последните плочки се хоризонтално, па истите можат да се наредат на 4 начини, па такви начини има вкупно $4A[n-2]$.

b. $A[0] = 1, A[1] = 2.$

8. Одговор: $A[n] = pA[n-2] + (1-p)A[n-3]$. Се разгледуваат два случаи: кога последниот чекор е со должина 2, кога веројатноста е $pA[n-2]$ и кога последниот чекор е со должина 2, кога веројатноста е $(1-p)A[n-3]$.

   Почетните услови се $A[0] = 1$, $A[1] = 0$ и $A[2] = p$.

2.2

1. Одговор: 10^{12}. Најлош случај е кога сите купуваат сами, па броевите треба да се помали од $\frac{10^{18}}{10^6}$.

2. Одговор: Не, затоа што $O(n)$ време е потребно да се внесат податоците.

3. Одговор: Не, овој податок не менува многу. Иако во оптималното решение не може да се случи двајца еден до друг да купуваат посебно, може да се случи во оптималното решение повеќе луѓе да купуваат сами. Од друга страна сложеноста на решението е иста со сложеноста која ни е потребна да ги внесеме податоците, па временски не можеме да направиме побрз алгоритам.

4. Одговор: Кога $n > 2$

$$A[n] = min\begin{cases} A[n-1] + t_n, \\ A[n-2] + p_{n-1}. \\ A[n-3] + q_{n-2} \end{cases}$$

   Со почетни услови: $A[-2] = A[-1] = A[0] = 0$.

5. Одговор:

$$A[n] = min\begin{cases} A[n-1] + x_n \\ A[n-2] + x_{n-1} - x_n \end{cases}.$$

Со почетни услови: $A[-1] = A[0] = 0$.

2.3

1. Одговор: Равенствата за $A_1[1], A_2[1]$ и B се исти како во оригиналниот проблем, додека рекурентната релација е дадена со

$$\begin{cases} A_1[i] = \min\{A_1[i-1] + s_{1,i-1} + p_{1,i-1}, A_2[i-1] + s_{,2i-1} + t_{2,i-1}\} \\ A_2[i] = \min\{A_2[i-1] + s_{,2i-1} + p_{2,i-1}, A_1[i-1] + s_{1,i-1} + t_{1,i-1}\} \end{cases}.$$

2. Одговор: Рекурентната релација е дадена со

$$A_j[k] = \min_{m=1,2,3}\{A_m[k-1] + s_{m,k-1} + t_{m,j,k-1}\}, j = 1,2,3$$

со почетни услови $A_j[1] = x_j$.

На крај оптималното решение се добива со:

$$B = \min\{A_1[n] + y_1, A_2[n] + y_2, A_3[n] + y_3\}.$$

3. Одговор: Рекурентната релација е дадена со

$$\begin{cases} V[i] = \max\{V[i-1]p_{1i-1}, N[i-1]p_{2i-1}\} \\ N[i] = \max\{V[i-1]q_{1i-1}, N[i-1]q_{2i-1}\} \end{cases},$$

со почетни услови $V[1] = x, N[1] = y$. На крај оптималното решение се добива со:

$$B = \max\{V[n], N[n]\}.$$

4. Помош: Треба да се реконструира најдоброто решение и за него да се пресмета колку денови врнело.

2.4

1. Одговор:

   a. Не. На пример распоредите 1, 2, 1, 2 и 2, 1, 2, 1, во Шарено оро се различни, а во овој проблем се исти.

   b. Да. Едно од децата ќе го одбереме како прво дете, со што проблемот се сведува на проблемот Шарено оро.

2. Одговор:

   a. Ако сукцесивно ја заменуваме рекурзивната равенка ќе добиеме:

$$A[i] = m(m-1)^i \sum_{k=1}^{i-1} \left(-\frac{1}{m-1}\right)^k$$

   b. $O(\log n)$, бидејќи ако горниот степенски ред се пресмета ќе се добие

$$A[i] = (m-1)^i - (-1)^{i-1}(m-1).$$

3. Одговор:

   a. Последниот број стриктно поголем од k во низа со должина i мора да биде на позиција од $i, i-1$ или $i-2$. Ако е на позиција $j < i$ тогаш сите броеви на позициите од $j+1$ до i се помали или еднакви на k, па секој од нив може да се одбере на k начини, додека i-тиот број се бира на $(n-k)$ начини. Од друга страна и низата со должина $j-1$ е $m-k$ убава. Оттука, бројот на $m-k$ убави низи со должина i е:

$$A[i] = (n-k)(A[i-1] + A[i-2]k + A[i-2]k^2).$$

   b. Системот равенки може да се запише во следнава матрична форма:

$$\begin{bmatrix} A[i] \\ A[i-1] \\ A[i-2] \end{bmatrix} = \begin{bmatrix} n-k & (n-k)k & (n-k)k^2 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} A[i-1] \\ A[i-2] \\ A[i-3] \end{bmatrix}.$$

Оттука,

$$\begin{bmatrix} A[n] \\ A[n-1] \\ A[n-2] \end{bmatrix} = (n-k) \begin{bmatrix} 1 & k & k^2 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}^{n-2} \begin{bmatrix} A[2] \\ A[1] \\ A[0] \end{bmatrix}.$$

Со логаритамско степенување на матрицата ќе се добие алгоритам кој работи во време $O(27 \ln n)$.

2.5

1. Одговор:

   a. Рекурентната релација е дадена со системот равенки:

   $$\begin{cases} A_0[i] = A_1[i-1] \\ A_9[i] = A_8[i-1] \\ A_k[i] = A_{k-1}[i-1] + A_{k-1}[i+1], \qquad 0 < k < 9 \end{cases}.$$

   Решението ќе се содржи во

   $$\sum_{k=0}^{9} A_k[n].$$

   b. Почетните услови се:

   $$\begin{cases} A_0[1] = 0 \\ A_k[1] = 1, \qquad 0 < k \leq 9 \end{cases}.$$

2. Одговор: За да изведеме единствена равенка за секое поле, ќе ја проширime таблата за две редици и две колони пред и после. Тогаш рекурзивната равенка ќе биде:

$$A_{x,y}[i+1] = \frac{1}{8}\big(A_{x+2,y+1}[i] + A_{x+2,y-1}[i] + A_{x-2,y+1}[i] + A_{x-2,y-1}[i] + A_{x+1,y+2}[i] + A_{x-1,y+2}[i] + A_{x+1,y-2}[i] + A_{x-1,y-2}[i]\big).$$

Почетните услови ќе бидат веројатностите да се биде во поле кое не е на таблата и веројатноста да не се направи чекор:

$$A_{x,y}[i] = \begin{cases} 1, & 1 \le x, y \le 8 \\ 0, & \text{инаку} \end{cases}, i > 0.$$

и

$$A_{x,y}[0] = \begin{cases} 1, & x = a, y = b \\ 0, & \text{инаку} \end{cases}.$$

Секоја вредност $A_{x,y}[k]$ ќе ја содржи веројатноста да се биде во тоа поле после k чекори, а бараната веројатност е:

$$\sum_{x,y=1}^{8} A_{x,y}[k].$$

3. Одговор:

   a. Рекурентната релација е дадена со матричната равенка:

$$\begin{bmatrix} V[i] \\ N[i] \end{bmatrix} = \begin{bmatrix} p & q \\ 1-p & 1-q \end{bmatrix} \begin{bmatrix} V[i-1] \\ N[i-1] \end{bmatrix}.$$

   Решението ќе се содржи во $V[n]$.

   b. $O(\ln n)$.

4. Одговор: Рекурентната релација е дадена со матричната равенка:

$$\overrightarrow{A[\imath]} = P\overrightarrow{A[\imath - 1]}.$$

5. Одговор:

   a. Нека $E[i]$ се оние низи кои завршуваат на единица, а $N[i]$ оние низи кои завршуваат на нула. Рекурентната релација е дадена со матричната равенка:

$$\begin{bmatrix} E[i] \\ E[i-1] \\ E[i-2] \\ N[i] \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} E[i-1] \\ E[i-2] \\ E[i-3] \\ N[i-1] \end{bmatrix}.$$

   Почетните услови се

$$E[0] = E[1] = E[2] = N[0] = 1, N[1] = N[2] = 0\,.$$

   Решението ќе биде $E[n] + N[n]$.

   b. $A[i] = 2A[i-1] - A[i-2] + A[i-4]$.

   Прв начин: да се забележи дека

$$A[i-1] = E[i] = E[i-1] + N[i-1]$$

$$N[i] = E[i-3] + N[i-1] + E[i-1] - E[i-1]$$

$$= A[i-3] + A[i-1] - A[i-1].$$

   Втор начин: да се забележи дека

$$A[i] = A[i-1] + \sum_{k=0}^{i-4} A[k].$$

   Оттука, $A[i] - A[i-1] = A[i-1] - A[i-2] + A[i-4]$

3.1

1. Одговор: Новата релација ќе биде:

$$A[i,j] = \begin{cases} min(A[i-1,j], A[i,j-1]) + m_{ij}, & i,j > 1 \\ A[i-1,j] + m_{ij}, & j = 1, i > 1 \\ A[i,j-1] + m_{ij}, & i = 1, j > 1 \end{cases}$$

   Со почетни услови $A[1,1] = m_{11}$.

