KÁTAI ZOLTÁN

ALGORITMUSOK FELÜLNÉZETBŐL

SAPIENTIA ERDÉLYI MAGYAR TUDOMÁNYEGYETEM
MŰSZAKI ÉS HUMÁNTUDOMÁNYOK KAR
MATEMATIKA–INFORMATIKA TANSZÉK

A kiadvány megjelenését a Sapientia Alapítvány támogatta.

KÁTAI ZOLTÁN

ALGORITMUSOK FELÜLNÉZETBŐL

Scientia Kiadó
Kolozsvár·2007

Lektor:
Ionescu Klára (Kolozsvár)

Sorozatborító:
Miklósi Dénes

Descrierea CIP a Bibliotecii Naționale a României
KÁTAI ZOLTÁN
Algoritmusok felülnézetből / Kátai Zoltán. – Cluj-Napoca: Scientia, 2007.
Bibliogr.
ISBN 978-973-7953-74-2

TARTALOM

CONTENTS

CUPRINS

BEVEZETÉS

Comenius, akit a modern oktatás létrehozójának tartanak, az alábbi kijelentést tette a tanítási módszerekre vonatkozóan: „Tanítani szinte nem is jelent mást, mint megmutatni, miben különböznek egymástól a dolgok a különböző céljukat, megjelenési formájukat és eredetüket illetően... Ezért aki jól megkülönbözteti egymástól a dolgokat, az jól is tanít." Ez a könyv elsősorban erre a didaktikai alapelvre épül. Egy olyan tanítási, illetve tanulási módszert ajánl, amely segít a tanulóknak úgymond felülnézetből látni a megvizsgált öt programozási módszert: *mohó* (*greedy*), *visszalépéses keresés* (*backtracking*), *oszd meg és uralkodj* (*divide et impera*), *dinamikus programozás, branch and bound*. Tehát nemcsak az a célunk, hogy bemutassuk e módszereket, hanem az is, hogy olyan nézőpontba juttassuk az olvasót, amelyből feltárulnak előtte a technikák közötti elvi, alapvető, sőt árnyalatbeli különbségek, illetve hasonlóságok. A comeniusi alapelvvel összhangban ez nélkülözhetetlen, ha uralni szeretnénk a programozás e területét.

A következőkben erre a megközelítési módra mint *felülnézet-módszer*re hivatkozunk.

A felülnézet-módszer általános leírása

Mit jelent „felülről látni" valamit?

Képzeljük el a következő helyzeteket: a rendőrségen, egy bűneset kapcsán, a különböző forrásokból érkező bizonyítékokat feltűzik egy hirdetőtáblára. Miért? A polgármesteri hivatal városrendezésért felelős szakosztálya elkészíti a város makettjét és azt körbeállják. Miért? Azért, hogy átfogó képet kapjanak az „egész"-ről, valamint, hogy jobban érzékelhetőek legyenek az egyes „részek" közötti hasonlóságok, különbségek, illetve kapcsolatok.

A két esetben különböző módon alakították ki az illetékesek a felülnézetet. A rendőröknek szükségük volt egy olyan „platformra" (a tábla), amelyen elhelyezve a bizonyítékokat, „egymás mellett" láthatták őket. A műépítészek kicsinyítést és absztrakciót használtak

a makett elkészítésekor. Az absztrakció azért fontos, mert elvonatkoztat attól, ami a tanulmányozás szempontjából lényegtelen.

A két módszer ötvözéséből látható, hogy a felülnézet kialakításához szükséges lehet egy úgynevezett „absztrakt platform", amelyen az elemzett entitások úgy helyezhetők egymás mellé, hogy szembetűnővé váljanak a vizsgálat szempontjából lényeges tulajdonságok és kapcsolatok.

Azt, hogy ebben a könyvben mit fogunk felülnézeten érteni, a következőképpen lehetne összefoglalni:

1. „Egymás mellett" látjuk a vizsgálat tárgyát képező entitásokat.
2. Csak az látszik, ami a vizsgálat szempontjából lényeges.
3. Nyilvánvalóak a hasonlóságok és a különbségek, szembetűnők a kapcsolatok.

A módszer célja olyan helyzetbe juttatni a tanulót, hogy ilyen rálátása legyen a tanulmányozás alatt álló anyagra. Természetesen különböző felülnézetek alakíthatók ki attól függően, miként valósítjuk meg az említett lépéseket. Ez kívánatos is, hiszen minden újabb felülnézet egy másik szemszöget jelent. A módszer egyik erősségének számít az is, hogy segít a diákoknak a vizsgált entitások egymáshoz viszonyított „értékét" is felmérni. Például az algoritmustervezési stratégiák tanulmányozása esetén nyilvánvalóvá válnak az egyes technikák erős, illetve gyenge pontjai, valamint az, hogy adott helyzetben melyiknek az alkalmazása a legcélszerűbb és miért.

Az alábbi egyszerű kísérlettel jól ellenőrizhető a módszer hatékonyságát igazoló alapelv. Mutassunk fel egy papírlapot, és kérjük meg a tanulókat, hogy nevezzék meg minél több tulajdonságát. Ezután – egy másik osztályban – ismételjük meg a kísérletet, de úgy, hogy a papírlappal együtt egy másik alakzatot is felmutatunk (például, ami fából készült, nem síkidom, nem egyszínű stb.).

Mit fogunk tapasztalni? A második osztályban a papírlapnak számottevően több tulajdonsága fogalmazódik meg a tanulókban. Például nem biztos, hogy az első osztályban felfigyelnek arra, hogy az ív egyszínű, síkidom, összegyűrhető, nyersanyaga fa stb. Ez az egyszerű kísérlet egy régóta ismert igazságot emel ki: *az ellentétek felhívják a figyelmet mind magukra, mind a hasonlóságokra.*

Mi volt a tanár szerepe ebben a kísérletben? Az, hogy a diák elméjében kialakítsa a felülnézetet. Ez azt feltételezte, hogy úgy válassza meg a tárgyat, amit a ív mellé helyezett, hogy szembeötlővé váljanak azok a szempontok, amelyek alapján szeretné, hogy a tanulók az összehasonlítást elvégezzék.

Felülnézetek az informatikaoktatásban

Hogyan lehet felülnézeteket kialakítani informatikaórán? A legtöbb tanár megteszi ezt – még ha nem is így nevezi –, amikor a rendezéseket tanítja. A fejezet végén veszünk egy konkrét számsorozatot, amelyen elmímeljük, vagy elmímeltetjük a tanulókkal, az összes megtanított rendezési algoritmust, és felhívjuk a figyelmet a hasonlóságokra, valamint a különbségekre. Amikor így járunk el, felülnézetet alakítunk ki a diákokban, hiszen:

- „Egymás mellé" helyezzük a rendezési algoritmusokat azáltal, hogy ugyanazon a számsorozaton mímeljük el őket.
- Felhívjuk a diákok figyelmét arra, hogy egy adott felülnézetből mi lényeges és mi nem. Például az összehasonlításon alapuló rendezések estén arra összpontosíthatnak a diákok egy adott felülnézetből, hogy milyen stratégiák szerint történnek az elemek hasonlítgatásai.
- Segítünk a tanulóknak meglátni a hasonlóságokat és a különbségeket, az algoritmusok erősségeit és gyenge pontjait.

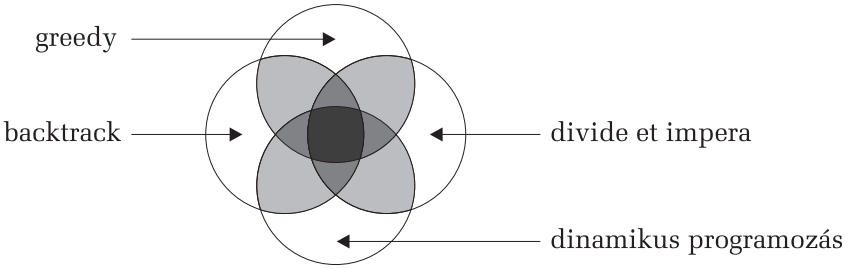
Kitűnő számítógépes szimulációk léteznek, amelyek megkönnyíthetik a felülnézet kialakítását a rendezési algoritmusokat illetően.

Algoritmustervezés felülnézetből

Hogyan alakíthatunk ki felülnézeteket az algoritmustervezési stratégiák tanításakor? A kihívás abban áll, hogy amíg a rendezési algoritmusok ugyanazt a feladatot oldják meg, addig minden egyes algoritmustervezési stratégiának megvan – többé-kevésbé – a saját felségterülete. Az 1. ábra ezt szemlélteti. Az egyes körök azon feladatok halmazát ábrázolják, amelyek megoldhatók az illető stratégiával, függetlenül attól, hogy optimális megoldást nyújtanak-e vagy sem, hatékony az algoritmus vagy nem.

Az, hogy a körök metszik egymást, azt szemlélteti, hogy léteznek olyan feladatok, amelyek több technikával is megoldhatók, sőt egyesek az összessel[1]. Ebből arra következtethetünk, hogy léteznie kell egy

1 A branch and bound technikát csak érintőlegesen tárgyaljuk az A* algoritmus révén.



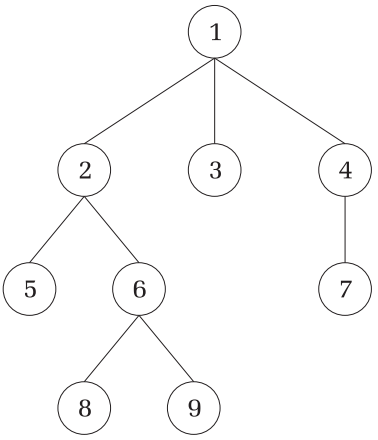
1. ábra.

„sík”-nak, amelyen az egyes technikák feladathalmazai alapvető hasonlóságokat mutatnak. Melyik ez a „sík”? Ahhoz, hogy megtaláljuk a választ erre a kérdésre, be kell vezetnünk a fastruktúra fogalmát.

Fastruktúrák

A fastruktúrának van egy kitüntetett csomópontja, amely a fa nulladik szintjén helyezkedik el, és amelyet gyökércsomópontnak nevezünk. Az első szinten találhatók a gyökér fiúcsomópontjai, a másodikon ezeknek a fiúcsomópontjai és így tovább. Azokat a csomópontokat, amelyekből nem ágazik le egyetlen fiúcsomópont sem, a fa leveleinek nevezzük. Tehát egy fastruktúra csomópontokból épül fel, és ezek között apa–fiú típusú kapcsolatok vannak. Minden csomópont egyetlen apacsomóponthoz kapcsolódik, amely a közvetlen felette lévő szinten található. Ez alól egyedül a gyökércsomópont kivétel. Ezzel szemben egy csomóponthoz több fiúcsomópont is kapcsolódhat, amelyek a közvetlen alatta lévő szinten helyezkednek el. A fiak között aszerint teszünk különbséget, hogy – balról jobbra számolva – hányadik fiuk az apjuknak (2. ábra).

Megfigyelhető, hogy bármely csomópont a leszármazottjaival együtt ugyancsak fa. Ez a következő rekurzív definíciót teszi lehetővé: Fastruktúrának nevezzük a csomópontoknak olyan halmazát és szerkezetét, amelyben létezik egy kitüntetett gyökércsomópont, és a többi csomópont olyan diszjunktív halmazokba van szétosztva, amelyek maguk is fák, és e részfák gyökerei, mint fiak, a fa gyökeréhez kapcsolódnak. Ha egy fastruktúrában mindenik csomópontnak legfentebb két fia van, akkor bináris fáról beszélünk. Ez esetben a fiúcsomópontokat jobb, illetve bal fiakként azonosítjuk.



2. ábra.

Egy „absztrakt platform”

A visszalépéses keresés és az oszd meg és uralkodj technikák általában rekurzívan közelítik meg a feladatot. A rekurzió gondolatmenete pedig a következő: Hogyan vezethető vissza a feladat hasonló, egyszerűbb részfeladatokra, majd ezek további hasonló még egyszerűbb részfeladatokra, egészen addig, míg triviális részfeladatokhoz nem jutunk? Ez a fajta lebontás azt feltételezi, hogy a feladatnak – felépítésében – fastruktúrája legyen. A gyökércsomópont nyilván magát a feladatot képviseli, az első szint csomópontjai azokat a részfeladatokat, amelyekre a feladat első lépésben lebontható, és így tovább. Végül a fa levelei fogják ábrázolni a lebontásból adódó triviális részfeladatokat.

A mohó, dinamikus programozás és branch and bound technikák egyik közös vonása, hogy általában olyan feladatok esetében alkalmazzuk őket, amelyek döntéssorozatként foghatók fel. Ez megint csak egy fastruktúrához vezet, amelyben a gyökér a feladat kezdeti állapotát jelképezi, az első szint csomópontjai azokat az állapotokat, amelyekbe a feladat az első döntés nyomán kerülhet, a második szinten lévők azokat, amelyek a második döntésből adódhatnak stb. Egy csomópontnak annyi fia lesz, ahány lehetőség közül történik a választás az illető döntés alkalmával.

Mindezt figyelembe véve elmondható, hogy az összes bemutatásra kerülő módszert elsősorban olyan feladatok esetében használjuk, amelyek valamilyen értelemben fastruktúrával rendelkeznek. A technikák

szemszögéből ez azt jelenti, hogy mindenik úgy fogja fel a feladatot, mint egy fát. Nos, ez a közös fastruktúra az a közös sík vagy absztrakt platform, amelyen a technikák egymás mellé helyezhetők, és amely a felülnézet kialakításához szükséges.

Eddig arra összpontosítottunk, hogy mi a közös az egyes technikákban, de a felülnézet azt is jelenti, hogy nyilvánvalóak a köztük lévő különbségek is. Amint látni fogjuk, az egyik ilyen különbség az, ahogyan bejárják a feladatnak megfelelő fát. Tekintettel azon olvasóinkra, akiknek még nincsenek gráfelméleti ismereteik, az alábbiakban bemutatjuk a fák azon bejárási módjait, amelyekre a későbbi fejezetekben gyakran hivatkozunk majd.

A fastruktúrák bejárása

Mit jelent bejárni egy fát? Alapvetően azt, hogy egy jól meghatározott sorrendben meglátogatjuk a csomópontjait úgy, hogy mindegyiket csak egyszer dolgozzuk fel. Általában kétféle bejárást különböztetünk meg: mélységi és szélességi bejárást.

Mélységi bejárások (DF – Depth First)

A mélységi bejárás a következő rekurzív algoritmust követi: a fa mélységi bejárását lebontjuk a gyökér fiúrészfáinak a mélységi bejárására. Ez azt jelenti, hogy indulunk a gyökérből, majd pedig balról jobbra haladva sorba bejárjuk a gyökér minden egyes fiúrészfájának összes csomópontját. A rekurzióból adódóan az egyes fiúrészfák bejárása hasonlóképpen megy végbe: indulunk az illető részfa gyökércsomópontjából, majd balról jobbra sorrendben bejárjuk ennek a fiúrészfáit. Ugyancsak a rekurzió következménye, hogy miután bejártuk egy csomópont összes fiúrészfáját, visszalépünk az apacsomópontjához, és folytatjuk a bejárást e csomópont következő fiúrészfájával (amennyiben ez létezik). A fentiekkel összhangban úgy foghatjuk fel a mélységi bejárást, mintha körbejárnánk a fa koronáját, szorosan követve annak vonalát. A mellékelt ábra azt is jól érzékelteti, hogy a bejárás során az egyes csomópontokat többször is érintjük. Egészen pontosan, ha egy csomópontnak f fia van, akkor $f+1$ alkalommal találkozunk vele a fa mélységi bejárása során. Attól függően, hogy melyik találkozáskor „látogatjuk meg”, vagyis mikor dolgozzuk fel a csomópontban tárolt információt, megkülönböztetünk preorder (első érintéskor látogatott), illetve postorder (utolsó érintéskor

látogatott) mélységi bejárásokat. Bináris fák esetén beszélhetünk inorder bejárásról is, amikor a két fiúrészfa bejárása közötti érintéskor látogatjuk meg a csomópontokat. A 3. ábra egy általános fa, a 4. egy bináris fa bejárásait szemlélteti.

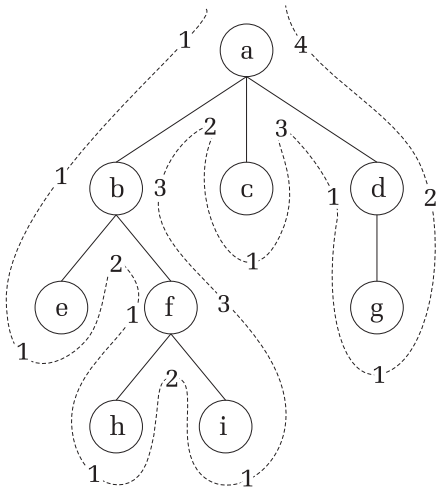
Megfigyelhető, hogy preorder bejárás esetén a fán lefelé haladva foglalkozunk a csomópontokkal, postorder bejáráskor viszont a visszautakon. Egy másik különbség, hogy preorder esetben először meglátogatjuk a csomópontot, és azután járjuk be ennek fiúrészfáit, postorder esetben pedig fordítva, először bejárjuk a csomópont fiúrészfáit, és csak azután látogatjuk meg magát a csomópontot.

Az alábbiakban megadjuk a bemutatott mélységi bejárásokat implementáló rekurzív eljárásokat (mindegyik az aktuális csomóponton dolgozik):

```
preorder(cs)
   meglátogat(cs)
   minden f_i fiára cs-nek végezd
      preorder(f_i)
   vége minden
vége preorder

postorder(cs)
   minden f_i fiára cs-nek végezd
      postorder(f_i)
   vége minden
   meglátogat(cs)
vége postorder

inorder(cs)
   inorder(f_bal)
   meglátogat(cs)
   inorder(f_jobb)
vége inorder
```

Mindhárom eljárást természetesen a gyökércsomópontra hívjuk meg:

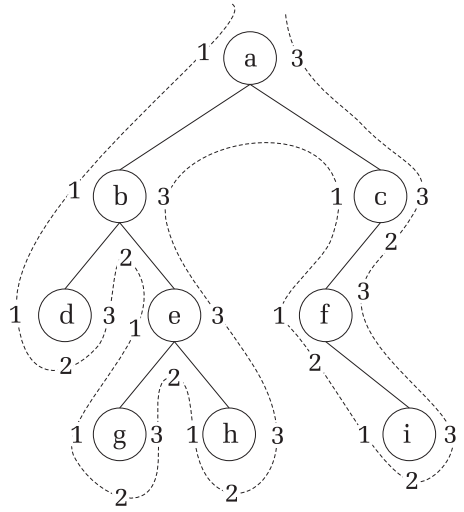
```
preorder(gyökér)
postorder(gyökér)
inorder(gyökér)
```



Preorder bejárás szerinti sorrend:
a, b, e, f, h, i, c, d, g

Postorder bejárás szerinti sorrend:
e, h, i, f, b, c, g, d, a

3. ábra.



Preorder bejárás szerinti sorrend:
a, b, d, e, g, h, c, f, i

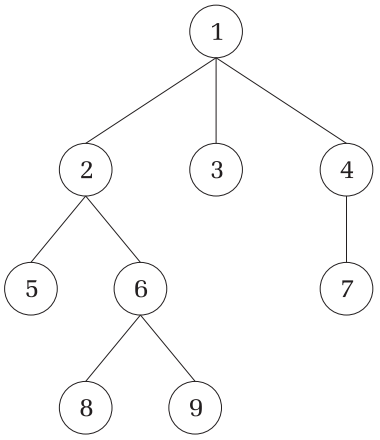
Inorder bejárás szerinti sorrend:
d, b, g, e, h, a, f, i, c

Postorder bejárás szerinti sorrend:
d, g, h, e, b, i, f, c, a

4. ábra.

Szélességi bejárás (BF – Breadth First)

A szélességi bejárás a következő gondolatmenetet követi: meglátogatjuk a gyökércsomópontot, majd ennek a fiúcsomópontjait (balról jobbra sorrendben), majd ezeknek a fiúcsomópontjait... Ahogy az 5. ábra is szemlélteti, szintről szintre haladunk a fában.



Szélességi bejárás szerinti sorrend:
1, 2, 3, 4, 5, 6, 7, 8, 9

5. ábra.

A szélességi bejárás implementálásához egy várakozási sor szerkezetet (Q) használunk (ellentétben a mélységi bejárással, amely – összhangban rekurzív voltával – veremszerkezetre épül). A várakozási sor struktúrára az jellemző, hogy betevéskor az elemek a sor végére kerülnek, a kivétel pedig a sor elejéről történik. Ebből adódik, hogy ugyanabban a sorrendben történik az elemek eltávolítása a sorból, mint amilyen sorrendben betettük őket (FIFO – **F**irst **I**n **F**irst **O**ut).

```
szélességi_bejárás(gyökér,Q)
   betesz(Q,gyökér)
   amíg (nem üres(Q)) végezd
      cs=kivesz(Q)
      meglátogat(cs)
      minden f_i fiára cs-nek végezd
         betesz(Q,f_i)
      vége minden
   vége amíg
vége szélességi_bejárás
```

Javasolt tanmenet

A felülnézet-módszernek az alábbi tanmenet szerinti alkalmazása feltételezi, hogy a tanulók rendelkeznek alapvető programozói készséggel:

0. A rekurzió átismétlése, a fastruktúrák és ezek bejárásainak bemutatása.
1. Egy bemutató feladaton keresztül, amely mindenik technikával megoldható, egy általános és átfogó képet nyújtunk a technikákról. Anélkül, hogy effektíve megoldanánk a feladatot, az osztállyal való beszélgetés formájában körvonalazzuk az egyes stratégiákat.
2. Bemutatjuk a backtracking technikát.
   Elmélyítjük a backtracking stratégiát.
   - Sajátos, backtrackinggel megoldható feladatok begyakorlása.
3. Bemutatjuk a divide et impera technikát.
   Elmélyítjük a divide et impera stratégiát.
   - Sajátos, divide et imperával megoldható feladatok begyakorlása.
4. *Divide et impera vagy backtracking*
   - Olyan feladatokat választunk, amelyek mindkét módszerrel megoldhatók.
5. Bemutatjuk a greedy-technikát (mohó algoritmust).
   Elmélyítjük a greedy-stratégiát.
   - Sajátos, greedy-módszerrel megoldható feladatok begyakorlása.
6. *Backtracking és greedy kéz a kézben*
   - Olyan feladatokat választunk, amelyek kiemelik, miként egészítheti ki a két módszer egymást.
7. Bemutatjuk a dinamikus programozás módszerét.
   Elmélyítjük a dinamikus programozás módszerét.
   - Sajátos, a dinamikus programozás módszerével megoldható feladatok begyakorlása.
8. *Divide et impera vagy dinamikus programozás*
   - Olyan feladatokat választunk, amelyek mindkét módszerrel megoldhatók.
   - A dinamikus programozás rekurzív változata.
9. *Greedy-módszer vagy dinamikus programozás*
   - Olyan feladatokat választunk, amelyek mindkét módszerrel megoldhatók.