2. Одговор:

$$A[i,j] = \min_k \min\{A[i,k] + A[k+1,j]\} + p_{i-1}p_k p_j.$$

3. Одговор: Новата релација ќе биде:

$$A[i,j] =$$

$$= \begin{cases} min(A[i-1,j-1], A[i-1,j], A[i,j-1]) + m_{ij}, i,j > 1 \\ A[i,j+1] + m_{ij}, & j = 1, i > 1 \\ A[i,j-1] + m_{ij}, & i = 1, j > 1 \end{cases}$$

   Со почетни услови $A[1,1] = m_{11}$.

4. Одговор: Дефинираме нова функција $B[i,j]$:

$$= \begin{cases} B[i-1,j] + B[i,j-1] - B[i-1,j-1] + A[i,j], & i,j > 1 \\ B[i-1,j] + A[i,j], & j = 1, i > 1 \\ B[i,j-1] + A[i,j], & i = 1, j > 1 \end{cases}$$

   Со почетни услови $B[1,1] = A[1,1]$.

5. Одговор: Релацијата ќе биде: $B[1,1] = A[1,1]$ и

$$A[i,j] = \begin{cases} min(A[i-1,j], A[i,j-1]) + m_{ij}, & i,j > 1 \\ A[i-1,j] + m_{ij}, & j = 1, i > 1 \\ min\big(A[i,j-1] + m_{ij}, m_{ij}\big), & i = 1, j > 1 \end{cases}$$

Решението ќе биде $min(A[n,j])$.

6. Одговор: Дефинираме нова функција $B[i,j]$ таква што $B[1,1] = 1$ и $B[j,0] = B[i,0] = 0$ за сите i,j.

   Ако $B[i-1,j] = 0, B[i,j-1] > 0$ и $A[i-1,j] < A[i,j]$, тогаш $B[i,j] = B[i-1,j] + 1$

   Ако $B[i,j-1] = 0, B[i-1,j] > 0$ и $A[i,j-1] < A[i,j]$, тогаш $B[i,j] = B[i,j-1] + 1$

   Ако $B[i,j-1] > 0, B[i-1,j] > 0, A[i,j-1], A[i-1,j] < A[i,j]$, тогаш $B[i,j] = \max\{B[i-1,j], B[i,j-1]\} + 1$

   Во сите други случаи $B[i,j] = 0$. Крајното решение е $\max\limits_{i,j} B[i,j]$.

6. Одговор:

П 3. 13. ПСЕВДОКОД ЗА МАКСИМАЛЕН ПАТ ВО ТРИАГОЛНИК

```
1  Внеси ја матрицата A[i,j];
2  B[1,1] = s_11;
3  за j = 2 до m прави A[1,j] = A[1,j-1] + s_1j
4  за i = 2 до n прави A[i,1] = A[i-1,1] + s_i1;
5  за j = 2 до m прави
6     за i = 2 до n − i прави
7           A[i,j] = max(A[i,j-1], A[i-1,j]) + s_ij
8  печати max{A[i,n-i]} и полето (i,n-i) за тој максимум
```

3.2

1. Одговор: Проблемот се сведува на наоѓање на најдолга заедничка подниза на дадениот стринг и неговиот обратен стринг.

2. Одговор:

   a. Треба да го најдеме она оптимално решение со кое не се стапнува на ниту едно поле во соодветната матрица кое е на дијагоналата. Всушност треба да се најде најдолгиот пат кој е целосно под или целосно над дијагоналата.

   b. Рекурзивната релација за $A[i,j]$ е:

$$\begin{cases} 0, & i=0, j=0 \text{ или } i \le j \\ A[i-1,j-1]+1, & i>j>0 \text{ и } x_i = x_j \\ \max(A[i,j-1], A[i-1,j]), & i>j>0 \text{ и } x_i \neq x_j \end{cases}.$$

3. Одговор:

   a. Треба да најдеме најдолга заедничка подниза на дадената низа и на истата таа низа, но подредена. Оваа стратегија ќе го реши проблемот бидејќи елементите во најдолгата заедничка подниза се во ист редослед и во оригиналната и во подредената низа.

   b. За подредување на низата ни треба време $O(nlnn)$, па временската сложеност зависи од алгоритамот за најдолга заедничка подниза, што е $O(n^2)$.

   c. Можеме да искористиме поинаква рекурентна равенка. Нека $A[j]$ е должината на најдолгата растечка подниза за поднизата до j -тиот елемент, која го вклучува j -тиот елемент. Тогаш $A[j] = A[k] + 1$ за некое k за кое k -тиот елемент од низата е помал од j-тиот:

$$A[j] = 1 + \max_{k<j, x_k<x_j} A[k].$$

   Најдолгата растечка подниза не мора да завршува со последниот елемент па затоа оптималното решение е

$$\max_k A[k].$$

4. Одговор:

   a. Нека $A[j]$ е сумата на елементите на растечката подниза со најголема сума, за поднизата до j-тиот елемент, која го вклучува j-тиот елемент. Тогаш

   $$A[j] = x_j + \max_{k<j, x_k<x_j} A[k].$$

   b. Оптималното својство е следново: Ако оптималната растечка подниза го содржи елементот на позиција k, тогаш поднизата до k-тиот елемент е оптимална подниза за низата до k-тиот елемент.

5. Одговор: Нека U^i ќе ја обележуваме подизата до i-тиот елемент од низата.

   a. Оптималната потструктура е дадена со следново својство:

   За дадени две низи $X = (x_1, x_2, \cdots x_m)$ и $Y = (y_1, y_2, \cdots y_n)$, нека $Z = (z_1, z_2, \cdots z_k)$ е најкратката нивна комбинација. Тогаш

      - Ако $x_m = y_n$, тогаш $z_k = x_m = y_n$ и Z^{k-1} е најкратка комбинација на X^{m-1} и Y^{n-1}.

      - Ако $x_m \neq y_n$ и $z_k \neq x_m$, тогаш мора $z_k = y_n$ и следува дека Z^{k-1} е најкратка комбинација на X^{m-1} и Y.

      - Ако $x_m \neq y_n$ и $z_k \neq y_n$ тогаш мора $z_k = x_m$ и следува дека Z^{k-1}е најкратка комбинација на X и Y^{n-1}.

   b. Ако последните две букви во низите се еднакви, $x_m = y_n$, тогаш во најкратката комбинација $z_k = x_m = y_n$. Ако не е така, тогаш бидејќи двете низи се поднизи на Z, некаде во Z има некој елемент $z_{k'}$, кој одговара на x_m и има некој елемент $z_{k''}$, кој одговара на y_n. Тогаш, и X и Y се е поднизи на $Z^{\max\{k', k''\}}$, која е пократка подниза од Z што заедничка комбинација, што е контрадикција. Ако пак $x_m \neq y_n$, тогаш z_k мора биде барем еден од x_m и y_n, бидејќи ако се разликува и од двата, тогаш бидејќи и X и Y се е поднизи на

Z , x_m и y_n ги има на некои позиции порано, k', k'' па повторно $Z^{\max\{k',k''\}}$, која е пократка подниза од Z што заедничка комбинација, што е контрадикција.

c. Рекурзивната релација за $A[i,j]$ е следнава:

$$\begin{cases} 0, & i = 0 \text{ или } j = 0 \\ A[i-1, j-1] + 1, & i, j > 0, x_i = y_j \\ min(A[i, j-1], A[i-1, j]) + 1, & i, j > 0, x_i \neq y_j \end{cases}$$

6. Одговор: Нека U^i ќе ја обележуваме подизата до i -тиот елемент од низата.

a. Секоја операција вметнување во втората низа може да се замени со операција бришење во првата низа, затоа што елементот што сакаме да го вметнеме во првата низа мора да го има во втората, па за иста цена можеме да го избришеме од втората. Слично, секоја операција бришење во втората низа може да се замени со операција вметнување во првата низа. Од друга страна не е битно дали промена на стринг ќе се направи во првата или во втората низа.

b. Оптималната потструктура е дадена со следново својство:

Нека $A[m,n]$ се минималниот број на операции со кои низата $X = (x_1, x_2, \cdots x_m)$ се претвара во низата $Y = (y_1, y_2, \cdots y_n)$. Тогаш

- Ако $x_m = y_n$, $A[m,n] = A[m-1, n-1]$.
- Ако $x_m \neq y_n$, тогаш или $A[m,n] = 1 + A[m-1, n-1]$, ако најдоброто решение е да се замени x_m со y_n, или $A[m,n] = 1 + A[m, n-1]$, ако најдоброто решение е да се додаде y_n или $A[m,n] = 1 + A[m-1, n]$, ако најдоброто решение е да се избрише x_m.

c. Рекурзивната релација за $A[i+1, j+1]$ е:

$$\begin{cases} A[i,j], & x_i = y_j \\ min(A[i,j-1], A[i-1,j], A[i-1,j-1]) + 1, & x_i \neq y_j \end{cases}$$

7. Одговор: Нека $A[m,n]$ е минималната цена за која низата $X = (x_1, x_2, \cdots x_m)$ се претвара во низата $Y = (y_1, y_2, \cdots y_n)$. Рекурзивната равенка е слична како равенката во претходната задача, со тоа што кога $x_i \neq y_j$ равенката е:

$$min(A[i,j-1] + a, A[i-1,j] + b, A[i-1,j-1] + \mathrm{c}).$$

3.3

1. Одговор: Бидејќи предмет кој ќе се стави ви ранецот се троши, мораме да воведеме уште една променлива која ќе води сметка кои предмети се пробани да се стават во ранецот. За да тоа можеме да го забележиме ќе ги подредиме елементите по некој редослед.

   a. Нека $A[i,r]$ е оптималното пополнување на ранец со големина i со првите r елементи. Сега наместо да итерираме и да наоѓаме максимум по сите елементи, задачата станува полесна: или го вклучуваме r-тиот елемент или не, па го земаме максимумот од овие две можности:

   $$A[i,r] = \max(A[i - t_r, r-1] + v_r, A[i,r-1])$$

   b. Сложеноста останува иста $\mathrm{O}(nm)$

2. Одговор:

   a. Не може да се случи да не се земе дел од некој предмет затоа што не го собира во ранецот, бидејќи можеме од секој предмет да ставаме толку колку што ќе собере во ранецот. Така, треба да ги подредиме предметите по опаѓачки редослед по цена на единица волумен, и да

ставаме од најскапиот предмет, додека собира во ранецот, кога ќе го потрошиме ќе ставиме од тој после него, и така се додека не го пополниме целиот ранец.

b. Сложеноста зависи од тоа како е дадена низата предмети. Ако е подредена по цена на единица количина, тогаш имаме линеарна сложеност, ако не, прво треба да ја подредиме, па сложеноста ќе биде $O(mlnm)$

3. Помош: Ќе се додаде уште една димензија, т.е. уште една променлива во рекурзивната равенка која ќе одговара на капацитетот на вториот ранец. Потоа за секој предмет ќе пробаме да го ставиме и во првиот и во вториот ранец (или во ниту еден од нив).

4. Одговор:

   a. Двете половини можеме да ги гледаме како ранци. Ако помалата половина ја разгледуваме како ранец со капацитет пола од сумата на елементите, тогаш проблемот се сведува на оптимално пополнување на тој ранец, ако сите предмети (броеви во низата) имаат цена еднаква на нивниот волумен (бројот), т.е. волуменот и цената се исти.

   b. Равенката е иста како 0-1 проблем на ранец, задача 2, само волуменот и цената се исти.

$$A[i,r] = \max(A[i - t_r, r-1] + t_r, A[i, r-1])$$

   c. $O(nm)$, каде n е половината од сумата на елементите, а m е должината на низата.

5. Одговор:

   a. Проблемот над множество $\{s_1, s_2, \dots, s_i\}$ може да се подели на проблем во кој во сумата има барем една монета со вредност s_i и проблем во кој во сумата нема ниту една монета со вредност s_i. Ако во сумата има барем една монета со вредност s_i, тогаш останатата сума $m - s_i$ треба да се плати со истото множество од монети. Ако ниту една

монета не е со вредност s_i, тогаш целата сума m треба да се плати со монети од мнжеството $\{s_1, s_2, \dots, s_{i-1}\}$.

b. Се сведува на проблем на ранец со неограничен број на елементи, каде волуменот и цената се исти. Нека $A[m, i]$ е бројот на начини на кои може да се плати сума од m денари со монети од множеството $\{s_1, s_2, \dots, s_i\}, i \leq k$. Тогаш $A[m, i]$ е

$$\begin{cases} A[m - s_i, i] + A[m, i-1], & m > 0, i > 0 \\ 0, & m < 0 \text{ или } m > 0, i \leq 0. \\ 1 & m = 0 \end{cases}$$

c. Бидејќи рекурзивната равенка е со две променливи, а за определување на решението се користат константен број на операции, сложеноста е $\Theta(n^2)$.

6. Упатство:

a. Се сведува на задачата да се подели низа на два дела така да разликата меѓу нив да биде најмала. Оптималното време е сумата во низата со поголема вредност.

b. Се сведува на 0-1 проблем на ранец, каде капацитетот на ранецот е даденото максимално време, а времињата за гласање се волумените на предметите. Потоа треба да се провери дали останатите елементи имаат сума помала од даденото максимално време.

3.4

1. Одговор: можеме да забележиме дека секој чекор нагоре по столбот одговара на отворена заграда, а секој чекор надолу на столбот на затворена заграда.

a. Ако n е непарен број, тогаш не е можно, а ако n е парен број тогаш тоа е број на начини на правилно распределени $\frac{n}{2}$ парови загради.

b. Упатство: Полжавот да се најде на позиција m е еквивалентно на веќе поставени n загради, од кои m се само отворени, а $\frac{n-m}{2}$ се целосно поставени загради.

2. Одговор: Проблемот се сведува на проблемот на правилно поставување на загради, но наместо n, треба да се постават $n-1$ парови на загради.

3. Одговор: Коренот на целосно бинарно дрво е внатрешно теме. Ако во неговото лево поддрво има од k внатрешни темиња, каде k се движи од 0 до $n-1$, тогаш во неговото десно поддрво има $n-1-k$ внатрешни темиња. Според ова, рекурзивната равенка за бројот на целосни бинарни дрва бројот е иста со рекурзивната равенка за број на начини на правилно запишани загради по првиот принцип.

4. Одговор: Нека луѓето околу масата ги обележиме со броевите од 1 до $2n$. Првиот човек мора да се поздрави со некој кој е на парна позиција, затоа што ако не е така меѓу нив (без разлика дали од едната или од другата страна) ќе останат непарен број на луѓе кои ќе треба да се поздраваат меѓу себе, а тоа не е можно. Ако првиот човек се поздрави со човекот на позиција $2i$, тогаш тие ќе ја поделат масата на два дела, првиот со $2(i-1)$ човек, а другата со останатите $2(n-i)$ луѓе. Секоја од овие две групи мора да направат поздравувања меѓу себе, бидејќи инаку би имало прекрстување со раката на првиот човек. Групата од лево на првиот човек може да има од 0 до $2(n-1)$ човек, па повторно ја добиваме равенката за број на правилно наредени загради по првиот принцип.

5. Нека темињата на n -аголниот полигон ги обележиме со броевите од 1 до n. Секоја страна од полигонот припаѓа на некој триаголник од триангулацијата. Така, страната со крајни темиња 1 и n припаѓа на еден триаголник, а третото теме на тој триаголник е некое друго теме од темињата од 2 до $n-1$. Ако е тоа темето k, тогаш триаголникот со темиња 1, k и n е едн триаголник во таа триангулација, а останатите триаголници се или во полигонот со темињата од 1 до k или во полигонот со темињата од k до n. (ова може и да не претставуваат вистински полигони ако се составени само од едно или две темиња).

a. Ако со $A[i]$ го означиме бројот на триангулации на полигон со n темиња, тогаш

$$A[i] = A[2] \cdot A[i-1] + A[2] \cdot A[i-1] + \cdots + A[i-1] \cdot A[2].$$

Почетните услови се за 3, т.е. $A[3] = 1$, но бидејќи во формулата се јавува $A[2]$ како почетен услов ќе земеме $A[2] = 1$, иако нема триангулација со две темиња, но има само еден начин тие да се поврзат.

b. Рекурзивната равенка има ист природа како и равенката за правилно поставување на загради дадена со вториот пристап, но со поголеми почетни услови, и всушност бројот на триангулации со n темиња е бројот на $n-2$ пара правилно поставени загради.

6. Бидејќи патот не смее да ја премине дијагоналата, никогаш не смеат да се направат повеќе чекори во лево отколку чекори надолу. Така, ако секој чекор во лево го обележиме со гледаме како отворена заграда, а секој чекор надолу со затворена заграда, бројот на вакви патишта ќе биде еднаков на бројот на правилно распределени загради.

    a. Упатство: патот од првата позиција која е над главната дијагонала на која поминува патот може да се преслика осносиметрично во однос на дијагоналата. Ваквиот пат има еден чекор во лево повеќе отколку патот кој се движи само под главната дијагонала, па е пат во матрица од ред $(n-1) \times (n+1)$. Од друга страна секој пат во матрица од ред $(n-1) \times (n+1)$ со обратната постапка може да се претвори во пат во матрица пат во матрица од ред $n \times n$ кој ја пресекува главната дијагонала.

    b. Од а. Имаме $\binom{2n}{n+1}$ патишта во матрица од ред $n \times n$ кои ја пресекува главната дијагонала се $\binom{2n}{n+1}$. Вкупниот број на патишта во матрица од ред $n \times n$ кои не ја пресекуваат главната дијагонала е еднаков на

вкупниот број на патишта минус оние патишта кои ја пресекуваат главната дијагонала

$$\binom{2n}{n} - \binom{2n}{n+1} = \binom{2n}{n} - \frac{n}{n+1}\binom{2n}{n} = \frac{1}{n+1}\binom{2n}{n}$$

Четврта глава

4.1

1. Одговор: Нека имаме оптимално решение на проблемот во кое главите се делат во $m-1$ наместо во $m<n$ томови. Тогаш постои том со барем две глави и него можеме даго поделиме на два тома, со што ќе добиеме друго оптимално решение.