10. *Beágyazzuk a branch and bound technikát az előbbi módszerek alkotta képbe.*
    - Áttekintjük a négy, már bemutatott technika fő jellegzetességeit, és rámutatunk, hogy milyen „űrt" tölt ki a branch and bound a kialakult képben.
    - Elmélyítjük a stratégiát specifikus, branch and bounddal megoldható feladatok által.
11. *Határátkelők a programozási technikák világában*
    - Áttekintjük felülnézetből a teljes anyag felépítését.
    - Néhány „határmenti algoritmus" megvizsgálásával teljesebbé tesszük a képet.

Az egyes technikákat bemutató órákon hangsúlyozni kell, mit jelent az illető stratégia szempontjából „fa"-ként felfogni egy feladatot.

A felülnézetórákra (dőlt betűvel jeleztük e programpontokat) olyan feladatokat választunk, amelyek „megoldhatók" mindkét összehasonlításra kerülő módszerrel. Ezek azok az órák, amelyeken kihangsúlyozásra kerülnek a stratégiák közti hasonlóságok, különbségek és kapcsolatok.

Mindehhez bőséges anyagot nyújt ez a könyv, amelyet az olvasó a kezében tart.

A könyvben használt pszeudókód nyelv leírása

Az algoritmusok leírására az alábbi pszeudókód nyelvet használjuk:

Operátorok

- aritmetikai operátorok: `+, -, * , / , %` (egész osztási maradék).
  Megjegyzés: Ha mindkét operandus egész szám, a / operátor egész osztást, különben valós osztást jelent.
- összetett operátorok: `+=, -=, *=, /=`
  `a+ = b` ⇔ `a = a + b`
- speciális operátorok eggyel való növelésre és csökkentésre: `++, --`
  `a++` ⇔ `a = a + 1`
  `a--` ⇔ `a = a - 1`
- abszolútérték-operátor (modulusz): `||`
  `|a|`

– egészrész operátorok: $[\,]$, $\lfloor\,\rfloor$, $\lceil\,\rceil$ (az első kettő le-, a harmadik felkerekít)
$$[a], \lfloor a \rfloor, \lceil a \rceil$$
– összehasonlítási operátorok: `<`, `<=`, `>`, `>=`, `==` (egyenlőségvizsgálat), $\neq$ (különbözőségvizsgálat)
– logikai operátorok: **`és`, `vagy`, `nem`**

Kommentek (megjegyzések)

Ha egy algoritmus valamely sorát dupla backslash jel (`//`) előzi meg, akkor megjegyzésnek tekintendő.

Műveletek

Értékadás:

```
<változó>=<kifejezés>
```

Elágazás:

```
ha <feltétel> akkor
   <műveletek1>
különben
   <műveletek2>
vége ha
```

Elöltesztelő ciklus:

```
amíg <feltétel> végezd
   <műveletek>
vége amíg
```

Hátultesztelő ciklus:

```
végezd
   <műveletek>
amíg <feltétel>
```

Megjegyzés: Mindkét `amíg` ciklusból akkor lépünk ki, ha a feltétel hamissá vált.

Ismert lépésszámú ciklus:

```
minden <változó>=<kezdőérték>,<végsőérték>,<lépés> végezd
   <műveletek>
vége minden
```

Megjegyzés: Ha a lépésszám hiányzik, implicit 1-nek tekintjük.

Beolvasási művelet:

be: <változó lista>

Kiírási művelet:

ki: <kifejezés lista>

Konstansok (példák)

13, -524 (egész)
-12.027, 0.22 (valós)
'A', 'c' , '1' , '!' (karakterek)
"alma", "123", "23 almafa" (karakterlánc)
IGAZ, HAMIS (logikai)

Adatszerkezetek

Egydimenziós tömb (vektor) (példák):
- a[] – egydimenziós tömb
- a[1..n] – egydimenziós tömb, amelynek elemei 1-től n-ig vannak indexelve
- a[i] – hivatkozás egy egydimenziós tömb i-edik elemére

Kétdimenziós tömb (példák):
- b[][] – kétdimenziós tömb
- b[1..n][1..m] – kétdimenziós tömb, amelynek sorai 1-től n-ig, oszlopai pedig 1-től m-ig vannak indexelve
- b[i][j] – hivatkozás egy kétdimenziós tömb i-edik sorának j-edik oszlopbeli elemére

Bejegyzés (rekord, struktúra) (példák):
- r.m – hivatkozás az r bejegyzés típusú változó m mezőjére
- a[i].x – az a bejegyzés típusú tömb i-edik elemének az x mezője

Mutató (pointer) (példák):
- p->m – hivatkozás a p pointer által megcímzett bejegyzés típusú változó m mezőjére

Eljárás

```
<eljárás neve> (<formális paraméter lista>)
   <műveletek>
vége <eljárás neve>
```

Megjegyzés: Egy eljárásnak lehet több visszatérési pontja is (tartalmazhat üres return utasítást).

Függvény

```
<függvény neve> (<formális paraméter lista>)
   <műveletek>
   return <eredmény>
vége <függvény neve>
```

Megjegyzés: Egy függvénynek lehet több visszatérési pontja is.

Paraméterátadás: A formális paraméterek listájában jelezni fogjuk, mely paraméterek egyszerű típusúak és melyek tömbök. Az érték szerinti és cím szerinti paraméterátadás között úgy teszünk különbséget, hogy az utóbbi esetében a formális paramétereket félkövér karakterekkel írjuk.

Példa:

```
eljárás(x[],y[][],a,b,c[])
   ...
vége eljárás
```

Megjegyzések:

1. `a` és `b` egyszerű típusú változók, `x` és `c` egydimenziós tömbök, `y` pedig kétdimenziós tömb. Az `x`, `y` és `b` paraméterek érték szerint, `a` és `c` cím szerint kerül átadásra.
2. Amikor tömböket adunk át, ha az algoritmus szempontjából nem lényeges, melyikfajta paraméterátadást használjuk, akkor az implementálásnál – memóriatakarékossági megfontolásból – célszerű a cím szerinti paraméterátadást használni. Egyes programozási nyelvekben (például a C nyelvben) a tömbök implicit cím szerint adódnak át.

1. FEJEZET

ÁLTALÁNOS KÉP AZ ÖT MÓDSZERRŐL

A bevezető fejezetben bemutatott felülnézet-módszer alkalmazásának első lépése a vizsgálat tárgyát képező entitások – jelen esetben a programozási módszerek – egymás mellé helyezése. A legegyszerűbben ezt úgy tudjuk megtenni, ha ugyanazt a feladatot oldjuk meg mind az öt módszerrel.

Például, sokatmondó felülnézet alakítható ki, ha feladatnak az alábbi optimalizálási[1] problémát választjuk, amelyet 1994-ben javasoltak megoldásra a Svédországban megrendezett Nemzetközi Informatika Olimpiászon.

Háromszög: Egy n soros négyzetes mátrix főátlóján és főátló alatti háromszögében természetes számok találhatók. Feltételezzük, hogy a mátrix egy a nevű kétdimenziós tömbben van tárolva. Határozzuk meg azt a „leghosszabb" utat, amely az $a[1][1]$ elemtől indul és az n-edik sorig vezet, figyelembe véve a következőket:

- egy úton az $a[i][j]$ elemet az $a[i+1][j]$ elem (függőlegesen le) vagy az $a[i+1][j+1]$ elem (átlósan jobbra le) követheti, ahol $1 \leqslant i < n$ és $1 \leqslant j < n$;
- egy út „hossza" alatt az út mentén található elemek összegét értjük.

Például, ha $n = 5$, az 1.1. ábrán látható mátrixban besatíroztuk a „leghosszabb" utat, amely a csúcsból az alapra vezet. Ennek az útnak a „hossza" 37.

A felülnézet kialakításának második lépése az absztrakció. Az előző fejezetben kifejtettük, hogy ez nem jelent egyebet, mint azonosítani a feladat fastruktúráját.

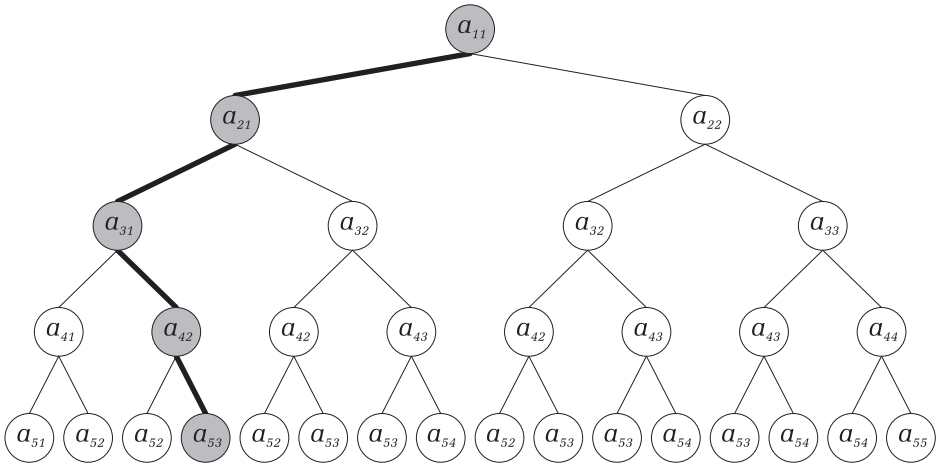
Elemezve a feladatot, észrevehetjük, hogy egy olyan optimalizálási problémáról van szó, amelyben az optimális megoldáshoz $n-1$ döntés nyomán juthatunk el, és mindenik döntésnél 2 választásunk van (melyik

1 A leginkább az optimalizálási feladatok között találunk olyanokat, amelyeket mind az öt technika meg tud oldani, hiszen ez olyan terület, amellyel mindenik foglalkozik valamilyen szinten.

7				
5	9			
10	1	4		
2	7	3	1	
2	5	8	3	1

1.1. ábra.

irányba lépjünk tovább, függőlegesen le vagy átlósan jobbra). Minden döntéssel a feladat egy hasonló, de egyszerűbb feladatra redukálódik. Tehát az absztrakt platform szerepét az alábbi bináris fa tölti be.



1.2. ábra.

Az optimális megoldás meghatározása az optimális döntéssorozat megtalálását jelenti. Úgy is mondhatnánk, hogy meg kell találnunk a megoldásfa 2^{n-1} darab gyökértől levélhez vezető útja közül a „legjobbat". Más szóval, meg kell keresnünk a „legjobb levelét" a megoldásfának, azt, amelyikhez a „legjobb út" vezet.

A megoldásfának egy alaposabb vizsgálata további észrevételekhez vezethet el:

1. A megoldásfa csomópontjainak száma, $1 + 2 + 2^2 + \ldots + 2^{n-1} = 2^n - 1$. Ez azt jelenti, hogy bármely algoritmus, amely bejárja a teljes fát, ahhoz, hogy megtalálja az optimális utat, exponenciális bonyolultságú lesz.

2. Amíg a fa a teljes feladatot képviseli, addig a részfái azokat a hasonló, de egyszerűbb részfeladatokat (a levelek a triviális részfeladatokat), amelyre ez lebontható. Konkrétan: az a_{ij} gyökerű részfa, az $a[i][j]$ elemtől az alapra vezető „leghosszabb út" meghatározásának feladatát ábrázolja.
3. A fenti ábra azt is kiemeli, hogy különböző döntéssorozatok azonos részfeladatokhoz vezethetnek, ami azt jelenti, hogy a megoldásfának vannak azonos részfái. Nem nehéz átlátni, hogy a különböző részfeladatok száma azonos a mátrix elemeinek számával, azaz $n(n+1)/2$. Tehát az algoritmus, amelynek sikerül elkerülni az azonos részfeladatok többszöri megoldását, négyzetes bonyolultságú lesz.

A következőkben meghívjuk az öt technikát egy képzeletbeli *talk-show*-ra, hadd mutatkozzanak be ők maguk, elmagyarázva, miként közelítenék meg a bemutatott feladatot.

Greedy (mohó algoritmus)

Jelszó: „Élj a mának!"

Indulok a csúcsból. Minden lépésben két lehetséges elem közül kell választanom. Melyiket válasszam? Természetesen mindig a nagyobbik elemet!

Megjegyzés: Mi a következménye mohóságának? Jelen esetben az, hogy egyáltalán nem biztos, hogy megtalálja a legjobb utat. A példamátrix esetében a mohó út 31 hosszú lesz, és ez a következő: $a[1][1]$, $a[2][2]$, $a[3][3]$, $a[4][3]$, $a[5][3]$.

Backtracking

Jelszó: „Lassan, de biztosan."

Generálom az összes olyan utat, amely a csúcsból az alapra vezet, és kiválasztom közülük a „legjobbat".

Divide et impera

Jelszó: „Oszd meg és uralkodj!"

Észrevehető, hogy bármely $a[i][j]$ $(i < n)$ elemtől induló „legjobb út" meghatározása leoszthatóaz $a[i+1][j]$, illetve $a[i+1][j+1]$ elemektől

induló „legjobb utak” meghatározására, ugyanis amennyiben ezek rendelkezésre állnak, nem marad más hátra, mint hogy „belépjünk” az $a[i][j]$ elemmel a hosszabbik elé. Más szóval, először lebontom rekurzívan a feladatot egyszerűbb, majd még egyszerűbb részfeladatokra, végül pedig a „visszaúton” – figyelembe véve az előbbi észrevételt – felépítem a legjobb utat.

Dinamikus programozás

Jelszó: „Az intelligens ember hosszú távon gondolkodik.”

Az én alapötletem az, hogy kiindulva a triviális részfeladatok kézenfekvő megoldásaiból, felépítsem az egyre bonyolultabb részfeladatok megoldásait, míg végül el nem jutok a fő feladat megoldásához. Mivel el szeretném kerülni az azonos részfeladatok többszöri megoldását, az optimális részmegoldásokat eltárolom egy c kétdimenziós tömbbe (a $c[i][j]$ tömbelem az $a[i][j]$ elemtől az alapra vezető „legjobb út” hosszát fogja tárolni). A c tömb főátlóját és a főátló alatti háromszögét lentről felfele, soronként töltöm fel, figyelembe véve, hogy

$$c[i][j] = a[i][j] + \max(c[i+1][j], c[i+1][j+1]), \quad i < n$$

Végül a $c[1][1]$-ben lesz az optimális út hossza. Ahhoz, hogy meglegyen maga az út is, annyi szükséges még, hogy végigmenjek a c tömbön mohó módon.

37				
30	25			
25	16	15		
7	15	11	4	
2	5	8	3	1

1.3. ábra.

Branch and bound

Jelszó: „Bátran, de nem vakmerően”

Párhuzamosan elindulok minden csúcsból alapra vezető úton lefele, mindig abba az irányba lépve először, amelyik a legígéretesebb. Amelyik úton hamarabb érem el az alapot, az a legjobb.

Megjegyzés: Bizonyítható, hogy e feladat esetében ez a megközelítés nem vezet el az optimális megoldáshoz.

A fenti bemutató természetesen csak azt a célt szolgálta, hogy fogalmat alkothassunk a módszerekről, melyeknek részletes bemutatása a következő fejezettel kezdődik.

2. FEJEZET

BACKTRACKING
Visszalépéses keresés

A backtracking módszer alapvetően a következő feladatot oldja meg: Egy ismert halmazsorozatból (A_1, A_2, ..., A_n), az összes lehetséges módon összeválogat elemeket, halmazonként egy-egy elemet, szem előtt tartva, hogy az összeválogatott sorozatok (vektorok) elemei között teljesülniük kell bizonyos feltételeknek.

Ezek az összeválogatott elemekből álló sorozatok lesznek a feladat megoldásai. Gyakran egyszerűen arról van szó, hogy meg kell keresnünk az $A_1 \times A_2 \times \ldots \times A_n$ Descartes-szorzat azon elemeit, amelyeket bizonyos *belső tulajdonságok* jellemeznek, és így megoldásnak számítanak. Jegyezzük meg (mint fontos következtetést ennél a pontnál), hogy *egy „backtracking-feladat" megoldásai vektor alakúak*.

Egy másik fontos szempont az, hogy az A_1, A_2, ..., A_n halmazokat általában nem kell tárolnunk a számítógép memóriájában, a backtracking algoritmus generálja ezeket. Erre a célra a módszer felhasznál egy egydimenziós tömböt, amelyet x-szel fogunk jelölni. Az A_i halmaz elemei rendre bekerülnek az $x[i]$ tömbelembe.

Mindezt szemléletesen mutatja be a 2.1. ábra, amelyet egy backtracking-feladat alapábrájának is nevezhetnénk.

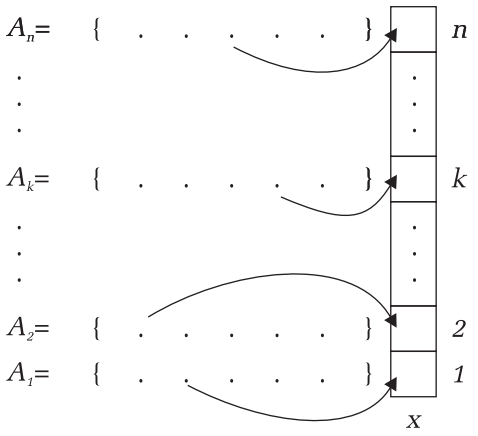
Hogyan helyezi el a backtracking az x[i] tömbelembe rendre az A_i halmaz egy-egy elemét? Egymás után állítja elő őket, egyikből a másikat. Ez azt jelenti, hogy az A_1, A_2, ..., A_n halmazok nem lehetnek akármilyenek, elemeik valamilyen szabályosság szerint kell hogy kövessék egymást.

Az eddig kifejtettek alapján elmondhatjuk, hogy backtracking-feladatként felfogni egy feladatot azt jelenti, hogy

1. a feladat megoldásait vektorokként kódoljuk,
2. azonosítjuk az A_1, A_2, ..., A_n halmazokat, ahonnan a megoldásvektorok elemei származnak.

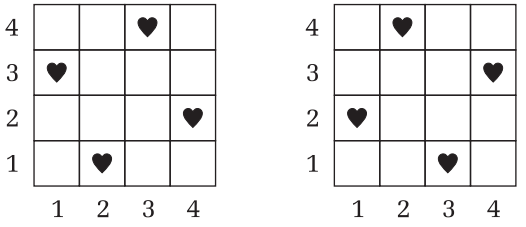
Szemléltetésül vizsgáljuk meg az „n királynő a sakktáblán" feladatot.

Királynők: Helyezzünk el n királynőt egy $n \times n$ méretű sakktáblán úgy, hogy ne támadják egymást.



2.1. ábra.

A 2.2. ábra bemutatja $n = 4$ esetén a megoldásokat.



2.2. ábra.

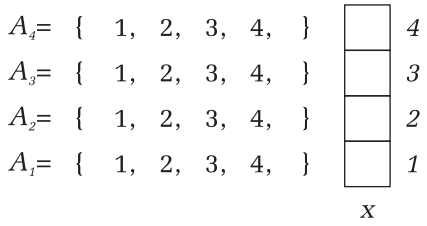
Hogyan fogható fel backtracking-feladatként az n királynő feladata?

A megoldások n elemű vektorokként $(x_1, x_2, \ldots, x_n)$ kódolhatók: az i-edik királynőt az i-edik sorkoordinátájú és az x_i-edik oszlopkoordinátájú négyzetre tesszük.

Például, ha $n = 4$, a megoldások kódjai:

1. (2, 4, 1, 3)
   1. királynő: 1. sor 2. oszlop
   2. királynő: 2. sor 4. oszlop
   3. királynő: 3. sor 1. oszlop
   4. királynő: 4. sor 3. oszlop
2. (3, 1, 4, 2)
   1. királynő: 1. sor 3. oszlop
   2. királynő: 2. sor 1. oszlop
   3. királynő: 3. sor 4. oszlop
   4. királynő: 4. sor 2. oszlop

Az alapábra:



2.3. ábra.

Tehát az A_1, A_2, ..., A_n halmazok mindegyike azonos az $A = \{1, 2, \ldots, n\}$ halmazzal.

Tömören: keressük az $A \times A \times \ldots \times A$ n-szeres Descartes-szorzat azon elemeit, amelyek helyes királynő-elhelyezés kódjai.

Összefoglalásként lássuk, mikor oldható meg egy feladat a backtracking módszer segítségével:

1. A feladat megoldásai vektor formájában kódolhatók.
2. Azonosítani tudjuk az A_1, A_2, ..., A_n halmazokat, ahonnan a megoldásvektorok elemei származnak, és ezek elemei valamilyen szabályosság szerint követik egymást.

Ugyanakkor figyelembe kell vennünk, hogy a backtracking algoritmusok általában exponenciális bonyolultságúak. Például, a backtracking legprimitívebb változata előállítja az $A_1 \times A_2 \times \ldots \times A_n$ Descartes-szorzat összes $n_1 \times n_2 \times \ldots \times n_n$ elemét (az n királynő feladata esetén ez n^n elemet jelent), és kiválasztja közülük a megoldásokat. Bár a módszer filozófiája az, hogy megpróbálja csökkenteni a generálandó megoldások számát, gyakran az optimalizálások után is megmarad az exponenciális bonyolultság. Ezért a backtrackinget általában csak akkor alkalmazzuk, ha nincs más választásunk.

2.1. A backtracking stratégiájának leírása

A backtracking módszer stratégiája jobban nyomon követhető, *ha a feladatot fa formájában ábrázoljuk.*

A gyökértől a levelekhez vezető utak az $A_1 \times A_2 \times \ldots \times A_n$ Descartes-szorzat elemeit ábrázolják. Az első szintre az $x[1]$ tömbelembe n_1-féleképpen választhatunk elemet az A_1 halmazból, tehát a fa gyökeréből n_1 első szinti fiúcsomópont ágazik le, amelyekhez az A_1 halmaz

elemeit rendeljük. Minden első szinti választáshoz a második szintre, az $x[2]$-be n_2-féleképpen választhatunk elemet az A_2 halmazból. A fán ez abban tükröződik, hogy minden első szinti csomópontnak n_2 fia lesz a második szinten, amelyekhez az A_2 halmaz elemeit rendeljük. Ez összesen $n_1 \times n_2$ második szinti csomópontot eredményez. Végül az n-edik szinten a fának $n_1 \times n_2 \times \ldots \times n_n$ csomópontja (levele) lesz. Általános szabályként megjegyezhető, hogy ha a csomópont a fa k-adik szintjén található, és az apacsomópontjának ez az i-edik fia, akkor az A_k halmaz i-edik elemét rendeljük hozzá. Mivel a fa bármely csomópontjához a gyökérből pontosan egy út vezet, ezért elmondható, hogy minden csomópont képviseli ezt az utat. A levelek mindegyike a Descartes-szorzat valamelyik elemét képviseli – azt, amelyiket a gyökérből az illető levélhez vezető út ábrázol.

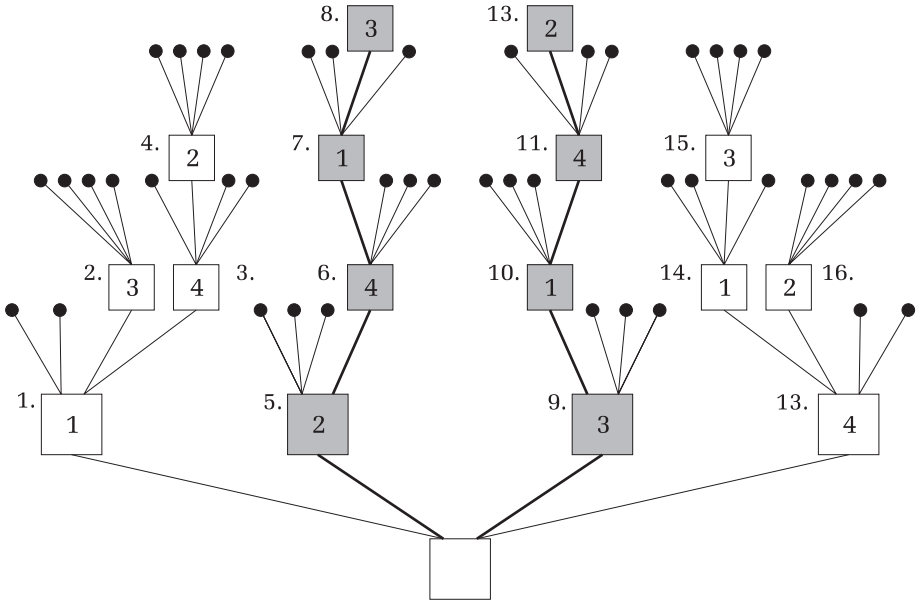
Hogyan jelennek meg a fában a feladat megoldásvektorai? Ha a megoldások a Descartes-szorzat adott tulajdonságú elemei, akkor ezeket azok az utak ábrázolják, amelyek a gyökértől egy-egy levélhez vezetnek. Nevezzük ezeket *megoldásutak*nak vagy *megoldás*oknak, a hozzájuk tartozó leveleket pedig *megoldáslevelek*nek. Egyes feladatokban a megoldásokat nem feltétlenül gyökér–levél utak ábrázolják, hanem a gyökértől adott tulajdonságú csomópontokhoz vezető utak. Általános esetben tehát *megoldáscsomópontok*ról beszélünk. Ezek után nem nehéz átlátni, miért nevezzük a feladathoz rendelt fát a *megoldások terének*.

Mindezeket figyelembe véve egy backtracking-feladat az alábbi módon is megfogalmazható: keressük meg a feladathoz rendelhető fa megoldáscsomópontjait, illetve építsük fel azokat a megoldásutakat, amelyek a gyökérből ezekhez vezetnek, és amelyekhez, természetesen, éppen a megoldásvektorok vannak rendelve. Mindezt az x tömbben fogjuk megvalósítani, amelyet – amint látni fogjuk – úgy használunk, mint egy vermet[1].

Szemléltetésül lássuk, hogyan működik a backtracking algoritmus a 4 királynő feladatának esetében. A következő ábrán, helyhiány miatt, nem rajzoltuk le a teljes fát (a negyedik szinten 256 levele van). Csak azt a részfát rajzoltuk le, amely potenciálisan tartalmazza a megoldásleveleket (ezt nevezzük *a fa élő részé*nek). A csomópontjait aszerint

1 A verem egy lineáris adatszerkezet, amelynek ugyanazon végén (a „tetején") történik mind a betevés, mind a kivevés. Ebből kifolyólag az utoljára betett elem lesz az elsőnek kivett. (**L**ast **I**n **F**irst **O**ut)

sorszámozzuk, hogy milyen sorrendben látogatjuk meg őket. A jobb áttekinthetőség kedvéért bejelöltük az „*élő* csomópontok” „*száraz* irányba” vivő fiait is. A fa alatt megadtuk a csomópontok meglátogatásának pillanatában a verem (az x tömb) tartalmát. A vastagított vonalat követve láthatjuk, mikor, mely elemek kerülnek be, illetve ki a veremből.



1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.	13.	14.	15.	16.
							3				2				
			2			1	1			4	4			3	
	3	4	4		4	4	4		1	1	1		1	1	2
1	1	1	1	2	2	2	2	3	3	3	3	4	4	4	4
x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

2.4. ábra.

Indulunk a gyökérből. Sorra megvizsgáljuk a gyökér utáni első szint csomópontjait, a gyökér fiait, és fellépünk ahhoz, amelyről elsőként találjuk azt, hogy potenciálisan megoldáscsomóponthoz vezet. Itt hasonlóképpen járunk el: megvizsgáljuk a fiait (a belőle ágazó második szint csomópontjait), és ahogy olyat találunk, amely ígéretes irányba vezet, oda lépünk. Így járunk el egészen addig, míg olyan csomóponthoz nem

jutunk, amelynek nyilvánvalóan egyetlen fia sem vezet megoldáshoz. Ilyenkor visszalépünk az előző szinti apacsomóponthoz, ahol folytatjuk a keresést, további ígéretes fiak után kutatva. Ha találunk egy újabb csomópontot, amely potenciálisan megoldáscsomóponthoz vezet, akkor újból fellépünk a következő szintre. Ha egy csomópontból már minden irányt ellenőriztünk – az ígéreteseket be is járva –, akkor innen is visszalépünk. Az algoritmus akkor ér véget, amikor a gyökérből ágazó összes első szinti csomóponttal foglalkoztunk már. Valahányszor megoldáscsomópontot találunk, a gyökérből hozzá vezető út egy megoldásvektornak felel meg.

A fa ily módon történő bejárását mélységi bejárásnak[2] nevezzük (lásd a bevezető fejezetet). Vegyük észre, hogy nem jártuk be a teljes fát, csak azt a részfát, amely potenciálisan tartalmazza a megoldásokat. Ezt neveztük a fa *élő részének,* a többi *száraz ág* a feladatra nézve. Nyilván, minél jobban sikerül leszűkíteni a bejárt részfát, annál hatékonyabb az algoritmusunk. Ez attól függ, hogy mennyire hatékonyan tudjuk meghatározni, hogy egy bizonyos irány ígéretes-e. A 4 királynő feladata esetén sikerült csupán 16 csomópont meglátogatása által megtalálni a megoldásokat, holott a teljes fának összesen 4 + 16 + 64 + 256 = 340 csomópontja van (nem számoltuk a gyökeret, lévén csupán virtuális).

De mit is jelent az, hogy egy irány ígéretes vagy hogy az illető fiúcsomópont potenciálisan megoldáscsomóponthoz vezet? Először is kövessük nyomon, miként alakul át, a keresés alatt, az aktuális út (a gyökérből az aktuális csomóponthoz vezető út) megoldásutakká. Amikor felfele lépünk a fában, új csomópont kerül az aktuális út végére. Visszalépéskor egy bizonyos csomópont eltűnik a végéről. Nos, egy csomópont akkor vezet potenciálisan megoldáscsomópont felé, ha ígéretesen bővíti a gyökérből az aktuális csomóponthoz vezető utat. Hogyan értendő ez? Emlékezzünk, hogy egy út attól megoldásút, hogy a csomópontjaihoz rendelt elemek megoldásvektort alkotnak. Más szóval, az illető úthoz rendelt elemek rendelkeznek a megoldásvektorokat azonosító belső tulajdonsággal. A backtracking módszer kulcsötlete az, hogy amennyiben a szóban forgó fiúcsomóponthoz rendelt elem máris nem fér össze – a megoldásvektorokat azonosító belső tulajdonság szempontjából – a már

2 A fákat általában a gyökerükkel fent és a leveleikkel lent szokás ábrázolni. Mivel mi megfordítottuk, ezért esetünkben találóbb lenne a „fa magassági bejárása" kifejezés.

az aktuális úton lévő csomópontokhoz rendelt elemekkel, akkor értelmetlen az illető csomópontra építve keresni az újabb megoldásokat. Tehát, egy megvizsgált fiúcsomópont akkor ígéretes, ha – az imént bemutatott értelemben – nem kerül „konfliktusba" az apacsomópontjához vezető aktuális út már ígéretesnek talált csomópontjaival. Például az n királynő feladata esetén a fa csomópontjaihoz rendelt elemek a sakktábla négyzeteit képviselik. Ha a szóban forgó csomópont által képviselt négyzetet támadják az aktuális úton lévő csomópontok négyzetein már elhelyezett királynők, akkor nyilván „száraz irányról" van szó.

Az a feltétel, amelynek alapján eldönthető, hogy az illető csomóponttal ígéretesen bővíthető-e az aktuális út, a feladat megoldásait azonosító belső tulajdonságok alapján határozható meg.

Amint megfigyelhettük, ennek a feltételnek kulcsszerepe van a backtrackingben, hiszen alapvetően ez határozza meg, mekkora lesz a bejárt részfa.

És most lássuk, miként valósítható meg a fán bemutatott algoritmus az x tömbben? Nem fogjuk felépíteni a fát a számítógép memóriájában, csak szimuláljuk a bejárását az x tömb segítségével.

A fenti ábra és az algoritmus tanulmányozása a következő fontos észrevételekhez vezet:

1. Az x tömböt úgy kell használnunk, mint egy vermet. Figyeljük meg, hogy a fa bejárásának egy adott pillanatában a tömb az aktuális úton található csomópontok képviselte elemeket tartalmazza. Amikor felfele lépünk a fában, a verem tetejére új elem kerül. Visszalépéskor eltűnik a verem tetején lévő elem.
2. A fa bejárásakor minden érintett csomópontban alapvetően ugyanúgy kellett eljárnunk: sorra megvizsgáltuk a fiait, és valahányszor ígéretest találtunk, felléptünk hozzá, ahol hasonlóképpen jártunk el. Természetesen minden csomópontban ellenőriznünk kell, hogy nem képvisel-e éppen megoldást.
3. A fának a fenti algoritmus szerinti bejárását úgy is felfoghatjuk, hogy egy csomóponthoz tartozó részfa bejárását visszavezetjük ígéretes fiúrészfáinak bejárására (egy részfa akkor ígéretes, ha potenciálisan tartalmaz megoldáscsomópontot).

Mindhárom észrevétel azt sugallja, hogy az algoritmust rekurzívan valósítsuk meg. Ez egy olyan eljárás megírását feltételezi, amely mindig az aktuális csomóponton dolgozik: sorra ellenőrzi a fiait, és valahányszor ígéretest talál, meghívja magát az illető fiúcsomópontra. A visszalépés

automatikusan történik a rekurzió mechanizmusa által, amikor az aktuális csomópont összes fiának a vizsgálata befejeződött. Ha az a részfa, amely potenciálisan tartalmazza a megoldásokat, véges, akkor nem kell tartanunk a végtelen rekurzív hívásoktól: mivel a leveleknek nincsenek ígéretes fiaik, ezekből az algoritmus nem hívja meg önmagát.

A kérdés már csak az, hogy miként tudjuk szimulálni mindezt az x tömbben.

Mivel az azonos szintű csomópontok fiaihoz ugyanazok az elemek vannak rendelve, a rekurzív eljárásnak mint paraméterre az x tömbön kívül (amit *cím szerint* kap meg) alapvetően csak *az aktuális szint értéké*re van még szüksége. Tegyük fel, hogy az aktuális csomópont a k-adik szinten található. Ebben az esetben a veremben, azaz az x tömb első k elemében, az aktuális úton lévő csomópontokhoz rendelt értékek találhatók. Mivel minden k-adik szinti csomópont fiaihoz az A_{k+1} halmaz elemei vannak rendelve, ezért a következőképpen járhatunk el: az x tömb $(k+1)$-edik szintjére sorra előállítjuk az A_{k+1} halmaz elemeit, és valahányszor ígéretest találunk, meghívjuk az eljárást rekurzívan a $(k+1)$-edik szintre. Természetesen emellett azt is ellenőriznünk kell, hogy nem tartunk-e megoldáscsomópontnál, azaz az x tömbben nem épült-e fel egy megoldásvektor, amelyet ki kell íratnunk. Hosszas vajúdás után íme a visszalépéses keresést implementáló rekurzív eljárás:

```
backtracking(x[],k)
   ha megoldás(x,k) akkor
      kiír(x,k)
   vége ha
   minden i=1,n_{k+1} végezd
      < előállítjuk x[k+1]-ben az A_{k+1} halmaz
       i-edik elemét >
      ha ígéretes(x,k+1) akkor
         backtracking(x,k+1)
      vége ha
   vége minden
vége backtracking
```

A legtöbb feladat esetében (ide tartozik az n királynő feladata is), ha $(x_1, x_2, \ldots, x_k)$ megoldásvektor, akkor nem bővíthető ígéretesen újabb megoldásvektorrá. Más szóval kizárt, hogy egyik megoldásvektor része legyen egy másiknak. Ilyenkor a **minden** ciklust tehetjük a **ha** utasítás **különben** ágára. Ha nem így járnánk el, és a fa valamely levele megoldáscsomópont lenne, akkor a fenti eljárás még generálná e levél

(képzeletbeli) fiait is, amelyek között persze egy ígéretest sem találna, és így ezt követően visszalépne.

```
backtracking(x[],k)
   ha megoldás(x,k) akkor
      kiír(x,k)
   különben
      minden i=1,n_{k+1} végezd
         < előállítjuk x[k+1]-ben az A_{k+1} halmaz
          i-edik elemét >
         ha ígéretes(x,k+1) akkor
            backtracking(x,k+1)
         vége ha
      vége minden
   vége ha
vége backtracking
```

A backtracking algoritmus kulcsfüggvénye az `ígéretes(x,k+1)` függvényhívás alkalmával ellenőrzi, hogy az `x[k+1]` rekeszbe elhelyezett elem ígéretesen összefér-e a megoldásvektorokat azonosító belső tulajdonság szempontjából a már ígéretesnek elfogadott és az `x[1]`, `x[2]`, ..., `x[k]` rekeszekben található elemekkel.

A `megoldás(x,k)` függvényhívás ellenőrzi, hogy az `x` tömb első `k` elemében megoldásvektor épült-e ki. Ez a függvény akkor lesz hatékony, ha figyelembe veszi, hogy amikor hozzá kerül az ellenőrzésre váró `x[1..k]` tömbszakasz, már átment az ígéretes függvény kezén.

A `kiír(x,k)` eljáráshívás kiírja az `x` tömb első `k` helyén felépült megoldásvektort.

Következzen a backtracking algoritmus az n királynő feladatának megoldására. Az eljárások és függvények paraméterezését a feladat sajátosságaihoz igazítottuk. Például a `backtracking` eljárás megkapja paraméterként a királynők számát is.

```
backtracking(x[],n,k)
   ha megoldás(n,k) akkor
      kiír(x,k)
   különben
      // A_{k+1} = {1,2,...,n} a (k+1)-dik királynő
      // lehetséges oszlop indexeit tartalmazza
      minden x[k+1]=1,n végezd
         ha ígéretes(x,k+1) akkor
            backtracking(x,n,k+1)
         vége ha
```

```
        vége minden
    vége ha
vége backtracking
```

A királynők garantáltan más-más sorba kerülnek, az i-edik királynő az i-edik sorba. Így a k-adik királynőt csakis akkor támadnák az első $k-1$ sorba már elhelyezett királynők, ha valamelyikkel ugyanabba az oszlopba, vagy ugyanarra az átlóra kerülne. Vegyük figyelembe azt is, hogy az `x[1..k-1]` tömbszakasz a már elhelyezett királynők oszlopindexeit, `x[k]` pedig a k-adik királynő számára javasolt hely oszlopindexét tartalmazza.

```
ígéretes(x[],k)
   minden i=1,k-1 végezd
      // ugyanazon oszlopban vagy ugyanazon az átlón
      ha x[i] == x[k] vagy (k - i) == |x[k] - x[i]| akkor
         return HAMIS
      vége ha
   vége minden
   return IGAZ
vége ígéretes
```

Mivel az `ígéretes` függvény már ellenőrizte, hogy a sakktáblára került királynők ne támadják egymást, a `megoldás` függvénynek csak azt kell ellenőriznie, hogy mind az n királynő el van-e helyezve.

```
megoldás(n,k)
   ha k == n akkor
      return IGAZ
   különben
      return HAMIS
   vége ha
vége megoldás

kiír(x[],k)
   minden i=1,k végezd
      ki: x[i]
   vége minden
vége kiír
```

A backtracking eljárást kezdetben a 0-adik szintre hívjuk meg (hiszen a gyökérből kell indulnia): `backtracking(x,n,0)`.

2.2. Hogyan közelítsünk meg egy backtracking feladatot?

Követhetjük az alábbi lépéssorozatot:

1. Hogyan kódolhatók a feladat megoldásai vektor formájában? Ez az első és legfontosabb lépés. Megtörténhet, hogy több lehetőség is kínálkozik. Válasszuk azt a kódolást, amelyik a legalkalmasabb. Például előnyös, ha a megoldásvektorok egyforma hosszúak. Hasznos lehet egy konkrét példából kiindulni. Ha egy egyszerű esetre el tudjuk képzelni a megoldásokat, akkor ez ihlető lehet a kódolást illetően.

Fogalmazzuk meg, milyen belső tulajdonságok azonosítják a megoldásvektorokat.

2. Azonosítsuk az A_1, A_2, ... halmazokat, és készítsük el a feladat alapábráját, esetleg a hozzá rendelhető fát is. Ebben segít, ha végiggondoljuk a következőket: a megoldásvektor k-adik eleme az $x[k]$ tömbelemben lesz előállítva. Tehát az A_k halmaz azokat az értékeket tartalmazza, amelyek elméletileg megjelenhetnek a megoldásvektorok k-adik pozícióin. Tartsuk szem előtt azt is, hogy az A_k halmazok elemei között léteznie kell egy szabályosságnak, amely szerint az elemek követik egymást. Például az n királynő feladatában azért volt bármely $k = 1, \ldots, n$ esetén $A_k = 1, 2, \ldots, n$, mert a k-adik királynő, a sakktábla k-adik sorában, elméletileg bármelyik oszlopba kerülhetett (soronként próbáltunk meg elhelyezni egy-egy királynőt). Az alapábra azt is sugalmazza, hogy milyen „mechanizmussal" állíthatók elő az $x[k]$ tömbelemben az A_k halmaz elemei rendre, egyik a másikból. Ez rendszerint az alábbi módon valósítható meg:

```
x[k] = <A_k_első_eleme>
amíg x[k] <= <A_k_utolsó_eleme> végezd
   ...
   <utasítás, amely x[k]-ban, A_k aktuális eleméből,
   a szabályosság alapján, előállítja a következő elemet>
vége amíg
```

Az A_k halmazok elemei gyakran – amint az „n királynő" feladatában is – egymást követő természetes számok. Ilyen esetben (abból a célból, hogy ezeket előállítsuk) a `backtracking` eljárásban használhatunk egy **minden** ciklust:

```
backtracking(x[],k)
   ha megoldás(x,k) akkor
     kiír(x,k)
```

```
        különben
           minden x[k+1] = <A_{k+1}_első_eleme>,<A_{k+1}_utolsó_eleme>
                                                              végezd
              ha ígéretes(x,k+1) akkor
                 backtracking(x,k+1)
              vége ha
           vége minden
        vége ha
     vége backtracking
```

3. Írjuk meg az ígéretes függvényt, amelynek döntő fontosságú szerep jut abban, hogy mennyire lesz hatékony az algoritmusunk. Emlékezzünk, hogy a backtracking eljárás a **minden** ciklusban kerül meghívásra, és ellenőriznie kell, hogy az `x[k+1]`-ban éppen előállított elem ígéretesen bővíti-e az `x[1..k]` tömbszakaszban már elfogadott elemeket. Bár az ígéretes függvényt a $(k+1)$-edik szintre hívjuk meg, a „forgatókönyvét" általánosan egy k-adik szintre írjuk meg. Az ígéretesség feltétele alapvetően a megoldásvektorokat azonosító belső tulajdonságokból következtethető ki.

4. A megoldásfüggvény megírása következik. Azt a feltételt kell tartalmaznia, amelynek alapján eldönthető, hogy egy ígéretes `x[1..k]` tömbszakasz megoldás-e.

5. Utolsóként szokás a `kiír` eljárást megírni. Tanácsos nem csupán a megoldásvektorokat kiírni, hanem, elvégezve a dekódolást, magukat a megoldásokat is. Például az n királynő feladata esetén „kirajzolhatnánk" a sakktáblát, rajta a helyesen elhelyezett királynőkkel.

A következőkben számos megoldott feladaton példázzuk, miért célszerű követni ezt a megközelítési módot.

2.3. Megoldott feladatok

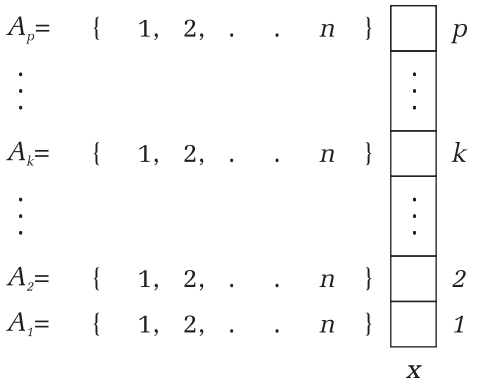
2.1. Halmazműveletek: Generáljuk az $\{1, 2, \ldots, n\}$ halmaz
1. p-szeres Descartes-szorzatának elemeit;
2. összes permutációit;
3. p-edrendű variációit;
4. p-edrendű kombinációit;
5. összes részhalmazát;
6. összes partícióját.