2. Одговор: За да се реши овој проблем треба да се заменат местата на операциите минимум и максимум.

   a. Рекурзивната равенка е следнава:

$$A[i,j]=\begin{cases}\sum_{r=1}^{j} x_r, & i=1\\ \max\limits_{k<j}\left\{\min\left\{A[i-1,k],\sum_{r=k+1}^{j} x_r\right\}\right\}, & 1<i\le j\end{cases}.$$

   b. Оптималното својство гласи потполно исто како и во проблемот кој го разгледуавме: Ако во оптималното разместување i-те први томови се составеи од j-те први глави, $i<j$, тогаш постои оптимално разместување на сите n глави во сите m томови кое го вклучува оптималното разместување на првите j глави во i томови.

   c. Нека во оптималното разместување минималниот број на страни во некој том е M и првите i томови биле разместени во j-те први глави, но не оптимално (има подобар начин кој ќе го зголеми бројот на страни во најтенкиот том од првите i томови. Можни се два случаи за тоа каде се наоѓа најтенкиот том: или е во првите i томови, или е во останатите. Во првиот случај, во првите i томови има том со дебелина M. Но, тие не се разместени

оптимално, па значи можеме да ги разместиме пооптимално, и да го зголемиме бројот на страни во најтенкиот том, односно да ги наредиме главите така да бројот на страни во сите тие томови е стриктно поголем од M, што е контрадикција. Ако пак томот со најмалку страници е во останатите глави, тогаш воопшто не е проблем првите j глави оптимално да ги наредиме во i-те томови, затоа што M нема да се смени.

d. Ако сите глави ги ставиме во еден том, тогаш тој секако ќе биде најголем можен.

3. Одговор: Нека последниот том ги содржи од $k+1$-тата до j-тата глава. Јасно е дека $A[i,k-1] \leq A[i,k] \leq A[i,k+1]$, бидејќи колку помалку глави треба да се наредат во ист број на томови толку томот со најголем број страни е помал.

Нека во оптималното решение последниот том е оној кој е најголем. Да претпоставиме дека сумата на главите од k-та до j-тата да е исто така поголема од $A[i,k-1]$. Тогаш од:

$$\sum_{r=k}^{j} x_r < \sum_{r=k-1}^{j} x_r$$

Следува дека ако последниот том е од $k+1$-та до j-тата е пооптимално решение отколку да е k-тата до j-тата глава, што е контрадикција.

Ако пак во оптималното решение најдебелиот том не е последниот, тогаш $A[i,j] = A[i,k]$. Ако во последниот том ја ставиме и k-тата глава, повторно се добива оптимално решение. Ова оптимално решение не може да биде подобро од претходното за кое претпоставивме дека е оптимални, па мора да биде исто. Но тогаш во последниот том ќе има повеќе страници, но сеуште ќе биде помал од $A[i,k]$. Тоа значи дека ако последниот том не е онаој со најголем број страници во него можеме да ставаме што е можно повеќе глави, т.е. да ставаме глави се до моментот додека $A[i,k-1]$ не стане помало од сумата на страните во последниот том

4. Одговор:

   a. Можеме да направиме низа од растојанијата помеѓу две соседни позиции, при што ќе добиеме низа која е за еден помала од претходната. Тогаш проблемот се сведува на поделба на $n-1$ броеви на m групи, така да сумата на групата со минимална сума биде најголема можна.

   b. Ако во оптималното решение нема човек на последната позиција, тогаш последниот во низата можеме да го поместиме на последната позиција и со тоа нема да намалиме некое растојание, односно, минималната сума нема да се намали. Слично, ако нема човек на првата позиција, првиот во низата можеме да го поместиме на првата дадена позиција.

   c. Нека во оптималното разместување i-тиот човек се наоѓа на j-тата позиција, Тогаш постои оптимално разместување на сите m луѓе на сите n позиции кое го вклучува оптималното разместување на првите i луѓе на првите j позиции.

4.2

1. Одговор: Не, кога го бараме вкупниот број на растечки поднизи не интересира која е нивната должина, па овој проблем може да се реши во време $\Theta(n^2)$ со рекурзивна равенка за $A[i]$ што го означува бројот на низи кои завршуваат во i -тиот елемент, дадена со:

$$A[i] = 1 + \sum_{j<i, x_j<x_i} A[j]$$

2. Одговор: Прв начин: Се е исто, само единствена разлика е наместо $x_j < x_i$ да се стави $x_j > x_i$.

   Втор начин: може да се најде максимална растечка подниза на обратната низа.

3. Одговор: Потребно е да се направат две рекурзивни функции, една за низи кај кои последниот елемент е помал од претпоследниот и една за низите кај кои последниот елемент е поголем од претпоследниот. Рекурзивните равенки на која се базира решението се следниве:

$$A[i,r] = \sum_{j<i, x_j<x_i} B[j, r-1],$$

$$B[i,r] = \sum_{j<i, x_j>x_i} A[j, r-1],$$

$$A[i,1] = B[i,1] = 1,$$

каде $A[i,r]$ се бројот на низи низи кај кои последниот елемент е поголем од претпоследниот, а $B[i,r]$ се бројот на низи кои последниот елемент е помал од претпоследниот.

Вкупниот број на такви низи е $A[n,k] = B[n,k]$.

4. Упатство: Искористи го следниов факт за да конструираш решение слично на решението на проблемот на број на растечки поднизи: Низите со должина r кои имаат сума M и кај кои последниот елемент е на позиција i, се низи со должина должина $r-1$ кои имаат сума $M - x_i$ на кои како последен елемент им е додаден елементот x_i.

4.3

1. Одговор: Нека минималното време се добива кога се множат од i-тата до k-тата матрица, па останатите. Откога се пресметани двата производи за да се помножат тие матрици потребно е време $p_{i-1}p_k p_j$. Претходно требало да се пресмета ат овие две матрици, но бидејќи тоа може да се прави паралелно, вкупното време е максимумот од времињата да се пресмета матрицата од i до k и matricata od $k+1$ до j. Оттука рекурзивната равенка ќе биде:

$$A[i,j] = \begin{cases} 0, & i = j \\ \min\limits_{i \le k < j} \{\max\{A[i,k], A[k+1,j]\} + p_{i-1}p_k p_j\}, & i < j \end{cases}$$

2. Одговор: Нека темињата ги обележиме со броевите од 1 до n, а далечината од темето i до темето j со $d[i,j]$. Нека $A[i,j]$ е големината на оптималната триангулација.

   a. Рекурзивната равенка е:

   $$A[i,j] = \min_{i<k<j}\{A[i,k] + A[k,j] + d[i,j] + d[i,k] + d[k,j]\}$$

   Со почетни услови $A[i,i] = 0$ и $A[i,i+1] = d[i,i+1]$. Оптималното решение се добива во $A[1,n]$.

   b. Сложеноста е $O(n^3)$

   c. Ако секоја страна се брои по еднаш, рекурзивната равенка е:

   $$A[i,j] = \min_{i<k<j}\{A[i,k] + A[k,j] + d[i,j]\}$$

   Со истите почетни услови како и со повеќе-кратно броење.

3. Одговор: Дефинираме две функции $Max[i,j]$, функција која ја зачувува максималната вредност ако заграда се стави пред a_i и после a_j и $Min[i,j]$, функција која ја зачувува минималната вредност ако заграда се стави пред a_i и после a_j.

   a. Ќе имаме систем од рекурзивни равенки:

   $$Max[i,j] = \max_{i \le k < j}\{Max[i,k] - Min[k+1,j]\};$$
   $$Min[i,j] = \min_{i \le k < j}\{Min[i,k] - Max[k+1,j]\}$$

Со почетни услови $Max[i,i] = Min[i,i] = a_i$. Оптималното решение се добива во $Max[1,n]$.

b. Сложеноста е $O(n^3)$

4. Одговор:

a. Во првиот потег можеме да го избереме топчето на позиција $k, k = \overline{1,n}$. Тогаш од низата ќе се отстрани некој палиндром $x_{k-r,} \dots, x_{k-1}, x_k, x_{k+1,} \dots, x_{k+r}$. Оттука, минималниот број на потези ако во првиот чекор сме го одбрале топчето на позиција k, е минималниот број на потези да се отстранат топчињата од поднизата $x_{1,} x_{2,} \dots, x_{k-r}$ плус минималниот број на потези да се отстранат топчињата од поднизата $x_{k+1,} \dots, x_{k+r}$ плус потезот кој веќе сме го искористиле. Оттука можеме да дојдеме до следнава рекурентна релација:

$$A[i,j] = 1 + \min_{i \le k \le j} \{A[i,k-r] + A[k+r,j]\},$$

каде $A[i,j]$ е минималниот број на потези да се отстрани поднизата $x_{i,} \dots, x_j$.

b. Треба да се увиди дека секако во некој потез ќе мора да се отстрани првиот елемент од низата. Тој ќе се отстрани со некој палиндром, кој во даденият момент е најголемиот палиндром со непарна должина на кој првиот елемент му е почеток, а е во рамките на некоја подниза. Постапката со оптимален број на чекори може да се преправи, така да прво се отстрани овој палиндром, па потоа останатите палиндроми по редоследот кој сме го имале во оптималното решение. Така го имаме следново својство:

> *Постои оптимално решение во кое прво ќе се отстрани палиндром кој почнува со првиот елемент на низата.*

Користејќи го ова својство, рекурзивната равенка ќе биде следнава:

$$A[i] = 1 + \min_{x_{i,\dots,}x_{j-1} \text{ е непарен палиндром}} \{A[j]\}.$$

4.4

1. Одговор: Треба да се разгледаат сите можни целосни бинарни дрва што е еднакво на Каталановиот број: $\frac{1}{n+1}\binom{2n}{n}$.

2. Одговор: Нека $T[n]$ е времето кое е потребно да се пресмета оптимално дрво со n внатрешни темиња. Тогаш

$$T[n] = \sum_{k=1}^{n}(T[k-1] + T[n-k]) + n = 2\sum_{k=0}^{n-1} T[k] + n.$$

Ако ги одземеме n -тиот и $N-1$-виот елемент ќе добиеме

$$T[n] - T[n-1] = 2T[n-1] + n - (n-1).$$

Оттука $T[n] = 3T[n-1] + 1$. Па сложеноста е $O(3^n)$.