A fent javasolt ötlépéses gondolatmenetet követjük.

Megoldás (p-szeres Descartes-szorzat):
I. Hogyan kódolhatók a feladat megoldásai vektorokként?
Példa a Descartes-szorzat elemeire $n = 3$ és $p = 2$ esetén:

$$\{(1,1),(1,2),(1,3),(2,1),(2,2),(2,3),(3,1),(3,2),(3,3)\}.$$

Észrevehető, hogy nincs szükség különösebb kódolásra, hiszen mindenik megoldás önmagában vektor alakú (kételemű – általánosan p elemű – vektorok).
II. A feladat alapábrája



2.5. ábra.

Tekintettel arra, hogy az A_k halmazok mindegyike $\{1, 2, \ldots, n\}$ alakú, a backtracking algoritmus a következőképpen fog kinézni:

```
descartes(x[],n,p,k)
   ha megoldás(p,k) akkor
      kiír(x,k)
   különben
      minden x[k+1]=1,n végezd
         ha ígéretes(x,k+1) akkor
            descartes(x,n,p,k+1)
         vége ha
      vége minden
   vége ha
vége descartes
```

III. Mivel a feladat a) pontja a Descartes-szorzat összes elemét kéri, nem beszélhetünk a feladat megoldásait azonosító belső tulajdonságokról. Ebből az is következik, hogy az x tömb bármely k-adik helyére az A_k

halmaz mindenik eleme megfelelő (ígéretes). Ily módon az ígéretes függvény mindig IGAZ-at fog visszatéríteni.

```
ígéretes(x[],k)
   return IGAZ
vége ígéretes
```

IV. Valahányszor ígéretes elemet találunk a tömb p-edik eleme számára, megoldáshoz jutottunk.

```
megoldás(p,k)
   ha k==p akkor
      return IGAZ
   különben
      return HAMIS
   vége ha
vége megoldás
```

V. Mivel a megoldásvektorok maguk a megoldások, ezért nincs szükség dekódolásra.

```
kiír(x[],k)
   minden i=1,k végezd
      ki: x[i]
   vége minden
vége kiír
```

Tekintettel az ígéretes és megoldás algoritmusok egyszerű voltára, lemondhatunk a használatukról. Ez esetben a backtracking eljárás a következőképpen fog mutatni:

```
descartes(x[],n,p,k)
   ha k == p akkor
      kiír(x,k)
   különben
      minden x[k+1]=1,n végezd
         descartes(x,n,p,k+1)
      vége minden
   vége ha
vége descartes
```

A backtracking eljárás meghívása: `descartes(x,n,p,0)`

Megoldás (permutációk):
I. Hogyan kódolhatók a feladat megoldásai vektorokként?

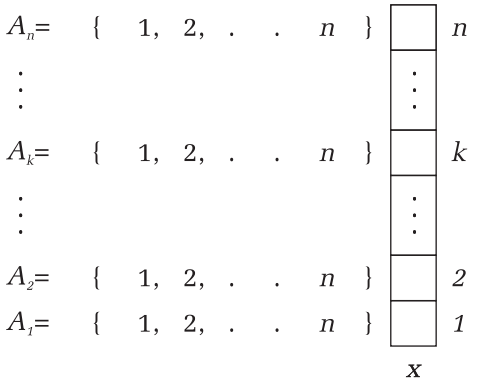
Példa a permutációkra $n = 3$ esetén:

$$(1,2,3),(1,3,2),(2,1,3),(2,3,1),(3,1,2),(3,2,1)$$

Észrevehető, hogy most sincs szükség különösebb kódolásra, hiszen mindenik megoldás önmagában vektor alakú (háromelemű – általánosan n elemű – vektorok).

Milyen belső tulajdonság jellemzi a megoldásvektorokat? A permutációk – a Descartes-szorzat elemeitől eltérően – nem tartalmazhatnak egy elemet többször, vagyis minden elemet pontosan egyszer tartalmaznak.

II. A feladat alapábrája



2.6. ábra.

Tekintettel arra, hogy az A_k halmazok mindegyike ez esetben is {1, 2, . . . , n} alakú, a backtracking algoritmus hasonlóképpen fog mutatni, mint a Descartes-szorzat esetén.

```
permutáció(x[],n,k)
   ha megoldás(n,k) akkor
      kiír(x,k)
   különben
      minden x[k+1]=1,n végezd
         ha ígéretes(x,k+1) akkor
            permutáció(x,n,k+1)
         vége ha
      vége minden
   vége ha
vége permutáció
```

III. Figyelembe véve a megoldásvektorokat azonosító belső tulajdonságokat, egy elem az x tömb k-adik szintjén akkor megfelelő (ígéretes), ha nem szerepel már a tömb $1 \ldots (k-1)$ szintjein.

```
ígéretes(x[],k)
   minden i=1,k-1 végezd
      ha x[i] == x[k] akkor
         return HAMIS
      vége ha
   vége minden
   return IGAZ
vége ígéretes
```

IV. Ahogy az a feladat alapábrájából is kiderül, akkor vagyunk megoldásnál, ha fel tudtunk lépni az x tömb n-edik szintjére.

```
megoldás(n,k)
   ha k == n akkor
      return IGAZ
   különben
      return HAMIS
   vége ha
vége megoldás
```

V. A kiír eljárás ugyanaz, mint a Descartes-szorzat esetén.

A backtracking eljárás meghívása: `permutáció(x,n,0)`

Érdekes megfigyelni, hogy a permutációk úgy foghatók fel, mint az n-szeres Descartes-szorzat azon elemei, amelyekben nincsenek ismétlődő elemek.

Megoldás (p-edrendű variációk):

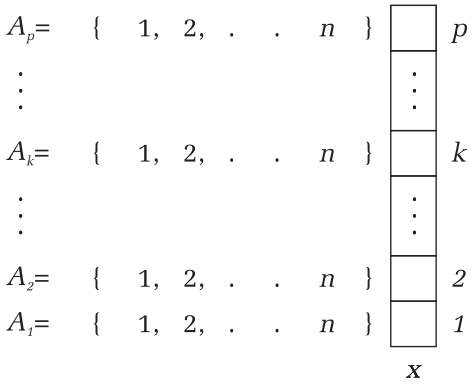
I. Hogyan kódolhatók a feladat megoldásai vektorokként?

Példa a variációkra $n = 3$ és $p = 2$ esetén:

$$(1,2), (1,3), (2,1), (2,3), (3,1), (3,2)$$

Amint látható, a megoldásvektorok hossza p. Ezeket ugyanaz a belső tulajdonság jellemzi, mint a permutációkat: nincsenek többszörös elemeik. Természetesen, mivel a variációk p hosszúságú vektorok, ha $p < n$, nem tartalmazzák az $\{1, 2, \ldots, n\}$ halmaz összes elemét.

II. A feladat alapábrája:



2.7. ábra.

III., IV., V. Tekintettel a fentiekre, a variációk generálása csak a megoldásfüggvényben különbözik a permutációkat előállító algoritmustól.

```
variáció(x[],n,p,k)
   ha megoldás(p,k) akkor
      kiír(x,k)
   különben
      minden x[k+1]=1,n végezd
         ha ígéretes(x,k+1) akkor
            variáció(x,n,p,k+1)
         vége ha
      vége minden
   vége ha
vége variáció

megoldás(p,k)
   ha k == p akkor
      return IGAZ
   különben
      return HAMIS
   vége ha
vége megoldás
```

A backtracking eljárás meghívása: `variáció(x,n,p,0)`

Megoldás (p-edrendű kombinációk):
I–V. Példa a kombinációkra $n = 3$ és $p = 2$ esetén:

$$(1,2),(1,3),(2,3)$$

Észrevehető, hogy a kombinációk úgy szűrhetők ki a variációk közül, hogy megköveteljük az elemek növekvő sorrendjét. Ez belső tulajdonságul szolgál. A növekvő sorrend biztosítása a legegyszerűbben úgy valósítható meg, hogy a backtracking eljárásban az `x` tömb $(k+1)$-edik szintjére az A_{k+1} halmaznak csak `x[k]`-nál nagyobb elemeit generáljuk. Mivel ily módon már a backtracking eljárás biztosítja a megoldásvektorokat azonosító belső feltétel teljesülését, az `ígéretes` függvény mindig `IGAZ`-at fog visszatéríteni. Íme a kombinációkat generáló backtracking eljárás:

```
kombináció(x[],n,p,k)
   ha k == p akkor
      kiír(x,k)
   különben
      minden x[k+1]= x[k]+1,n végezd
         kombináció(x,n,p,k+1)
      vége minden
   vége ha
vége kombináció
```

A backtracking eljárás meghívása: `kombináció(x,n,p,0)`. Ugyanakkor a hívás előtt gondoskodunk az `x[0]` elemről: `x[0] = 0` – ezzel biztosítjuk, hogy az `x[1]` elemben az $\{1, 2, \ldots, n\}$ halmaz összes eleme elő legyen állítva.

Megoldás (részhalmazok):
I–V. Példa a részhalmazokra $n = 3$ esetén:

$$\{\},\{1\},\{2\},\{3\},\{1,2\},\{1,3\},\{2,3\},\{1,2,3\}$$

Figyeljük meg, hogy a részhalmazok előállíthatók úgy, hogy generáljuk sorra az első-, a másod-, ..., és az n-edrendű kombinációkat. Ez úgy valósítható meg, hogy a kombináció-eljárásnak átadjuk paraméterként azt is, hogy hányadrendű kombinációkat generáljon, majd meghívjuk ezt az eljárást ciklusból.

```
részhalmaz(x[],n)
   ki: "{}"
   minden p=1,n végezd
```

```
        kombináció(x,n,p,0)
    vége minden
vége részhalmaz
```

Ha azt szeretnénk, hogy halmazok alakjában jelenjenek meg az eredmények, használjuk az alábbi kiír függvényt:

```
kiír(x[],k)
   ki: '{'
   minden i=1,k végezd
      ki: x[i], ','
   vége minden
   ki: '}'
vége kiír
```

Megoldás (halmazpartíciók):
I. Hogyan kódolhatók a feladat megoldásai vektorként?
Példa a halmazpartícióira $n = 3$ esetén:

$$\{1,2,3\}$$
$$\{1,2\}\{3\}$$
$$\{1,3\}\{2\}$$
$$\{1\}\{2,3\}$$
$$\{1\}\{2\}\{3\}$$

Először címkézzük a partíciók részhalmazait, majd aszerint kódoljuk őket, hogy az egyes elemek melyik részhalmazban találhatók:

2.1. táblázat.

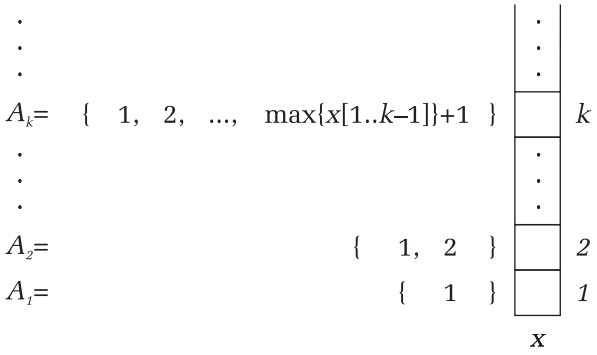
Címkézett partíciók	Vektor formájába kódoltan
$\{1,2,3\}^1$	$(1,1,1)$
$\{1,2\}^1\{3\}^2$	$(1,1,2)$
$\{1,3\}^1\{2\}^2$	$(1,2,1)$
$\{1\}^1\{2,3\}^2$	$(1,2,2)$
$\{1\}^1\{2\}^2\{3\}^3$	$(1,2,3)$

Tehát a megoldások n elemű vektorok formájában kódolhatók a következő jelentéssel: az i elem a partíció `x[i]` címkéjű részhalmazában található.

II–IV. Milyen címkéjű részhalmazokba kerülhetnek a partíciók elemei?

Észrevehető, hogy az 1-es mindig az első részhalmazba kerül. A 2-es lehet ugyanabban a részhalmazban, mint az 1-es, különben megnyitja a második részhalmazt. Ha az 1-es és 2-es együtt vannak, akkor a 3-as lehet velük, különben megnyitja a második részhalmazt. Ha az 1-es és 2-es külön vannak, akkor a 3-as lehet valamelyikükkel, különben megnyitja a 3. részhalmazt...

Általánosan: k kerülhet valamelyik már megnyitott részhalmazba, vagy nyithat egy új részhalmazt a következő sorszámú címkével. Mindez az alábbi ábrához vezet el:



2.8. ábra.

Figyeljük meg, hogy az ily módon meghatározott A_k halmazok csak ígéretes elemeket fognak tartalmazni, ami szükségtelenné teszi az `ígéretes` függvény használatát.

Mivel egy adott k-adik szinten az A_k halmaz felső határa az előző szinteken kiválasztott értékek maximumától függ, célszerű lenne mindig átadni a következő szintnek az addig használt legnagyobb címke értékét. Mindez a következő backtracking eljáráshoz vezet el:

```
partíció(x[],n,k,max)
   ha k == n akkor
      kiír(x,k,max)
   különben
      minden x[k+1]=1,max végezd
         partíció(x,n,k+1,max)
      vége minden
      x[k+1] = max+1
      partíció(x,n,k+1,max+1)
   vége ha
vége partíció
```

V. A `kiír` függvény dekódolja a megoldásokat.

```
kiír(x[],k,max)
   minden c=1,max végezd
      ki: '{'
      minden i=1,k végezd
         ha x[i] == c akkor
            ki: i,','
         vége ha
      vége minden
      ki: '}'
   vége minden
vége kiír
```

A backtracking eljárás meghívása: `partíció(x,n,0,0)`

Az imént bemutatott feladat azért is fontos, mert gyakoriak azok a feladatok, amelyek visszavezethetők rá. Például az n királynő feladata a következőképpen is felfogható: keressük meg az $\{1, 2, \ldots, n\}$ halmaz azon permutációit, amelyek helyes királynő-elhelyezésnek felelnek meg. Következzen további két példa.

2.2. Osztályterem: Adottak n tanuló nevei a `tanuló[1..n]` tömbben, akiket kétszemélyes padokba szeretnénk leültetni. Pontosan annyi pad van, amennyi szükséges (ha a tanulók száma páratlan, egy ülőhely üres marad). Írjuk ki az összes lehetőséget, ahogy a tanulók leültethetők az $(n+1)/2$ padba!

Megoldás: A feladat felfogható mint permutáció-feladat. Ha a tanulók száma páratlan, felveszünk az osztályba egy $(n+1)$-edik tanulót „üres” néven:

```
...
ha n%2 == 0 akkor
   tanulószám = n
különben
   tanuló[n+1] = "üres"
   tanulószám = n+1
vége ha
...
```

Generáljuk a tanulók *sorszámainak* összes permutációit, és a `kiír` eljárást a következőképpen módosítjuk (a `megoldás` és az `ígéretes` függvények azonosak a permutációk generálásánál bemutatott változatokkal):

```
diákok(x[],tanuló[],tanulószám,k)
   ha megoldás(tanulószám,k) akkor
      kiír(x,tanuló,k)
   különben
      minden x[k+1]=1,tanulószám végezd
         ha ígéretes(x,k+1) akkor
            diákok(x,tanuló,tanulószám,k+1)
         vége ha
      vége minden
   vége ha
vége diákok

kiír (x[],tanuló[],k)
   minden i=1,k/2 végezd
      ki: "Az ",i," padban ",tanuló[x[2*i-1]]," és ",
      tanuló[x[2*i]]," tanulók ülnek"
   vége minden
vége kiír
```

Megjegyzés: A `kiír` eljárás meghívásakor `k` éppen `tanulószám`-mal lesz egyenlő. A `tanuló` tömbben tároltuk a tanulók nevét.

A backtracking eljárás meghívása: `diákok(x,tanuló,tanulószám,0)`

2.3. Urna: Adott egy urna, amelyben n piros és m fekete golyó található. Generáljuk az összes lehetőséget, ahogyan $p (p \leqslant n+m)$ golyó kivehető az urnából!

Megoldás: A feladat értelmezhető az $\{0,1\}$ halmaz p-szeres Descartes-szorzat azon elemeinek az előállításaként, amelyek nem tartalmaznak n-nél több 0-t és m-nél több 1-est.

Első megközelítésben megoldhatnánk a feladatot úgy is, hogy előállítjuk a Descartes-szorzat összes elemét, és a `kiír` eljárásban kiszűrjük közülük a helyeseket. Azonban hatékonyabb algoritmust kapunk, ha megpróbáljuk csak a helyes Descartes-szorzat elemeket generálni. Ennek egyik módja, ha csak akkor fogadunk el ígéretesnek egy elemet (0-t vagy 1-est) egy adott k-adik szinten, ha nincs belőlük már n, illetve m darab az előző szinteken. Ezért célszerű lehet mindig átadni a következő szintnek az eddigi 0-k, illetve 1-esek számát.

```
urna(x[],n,m,p,k,nulla,egy)
   ha k == p akkor
      kiír(x,k)
   különben
```

```
        ha nulla < n akkor
           x[k+1] = 0
           urna(x,n,m,p,k+1,nulla+1,egy)
        vége ha
        ha egy < m akkor
           x[k+1] = 1
           urna(x,n,m,p,k+1,nulla,egy+1)
        vége ha
     vége ha
vége urna
```

Mivel egy kételemű halmaz Descartes-szorzatáról van szó, lemondtunk a **minden** ciklus használatáról. Továbbá egyszerűbbnek találtuk azt is, hogy az ígéretesség feltételét ide a backtracking (urna) eljárásba építsük bele. A kiír eljárásban a következőképpen valósítható meg a megoldásvektorok dekódolása:

```
kiír(x,k)
   minden i=1,k végezd
      ha x[i] == 0 akkor
         ki: "piros"
      különben
         ki: "fekete"
      vége ha
   vége minden
vége kiír
```

A backtracking eljárás meghívása: urna(x,n,m,p,0,0,0)

2.4. Pénzérmék: Adottak az a_1, a_2, ..., a_n értékű pénzérmék, mindenik típusból bármennyi. Írjuk ki az összes lehetséges módot, ahogyan egy s összeg kifizethető ezen pénzérmék segítségével.

Első megoldás:

I. Hogyan kódolhatók a feladat megoldásai vektorokként?

Legyen a következő példa: $n = 3$, $a = \{2, 3, 5\}$, $s = 10$ (2.2. táblázat).

Megjegyzések:

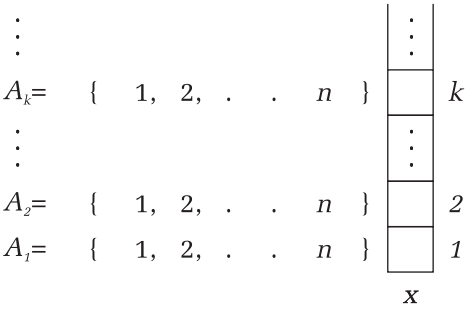
1. A megoldások kódolásánál a pénzérmék sorszámát használtuk.
2. Ezen kódolási mód mellett a megoldásvektorok különböző hosszúságúak.

Tehát a megoldásvektorok, az $\{1, 2, \ldots, n\}$ halmaz elemeiből kialakítható azon *monoton növekvő* sorozatok, amelyeknek megfelelő pénzérmék összege s. A monoton növekvő kitétellel kizárjuk egyes megoldások többszöri előállítását.

2.2. táblázat.

Pénzérmék	A megoldások kódvektorai	
$\{2,2,2,2,2\}$	$(1,1,1,1,1)$	az első pénzérméből használok ötöt
$\{2,2,3,3\}$	$(1,1,2,2)$	használok kettőt az első pénzérméből és kettőt a másodikból
$\{2,3,5\}$	$(1,2,3)$	mindenikfajta pénzérméből használok egyet-egyet
$\{5,5\}$	$(3,3)$	használok kettőt a harmadik pénzérméből

II–IV. A feladat alapábrája:



2.9. ábra.

Mikor ígéretes a k-adik szintre javasolt `x[k]` elem? Ha az előző szinteken ígéretesnek elfogadott elemeknek megfelelő részösszeghez hozzáadva a neki megfelelő pénzértéket, *nem haladjuk túl* az s összeget.

Milyen feltétel mellett jelenthetjük ki, hogy a k-adik szinten ígéretesnek elfogadott pénzérmével egyben megoldáshoz is jutottunk? Ha *pontosan* az s összegre pótolta ki az előző szinteknek megfelelő részösszeget!

Tekintettel arra, hogy az ígéretesség, illetve annak a feltétele, hogy az eredmény megoldás, függ az addig összegyűlt részösszegtől, előnyös, ha mindenik szint megkapja paraméterként ezt az értéket.

Figyelembe véve az alapábrát, és azt, hogy a megoldásvektorok elemei monoton növekvő sorrendben követik egymást, valamint a fenti észrevételeket, az alábbi backtracking eljáráshoz jutunk:

```
fizetés1(x[],a[],n,s,k,részösszeg)
   ha részösszeg == s akkor
      kiír(x,a,k)
```

```
        különben
           minden x[k+1]=x[k],n végezd
              ha részösszeg + a[x[k+1]] <= s akkor
                 fizetés1(x,a,n,s,k+1,részösszeg+a[x[k+1]])
              vége ha
           vége minden
        vége ha
     vége fizetés1
```

V. A `kiír` függvényben dekódoljuk a megoldásvektorokat.

```
kiír(x[],a[],k)
   minden i=1,k végezd
      ki: a[x[i]]
   vége minden
vége kiír
```

A backtracking eljárás meghívása: `fizetés1(x,a,n,s,0,0).` Ugyanakkor a hívás előtt gondoskodunk az `x[0]` elemről is: `x[0]=0.` Így biztosítjuk, hogy az `x[1]` elemben az $\{1, 2, \ldots, n\}$ halmaz összes eleme elő legyen állítva.

Második megoldás:
I. Egy másik lehetősége a megoldások kódolásának az alábbi.
Legyen ugyanaz a példa: $n = 3$, $a = \{2, 3, 5\}$, $s = 10$.

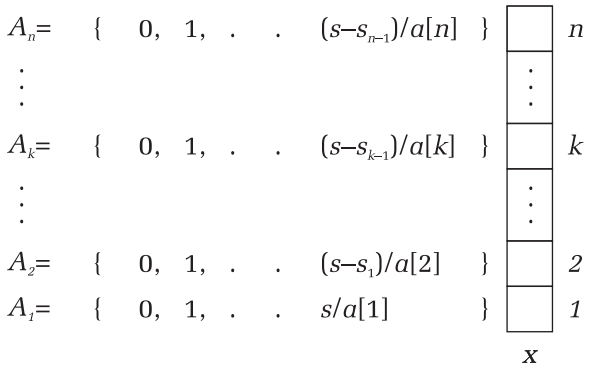
2.3. táblázat.

Pénzérmék	A megoldások kódvektorai	
$\{2, 2, 2, 2, 2\}$	$(5, 0, 0)$	az első pénzérméből öt, a másodikból és harmadikból egy sem
$\{2, 2, 3, 3\}$	$(2, 2, 0)$	kettő az első és a második pénzérméből, egy sem a harmadikból
$\{2, 3, 5\}$	$(1, 1, 1)$	mindhárom pénzérméből egy-egy
$\{5, 5\}$	$(0, 0, 2)$	kettő a harmadik pénzérméből

Általánosan: A megoldásvektorok k-adik eleme azt tárolja, hogy hány darabot használtunk a k-adik pénzérméből az s összeg kialakításában. Mindenik pénzérméből legkevesebb nulla és legtöbb $s/a[k]$ (ahol $k = 1, 2, \ldots, n$) darabot. Sőt, ha figyelembe vesszük, hogy amikor a k-adik pénzérméhez érkezünk, már rendelkezésre áll egy s_{k-1} részösszeg, amely az $1, \ldots, k-1$ pénzérme típusokból áll össze, akkor a k-adik pénzérméből

már csak legtöbb $(s - s_{k-1})/a[k]$ darab használható. Mindez a következő alapábrához vezet.

II. A feladat alapábrája:



2.10. ábra.

```
fizetés2(x[],a[],n,s,k,részösszeg)
   ha megoldás(n,k) akkor
      kiír(x,a,k)
   különben
      minden x[k+1]=0,(s-részösszeg)/a[k+1] végezd
         ha ígéretes(x,a,n,s,k+1,részösszeg) akkor
            fizetés2(x,a,n,s,k+1,részösszeg+x[k+1]*a[k+1])
         vége ha
      vége minden
   vége ha
vége fizetés2
```

A rekurzív eljárás k-adik szintre való meghívásakor a `részösszeg` paraméter az s_{k-1} értéket kapja meg.

III. Mikor ígéretes a k-adik szintre javasolt $x[k]$ elem? Ha az addigi részösszeghez hozzáadva az $x[k]$ darab $a[k]$ értékű pénzérmének megfelelő összeget ($x[k]$ * $a[k]$), *nem haladjuk túl* az s összeget! Sőt, az n-edik szinten még igényesebbeknek kell lennünk, és csak akkor fogadhatunk el ígéretesnek egy elemet, ha *pontosan* az s összeghez vezet. Ha az n-edik szinten egy s-nél kisebb összegre is rábólintana az ígéretes függvény, tekintetbe véve, hogy ez nem megoldás, a backtracking eljárás úgy reagálná le ezt a helyzetet, hogy fellépne az $(n+1)$-edik szintre, holott nincs $(n+1)$-edik pénzérme.

```
ígéretes(x[],a[],n,s,k,részösszeg)
   return (k < n és részösszeg+x[k]*a[k] <= s) vagy
             (k == n és részösszeg+x[k]*a[k] == s)
vége ígéretes
```

IV. Milyen feltétel mellett jelenthetjük ki, hogy a k-adik szinten ígéretesnek elfogadott pénzérme számmal egyben megoldáshoz is jutottunk? Tekintettel az ígéretes függvény megerősített feltételére, ha fel tudunk lépni az n-edik szintre, akkor megoldást találtunk.

```
megoldás(n,k)
   return k == n
vége megoldás
```

A backtracking eljárás meghívása: `fizetés2(x,a,n,s,0,0)`

2.5. Szuperprímek: Generáljuk az összes n számjegyű szuperprímet. Egy természetes számot szuperprímnek nevezünk, ha prím és a számjegyei jobbról balra sorrendben történő egyenkénti levágásával nyert számok (tekintsük ezeket a szám prefixeinek) is mind prímek. Például 239 szuper prím, mert 239, 23 és 2 mind prímek.

Első megoldás:

I. Hogyan kódolhatók a feladat megoldásai?

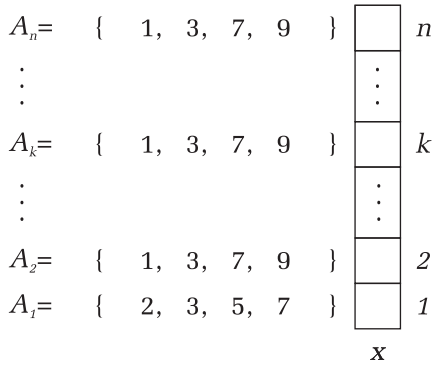
Mivel a megoldások n számjegyű természetes számok, felfoghatók úgy mint n elemű vektorok, amelyeknek elemei a $\{0, 1, \ldots, 9\}$ halmazból származnak. Bár már ezen megállapítás alapján is felrajzolható az alapábra, és megoldható a feladat, vegyük észre, hogy hatékonyabb algoritmushoz jutunk, ha további megszorításokat teszünk. Tekintettel a szuperprímek értelmezésére, a megoldásvektorok nem tartalmazhatnak csak 5-től különböző páratlan számjegyeket, kivéve az első szintet, ahol viszont csak a $2, 3, 5, 7$ értékek jelenhetnek meg.

II–V. A feladat alapábráját lásd a 2.11. ábrán.

```
szuperprím1(x[],n,k,prefix)
   ha k == n akkor
      ki: prefix
   különben
      ha k == 1 akkor
         x[k+1] = 2
         szuperprím1 (x,n,k+1,2)
         x[k+1] = 3
         szuperprím1 (x,n,k+1,3)
         x[k+1] = 5
```

```
            szuperprím1 (x,n,k+1,5)
            x[k+1] = 7
            szuperprím1 (x,n,k+1,7)
         különben
            x[k+1] = 1
            ha prím(prefix*10+1) akkor
               szuperprím1(x,n,k+1,prefix*10+1)
            vége ha
            x[k+1] = 3
            ha prím(prefix*10+3) akkor
               szuperprím1(x,n,k+1,prefix*10+3)
            vége ha
            x[k+1] = 7
            ha prím(prefix*10+7) akkor
               szuperprím1(x,n,k+1,prefix*10+7)
            vége ha
            x[k+1] = 9
            ha prím(prefix*10+9) akkor
               szuperprím1(x,n,k+1,prefix*10+9)
            vége ha
         vége ha
      vége ha
   vége szuperprím1
```

Mivel az A_k halmazok csupán négyeleműek és nem egy pontos szabályosság szerint követik egymást, nem tettük ciklusba a rekurzív hívásokat. Továbbá, tekintettel arra, hogy az A_1 halmaz eltér a többitől, ezt az esetet külön kezeltük.



2.11. ábra.

Mikor ígéretes egy számjegy az x verem $(k+1)$-edik szintjén? Ha az épülőben lévő számnak a verem k-adik szintjéig kialakított prefixét egy következő prímprefixszé egészíti ki. Ahhoz viszont, hogy ezt ellenőrizni tudjuk (prím függvény), szükséges az illető vektorprefixet először számmá alakítani. Azért, hogy ne kelljen ezt újra meg újra megtenni, mindenik szinti függvényhívás paraméterként megkapja az addig kialakult prefix értékét számként is. Így egy következő prefix számmá alakítása csupán annyiból áll, hogy az aktuális prefix végéhez egy következő számjegyet ragasztunk. Úgy is fogalmazhatnánk, hogy a megoldásvektorokkal párhuzamosan számként is felépítjük az eredményeket.

A backtracking eljárás meghívása: `szuperprím1(x,n,0,0)`

Második megoldás:

Bár eddig minden esetben az x tömbben építettük fel a megoldásokat, nem szükséges ehhez mereven ragaszkodni. Például a jelen feladat esetében szükségtelen a megoldásokat számjegysorozatként is és számként is felépíteni. Elvégre a számok önmagukban is felfoghatók vektorként, anélkül hogy szétbontanánk számjegyeikre.

```
szuperprím2(k,n,prefix)
   ha k == n akkor
      ki: prefix
   különben
      ha k == 0 akkor
         szuperprím2(k+1,n,2)
         szuperprím2(k+1,n,3)
         szuperprím2(k+1,n,5)
         szuperprím2(k+1,n,7)
      különben
         ha prím(prefix*10+1) akkor
            szuperprím2(k+1,n,prefix*10+1)
         vége ha
         ha prím(prefix*10+3) akkor
            szuperprím2(k+1,n,prefix*10+3)
         vége ha
         ha prím(prefix*10+7) akkor
            szuperprím2(k+1,n,prefix*10+7)
         vége ha
         ha prím(prefix*10+9) akkor
            szuperprím2(k+1,n,prefix*10+9)
         vége ha
      vége ha
```

```
        vége ha
    vége szuperprím2
```

A backtracking eljárás meghívása: `szuperprím2(0,n,0)`

2.6. 1/2 halmazok: Határozzuk meg az összes p elemű halmazt, amely rendelkezik az alábbi tulajdonságokkal:

- minden eleme n ($n \leqslant 32$) számjegyű;
- az elemek csak 1-es és 2-es számjegyeket tartalmaznak;
- bármely két eleme pontosan m ($m < n$) pozíción tartalmaz azonos számjegyeket;
- a számok egyetlen pozícióban sem tartalmaznak azonos számjegyeket.

Megoldások:

I. Hogyan kódolhatók a feladat megoldásai?

Legyen a következő példa: $n = 3$, $m = 1$, $p = 3$.

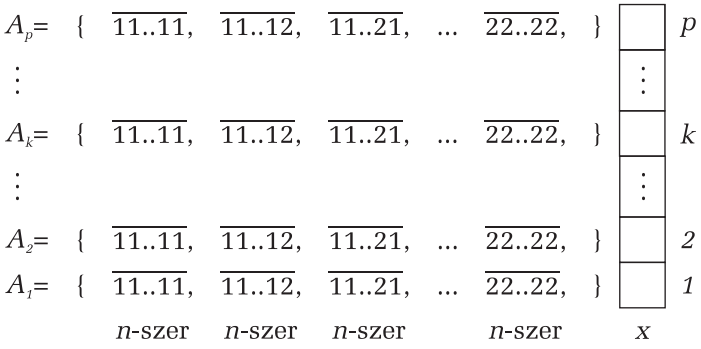
2.4. táblázat.

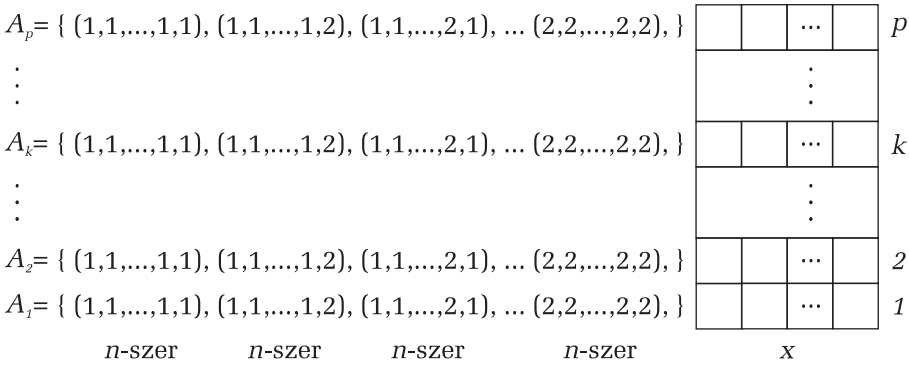
Megoldások	$n \leqslant 10$	bármely n-re	$10 < n \leqslant 32$
$\{112, 121, 211\}$	$(112, 121, 211)$	$((1,1,2),(1,2,1),(2,1,1))$	$(1, 2, 4)$
$\{221, 212, 122\}$	$(221, 212, 122)$	$((2,2,1),(2,1,2),(1,2,2))$	$(6, 5, 3)$
$\{111, 122, 221\}$	$(111, 122, 221)$	$((1,1,1),(1,2,2),(2,2,1))$	$(0, 3, 6)$
$\{111, 122, 212\}$	$(111, 122, 212)$	$((1,1,1),(1,2,2),(2,1,2))$	$(0, 3, 5)$
$\{111, 221, 212\}$	$(111, 221, 212)$	$((1,1,1),(2,2,1),(2,1,2))$	$(0, 6, 5)$
$\{222, 211, 112\}$	$(222, 211, 112)$	$((2,2,2),(2,1,1),(1,1,2))$	$(7, 4, 1)$
$\{222, 211, 121\}$	$(222, 211, 121)$	$((2,2,2),(2,1,1),(1,2,1))$	$(7, 4, 2)$
$\{222, 112, 121\}$	$(222, 112, 121)$	$((2,2,2),(1,1,2),(1,2,1))$	$(7, 1, 2)$

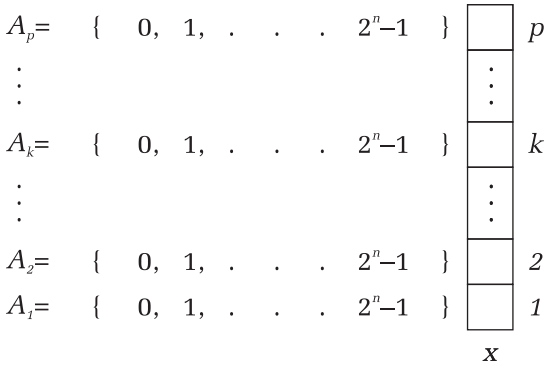
A fenti példa bemutatja az eredmény kódolásának három különböző módját. Ha $n \leqslant 10$, akkor a megoldáshalmazok elemei eltárolhatók számként a számítógép memóriájában (például, ha C programozási nyelven történik a kódolás, használhatunk unsigned long int típust), és nincs szükség különösebb kódolásra (második oszlop). Ha $n \leqslant 32$, a megoldáshalmazok elemeit felfoghatjuk úgy, mint természetes számok n biten való belső ábrázolásait (az 1-es számjegynek a 0-ás bitet, a 2-esnek az 1-es bitet feleltettük meg) (negyedik oszlop). Bármely n esetén tekinthetjük a megoldásokat olyan p elemű tömböknek, amelyeknek elemei n elemű, egyest és kettest tartalmazó tömbök (harmadik oszlop).

Egy megoldáson belül az elemeket növekvő sorrendben generáljuk ahhoz, hogy elkerüljük ugyanazon megoldás többszöri előállítását.

II. A feladat alapábrája a három esetben:







2.12. ábra.

```
számok(x[],n,p,m,utolsó,k)
   ha megoldás(p,k) akkor
      kiír(x,n,k)
   különben
      x[k+1] = következő(x[k])
      amíg x[k+1] <= utolsó végezd
         ha ígéretes(x,n,m,k+1,részösszeg) akkor
            számok(x,n,p,m,utolsó,k+1)
         vége ha
         x[k+1] = következő(x[k+1])
      vége amíg
   vége ha
vége számok
```

Az `utolsó` nevű paraméterben az eljárás az A_k halmazok utolsó (legnagyobb) elemét kapja meg. Az első esetben ez a $\overline{22..22}$, n számjegyű szám, a másodikban a `(2,2,...,2)` n elemű vektor és a harmadikban a $2^n - 1$ érték.

Az első két esetben nem triviális, hogy miként lehet az A_k halmaz elemeit, az `x` tömb k-adik szintjére, egymást követően – egyikből a másikat – előállítani. Az alapötlet az, hogy szimuláljuk a számok eggyel való növelését „kettes számrendszerben” (jobbról balra haladva végig a számon, amíg az első 1-est találjuk, minden 2-est 1-re állítunk, majd ezt 2-esre növeljük). Az alábbiakban megadjuk a `következő` függvényt a három esetre. A második esetben az `x` tömb elemei maguk is n elemű vektorok (ami azt jelenti, hogy `x` alapvetően kétdimenziós tömb, és az `utolsó` nevű paramétere egydimenziós). Ezért kap a `következő2` eljárás paraméterként egy egydimenziós tömböt és az n értékét. Ebben az esetben természetesen módosul a `következő` függvénynek a számok főeljárásbeli meghívása is (például: `x[k+1]=következő2(x[k],n)`), és az **amíg** ciklus feltételében az összehasonlítás sem oldható meg egyszerű összehasonlítási operátorral (használhatunk egy összehasonlító függvényt az `x[k+1]` és `utolsó` vektorok összehasonlítására).

```
következő1(w)
   v = w
   h = 1
   amíg v > 0 és v%10 == 2 végezd
      w = w - h
      h = h*10
      v = v/10
   vége amíg
```

```
   ha v > 0 akkor
      w = w + h
   vége ha
   return w
vége következő1

következő2(a[],n)
   minden i=n,1,-1 végezd
      ha a[i] == 2 akkor
         a[i] = 1
      különben
         a[i] = 2
         return
      vége ha
   vége minden
   return a
vége következő2

következő3(w)
   return w+1
vége következő3
```

III. Az A_k halmaz valamely eleme akkor ígéretes az x tömb k-adik szintjén, ha pontosan m pozíción talál az alábbi szinteken ígéretesnek elfogadott mindenik elemmel. Az utolsó esetben ez azt jelenti, hogy a tömbelemek n biten való belső ábrázolásai találnak m bitpozíción. Sőt, $k = p$ esetén ezenfelül annak a feltételnek is teljesülnie kell, hogy egyetlen pozícióban se tartalmazza a tömb mindenik eleme ugyanazt a számjegyet, illetve bitet.

```
ígéretes(x[],n,m,k)
   minden i=1,k-1 végezd
      ha talál(x[k],x[i],n) ≠ m akkor
         return HAMIS
      vége ha
   vége minden
   ha ugyanaz(x,n,k) akkor
      return HAMIS
   vége ha
   return IGAZ
vége ígéretes
```

A talál függvény meghatározza, hogy n-ből hány pozíción talál az első két paramétere. Az ugyanaz függvénynek azt kell ellenőriznie, hogy

van-e olyan pozíció, amelyben az x tömbnek mind a k eleme ugyanazt a számjegyet, illetve bitet tartalmazza. Ezen egyszerű függvények megírását az olvasóra hagyjuk.

IV. Nyilván mindhárom esetben akkor jutottunk megoldáshoz, ha $k = p$.

V. A kiír eljárás csak a harmadik esetben tartalmaz különösebb dekódolást, amikor is a megoldásvektorok elemeinek n biten való belső ábrázolásait kell megjelenítenünk oly módon, hogy minden 0 bit helyett 1-es számjegyet, és minden 1-es bit helyett 2-s számjegyet írunk ki.

```
kiír3(x[],n,k)
   minden i=1,k végezd
      bitenként(x[i],n)
   vége minden
vége kiír3

bitenként(w,n)
   ha n > 0 akkor
      bitenként(w/2,n-1)
      ki: w%2 + 1
   vége ha
vége bitenként
```

A backtracking eljárás meghívása:

Megjegyzések:

1. Az x[0] tömbelemnek olyan kezdőértéket adunk, hogy x[1]-ben az A_1 halmaz összes eleme előállításra kerüljön.
2. A következő1 és következő2 eljárásokat úgy írtuk meg, hogy az A_k halmazok utolsó elemét a halmazok első eleme kövesse, például: következő1($\overline{22..22}$) = $\overline{11..11}$

Ezért inicializáltuk ezen esetekben x[0]-t az utolsó változó, illetve vektor értékével.

1. eset (az utolsó változóban a $\overline{22..2}$, n számjegyű számot alakítjuk ki)

```
utolsó = 0
minden i=1,n végezd
   utolsó = utolsó*10+2
vége minden
x[0] = utolsó
számok1(x,n,p,m,utolsó,0)
```

2. eset (az utolsó tömbben a `(2,2,...,2)`, n elemű vektort alakítjuk ki)

```
minden i=1,n végezd
   utolsó[i] = 2
   x[0][i] = 2
vége minden
számok1(x,n,p,m,utolsó,0)
```

3. eset (az utolsó változóban a $2^n - 1$ értéket alakítjuk ki)

```
utolsó = 1
minden i=1,n végezd
   utolsó = utolsó*2
vége minden
utolsó = utolsó-1
x[0] = -1
számok1(x,n,p,m,utolsó,0)
```

2.7. Békák: Legyen egy $s[1 \,..\, 2n+1]$ karakterlánc, amelyben az első n elem 0, az $(n+1)$-edik szóköz, a többi pedig 1-es. Generáljuk az összes lehetőséget, ahogyan a nullások (fehér békák) helyet cserélhetnek az egyesekkel (fekete békák), a következő feltételek mellett:

- a fehér békák csak balról jobbra, a feketék pedig csak jobbról balra szökdöshetnek.
- egy béka akkor haladhat előre, ha előtte szóköz van, vagy ha az előtte lévő béka előtt szóköz van, ez utóbbi esetben átugorhatja az előtte lévő békát.

Megoldás:

I. Hogyan kódolhatók a feladat megoldásai?

Figyeljük meg, hogy a feladat úgy is felfogható, mintha a szóköz ugrálna. Jelöljük p-vel a szóköz pozícióját a karaktertömbben. Általában négyféle lépés lehetséges:

1. A szóköz helyet cserél a tőle jobbra lévő békával (a $(p+1)$-edik pozícióban lévővel), amennyiben létezik ilyen béka ($p \leqslant 2n$) és az fekete.
2. A szóköz helyet cserél a $(p+2)$-edik pozícióban lévő békával, amennyiben létezik ilyen béka ($p \leqslant 2n-1$) és az fekete.
3. A szóköz helyet cserél a tőle balra lévő békával (a $(p-1)$-edik pozícióban lévővel), amennyiben létezik ilyen béka ($p \geqslant 2$) és az fehér.

4. A szóköz helyet cserél a $(p-2)$-edik pozícióban lévő békával, amennyiben létezik ilyen béka ($p \geqslant 3$) és az fehér.