3. Одговор: За некои n нема целосни бинарни дрва, како на пример за 2 или за 4. Ќе се добијат дрва кои се само бинарни.

4. Одговор: Секој бинарен куп почнува со максималниот елемент, а едно дете му е претпоследниот елемент по големина. Ако сите елементи освен последниот ги поделиме на две групи, едната група ќе бидат во едното, а другата во другото дрво.

Нека $T[n]$ е вкупниот број на дрва со n темиња, тогаш коренот може да се избере на еден начин, левото и десното дрво може да имаат од 1 до $n-2$ темиња (освен кога е n=1 и n=2 кога постои само едно дрво). Ако левото дрво има k темиња, десното има $(n-k-1)$, па рекурзивната равенка е:

$$T[n] = \sum_{k=1}^{n}\binom{n-1}{k}(T[k]T[n-k-1]).$$

Сложеноста на решението со динамичко програмирање е $O(n^2)$.

Петта глава

5.1

1. Одговор: Пермутациите кои почнуваат со 1 се порано, па ако имаме парички со сума i, тогаш $A[i-1]$ почнуваат со 1, а останатите $A[i-2]$ почнуваат со 2.

   a. Нека со $B[i,j]$ ја обележиме j-тата пермутација за сума на парички i. Тогаш:

$$B[i,j] = \begin{cases} 1, & i = 1, j = 1 \\ \lambda, & i = 0 \\ 1 \circ B[i-1,j], & i > 1, j \leq A[i-1], \\ 2 \circ B\big[i-2, j-A[i-1]\big] & i > 1, j > A[i-1] \end{cases}$$

   каде λ е празен стринг, т.е. ништо не се допишува.

   b. Нека $C[i,\alpha]$ го претставува редниот број на пермутацијата на кои сумата им е i. Секако α е таква пермутација. Тогаш,

$$C[i,\alpha] = \begin{cases} 1, & i = 1 \\ A[i-1] + C[i-2,\alpha'], & i > 1, \alpha = 2\alpha'. \\ C[i-1,\alpha'] & i > 1, \alpha = 1\alpha' \end{cases}$$

2. Одговор: Решението е исто како на претходната задача, само треба да се додадат нули на крај за да бројот стане n-цифрен.

3. Одговор: Рекурзивната релација за пресметување на бројот на битстрингови кои немаат две последователни нули е иста со бројот на низи од единици и двојки кои имаат сума 2. Според тоа пак ја користиме рекурзивната функција $A[0] = 1, A[1] = 2$ и $A[i] = A[i-1] + A[i-2]$.

   a. Нека со $B[i,j]$ ја обележиме j-тата низа за битстринг со должина i. Тогаш:

$$B[i,j] = \begin{cases} j-1, & i=1, j=1,2 \\ \lambda, & i=0 \\ 01 \circ B[i-2,j], & i>1, j \le A[i-2], \\ 1 \circ B\big[i-1, j-A[i-2]\big] & i>1, j>A[i-2] \end{cases}$$

каде λ е празен стринг, т.е ништо не се допишува.

b. Нека $C[i,\alpha]$ го претставува редниот број на пермутацијата на кои сумата е i. Секако α е таква пермутација. Тогаш,

$$C[i,\alpha] = \begin{cases} \alpha+1, & i=1 \\ A[i-2] + C[i-1,\alpha'], & i>1, \alpha = 1\alpha' \\ C[i-2,\alpha'] & i>1, \alpha = 01\alpha' \end{cases}.$$

c. Во двата проблеми кога ќе се стави единица напред тогаш знаеме дека пред него се сите стрингови кои почнуваат со 0, Во овој проблем тоа се сите низи кои почнуваат со 01 и тие се $A[i-2]$, а во проблемот со бит низи во кои нема две последователни единици, кога ќе се стави единица напред, тие се $A[i-1]$,

5.2

1. Одговор: $O(nr+n^2)$.

2. Упатство: Во кодот за генерирање на r-тиот битстринг треба деловите каде што се печати 0 да се игнорираат, а секаде каде што се печати 1 да се печати i. Во кодот за определување на редниот број на битстринг треба секаде каде што се проверува дали битот почнува со единица да се промени и наместо тоа да се види кој е најголемиот елемент кој го има во комбинацијата и ако тоа е i за него да се додаде $A[i,j]$, каде j е бројто на моментални елементи во комбинацијата.

3. Одговор: Бројот на пермутации со повторување од множество со n елементи со должина i се n^i. Ако бројот со должина i почнува со цифрета j тогаш пред него се $(j-1)n^{i-1}$елементи.

a. Рекурзивната равенка за определување на k тиот елемент е:

$$B[i,k] = jB\left[i-1, k-(j-1)n^{i-1}\right],$$

каде j е такво што $(j-1)n^{i-1} < k \leq jn^{i-1}$

b. Рекурзивната равенка за определување на редниот број на пермутацијата α:

$$C[i,\alpha] = (j-1)n^{i-1} + C[i-1,\alpha'],$$

каде $\alpha = j\alpha'$.

c. Сложеноста е $O(n)$, заради тоа што функцијата n^i која се користи овде како подрутина се пресметува во логаритамско време.

4. Одговор:

a. Нека бројот на елементи во S_i го обележиме со $A[i]$. Тогаш

$$A[i] = iA[i-1].$$

Со почетен услов $A[1] = 1$

b. Рекурзивната равенка за определување на редниот број на пермутацијата α:

$$C[i,\alpha] = (j-1)A[i-1] + C[i-1,\alpha'],$$

каде $\alpha = j\alpha'$.

$C[i,\alpha]$ дава колку пермутации има пред нашата, па редниот број на пермутацијата што ја бараме е $C[i,\alpha]+1$.

c. Рекурзивната равенка за определување на k тиот елемент е:

$$B[i,k] = jB[i-1, k-(j-1)A[i-1]],$$

каде j е такво што $(j-1)A[i-1] < k \leq jA[i-1]$. Почетен услов е $B[1,1] = 1$.

d. Сложеноста е $O(n)$.

5. Упатство:

   a. Секоја низа од задача 3, читана од лево на десно кажува кој број од множеството елементи ќе го ставиме на таа позиција. Потоа тој број го бришеме. На пример 221 значи од множеството елементи {1, 2, 3} прво го земаме вториот елемент 2, и останува {1, 3}. Потоа го земаме вториот елемент 3, и на крај останува да се земе 1. Значи одговара на пермутацијата 231.

   b. За генерирање на елемент, откога ќе се добие решението во задача 3 ќе се конвертира во пермутација. За да се најде редниот број на дадена пермутација, прво таа ќе се конвертира во низа од задача 3, а потоа ќе се реши задача 3.

   c. Треба да се види која е сложеноста да се конвертира пермутација од задача 3 во пермутација со повторување и обратно. Најбргу може да се направи во логаритамско време.

5.3

1. Одговор: Нека α е дадениот стринг на кој првата затворена заграда се наоѓа на позиција k_α и нека α' се добива од α со вадење на заградите на првата и k_α –та позиција. Тогаш функцијата $C[\alpha, i]$, каде i е бројот на парови загради е:

$$C[\alpha, i] = \begin{cases} C[\alpha', i-1] + \sum_{j=k_\alpha+1}^{i+1} A[i,j], & k_\alpha \leq i \\ 1, & k_\alpha = i+1 \end{cases}$$

2. Упатство: Ако првите n броеви ги гледаме како отворени загради, а вторите n броеви како затворени загради, задачата се сведува на правилно поставени загради.

a. Се генерира стринг од правилно поставени загради кој одговара на дадената позиција и тој се претвара во бараната низа.

b. Низата се претвара во правилно поставени загради и потоа за тој стринг се определува на која позиција се наоѓа.

3. Треба да се забележи дека пермутациите одговараат на правилно подредени загради, со тоа што секое прво појавување на еден број е отворена заграда, а секое второ појавување на некој број е затворена заграда. Подредувањата одговараат на тоа колку порано се отворената заграда се затвара, толку понапред е таа пермутација.

Шеста глава

6.1

1. Одговор: Ако некој чувар го ставиме на позиција a, тогаш тој може да ги чува сликите од во интервалот од $[x-a, x+a]$.

   a. Оптималното својство гласи: Ако во оптималното решение некој чувар е на позиција a, тогаш останатите чувари треба оптимално да се наредат да ги чуваат сликите во интервалите $[x_1, x-a]$ и $[x+a, x_n]$.

   b. Алчното својство гласи: Во оптималното решение последниот чувар секогаш може да се стави на позиција $x_n - x$.

      Ако во оптималното решение има m чувари, тогаш последниот чувар ја чува последната слика. Тој мора да е на позиција поголема од $x_n - x$, но бидејќи останатите чувари ги чуваат сликите кои не ги чува тој, тогаш него без проблем можеме да го поместиме на позиција $x_n - x$

   c. Последниот чувар ќе го позиционираме на позиција $x_n - x$ и од низата ќе ги извадиме сите слики кои тој може да ги чува. Процедурата ќе ја продолжиме за останатите слики.

   d. $O(n)$.

2. Одговор:

   a. Ако во оптималното решение полицаецот на позиција i го фаќа крадецот на позиција j, тогаш останатите полицајци треба оптимално да ги фатат останатите крадци.

   b. Алчното својство гласи: Во оптималното решение секогаш последниот полицаец го фаќа последниот крадец, ако тоа е можно.

c. Проверуваме дали последниот полицаец може да го фати последниот крадец. Ако тоа е можно, тогаш тие двајца ги вадиме од низата, забележуваме дека имаме еден фатен крадец, и постапката ја повторуваме на останатите елементи. Ако не е можно, и последен во низата е полицаец, тогаш тој полицаец не може да фати крадец, а ако последен во низата е крадец, тогаш тој крадец не може да биде фатен.

d. $O(n)$

6.2

1. Одговор:

   a. Пример каде нема да работи дадениот алчен пристап:

   $$\begin{matrix} 2 & 1 & 3 \\ 2 & 3 & 3 \\ 1 & 1 & 1 \end{matrix}$$

   b. Од првото поле одиме до двете соседни полиња и ја бележиме вкупната сума. Потоа продолжуваме од полето со помала вредност. Во секој чекор понатаму, ја наоѓаме сумата на соседите на полето до кое сме стигнале и во тој момент има најмала вредност, се додека прв пат не стигнеме до крајното поле.

   c. $O(n^2)$

2. Одговор: Не, на пример ако имаме низа од 4-ца во која секој сам купува за време 5, првиот и вториот споени купуваат за време 7, вториот и третиот за време 6, а последните двајца за време 7, тогаш со оваа стратегија прво ќе се спојат вториот и третиот за време 6 и останатите ќе бидат сами, што ќе фати време 16. Подобрата опција е спојување на првите двајца и вторите двајца за што е потребно време 14.

6.3

1. Одговор:

   a. Оптималното својство гласи: Ако во оптималното решение бродот i се наоѓа во интервалот $[a, b]$, тогаш останатите бродови треба оптимално да се наредат на алките кои се во интервалите $[0, a]$ и $[b, \infty]$.

      Нека во оптималното решение бродот i се наоѓа во интервалот $[a, b]$, но во некој од интервалите $[0, a]$ и $[b, \infty]$ бродовите не се оптимално наредени. Тогаш можеме оптимално да ги наредиме и ќе добиеме друга оптимална комбинација.

   b. Алчното својство гласи: Нека $[a, b]$ е непразен интервал и нека x_k е бродот со должина d_k кој треба да се закачи на алката на позиција $a \leq y_k \leq b$ таков што $y_k \leq a + d_k$ и таков што за сите бродови x_j со должина d_j кои треба да се закачат на алка на позиција y_j $a \leq y_j \leq a + d_j$ важи дека што $a + d_k = \min\{ a + d_j\}$, тогаш постои оптимално множество од наредени бродови во $[a, b]$ во кое е бродот x_k.

      Ако постои оптимално решение во кое не е овој брод, тогаш првиот брод во тоа оптимално решение може да се замени со овој брод.

   c. Прво ќе ги најдеме сите бродови со должина d_j на кои им е доделена алката на позиција y_j за која важи $y_j - d_j \leq y_1$ и во нашето множество ќе го ставиме бродот со најмала должина. Потоа од множеството бродови ќе ги извадиме сите кои треба да се закачат на алка во интервалот $[0, d_j]$ и множеството алки ќе ги извадиме сите кои се во интервалот $[0, d_j]$. Постапката понатаму ќе ја повторуваме со почеток во d_j.

2. Одговор: Во рекурзивната равенка наместо $1 + \max_{j \in S_i} A[j]$ ќе се замени $\max_{j \in S_i}\{y_j + A[j]\}$.

3. Упатство: Треба да се сортира низата по опаѓачки редослед во однос на заработката и по тој редослед ќе се проверува дали некоја активност можеме да ја ставиме во оптималната низа. Во дадено време кога временскиот термин кој ни е даден е $[a, \infty)$ за секоја активност која е на ред проверуваме дали може да се стави во низата, т.е. дали $a + 1 \leq f_i$. Ако е можно ќе ја ставиме таа активност во листата на активности, ако не ја игнорираме.

6.4

1. Одговор: Прво пресметајте ја фреквенцијата на знаците ако не е дадена. Потоа генерирајте го дрвото на Хафман. Пресметајте го бројот на битови како сума од производи на фреквенцијата на знаци и број на битови потребни за претставување на тие знаци.
2. Одговор: Откога ќе се пресмета бројот на битови потребни со хофманов код, тој број ќе се одземе од $m2^{\lceil \ln n \rceil}$, каде m е должината на пораката, а n е бројот на карактери.
3. Одговор: $n - 1$.
4. Одговор: Нека е даден текстот σ и функцијата со шифри $M[a]$ за секој карактер a. Псевдокодот ќе биде следниов:

```
1  Внеси го текстот σ и функцијата со кодови M[a]
2  за секој карактер од σ, a, печати M[a].
```

5. Одговор: Нека е даден текстот σ и дрвото T, така што секое лево дете на a е обележано со 0 и е дадено со функцијата $levo(a)$, а секое десно дете на a е обележано со 1 и е дадено со функцијата $desno(a)$. Секој лист на T е обележан со $sifra(a)$. Псевдокодот ќе биде следниов:

```
1  Внеси го текстот σ и хафмановото дрво T;
2  додека има карактери во σ, прави
3        s = koren(T);
4        додека s не е лист прави
```

```
5       {
6         земи нов карактер од σ, a
7         ако a = 0 тогаш s = levo(s) инаку s = desno(s);
8       }
9       испечати го sifra(s).
```

7. 1

1. Одговор: Сложеноста на алгоритамот е $O(|V|^2)$, која зависи од бројот на полиња во матрицата на соседство кои треба да се пополнат. Псевдокодот е:

```
1  за i = 1 до n прави внеси lista[i]
2  за i = 1 до n прави
3     за j = 1 до n прави M[i,j] =0
4  за i = 1 до n прави
5     {
6       земи елемент j од lista[i]
7       M[i,j] = 1;
8  }
```

2. Одговор: Сложеноста на алгоритамот е $O(|V|^2)$, која зависи од бројот на полиња во матрицата на соседство кои треба да се прочитаат. Псевдокодот е:

```
1  за i = 1 до n прави
2     за j = 1 до n прави внеси M[i,j]
3  за i = 1 до n прави
4     {
5       постави lista[i] на празно;
6       за j = 1 до n прави
7           ако M[i,j] = 1 тогаш додај j во lista[i];
8  }
```

3. Одговор: Сложеноста на алгоритамот е $O(|V| + |E|)$, која зависи од бројот на елементи во листата на соседство кои треба да се пополнат. Псевдокодот е:

```
1  за k = 1 до |V| прави внеси rebra[k]
2  за i = 1 до n прави иницирај lista[i] на празно;
3  за k = 1 до |V| прави
4      ако rebra[k] = (i, j) тогаш додај j во lista[i].
```

4. Одговор: Сложеноста на алгоритамот е $O(|V|^2)$, која зависи од бројот на полиња во матрицата на соседство кои треба да се пополнат.
5. Одговор: Функцијата родител ќе ја обележиме со $\pi[v]$. Псевдокодот на алгоритемот е следен:

```
1  за секое теме од V прави π[v] = nill
2  за секое ребро од (u, v) прави π[v] = u
```

   Сложеноста на алгоритамот е $O(|V|)$, која зависи од бројот на темиња, затоа што секое теме има по најмногу еден родител.
6. Одговор: $O(|V| + |E|)$.
7. Одговор: $O(|V| + |E|)$.
8. Одговор: $O(|V|^2)$.
9. Одговор: $O(|V|^2)$.

7.2

1. Одговор: Да, ако тој циклус е со тежина помала од 0.
2. Одговор: Да.
3. Одговор: $-\infty$. Циклусот се повеќе и повеќе ќе го намалува патот, па неговата тежина ќе тежи кон бескрај во минус.
4. Одговор: Ќе ја прошириме со $w(u, v) = \infty$ кога од u до v не постои ребро.
5. Одговор: Бидејќи $\delta(i, k)$е најкраткиот пат од темето i до темето k, тој не може да има поголема тежина од патот од темето i до темето j, плус должината на реброто (j, k).

7.3

1. Одговор: Нека ребрата ги гледаме по следниов редослед: $(1,2)$, $(1,3)$, $(2,3)$, $(2,4)$, $(3,4)$, $(5,4)$ и $(5,3)$.

   Итерација 0: $d[1] = d[2] = d[3] = d[4] = \infty$, $d[5] = 0$.

   Итерација 1: $d[1] = d[2] = \infty$, $d[3] = 2$, $d[4] = 3$, $d[5] = 0$.

   Итерација 2: $d[1] = d[2] = \infty$, $d[3] = 2$, $d[4] = 3$ $d[5] = 0$.

   Во последната итерација нема промена и алгоритамот застанува овде.

2. Упатство: Во секоја итерација ќе се провери дали има промена на функцијата која ги зачувува најкратките патишта, и ако нема, тогаш можеме да престанеме со итерирање. Најлош случај би било да се итерира толку колку што е должината на најдолгиот циклус. Понатаму нема потреба да се проверува со уште една итерација, затоа што ако имало промена во функцијата δ, таа ќе се случела и во последната итрација која сме ја направиле.

3. Одговор: Псевдокодот е следниов:

   1 Изворот го обележуваме со далечина 0, а сите други темиња со далечина 1

   2 за секое теме i различно од изворот, следејќи го тополошкиот редослед пресметуваме: $d[i] = \min\{d[j] + w[i,j]\}$

4. Упатство: За секоја променлива се става по едно теме во графот. За секое ограничување $x_i - x_j \leq a$ се става ребро (j,i) со тежина a. Се додава ново теме, извор, и од него ребра со тежина 0 до сите останати ребра. Од новододадениот извор се пушта алгоритамот на Белман- Форд. Ако алгоритамот покаже дека во така добиениот граф има негативен циклус, тогаш системот од ограничувања нема решение, во спротивно најкратките патишта до секое теме се едно решение.