A lépéseket a sorszámukkal kódoljuk, és egy megoldást úgy fogunk fel, mint a szóköznek egy olyan lépéssorozatát, amely a békák helycseréjét eredményezi.

Példa $n = 3$ esetén:

2.5. táblázat.

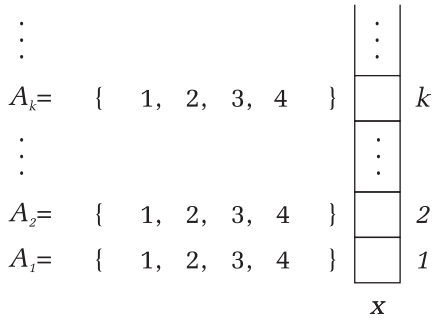
A megoldás mint állapotsor	A megoldás mint lépéssorozat (a megoldás kódja)
000 111	3
00 0111	2
0010 11	1
00101 1	4
001 101	4
0 10101	3
010101	2
10 0101	2
1010 01	2
101010	3
10101 0	4
101 100	4
1 10100	1
11 0100	2
1110 00	3
111 000	

Tehát a feladat megoldásai olyan vektorok formájában kódolhatók, amelyeknek elemei az {1, 2, 3, 4} halmazból származnak.

II. A 2.13. ábra a feladat alapábrája.

Hogy mikor ígéretes (lehetséges) egy lépés, azt már a lépéstípusok leírásánál megfogalmaztuk. Nyilván akkor vagyunk megoldásnál, ha a békák helyet cseréltek és a szóköz középen van.

```
békák(x[],n,s[],k,p)
   ha megoldás(n,s) akkor
      kiír(x,k)
```



2.13. ábra.

```
különben
   ha p <= 2n és s[p+1] == '1' akkor
      x[k+1]=1
      s[p]='1'
      s[p+1]=' '
      békák(x,n,s,k+1,p+1)
      s[p]=' '
      s[p+1]='1'
   vége ha
   ha p <= 2n-1 és s[p+2] == '1' akkor
      x[k+1]=2
      s[p]='1'
      s[p+2]=' '
      békák(x,n,s,k+1,p+2)
      s[p]=' '
      s[p+2]='1'
   vége ha
   ha p >= 2 és s[p-1] == '0' akkor
      x[k+1]=3
      s[p]='0'
      s[p-1]=' '
      békák(x,n,s,k+1,p-1)
      s[p]=' '
      s[p-1]='0'
   vége ha
   ha p >= 3 és s[p-2] == '0' akkor
      x[k+1]=4
      s[p]='0'
      s[p-2]=' '
      békák(x,n,s,k+1,p-2)
```

```
            s[p]=' '
            s[p-2]='0'
         vége ha
      vége ha
   vége békák

   megoldás(n,s[])
      minden i=1,n+1 végezd
         ha i <= n és s[i] ≠ '1' akkor
            return HAMIS
         vége ha
         ha i == n+1 és s[i] ≠ ' ' akkor
            return HAMIS
         vége ha
      vége minden
      return IGAZ
   vége megoldás
```

A feladat állapotát egy adott pillanatban az s tömb alakja határozza meg. Memóriatakarékossági szempontok miatt az s tömböt és az n-t a békák `eljárás` cím szerinti paraméterekként kapja meg. Ezzel szemben a szóköz aktuális helyét mindenik szinti hívásnak érték szerint adjuk át (így nem kell minden békaugrás előtt újra meg újra megkeresni az s tömbben a szóköz helyét). Az a tény, hogy az s tömb cím szerint lett átadva, szükségessé tette, hogy valahányszor visszalépünk, először rekonstruáljuk az s tömböt (a rekurzív hívások után), és csak azután lépjünk egy újabb ígéretes irányba. Más szóval, a rekurzió visszaútjain visszaugráltatjuk a békákat. Úgy is mondhatnánk, hogy azzal párhuzamosan, ahogy a backtracking eljárás bejárja mélységében a feladathoz rendelhető fastruktúra ígéretes részfáját, az s globális tömbben szimuláljuk a békák előre-visszaugrálását.

A `kiír` eljárásnak rekonstruálnia kell az x tömb alapján a megoldást mint állapotsort. Ez maradjon gyakorlatként az olvasó részére.

2.8. Labirintus: Adott egy labirintus az $a[1..n][1..m]$ bináris tömbben (1-essel kódoljuk a falat, 0-val a folyosót). Adott még egy személy pozíciója a labirintusban (az x koordináta a sorindex, az y az oszlopindex). Generáljuk az összes hurokmentes utat, amelyen a személy kijuthat a labirintusból.

Példa: $n = 5$, $m = 6$, $x = 3$, $y = 3$ (2.14. ábra).

Megoldás: Bár a feladatot az előző séma szerint is felfoghatnánk (kódoljuk a lépéseket: 1 – felfelé, 2 – jobbra, 3 – lefelé, 4 – balra),

a

	0	1	2	3	4	5	6	7
0	2	2	2	2	2	2	2	2
1	2	1	1	1	0	1	1	2
2	2	1	0	0	0	0	1	2
3	2	0	1	**e**	1	0	1	2
4	2	0	1	0	0	0	0	2
5	2	0	0	1	0	1	0	2
6	2	2	2	2	2	2	2	2

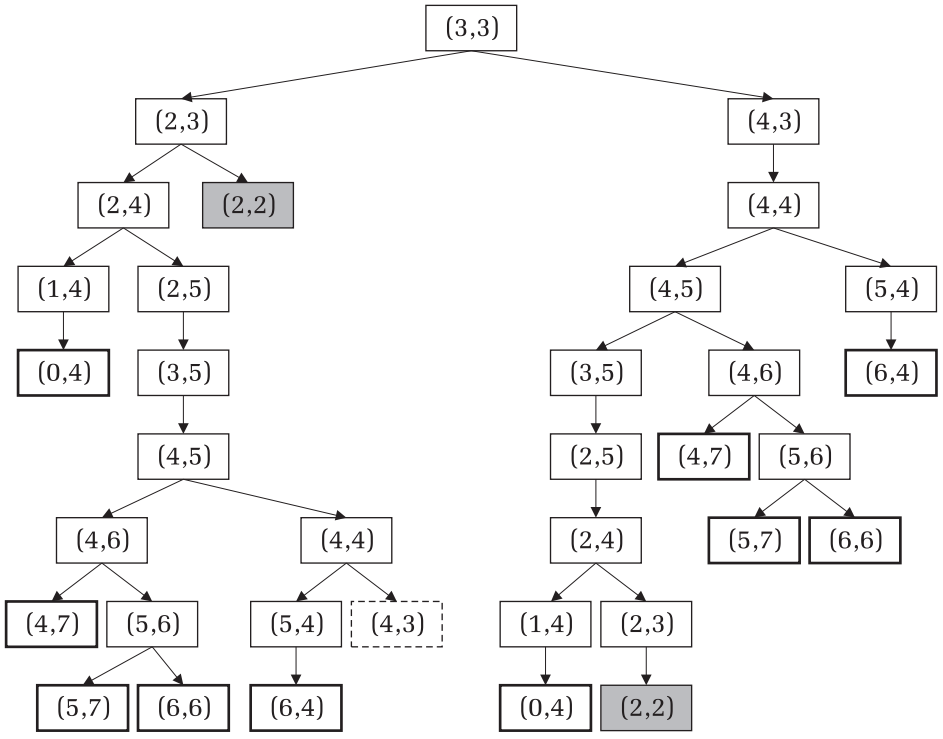
2.14. ábra.

ez alkalommal a megoldást egy az egyben úgy tekintjük, mint egy fa mélységi bejárását. Ahhoz, hogy könnyebben érzékelhessük, ha kijutottunk a labirintusból, szegélyezzük a tömböt egy kettesekből álló kerettel (0-dik sor, 0-dik oszlop, $(n+1)$-edik sor, $(n+1)$-dik oszlop). A fa gyökere a személy kezdeti pozíciója (x, y), az első szinti csomópontok azok a helyek lesznek, ahova első lépésben léphet, a második szintiek azok, ahova ezekből tovább léphet stb. Akkor jutottunk levélhez, ha kiléptünk a keretre (megoldáslevél) vagy ha „zsákutcába" jutottunk. Úgy fogjuk elkerülni a hurkokon való végtelen körbejárást, hogy miközben haladunk előre a fában, „befalazzuk" magunk mögött az utat, visszalépéskor pedig helyreállítjuk a „befalazott" elemeket. Alább megadjuk a feladathoz rendelhető fastruktúrát, a fenti példára vonatkoztatva. A megoldásleveleket vastagított kerettel rajzoltuk, a zsákutca-leveleket pedig besatíroztuk. Szaggatott vonallal kereteztük azokat a leveleket, ahonnan a továbblépés hurokba vezet.

Emlékezzünk, hogy a mélységi bejárás lényege az, hogy minden pozícióból, ahova eljutottunk, először az első irányban (legyen ez felfele) próbálunk továbblépni. Ha egy adott pillanatban az első irányban fal van, próbálkozzunk a második irányban (jobbra) „egérutat" találni, és így tovább, míg végül abba a helyzetbe nem jutunk, hogy vagy mind a négy irányban fal van, vagy találtunk egy kiutat (kiléptünk a keretre). Ha a fát képzeljük magunk elé, úgy is mondhatnánk, hogy a bal oldali ágán lemegyünk a lehető legmélyebbre (míg egy levélhez nem jutunk). Mit teszünk, ha egy adott pozícióból már nem tudunk továbblépni? Visszalépünk oda, ahonnan ideléptünk (az apacsomópontba), ahol folyatjuk egy következő irányban a labirintusból való kivezető utak keresését. Ha

abba a helyzetbe kerülünk, hogy egy (i, j) pozícióból már minden lehetséges irányban sorra elmentünk ($(i-1, j)$, $(i, j+1)$, $(i+1, j)$, $(i, j-1)$) és visszajöttünk, akkor az (i, j) pozícióból is visszalépünk. Az algoritmus akkor ér véget, amikor úgy értünk vissza a személy kezdeti helyére, hogy már nincs olyan irány, amit ne próbáltunk volna ki.

A rekurzív backtracking eljárás a személy aktuális pozícióját (i, j) érték szerinti paraméterként kapja meg, valamint azt, hogy hányadik állomásnál (k) tartunk az aktuális úton. Az aktuális utat egy d nevű egydimenziós tömbben regisztráljuk (ez fogja betölteni az általános sémabeli x tömb szerepét), amelynek elemei az út menti tömbelemek koordinátáit (s a sor indexe, o az oszlop indexe) tárolják. Az a és d tömböket a labirintuseljárás cím szerint kapja meg.



2.15. ábra.

Figyeljük meg, hogy a tömb bármely (i, j) pozíciójában, ahova eljutottunk (más szóval a fa minden egyes csomópontjában) alapvetően ugyanazt kell tennünk:

a) Ellenőrizzük, hogy a kereten vagyunk-e. Ha igen, akkor kiírjuk a d tömbben regisztrált megoldást.
b) Ha nem vagyunk még a kereten, regisztráljuk a d tömbben az aktuális út k-adik állomásaként az (i, j) pozíciót, „befalazzuk" az $a[i][j]$ elemet, ahol éppen állunk, és megpróbálunk – rekurzív hívások által – sorra mind a négy szomszédos irányban továbblépni.
c) Miután a rekurzió minden ígéretes irányból visszahozott, mielőtt az aktuális (i, j) pozícióból is visszaléptetne, helyreállítjuk a „befalazott" $a[i][j]$ elemet.

```
labirintus(a[][],d[],i,j,k)
   ha a[i][j] == 2 akkor
      minden r=1,k-1 végezd
         ki: d[r].s,d[r].o
      vége minden
   különben
      d[k].s=i
      d[k].o=j
      a[i][j]=1
      ha a[i-1][j] == 0 akkor
         labirintus(a,d,i-1,j,k+1)
      vége ha
      ha a[i][j+1] == 0 akkor
         labirintus(a,d,i,j+1,k+1)
      vége ha
      ha a[i+1][j] == 0 akkor
         labirintus(a,d,i+1,j,k+1)
      vége ha
      ha a[i][j-1] == 0 akkor
         labirintus(a,d,i,j-1,k+1)
      vége ha
      a[i][j]=0
   vége ha
vége labirintus
```

2.9. Fénykép: Adott egy fekete-fehér kép, amelyet az $a[1..n][1..m]$ bináris mátrixban tároltunk (a 0 a fehér tartományokat, az 1 a fekete foltokat /tárgyakat/ ábrázolja). Állapítsuk meg a képen található tárgyak számát!

Az alábbi példákban mindkét fénykép két tárgyat tartalmaz:

```
1000      110001
0100      100101
0001      111001
```

Megoldás: Végigpásztázzuk a képet (fentről lefele, sorról sorra), és valahányszor elérünk egy tárgyat (találunk egy 1-es értékű elemet), egy backtracking eljárással – az illető 1-esből kiindulva – letöröljük (átállítjuk az 1-eseit 0-ra), majd folytatjuk a pásztázást, további tárgyak után kutatva. Nyilván annyi tárgy van a képen, ahányat letöröltünk. A backtracking típusú töröl eljárás tanulmányozását az olvasóra hagyjuk.

Megjegyzés: Azért, hogy a töröl eljárás minden tárgyat úgy kezelhessen mint 0-kkal körülvett összefüggő 1-es területet, a mátrixot egy 0-kból álló kerettel szegélyeztük.

```
kép(a[][],n,m)
   minden i=0,n+1 végezd
      a[i][0]=0
      a[i][m+1]=0
   vége minden
   minden j=0,m+1 végezd
      a[0][j]=0
      a[n+1][j]=0
   vége minden
   k=0
   minden i=1,n végezd
      minden j=1,m végezd
         ha a[i][j]==1 akkor
            töröl(a,i,j)
            k=k+1
         vége ha
      vége minden
   vége minden
   ki: k," tárgy van a képen"
vége kép

töröl(a[][],i,j)
   a[i][j]=0
   ha a[i-1][j] == 1 akkor
      töröl(a,i-1,j)
   vége ha
   ha a[i-1][j+1] == 1 akkor
      töröl(a,i-1,j+1)
```

```
        vége ha
        ha a[i][j+1] == 1 akkor
           töröl(a,i,j+1)
        vége ha
        ha a[i+1][j+1] == 1 akkor
           töröl(a,i+1,j+1)
        vége ha
        ha a[i+1][j] == 1 akkor
           töröl(a,i+1,j)
        vége ha
        ha a[i+1][j-1] == 1 akkor
           töröl(a,i+1,j-1)
        vége ha
        ha a[i][j-1] == 1 akkor
           töröl(a,i,j-1)
        vége ha
        ha a[i-1][j-1] == 1 akkor
           töröl(a,i-1,j-1)
        vége ha
    vége töröl
```

A töröl eljárást *fill* eljárásnak is nevezik, hiszen alapvetően úgy töröltünk le vele egy tárgyat, hogy a tárgy 1-esei képezte zárt területet kifestettük 0-kkal.

2.4. Kitűzött feladatok

2.1. Zászlók: Generáljuk ki az összes *három*színű zászlót, amelyekben a következő 6 szín (fehér, fekete, piros, kék, zöld, sárga) szerepelhet és amelyek középső színe vagy fekete vagy fehér.

2.2. Összegre bontás: Bontsuk fel az n természetes számot p darab természetes szám összegévé ($p <= n$) az összes lehetséges módon.

2.3. Delegáció: Egy n tagú csoportban a személyek 1-től n-ig vannak megszámozva. Tudva azt, hogy az $1..p$ személyek nők, képezzük az összes olyan k tagú delegációt, amelyben pontosan q nő található.

2.4. Egyenlet: Oldjuk meg a $3x + y + 4xz = 100$ egyenletet a természetes számok halmazán. Írjuk ki az összes lehetséges megoldást.

2.5. „Rendezés”: Képezzük az `a[1..n]` tömb elemeinek az összes olyan átrendezését, amelyekben az `a[i..i+k]` tömbszakasz elemei egymás után szerepelnek az eredeti sorrendben.

2.6. Részcsoportok: Egy n tagú csoportban a személyek 1-től n-ig vannak megszámozva. Generáljuk az összes olyan k tagú részcsoportot, amely:

- tartalmaz adott p személyt;
- nem tartalmaz adott q személyt;
- tartalmaz egy adott személyt, de nem tartalmaz egy másikat;
- tartalmaz egyet adott p személy közül;
- tartalmaz legalább egyet adott p személy közül;
- tartalmaz r személyt adott p közül, de nem tartalmaz másik q személyt.

2.7. Prímek összege: Adott egy természetes szám. Bontsuk az összes lehetséges módokon prímszámok összegére.

2.8. A király és lova: Ismert egy $n \times n$ sakktáblán egy ló és a király pozíciója. A tábla egyes négyzetei égnek, mások pedig nem. Generáljuk az összes lehetőséget, ahogy a ló a királyért mehet, majd visszatér – a hátán a királlyal – eredeti helyére, ha tudjuk, hogy nem léphet égő négyzetre, és ahova egyszer lépett, az a négyzet felgyúl mögötte. Természetesen a ló eredeti pozíciója, illetve a király helye kezdetben nem ég.

2.9. Pontok a síkban: Adott n pont a síkban a koordinátáik által. Képezzük az összes lehetőséget, ahogy a pontok összeköthetők p egyenes által úgy, hogy az egyenesek metszéspontjainak halmaza részhalmaza legyen az n pont halmazának.

2.10. Maximális prefix: Adott egy $n \times m$ méretű mátrix, amelynek elemei karakterek, valamint egy s karakterlánc. Keressük meg a mátrixban s leghosszabb prefixét. A mátrixban a prefix betűi le, fel, bal, jobb irányban követhetik egymást.

2.11. Golyó: Egy $n \times m$ centiméter méretű, téglalap alakú területet úgy fogunk fel, mint egy mátrixot. Az (i, j) pozíciójú elem a megfelelő helyzetű négyzetcentiméter magasságát tárolja. Generáljuk az összes útvonalat, amelyen egy (x, y) pozíciójú golyó kigurulhat a területről.

2.12. Törékeny béke: Legyen n szék és n személy 1-től n-ig megszámozva. Kezdetben az i-edik személy az i-edik széken ül, de minden szomszéd összevesz. Képezzük a személyek összes olyan átültetését, amelyben az ellenségek között legalább egy, de legfennebb két másik személy ül.

2.13. Bitszomszédok: Töltsünk fel egy $2^n \times 2^m$ méretű mátrixot az összes lehetséges módon a $0..2^{n+m}-1$ természetes számokkal úgy, hogy bármely két szomszédos elem (vízszintes és függőleges irányban) bináris ábrázolása pontosan egy pozíción különbözzön.

2.14. Hamilton és Euler: Adott egy n csomópontú gráf. Határozzuk meg az összes Hamilton-kört (minden csomópontot érint egyszer, de csakis egyszer), illetve az összes zárt Euler-vonalat (minden élen áthalad egyszer, de csakis egyszer) a gráfban.

2.15. Kifejezések: Adott egy n elemű egész számsorozat. Helyezzünk a számok közé az összes lehetséges módon $n-1$ operátort a 4 aritmetikai operátor (+,−,·,/) közül úgy, hogy a kapott kifejezés értéke egy adott egész szám legyen.

2.16. Dobozok és tárgyak: Legyen n tárgy 1-től n-ig megszámozva, és legyen n doboz. Tudva azt, hogy egy dobozba legfennebb m tárgy fér bele, generáljuk az összes lehetőséget, ahogy a tárgyak elhelyezhetők a dobozokban.

2.17. Raktár: Egy $n \times m$ méretű mátrix, amelynek elemei az $\{0,1,2,3\}$ halmazból származnak, egy raktárt ábrázol. A 0 szabad, az 1 eltorlaszolt és a 2 égő pozíciókat ábrázol. Az egyetlen 3-as érték a raktáros helyzetét adja meg. Generáljuk az összes útvonalat, amelyen a raktáros kijuthat az épületből, ha:

- a kijárat az (n,m) pozícióban van;
- le, fel, jobbra, balra irányba haladhat;
- nem haladhat át az 1-es pozíciókon, valamint a 2-es és az ezekkel szomszédos pozíciókon.

2.18. A futó és a ló: Egy sakktáblán ismert egy futó helyzete. Határozzuk meg az összes útvonalat, amelyen egy ló eljuthat egy adott

kezdeti pozícióból egy adott végső pozícióba úgy, hogy ne lépjen a futó által támadott négyzetekre.

2.19. Zárójelek: Írjunk ki minden helyesen nyitó és csukó n zárójelet tartalmazó karakterláncot!

2.20. A „legnagyobb" legrövidebb út: Adott egy $n \times m$ méretű mátrix, amely egész számokat tartalmaz. Határozzuk meg az (i, j) pozíciótól az (u, v) pozícióhoz vezető összes legrövidebb út közül azokat, amelyek mentén az összeg pozitív szám.

2.21. Szürjektív függvények: Generáljuk az összes $f : A \to B$ szürjektív függvényt, ahol $A = \{1, 2, \ldots, n\}$ és $B = \{1, 2, \ldots, m\}$!

2.22. Színes fénykép: Adott egy színes fénykép egy $n \times m$ méretű mátrix által. A háttér fehér, amelyet nullák ábrázolnak, a különböző tárgyakat pedig nem nulla „egymás mellett" elhelyezkedő számok kódolják. Hány képpontból áll a legnagyobb tárgy képe?

2.23. Legkisebb összeg: Adott egy $n \times m$ méretű mátrix. Írjuk ki azt a legkisebb n tagú összeget, amelynek elemeit különböző sorokból és különböző oszlopokból szedjük össze!

2.24. Tornyok: Ismert n építőkocka oldalhossza és színe. Határozzuk meg a legmagasabb olyan *„tornyot",* amelyben nincsenek azonos színű szomszédos „emeletek".

2.25. Rúddarabolás: Adott egy n méter hosszú rúd. Írjuk ki az összes lehetséges módot, ahogy feldarabolható egy és két méter hosszú rudacskákra.