7.4

1. Одговор: $G(\{s,a,b\},\{(s,a),(s,b),(a,b)\}.w(s,a)=2,w(s,b)=1,w(a,b)=-1.$

2. Одговор: Решението е дадено со следниов псевдокод, кој е ист како и псевдокодот за алгоритамот на Дикстра, со некои промени: Вредностите на функцијата се иницијализираат на 0, наместо на бескрај. Се користи максимален куп наместо минимален, односно секогаш се ажурира вредноста на темето со максимална вредност наместо минимална. И на крај, ажурирањето не се прави со собирање, туку со множење, т.е. наместо $A[v]=A[u]+w(u,v)$, $l[v]=A[u]\cdot w(u,v)$:

```
1  Внеси ги V и E;
2  за секое u ∈ E прави {
3     l[u] = A[u] = 0;
4     π[u] = NIL;}
5  l[s] = 1; S = ∅; Q = V;
6  додека Q ≠ ∅ прави {
7        u = izvadi_max(Q);
8        S = S ∪ {u}
9        A[u] = l[u];
10       за секое v од листата на соседи на u, v ∈ Adj[u]
11          ако l[v] < A[u] · w(u,v) тогаш {
12             l[v] = A[u] · w(u,v).
13             π[v] = u.}}
```

3. Одговор: Ќе се направи модификација на графот на тој начин што на секое ребро кое излегува од даденото теме ќе се стави тежината на темето.

4. Одговор: Без разлика дали се работи со низа или листа, минимумот ќе се пребарува во линеарно време. Секогаш може ажурирањето да се направи во константно време, па сложеноста на алгоритамот ќе биде: $O(|E||V|)$.

7.5

1. Упатство: За да се пресмета кое е претпоследното теме на патот од темето u до темето v, треба а се пресмета $l[u,v] - w[u,v]$. Темето кое е претпоследно е она теме a за кое оваа вредност е еднаква на $l[u,a]$.

2. Одговор: ќе се добијат негативни броеви на дијагоналата на D.

3. Одговор: Треба да се изврши алгоритамот а Флојд – Варшал на граф на кој секое ребро има тежина 1. Со тоа ќе се добие матрица која има вредности позитивни реални броеви или бескрај. Матрицата за транзитивниот затварач ја добиваме така што во оваа матрица секој реален број го заменуваме со 1, а секоја вредност бескрај со 0.

4. Упатство: Најкратките патишта на иницијалното дрво ќе се пресметаат со алгоритамот на Дикстра. Потоа, со еден чекор од алгоритамот на Флојд-Варшал ќе се најдат најкратките патишта од темето u до коренот, а потоа од коренот до темето v.

5. Упатство: Подреди ги состојбите на конечниот автомат при што стартната состојба ќе ја обележиш со 1. Нека $L(i,j,k)$ е множеството од зборови кои може да се генерираат со конечниот автомат со прошетка во автоматот од состојбата i до состојбата j при што ќе се поминува само низ состојби обележани со стриктно помала вредност од k. Јазикот кој се прифаќа со овој автомат е точно $\bigcup_{i \in F} L(1,j,|Q|+1)$.

6. Одговор:

   a. За секој пат $p = v_0, \ldots, v_k$ имаме дека

$$w'(p) = \sum_{i=1}^{k} w'(v_{i-1}, v_i)$$
$$= \sum_{i=1}^{k} \big(w'(v_{i-1}, v_i) - h(v_{i-1}) + h(v_i)\big) =$$
$$= \sum_{i=1}^{k} \big(w'(v_{i-1}, v_i)\big) + h(v_0) - h(v_k)$$
$$= w(p) + h(v_0) - h(v_k)$$

Оттука следува дека ако некој пат од v_0 до v_k е пократок од друг со користење на тежинската функција w, тој е пократок и користејќи ја тежинската функција w'. Оттука, $w(p) = A(v_0, v_k)$ акко $w'(p) = A'(v_0, v_k)$.

b. Ако патот p е циклус, од претходното равенство имаме

$$w'(p) = w(p) + h(v_0) - h(v_0),$$

од каде следува дека во G постои циклус со негативна тежина во однос на функцијата w акко G има циклус со негативна тежина во однос на функцијата на w'.

c. Доколку сите тежини на ребрата во графот $G = (V, E)$ се ненегативни, најкраткиот пат помеѓу секој пар темиња се наоѓаат со извршување на алгоритамот на Дикстра еднаш од секое теме. Ако G има негативни ребра, но нема циклус со негативна тежина, прво се пресметуваат нови тежини на ребрата w', што овозможува повторно од секое теме да се пушти алгоритамот на Дикстра.

d. $O(|V|2ln|V|)$.

8. 1

1. Одговор: Псевдокодот е следниов:

```
1  Внеси ги елементите од матрицата x_{i,j}
2  max = ∞;
3  генерирај ги сите пермутации до n!
4  за секоја пермутација a_1, a_2, ... , a_n прави
5    {
6      b = 0;
7      за i = 1 до n прави b = b + x_{i,a_i};
8      ако b > max тогаш max = b;
9     }
10 печати max.
```

2. Одговор: $\Theta(n \cdot n!)$. Генерирањето на пермутациите зазема време $\Theta(n!)$, Бројот на сите пермутации без повторување е $n!$ и за секоја пермутација се повикува циклус n пати.

3. Одговор:

   a. Ако m е бит низа со големина n, а M е реален број, тогаш рекурзивната релација е следнава:

$$A[m, M] = \begin{cases} \min\limits_{m,m\&2^i>0} \{A[m\&\overline{2^i}, M - y_{|m|,i}] + x_{|m|,i}\}, & M \geq 0 \\ \infty, & M < 0 \\ 0 & m = 0 \end{cases},$$

   каде $|m|$ е бројот на единици во маската m.

   b. Во следново решение се избегнува да се меморираат вредностите кога $M < 0$. Псевдокодот е следниов:

```
1  Внеси n и матриците X и Y;
```

```
2   за M = 0 до N прави
3     {
4       A[0, M] = 0;
5       за m = 1 до 2^n − 1 прави A[m, M] = ∞;
6     }
7     за mask од 0 до 2^n − 1 прави
8       {
9         i = broj_elem[mask] + 1;
10        за j од 1 до n прави
11        ако mask & 2^(j−1) = 0 прави
12            {
13              за M = 0 до N прави и M − y_ij > 0 прави
14                      mask1 = mask|2^(j−1);
15              A[mask1, M] = min{A[mask1, M], A[mask, M −
    y_ij] + x_ij};
16            }
17        }
18  врати A[2^n − 1, N].
```

c. За маска со k нули се можни повикувања за $k!$ вредности за различни суми од елементи од матрицата Y.

4. Одговор:

a. Нека i е редниот број на множеството. Секое множество нека е претставено со бит стринг со должина n. Нека a е број од 1 до 2^n. Решението се добива за $A[m, 0]$, а рекурзивната релација над која се базира решението е:

$$A[i, a] = \begin{cases} 0, & a = 2^n \\ \min\{A[i-1, a], A[i-1, a \& x_i] + 1, & a \neq 2^n \end{cases}.$$

b. Сложеноста е $O(2^m)$, ако работиме со мемоизација и не ги пресметуваме сите вредности за a. Ако иницираме $f(i, a)$, за секое a, тогаш сложеноста би била $\Theta(2^n m)$, што е најлоша можна реализација.

5. Одговор:

   a. Нека i е редниот број на множеството. Секое множество нека е претставено со бит стринг со должина n. Нека a е број од 1 до 2^n. Решението се добива за $A[m, 0]$, а рекурзивната релација е следнава:

$$A[i,a] = \begin{cases} 0, & i = 0 \\ A[i-1,a], & a\&x_i \neq 0 \\ \max\{A[i-1,a], A[i-1,a|x_i] + 1, & a\&x_i = 0 \text{ и } i \neq 0 \end{cases}$$

   b. Сложеноста е $O(2^m)$, ако работиме со мемоизација и не ги пресметуваме сите вредности за a. Ако иницираме $A[i,a]$, за секое a, тогаш сложеноста би била $\Theta(2^n m)$, што е најлоша можна реализација.

6. Одговор: Треба да се направи $n \times n$ матрица во која на полето (i,j) ќе се стави растојанието од i-тиот војник до j-тата база. Потоа проблемот се сведува на избирање по еден број од секоја редица и секоја колона, така да сумата на броевите биде минимална.

8. 2

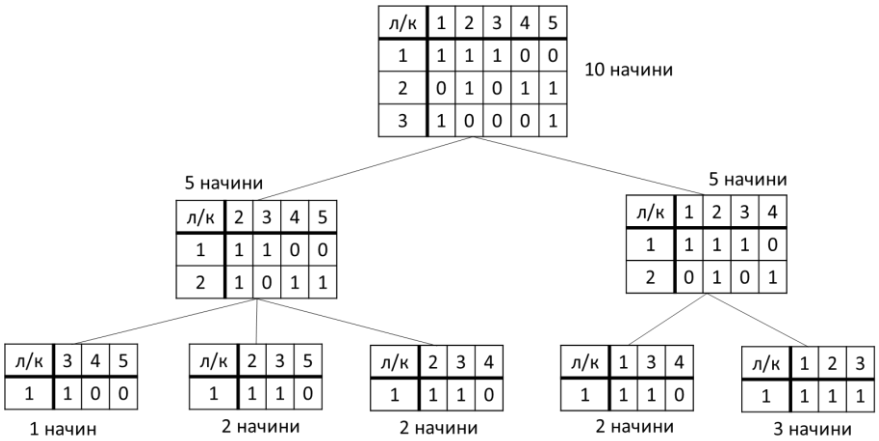
1. Одговор:

   a. Илустрацијата е дадена на Слика 9. 1

   b. Ако $mask$ е маска која е одговара на множеството различни видови капи, а i е бројот на луѓе кои одат на забавата, тогаш рекурзивната релација е следнава:

$$A[mask, j] = \begin{cases} \sum_{\substack{i, mask \wedge `2^i \neq 0 \\ \text{и } i \in Covek[j]}} A[mask|\overline{2^i}, j-1], & |mask| > 0 \\ 1, & |mask| = 0 \end{cases},$$

каде $|mask|$ е бројот на единици во маската, $Covek[j]$ е множеството на луѓе кои го поседуваат j-тиот вид на капа, $\overline{2^i}$ е бит стринг со должина како и $mask$ со 0-ла само на позицијата i, а $\wedge$ е конјункција на бит стрингови, (т.е. покажува дали во маската $mask$ има 1-ца на i -та позиција.



Слика 9. 1. Илустрација на решение на проблемот со капи со рекурзија по различните капи, наместо по различните луѓе

c. Ако динамичкото програмирање се базира на множеството луѓе кои присуствуваат на забавата, наместо на множеството на видови капи, тогаш имаме маска со поголема должина. За сите тие маски ќе мораме да иницираме почетни вредности, па решението наместо да има сложеност $O(2^n n)$ ќе има сложеност $O(2^m m)$, каде m е бројот на различни видови капи, што е многу поголема сложеност, затоа што бројот на видови капи е поголем од бројот на луѓе.

2. Идејата на решението е иста како и на проблемот разгледуван во поглавјето 8.1, само матрицата ќе го содржи бројот на капи од секој тип кој го поседува одреден човек наместо 1-ци и 0-ли.

   a. Модифицираниот псевдокод се базира на псевдокодот за оригиналниот проблем кој го решававме:

```
1  внеси го бројот на видови капи n и бројот на луѓе m;
2  за j од 1 до n прави
```

```
3    {
4      внеси ја матрицата Kapi[i,j];
5      за i од 1 до 2^m − 1 прави B[mask,i] = – 1;
6      B[0,j] = 1;
7    }
8  broj_elem[2^n − 1] = n;
9  врати A[2^n − 1, m];
10 функција A[mask,j]
11  {
12    ако B[mask,j] ≠ – 1 врати B[mask,j] инаку
13      ако broj_el[mask] > n тогаш B[mask,n] = 0 инаку
14        {
15          B[mask,j] = A[mask,j – 1]
16          за i = 0 до j прави
17          ако mask & 2^i > 0 тогаш
18              {
19                mask1 = mask & ! (2^i);
20                broj_el[mask1] = broj_el[mask] – 1;
21                 B[mask,j] = + Kapi[i,j]·A[mask1,j – 1]
22              }
23          }
24   врати B[mask,j];
25  }
```

b. Нема никаква промена во временската сложеност.

c. Се очекува да се добијат многу поголеми броеви како решение отколку за проблемот со една капа од секој вид по човек. Според тоа, може да се случи бројот кој треба да се добие да не може да го собере во мемориската локација која сме ја резервирале за неа. Оттука многу често во задачи не се бара да се отпечати самиот број на начини на распределување, туку тој број да се отпечати по некој модул.

3. Одговор: Нека различните карактери ги обележиме со броевите од 1 до k, а различните коцки со броевите од 1 до n. Треба да се направи матрица X во која x_{ij} е бројот на страни i на кои е напишан карактерот j. Алгоритамот за пресметување на бројот на начини на испишување на овој стринг е ист со алгоритамот за броење на капи, кога еден човек поседува повеќе капи од еден вид, проблемот од претходната задача, со тоа што овде карактерите на стрингот одговараат на луѓето, затоа што секој карактер мора да се јави во стрингот испишан од коцките, исто како што секој човек мора да отиде на забавата, а различните коцки одговараат на видот на капите. Ако има исти стрингови нема да има никаква разлика, затоа што коцките се различни.

8. 3

1. Одговор:

   a. Прво треба да се искористи некој алгоритам за наоѓање на најкраток пат од секое до секое теме, и бидејќи графот е со ненегативни ребра, најдобро е да се искористи алгоритамот на Дикстра од секое теме. Матрицата од најкратки патишта која при тоа ќе се добие е графот од најкратки патишта над кој треба да се пушти алгоритамот за оптимален хамилтонов циклус во комплетен граф.

   b. Графот мора да биде сврзан, бидејќи во спротивно во матрицата од најкратки патишта од секое до секое теме, меѓу некои од темињата ќе се добие ∞.

   c. Не мора.

   d. Решение ќе има за секој сврзан граф, без разлика дали во него има хамилтонов циклус.

2. Помош:

a. За секое теме, треба да се најде најоптималниот пат со почеток од тоа теме, слично како за хамилтонов циклус само без пресметките кои го комплетираат циклусот.

b. Сложеноста ќе биде потполно иста. Бројот на чекори кои се користат да се определи од кое теме ќе започне Хамилтоновиот пат е еднаков на бројот на чекори кои се користат да се пресмета кое е последното теме на Хамилтоновиот циклус кој започнува од темето обележано со 1, или некое друго произволно теме.

3. Одговор: Да, ќе се работи на иницијалната матрица. Ако нема хамилтонов циклус ќе отпечати бескрај.

4. Одговор: Ќе направиме граф со тежинска функција 1 ако постои ребро помеѓу двете темиња и 0, ако не постои. Во општ случај, посебно ако не постои хамилтонов циклус нема да се очекува никакво убрзување.

5. Броевите можеме да ги гледаме како темиња од комплетен граф, така што реброто меѓу темињата кои одговараат на броевите a и b има тежина $a \oplus b$. Врз овој граф, со овие тежини на ребрата треба да се најде најкраток Хамилтонов пат.

Референци

[1] K. H. Rosen, Discrete mathematics and its applications, WCB/McGraw-Hill, 1999.

[2] Б. Јанева, Вовед во теоријата на множествата и математичката логика, Скопје: Природно математички факултет, 1996.

[3] D. E. Knuth, The Art of Computer Programming (TAOCP), Addison-Wesley, 1997.

[4] R. Raz, " On the complexity of matrix product," in *Proceedings of the thirty-fourth annual ACM symposium on Theory of computing*, 2002.

[5] L. E. P. T. Baum, "Statistical Inference for Probabilistic Functions of Finite State Markov Chains," *The Annals of Mathematical Statistics,* vol. 37, no. 6, p. 1554–1563, 1966.

[6] L. E. Baum, T. Petrie, G. Soules and N. Weiss, "A Maximization Technique Occurring in the Statistical Analysis of Probabilistic Functions of Markov Chains," *The Annals of Mathematical Statistics.,* vol. 41, no. 1, p. 164–171, 1970.

[7] I. R. Sipos, A. Ceffer and J. Levendovszky, "Parallel Optimization of Sparse Portfolios with AR-HMMs," *Computational Economics,* vol. 49 , no. 4, p. 563–578, 2016.

[8] P. Domingos, The Master Algorithm: How the Quest for the Ultimate Learning Machine Will Remake Our World, Basic Books, 2015.

[9] H. Sakoe and S. Chiba, ""Dynamic programming algorithm optimization for spoken word recognition"," *IEEE Transactions on*

Acoustics, Speech, and Signal Processing. , vol. 26, no. 1, pp. 43-49, 1978.

[10] T. Davis, "Catalan Numbers," 2016.

[11] A. d. Segner, " Enumeratio modorum, quibus figurae planae rectilineae per diagonales dividuntur in triangula.," *Novi commentarii academiae scientiarum Petropolitanae,* vol. 7, no. 1758/59, pp. 203-201.

[12] N. Dershowitz and S. Zaks, "Enumerations of ordered trees," *Discrete Mathematics,* vol. 31, p. 9–28, 1980.

[13] D. E. Knuth, " Optimum binary search trees," *Acta Informatica,* p. 14–25, 1971.

[14] K. Mehlhorn, "Nearly optimal binary search trees," *Acta Informatica,* vol. 5, no. 4, p. 287–295, 1975.

[15] T. H. L. C. E. Cormen, R. Rivest and C. Stein, Introduction to algorithms (Third ed.), MIT Press, 2017.

[16] "https://code.google.com/codejam/contest/4214486/dashboard#s=p3," [Online].

[17] "https://mendo.mk/algoritmi/," [Online].

[18] A. Shimbel, "Structure in communication net," in *Proceedings of the Symposium on Information Networks.* , New York: Polytechnic Press of the Polytechnic Institute of Brooklyn, 1955.

[19] R. Bellman, "On a routing problem," *Quarterly of Applied Mathematics,* vol. 16, p. 87–90, 1958.

[20] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik,* vol. 1, p. 269–271, 1959.

[21] "https://www.hackerearth.com/practice/algorithms/dynamic-programming/bit-masking/tutorial/," [Online].

[22] "www.geeksforgeeks.org," [Online].