3. FEJEZET

DIVIDE ET IMPERA
„Oszd meg és uralkodj”

Milyen feladatok oldhatók meg a *divide et impera* („oszd meg és uralkodj”) módszer segítségével? Azok a feladatok, amelyek visszavezethetők, más szóval lebonthatók, két vagy több hasonló, de egyszerűbb (kisebb méretű) részfeladatra. Ezen részfeladatok hasonlóak lévén az eredeti feladathoz, maguk is visszavezethetők további hasonló, de még egyszerűbb részfeladatokra. Addig járunk így el, míg banálisan egyszerű (triviális) részfeladatokhoz jutunk, amelyek tovább nem bonthatók. Megint csak egy fát látunk magunk előtt. A gyökérben található az eredeti feladat. A fa első szintjén helyezkednek el azok a részfeladatok, amelyekre első lépésből bontható le a feladat. Mely részfeladatok kerülnek a második szintre? Nyilván azok, amelyek közvetlenül adódnak az első szinti részfeladatok lebontásából. Végül a fa leveleibe a lebontásból adódó triviális részfeladatok kerülnek.

Mivel az imént bemutatott fa minden csomópontjában hasonló feladatok találhatók, a részfeladatok általános alakjáról beszélhetünk. Az eredeti feladat úgy tekinthető, mint az általános feladat határesete, abban az értelemben, hogy méretben a legnagyobb. A triviális részfeladatok a másik határesetként foghatók fel, hiszen méretben a lehető legkisebbek (tovább nem darabolhatók).

Szemléltessük mindezt a *„hanoi tornyok”* feladatán keresztül.

Hanoi tornyok: Képzeljünk magunk elé három rudat, amelyeket a-val, b-vel és c-vel fogunk jelölni. Az a rúdra n különböző méretű korong van húzva, méretük szerint csökkenő sorrendben (legalul található a legnagyobb átmérőjű). Helyezzük át az n korongot az a rúdról a b rúdra a c rúd felhasználásával, figyelembe véve a következőket:

- egy lépésben egyetlen korong mozdítható el, valamelyik korongtorony tetejéről egy másik tetejére;
- tilos nagyobb átmérőjű korongot kisebb átmérőjűre helyezni.

A fenti feladat első lépésben két hasonló, de egyszerűbb részfeladatra bontható (amelyek mindegyike további kettőre, és így tovább), ugyanis

n korong áthelyezése a-ról b-re a c segítségével visszavezethető $n-1$ korong kétszeri áthelyezésére:

1. áthelyezzük a felső $n-1$ korongot a-ról c-re b segítségével;
2. áthelyezzük a legnagyobb korongot a-ról b-re;
3. áthelyezzük az $n-1$ korongot c-ről b-re a segítségével.

A részfeladatok általános alakja: k korong áthelyezése az s (*source*) rúdról a d (*destination*) rúdra a h (*help*) segítségével.

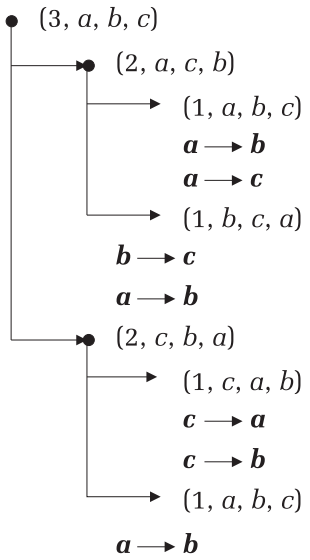
Határesetek a méretet tekintve: ha $k = n$, akkor az eredeti feladatot kapjuk, $k = 1$ esetén pedig triviális feladatokkal van dolgunk.

Példa gyanánt legyen az $n = 3$ eset. Az alábbi ábra szemléletesen mutatja be, miként bontható le az eredeti feladat részfeladatokra, és érzékelteti a feladat mögött meghúzódó fasztruktúrát is. Félkövér betűtípust használtunk a feladatot megoldó lépéssorozat kiemelésére (***a*** → ***b***, ***a*** → ***c***, ***b*** → ***c***, ***a*** → ***b***, ***c*** → ***a***, ***c*** → ***b***, ***a*** → ***b***).

A jelölések jelentése:

(k, s, d, h) – áthelyezünk k korongot az s rúdról a d rúdra a h rúd segítségével.

$s \to d$ – áthelyezünk 1 korongot az s rúdról a d rúdra.



3.1. ábra.

3.1. A divide et impera módszer stratégiája

Egy divide et impera algoritmus rekurzívan közelíti meg a feladatot. Ebből adódik eleganciája. Az algoritmust megvalósító rekurzív eljárást (vagy függvényt) nyilván az általános feladatra kell megírnunk és az eredeti feladatra meghívnunk. Az eljárásnak (függvénynek) különbséget kell tennie triviális és nem triviális részfeladat között. Más szóval két forgatókönyvet tartalmaz:

- Triviális esetben fel kell vállalnia az illető részfeladat teljes megoldását. Ez fog a rekurzió megállási feltételeként szolgálni.
- Ha nem triviális a feladat, megoldását, rekurzív hívások által, vissza kell vezetnie a részfeladatai megoldására. Ez három lépésben valósítható meg:
  1. Meghatározzuk a részfeladatokat, amelyekre az általános feladat lebontható („Oszd meg . . . ”).
  2. Rekurzív hívások által megoldjuk az így nyert részfeladatokat („. . . és uralkodj”).
  3. A részfeladatok megoldásaiból felépítjük az általános feladat megoldását.

Íme egy divide et impera algoritmus váza:

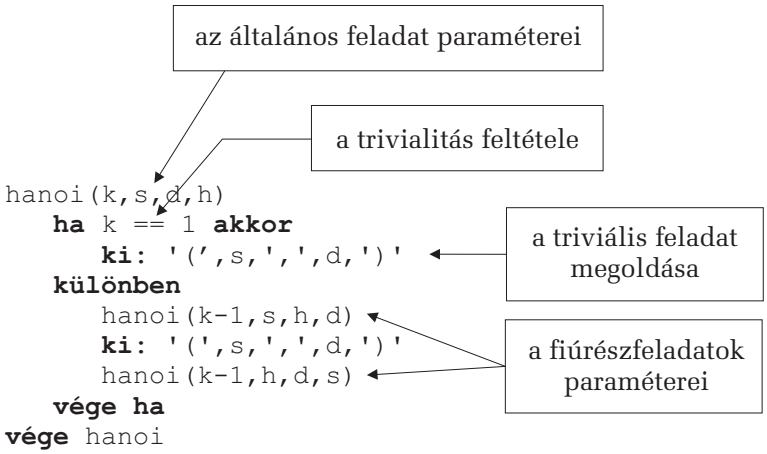
```
eljárás Név(az_általános_feladat_paraméterei)
   ha triviális akkor
      < oldd meg >
   különben
      < határozd meg a részfeladatait >
      < rekurzív hívások által oldd meg ezeket >
      < építsd fel a megoldást >
   vége ha
vége eljárás
```

Hogyan kerül bejárásra egy divide et impera algoritmusban a feladat lebontásából adódó fastruktúra? Nyilván mélységében, hiszen mindenik csomópont esetén először sorra megoldjuk a fiúrészfák által (azon részfák, amelyeknek gyökerei a csomópont fiai) képviselt részfeladatokat, majd ezt követően az illető csomóponthoz kapcsolódó feladatot.

Mi történik, ha a feladat lebontásakor a fa különböző ágain azonos részfeladatok jelennek meg? Példaként tegyük fel, hogy n elem k-adrendű kombinációinak számát szeretnénk kiszámítani divide et impera algoritmussal a $C(n,k) = C(n-1,k) + C(n-1,k-1)$ képlet alapján. Első lépésben a $C(n,k)$ kiszámítása a $C(n-1,k)$ és a $C(n-1,k-1)$

kiszámítására vezetődik vissza. A második lépésben megjelenő részfeladatok pedig a következők lesznek: $C(n-2,k)$ és $C(n-2,k-1)$, valamint $C(n-2,k-1)$ és $C(n-2,k-2)$. Íme, máris megjelent két azonos részfeladat. Mivel a divide et impera a részfeladatokat, amelyekre egy feladat visszavezetődik, egymástól függetlenül oldja meg, az azonos részfeladatok újra és újra megoldásra kerülnek, ahányszor csak találkozunk velük a fa bejárása közben. Ilyen feladatok megoldására előnytelen divide et impera algoritmust írni. Például megtörténhet, hogy a fa összes csomópontjának száma exponenciálisan függ a bemenet méretétől, bár a különböző részfeladatok képviselőinek száma csak polinomiális érték. Egy későbbi fejezetben megismerkedünk majd a dinamikus programozás módszerével, mely hatékonyan oldja meg az ilyen feladatok némelyikét.

Következzen a divide et impera algoritmust implementáló rekurzív eljárás a „hanoi tornyok” feladatára:



3.2. ábra.

Az eljárást az eredeti feladatra hívjuk meg: `hanoi(n,'a','b','c')`.

3.2. Hogyan közelítsünk meg egy divide et impera feladatot?

Ha tisztázzuk az alábbi kérdéseket, akkor ezekből az algoritmus természetszerűen fog adódni:

I. Hogyan vezethető vissza a feladat hasonló, de egyszerűbb részfeladatokra?

II. Mi az általános feladat alakja? Mik a paraméterei? Ezek lesznek az eljárás formális paraméterei!

III. Milyen paraméterértékekre kapjuk az eredeti feladatot? Ezekre hívjuk meg kezdetben az eljárást!

IV. Mi a triviálitás feltétele? Hogyan oldhatók meg a triviális részfeladatok?

V. Nem triviális esetben mik az általános feladat részfeladatainak a paraméterei? Ezek lesznek a rekurzív hívások aktuális paraméterei!

VI. Hogyan építhető fel a részfeladatok megoldásaiból az általános feladat megoldása?

A fenti sablont természetesen nem lehet egy az egyben minden feladatra ráhúzni. Az elv megértésében segítenek a most következő megoldott feladatok.

3.3. Megoldott feladatok

3.1. Maximumkeresés: Adott egy számsorozat, amelyet az $a[1..n]$ tömbben tároltunk. Határozzuk meg a legnagyobb elem indexét!

Megoldás:

I. Az $a[1..n]$ tömbszakaszban a maximum keresése visszavezethető az $a[1..n/2]$ és $a[n/2+1..n]$ tömbszakaszok maximumának meghatározására, amelyek a maguk rendjén tovább bonthatók az $a[1..n/4]$ és $a[n/4+1..n/2]$, illetve az $a[n/2+1..3n/4]$ és $a[3n/4+1..n]$ tömbszakaszok maximumainak megtalálására stb.

II. Nyilvánvaló, hogy az általános részfeladatnak, amelynek megoldására meg kell írnunk a divide et impera forgatókönyvet, meg kell határoznia egy $a[i..j]$ tömbszakasz maximumértékének indexét. Az általános feladat paraméterei természetesen az i és j indexek lesznek.

III. Az eredeti feladatot az $i = 1$ és $j = n$ értékekre kapjuk.

IV. A feladat akkor triviális, ha a tömbszakasz egyelemű ($i = j$). Ez esetben éppen az i (vagy j) indexű elem lesz az illető szakasz maximuma.

V. Nem triviális esetben ($i < j$) az (i, j) paraméterű feladatot visszavezetjük az $(i, (i+j)/2)$ és $((i+j)/2+1, j)$ feladatokra.

VI. A rekurzív hívásokból visszatérve, rendelkezésre állnak az $a[i\ ..\ (i+j)/2]$ és $a[(i+j)/2+1\ ..\ j]$ tömbszakaszok maximumainak

indexei. Magától értetődően, a két maximum közül a nagyobbik lesz az $a[i\,..\,j]$ tömbszakasz legnagyobb eleme.

```
maxindex(a[],i,j)
   ha i == j akkor
      return i
   különben
      m1=maxindex(a,i,(i+j)/2)
      m2=maxindex(a,(i+j)/2+1,j)
      ha a[m1] > a[m2] akkor
         return m1
      különben
         return m2
      vége ha
   vége ha
vége maxindex
```

A függvényt az eredeti számsorozatra hívjuk meg: `maxindex(a,1,n)`

3.2. Bináris keresés: Adott egy szigorúan növekvő számsorozat, amelyet az $a[1\,..\,n]$ tömbben tároltunk, valamint egy x szám. Keressük meg a számot a számsorozatban, és ha megtalálható, térítsük vissza a sorszámát, különben nullát!

Megoldás:

I. Megcélozzuk az $a[1\,..\,n]$ tömbszakasz középső elemét, az $a[n/2]$ elemet. Ha ez éppen a keresett elem, a keresést befejeztük. Ellenkező esetben, attól függően, hogy x kisebb vagy nagyobb, mint ezen középső elem, a keresést vagy a számsorozat alsó ($a[1\,..\,n/2-1]$), vagy a felső ($a[n/2+1\,..\,n]$) felében folytatjuk, ahol természetesen hasonlóan járunk el.

II. Tehát a részfeladatok általános alakja megint csak egy $a[i\,..\,j]$ tömbszakaszra vonatkozik, itt kell megkeresni az x számot. Az általános feladat paraméterei ez esetben is az i és j indexhatárok lesznek.

III. Az eredeti feladatot az $i = 1$ és $j = n$ értékekre kapjuk.

IV. A feladat két esetben triviális: ha a tömbszakasz nem létezik ($i > j$), vagy ha az $i \leqslant j$ esetben, megcélozva a tömbszakasz középső elemét, eltaláljuk a keresett számot. Az első esetben nullát, a másodikban a középső elem indexét ($(i + j)/2$) térítjük vissza.

V. Nem triviális esetben ($i \leqslant j$ és a keresett szám nem egyenlő $a[(i + j)/2]$-vel), attól függően, hogy x kisebb vagy nagyobb, mint $a[(i+j)/2]$, a keresést vagy az $a[i\,..\,(i+j)/2-1]$, vagy az $a[(i+j)/2+1\,..\,j]$ tömbszakaszban folytatjuk.

VI. Nem triviális esetben a feladat eredménye egy az egyben a rekurzív hívás által visszatérített érték lesz.

```
bináriskeresés(a[],x,i,j)
   ha i > j akkor
      return 0
   különben
      k = (i+j)/2
      ha x == a[k] akkor
         return k
      különben
         ha x < a[k] akkor
            return bináriskeresés(a,x,i,k-1)
         különben
            return bináriskeresés(a,x,k+1,j)
         vége ha
      vége ha
   vége ha
vége bináriskeresés
```

A függvényt az eredeti számsorozatra a következőképpen hívjuk meg: `bináriskeresés(a,x,1,n)`

3.3. Mergesort: Rendezzük növekvő sorrendbe az $x[1..n]$ tömbben tárolt számsorozatot, a növekvő számsorozatok összefésülésére vonatkozó algoritmusra alapozva.

Megoldás:

I–VI. Az $x[1\ .\ .\ n]$ tömbszakasz rendezése visszavezethető az $x[1\ .\ .\ n/2]$ és $x[n/2+1\ .\ .\ n]$ tömbszakaszok külön-külön való rendezéseire és az ezt követő összefésülésükre. Akárcsak az előző feladatok esetében, a divide et impera forgatókönyvet egy $x[i\ .\ .\ j]$ tömbszakasz rendezésére kell megírnunk. A feladat akkor triviális, ha a rendezendő tömbszakasz egyelemű ($i = j$).

```
mergesort(x[],i,j)
   ha i < j akkor
      k = (i+j)/2
      mergesort(x,i,k)
      mergesort(x,k+1,j)
      összefésül(x,i,k,j)
   vége ha
vége mergesort
```

```
összefésül(x[],bal,közép,jobb)
   minden i=bal,közép végezd
      a[i] = x[i]
   vége minden
   minden i=közép+1,jobb végezd
      b[i] = x[i]
   vége minden
   a[közép+1] = ∞
   b[jobb+1] = ∞                                    {strázsák}
   i = bal
   j = közép+1
   minden k=bal,jobb végezd
      ha a[i] < b[j] akkor
         x[k] = a[i]
         i = i + 1
      különben
         x[k] = b[j]
         j = j + 1
      vége ha
   vége minden
vége összefésül
```

Az összefésül eljárás összefésüli az $x[i \,..\, j]$ tömbszakasz alsó ($x[i \,..\, k]$) és felső ($x[k+1 \,..\, j]$), külön-külön már rendezett feleit az a és b tömbök segítségével.

Az eljárást a teljes számsorozatra hívjuk meg: `mergesort(x,1,n)`

3.4. Quicksort: Rendezzük növekvő sorrendbe az $a[1 \,..\, n]$ tömbben tárolt számsorozatot, a számsorozatok szétválogatására vonatkozó algoritmusra alapozva.

Megoldás:

I–VI. Szétválogatjuk az $a[1 \,..\, n]$ tömbszakaszt az első eleme szerint. Ennek eredményeként a számsorozat első eleme a növekvő sorrend szerinti helyére (legyen ez a k-adik pozíció), a nála kisebbek eléje (nem föltétlenül növekvő sorrendben), a nagyobbak pedig mögéje (nem föltétlenül növekvő sorrendben) kerülnek. Ha a szétválogatás után különkülön rendezzük az $a[1 \,..\, k-1]$ és $a[k+1 \,..\, n]$ tömbszakaszok elemeit, ez a teljes számsorozat rendezését jelenti. Tehát, szétválogatás által visszavezethető az eredeti feladat két hasonló, de egyszerűbb feladatra.

Az általános feladatot megint csak az $a[i \,..\, j]$ tömbszakasz rendezése fogja jelenteni, amelyet az $i \geqslant j$ esetben tekintünk triviálisnak (egy egyelemű vagy egy nem létező tömbszakasz nyilván rendezett).

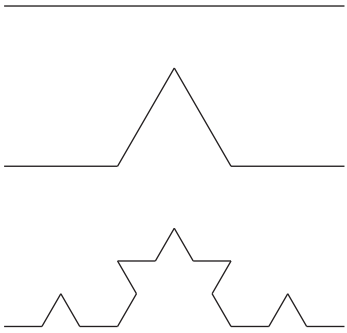
```
quicksort(a[],i,j)
   ha i < j akkor
      k=szétválogat(a,i,j)
      quicksort(a,i,k-1)
      quicksort(a,k+1,j)
   vége ha
vége quicksort

szétválogat(a[],i,j)
   p=i
   q=j
   w=a[i]
   amíg p < q végezd
      ha a[p] <= a[q] akkor
         ha w == a[p] akkor
            q=q-1
         különben
            p=p+1
         vége ha
      különben
         v=a[p]
         a[p]=a[q]
         a[q]=v
      vége ha
   vége amíg
   return p
vége szétválogat
```

Az eljárást a teljes számsorozatra hívjuk meg: `quicksort(a,1,n)`

3.5. Koch-fraktál: Rajzoljuk ki a képernyőre az n-edik szintű, D széles Koch-fraktált.

A 3.3. ábrán 0, 1 és 2 szintű Koch-fraktálokra adunk példát.

Megoldás: Az n-edik szintű Koch-vonalat úgy fogjuk fel, mintha a nulladik szintű helyett rajzolnánk négy első szintűt, és ezek mindegyike helyett négy-négy második szintűt, . . . , egészen addig, míg elérkezünk az n-edik mélységű vonalakig, amelyeket tovább már nem helyettesítünk, hanem megrajzoljuk őket. Tehát vonalrajzolásra csak a feladathoz



3.3. ábra.

rendelhető fa leveleiben kerül sor. Mivel a mélységi bejárás a fa leveleit balról jobbra sorrendben látogatja meg, célszerű olyan vonalrajzoló eljárást használni, amely az aktuális képpontból kiindulva, az aktuális képpontot mozgatva rajzol. Ez oda vezet, hogy úgymond a ceruza felemelése nélkül rajzoljuk ki a Koch-vonalat. A alábbi Koch-eljárás a `rajzol(d,`α`)` függvényt használja, amely vonalat húzva, elmozdítja az aktuális képpontot egy d hosszú szakaszon, α szög alatt.

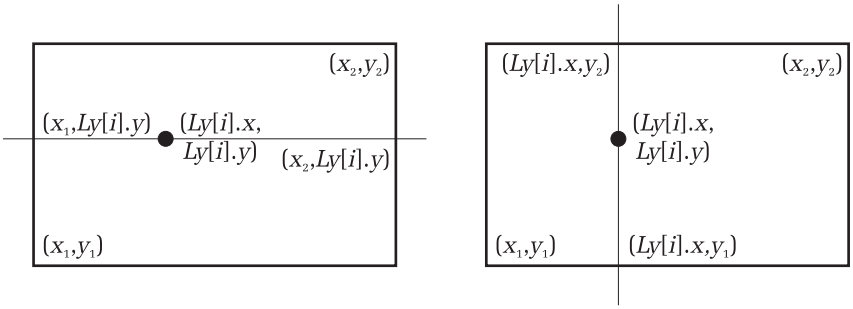
```
koch(d,α,k)
   ha k == n akkor
      rajzol(d,α)
   különben
      koch(d/3,α,k+1)
      koch(d/3,α + π/3,k+1)
      koch(d/3,α - π/3,k+1)
      koch(d/3,α,k+1)
   vége ha
vége koch
```

Mielőtt az eljárást meghívnánk, természetesen aktuális képpontnak azt a pontot kell beállítanunk, ahol szeretnénk, hogy a Koch-vonal kezdődjön. Az eljárást a következőképpen hívjuk meg: `koch(D,0,0)`

3.6. Lemezdarabolás: Adott egy n méter hosszú és m méter széles téglalap alakú lemezdarab. A lemez bal alsó sarka a $(0,0)$, a jobb felső pedig az (n,m) koordinátájú pontban van. A lemezen p pontszerű lyuk található egész koordinátájú pozíciókban, amelyeket az $Ly[1..p]$ tömb tárol. Az i-edik lyuk koordinátáit az $Ly[i].x$ és $Ly[i].y$ mezők tárolják ($i =$

$1, n$). A koordinátatengelyekkel párhuzamos teljes vágásokat (minden vágással a lemez két részre esik) alkalmazva, vágjuk ki a legnagyobb lyukmentes területet a lemezből.

Megoldás: Észrevehető, hogy a legnagyobb lyukmentes lemezdarabhoz vezető vágássorban a vágások sorrendje tetszőleges lehet. Az általános feladat nyilvánvalóan a következő: az (x_1, y_1) bal alsó sarkú és (x_2, y_2) jobb felső sarkú lemezdarabból a legnagyobb lyukmentes darab kivágása. Az eredeti feladatot az $x_1 = 0$, $y_1 = 0$, $x_2 = n$ és $y_2 = m$ értékekre kapjuk.



3.4. ábra.

A megoldás alapgondolata az, hogy ha nem lyukmentes az aktuális darab, akkor *valamelyik* lyuk mentén elvágjuk (vagy vízszintesen, vagy függőlegesen). A vágásból adódó darabok már nem fogják tartalmazni az illető lyukat. Tehát minden vágással a feladat hasonló és egyszerűbb részfeladatokra bomlik. Mivel mindenik lyuk mentén két vágás lehetséges, és ezek mindenikéből két-két darab lemez adódik, ezért minden lépésben a feladatot további négy részfeladatra vezetjük vissza. Egy részfeladat akkor triviális, ha lyukmentes darabnak felel meg. Ha megkapjuk, hogy a fiúdarabok mindenikén melyik a legnagyobb lyukmentes terület, akkor ezek közül a maximális lesz az apadarab legnagyobb lyukmentes területe. Az alábbi ábra azt mutatja be, hogy az i-edik lyuk menti vágásból milyen koordinátájú darabok származhatnak.

A `darabolás` eljárás egy bejegyzés típusú változó (`lnd`) értékét téríti vissza, amelynek mezői az eljárás által paraméterként kapott lemezdarabból kivágható legnagyobb terület koordinátáit (`lnd.x1`, `lnd.y1`, `lnd.x2`, `lnd.y2`) és méretét (`lnd.t`) tartalmazzák. Ha az illető programozási nyelven, amelyen az algoritmust implementálni szeretnénk, nem

lehetséges bejegyzés típusú függvények használata, akkor cím szerint átadott paramétert használhatunk.

A `lyukmentes` függvény ellenőrzi, hogy lyukmentes-e a paraméterként kapott darab. Ha lyukmentes, akkor nullát, különben valamelyik lyuk sorszámát téríti vissza.

A `terület` függvény meghatározza a paraméterként kapott darab területértékét.

A `maximális_területű` függvény visszatéríti a négy paramétere közül azt, amelyiknek a legnagyobb a `t`-mezője.

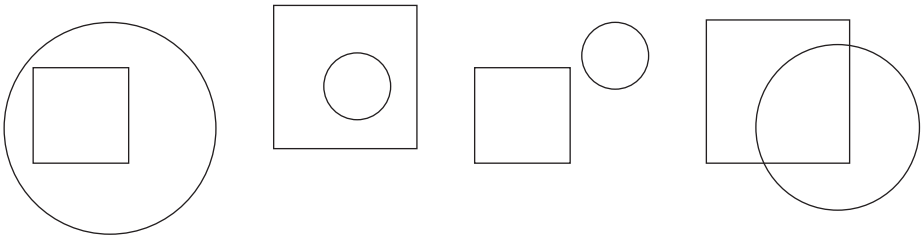
```
darabolás(Ly[],p,x1,y1,x2,y2)
   i=lyukmentes(Ly,p,x1,y1,x2,y2)
   ha i == 0 akkor
      lnd.t=terület(x1,y1,x2,y2)
      lnd.x1=x1
      lnd.y1=y1
      lnd.x2=x2
      lnd.y2=y2
   különben
      lnd1=darabolás(Ly,p,x1,y1,x2,Ly[i].y)
         // vízszintes vágás, alsó darab
      lnd2=darabolás(Ly,p,x1,Ly[i].y,x2,y2)
         // vízszintes vágás, felső darab
      lnd3=darabolás(Ly,p,x1,y1,Ly[i].x,y2)
         // függőleges vágás, bal darab
      lnd4=darabolás(Ly,p,Ly[i].x,y1,x2,y2)
         // függőleges vágás, jobb darab
      lnd=maximális_területű(lnd1,lnd2,lnd3,lnd4)
   vége ha
   return lnd
vége darabolás

lyukmentes(Ly[],p,x1,y1,x2,y2)
   minden i=1,p végezd
      ha Ly[i].x>x1 és Ly[i].x<x2 és Ly[i].y>y1 és Ly[i]<y2
         akkor return i
      vége ha
   vége minden
   return 0
vége lyukmentes
```

A `terület` és `maximális_területű` függvények megírását az olvasóra hagyjuk.

A kulcsfüggvényt az eredeti lemezdarabra hívjuk meg: `darabolás(Ly,p,0,0,n,m)`

3.7. Négyzet és kör: Adott egy négyzet a bal alsó sarkának koordinátái és az oldalhossza által (a négyzet oldalai párhuzamosak a koordinátatengelyekkel), valamint egy kör a középpontja koordinátái és a sugara által. Határozzuk meg a metszési felületük területét háromtizedes pontossággal.

Megoldás: Legyenek a négyzet bal alsó sarkának koordinátái, valamint oldalhossza: `xn`, `yn`, `on`. Továbbá, jelöljük a kör középpontjának koordinátáit, valamint sugarát a következőképpen: `xk`, `yk`, `rk`. Egy négyzet és egy kör – egymáshoz viszonyítva négy különböző helyzetben lehetnek: 1. a négyzet része a körnek, 2. a kör része a négyzetnek, 3. a négyzet a körön kívül van, 4. a négyzet átfed valamennyit a körből.

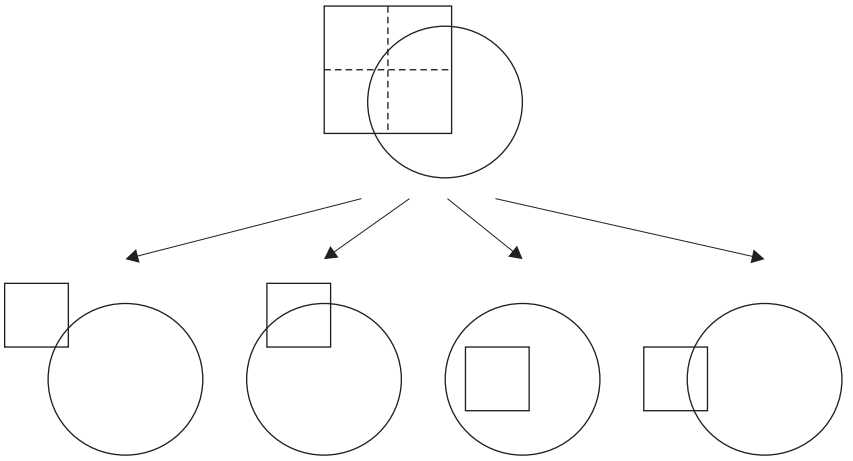


3.5. ábra.

Az első esetben a metszési felület a teljes négyzet területe. A második esetben a közös terület a teljes kör. A harmadik esetben nincs közös részük. A feladat akkor érdekes, ha negyedik eset áll fent.

A feladat a következőképpen vezethető vissza hasonló és egyszerűbb részfeladatokra. Ha a négyzet metszi a kört (4. eset), négybe vágjuk. Ha valamelyik négyzetecske teljesen beleesik a körbe, vagy ha teljesen kívül esik rajta, akkor triviális részfeladatként fogjuk kezelni. A körrel metsző négyzetecskékre mint hasonló és egyszerűbb részfeladatokra rekurzív hívást alkalmazunk. Addig folytatjuk a rekurzív hívások láncolatát, míg 0,001-nél kisebb területű négyzetecskékhez jutunk. Természetesen az apanégyzet–kör metszési felület a négy fiúnégyzet–kör metszési terület összege lesz.

A részfeladatok általános alakja: az (x, y, o) négyzet és az (xk, yk, rk) kör metszési felületének háromtizedes pontossággal való meghatározása. Az eredeti feladatot az $x = x_n, y = y_n, o = o_n$ paraméterértékekre kapjuk.



3.6. ábra.

```
négyzet_és_kör(x,y,o,xk,yk,rk)
   ha o*o < 0.001 akkor
      return 0
   vége ha
   i=kör_négyzet_helyzete(x,y,o,xk,yk,rk)
   ha i == 1 akkor
      return o*o
   vége ha
   ha i == 2 akkor
      return π*rk*rk
   vége ha
   ha i == 3 akkor
      return 0
   vége ha
   ha i == 4 akkor
      t1=négyzet_és_kör(x1+o/2,y1,o/2,xk,yk,rk)
      t1=négyzet_és_kör(x1+o/2,y1+o/2,o/2,xk,yk,rk)
      t1=négyzet_és_kör(x1,y1+o/2,o/2,xk,yk,rk)
      t1=négyzet_és_kör(x1,y1,o/2,xk,yk,rk)
      return t1+t2+t3+t4
   vége ha
vége négyzet_és_kör
```

A `kör_négyzet_helyzete` függvény egyet, kettőt, hármat vagy négyet térít vissza, attól függően, hogy melyik eset áll fent a kör és a négyzet helyzetét illetően. Megírását az olvasóra hagyjuk.

3.8. Mátrixszorzás: Adott két $n \times n$ méretű (n kettőnek a hatványa) mátrix, A és B. Szorozzuk össze őket hatékonyabban, mint a klasszikus módszer, amelyekkel mátrixokat szorzunk.

Megoldás: A mátrixokat négybe vágjuk, és úgy tekintjük őket, mint (2×2) méretű mátrixokat, amelyeknek elemei a maguk rendjén ugyancsak $(n/2) \times (n/2)$ méretű mátrixok.

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix},$$

ahol:

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\ C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\ C_{22} &= A_{21}B_{12} + A_{22}B_{22} \end{aligned}$$

Ily módon az $n \times n$ méretű mátrixok összeszorzását visszavezettük nyolc darab $(n/2) \times (n/2)$ méretű mátrix közötti szorzásra (és még négy mátrixösszeadásra). Ha megvizsgáljuk ezen divide et impera algoritmus bonyolultságát, azt találjuk, hogy ez $O(n^3)$, tehát ugyanannyi, mint a klasszikus mátrixszorzásé.

Köztudott, hogy a mátrixszorzás lényegesen több elemi műveletet igényel, mint a mátrixösszeadás. 1969-ben *Strassen* talált egy olyan módszert a C_{ij} részmátrixok kiszámítására, amely csak 7 szorzást és 18 összeadást, illetve kivonást alkalmaz. Ez a módszer egy $O(n^{lg7}) < O(n^3)$ bonyolultságú algoritmushoz vezet.

$$\begin{aligned} P &= (A_{11} + A_{22})(B_{11} + B_{22}) \\ Q &= (A_{21} + A_{22})B_{11} \\ R &= A_{11}(B_{12} - B_{22}) \\ S &= A_{22}(B_{21} - B_{11}) \\ T &= (A_{11} + A_{12})B_{22} \\ U &= (A_{21} - A_{11})(B_{11} + B_{22}) \\ V &= (A_{12} - A_{22})(B_{21} + B_{22}) \end{aligned}$$

Továbbá:

$$C_{11} = P + S - T + V$$
$$C_{12} = R + T$$
$$C_{21} = Q + S$$
$$C_{22} = P + R - Q + U$$

Az első esetben, amikor a C_{ij} részmátrixokat a hagyományos módon határozzuk meg, a következőképpen valósítható meg a divide et impera algoritmus. Az A és B mátrixokat az $A[1..n][1..n]$ és $B[1..n][1..n]$ tömbök tárolják. A C eredménymátrix a $C[1..n][1..n]$-be kerül, amelyet kezdetben lenullázunk. A részfeladatok általános alakja: az `A` tömb `(ia,ja)` bal felső sarkú $k \times k$ méretű részmátrixának és a `B` tömb `(ib,jb)` bal felső sarkú $k \times k$ méretű részmátrixának az összeszorzása. Ezen általános feladat megoldásának forgatókönyvét implementáltuk a `mátrixszorzás` eljárás révén. A feladat akkor triviális, ha $k = 1$. Az algoritmus egyszerűsége azon észrevételen alapszik, hogy az eredménymátrix elemei elemi szorzatok összegei, és az `A[ia][ja]·B[ib][jb]` alakú elemi szorzatok az eredménymátrix `C[ia][jb]` elemének mint összegnek a tagjai.

```
mátrixszorzás(A[][],B[][],C[][],ia,ja,ib,jb,k)
   ha k == 1 akkor
      C[ia][jb] = C[ia][jb] + A[ia][ja] * B [ib][jb]
   különben
      mátrixszorzás(A,B,C,ia,ja,ib,jb,k/2)
      mátrixszorzás(A,B,C,ia,ja+k/2,ib+k/2,jb,k/2)
      mátrixszorzás(A,B,C,ia,ja,ib,jb+k/2,k/2)
      mátrixszorzás(A,B,C,ia,ja+k/2,ib+k/2,jb+k/2,k/2)
      mátrixszorzás(A,B,C,ia+k/2,ja,ib,jb,k/2)
      mátrixszorzás(A,B,C,ia+k/2,ja+k/2,ib+k/2,jb,k/2)
      mátrixszorzás(A,B,C,ia+k/2,ja,ib,jb+/2,k/2)
      mátrixszorzás(A,B,C,ia+k/2,ja+k/2,ib+k/2,jb+k/2,k/2)
   vége ha
vége mátrixszorzás
```

Az eredeti feladatra a következőképpen hívjuk meg a fenti eljárást: `mátrixszorzás(A,B,C,1,1,1,1,n)`

A *Strassen* ötletére alapuló megoldás maradjon gyakorlat az olvasónak.

3.4. Kitűzött feladatok

3.1. Hajtogatás_1: Legyen egy n elemű egydimenziós tömb. Hajtsuk rá a tömb egyik felét a másik felére. A tömb felül kerülő felét adónak, az alul kerülőt pedig vevőnek nevezzük (ha páratlan a tömb elemszáma, a közbülső elemet eltávolítjuk). Az összetűrt tömb elemei a vevő fél indexeit „öröklik". Például az $x[1..5]$ tömb kétféleképpen tűrhető kettőbe, egyszer az $x[1..2]$ tömböt (a második felét tűrjük rá az első felére), másodszor pedig az $x[4..5]$ tömböt (az első felét tűrjük rá a második felére) eredményezve (a harmadik elemet eltávolítjuk). A hajtogatást addig folytatjuk, amíg egyelemű tömbhöz jutunk.

Feltételezve, hogy az $x[1..n]$ tömbből indulunk ki:

- Létezik-e olyan hajtogatási sor, amely eredményeként az $x[i]$ egyelemű tömbhöz jutunk? Nevezzük ezt végső elemnek.
- Mely tömbelemek lehetnek végső elemek?
- Egy adott i index esetén írjunk ki egy olyan hajtogatási sort, amely őt eredményezi mint végső elemet.
- Feltételezve, hogy a tömb elemei egész számok, valamint, hogy a kettéhajtásból adódó tömb elemei értékét úgy számítjuk ki, hogy a megfelelő vevőelem kétszereséből kivonjuk a megfelelő adóelem értékét, juthatunk-e nulla értékű végső elemhez?

3.2. Kitalálósdi: Képzeljük el az alábbi játékot: Az első játékos gondol egy számra 0-tól 1000-ig, a másodiknak pedig ki kell találnia a számot rá-rákérdezve. Az első játékos háromféleképpen válaszolhat: kisebb, nagyobb, eltaláltad. Határozzunk meg egy olyan stratégiát a második játékosnak, hogy bármelyik legyen is az elrejtett szám, minimális találgatással találja meg. Az első játékos szerepét játszodja a felhasználó, a másodikét pedig a számítógép.

3.3. Nagy számok: Legyen két nagyon nagy egész szám, u és v, amelyeket az $u[0\,..\,n-1]$ és $v[0\,..\,n-1]$ egydimenziós tömbökben tárolunk. Legyen továbbá $s = \lfloor n/2 \rfloor$. Vágjuk kettőbe az u és v tömböket és nevezzük u_1-gyel és v_1-gyel ezek $\lceil n/2 \rceil$ számjegyű alsó (kisebb helyértékű) feleit (az $u[0\,..\,\lceil n/2 \rceil - 1]$, valamint $v[0\,..\,\lceil n/2 \rceil - 1]$ tömbszakaszok), és u_2-vel, illetve v_2-vel a $\lfloor n/2 \rfloor$ számjegyű felső (nagyobb helyértékű) feleit (az $u[\lceil n/2 \rceil\,..\,n-1]$, valamint $v[\lceil n/2 \rceil\,..\,n-1]$ tömbszakaszok). Szem előtt tartva, hogy:

$$u = 10^s u_1 + u_2, v = 10^s v_1 + v_2, \text{ ahol } 0 < u_2, v_2 < 10^s$$
$$uv = 10^{2s} u_1 v_1 + 10^s (u_1 v_2 + u_2 v_1) + u_2 v_2,$$

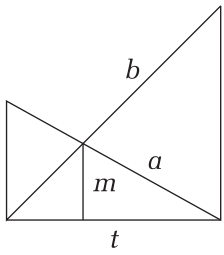
írjunk hatékony divide et impera algoritmust az uv szorzat kiszámítására.

3.4. Hatvány: Írj divide et impera algoritmust az a^n-en kiszámítására $\log(n)$ idő alatt.

3.5. Mátrixhatvány: Legyen A egy négyzetes mátrix. Írj divide et impera algoritmust az A^n kiszámítására.

3.6. Négyzetszámok: Számoljuk meg egy n ($1 \leqslant n \leqslant 1\ 000\ 000$) elemű számsorozat négyzetszámait! A számok nem nagyobbak, mint 1 000 000.

3.7. Mértan: Figyelembe véve a mellékelt ábrát, számítsuk ki a t hosszúságot az m, a és b méretek függvényében 10^{-5} pontossággal.



3.7. ábra.

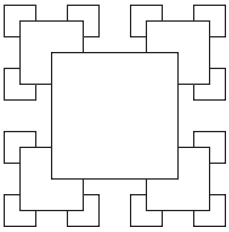
3.8. L-tapétázás: Ismert egy $2^n \times 2^n$ méretű bináris mátrixban az egyetlen 1-es elem pozíciója. Tapétázzuk ki a nullásokat tartalmazó felületet egyenlő szárú, 2×2 méretű, L alakú lapokkal. Az eredményt az alábbi formában írjuk ki (1-nél nagyobb egymás utáni egészekkel kódoljuk az egyes tapétalapokat):

```
2 2 3 3
2 4 1 3
5 4 4 6
5 5 6 6
```

A példában $n = 2$ és az 1-es a $(2, 3)$ pozícióban található.

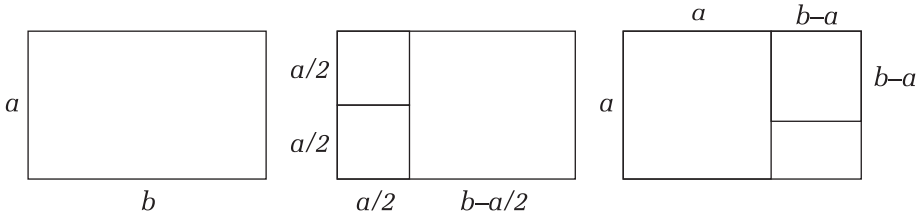
3.9. LNKO: Egy n elemű természetes számokat tartalmazó számsorozat elemeit az $a[1\,..\,n]$ tömbben tároljuk el. Határozzuk meg a számsorozat elemeinek legnagyobb közös osztóját, tudva, hogy: $\mathrm{lnko}(a[1\,..\,n]) = \mathrm{lnko}(\mathrm{lnko}(a[1\,..\,n/2]), \mathrm{lnko}(a[n/2{+}1\,..\,n]))$.

3.10. Négyzet-fraktál: Írjunk divide et impera algoritmust az alábbi fraktál kirajzolására. Beolvassuk a legnagyobb négyzet oldalhosszát, a legkisebb négyzetek pedig képpont méretűek legyenek.



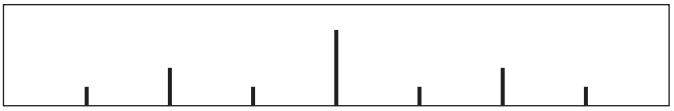
3.8. ábra.

3.11. Téglalap: Adott egy $a \times b$ méretű téglalap ($a, b \in N$ és $b - a < a < b$). Egy ilyen alakzat kétféleképpen bontható két négyzetre és egy téglalapra. A következő lépésben a kapott téglalapot tovább bontjuk, és ezt addig folytatjuk, míg „felbonthatatlan" téglalapokat nem nyerünk (nem elégítik ki a fenti összefüggést). Határozzuk meg azt a lépéssorozatot, amely minimális számú „felbonthatatlan" téglalapot eredményez.



3.9. ábra.

3.12. Vonalzó: Egy L képpont hosszú vonalzót az alábbi ábra szerint szeretnénk felosztani. Összesen legyen $(2^n + 1)$ beosztás, egymástól azonos távolságokra, amelyek közül a legmagasabb K képpont, a következők $K/2$ képpont, és így tovább, hosszúak legyenek. Rajzoljuk ki a vonalzót a képernyőre egy divide et impera algoritmussal.



3.10. ábra.

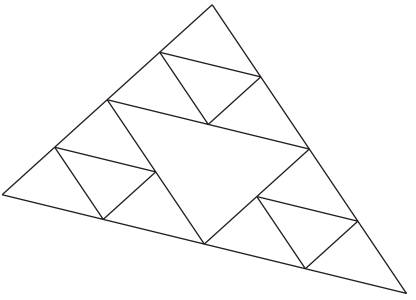
3.13. K.-legkisebb: Adott egy n elemű számsorozat. Írjunk divide et impera algoritmust a k-adik legkisebb elem meghatározására.

3.14. Hajtogatás_2: Adott egy $n \times m$ méretű mátrix. Hajtogassuk össze a mátrixot oly módon, hogy felülre a legnagyobb azonos elemeket tartalmazó terület kerüljön. A következő megkötések léteznek:

- Minden lépésben a mátrixot pontosan kettőbe tűrjük, vagy vízszintesen, vagy függőlegesen (persze, ha valamelyik irányban a mérete páratlan, akkor a másik irányú tűrés nem lehetséges).
- Mindkét irányú tűrés kétféleképpen végezhető el, attól függően, hogy melyik fele kerül felül és melyik marad alul.

Ha létezik megoldás, akkor írjuk ki a megfelelő lépéssorozatot könnyen érthető formában. Ellenkező esetben írjunk ki egy megfelelő üzenetet.

3.15. Sierpinski: Rajzoljuk ki az n-edrendű Sierpinski-fraktált!



3.11. ábra.

4. FEJEZET

BACKTRACKING VAGY DIVIDE ET IMPERA?

Az előbbi két fejezetben felfigyelhettünk arra, hogy mind a backtracking, mind a divide et impera algoritmusok mélységükben járják be a feladatok szerkezetét ábrázoló fát. Ez a fő ok, amiért a rekurzív megvalósítás annyira kézenfekvő mindkét esetben. Mi akkor az alapvető különbség a két módszer között?

A backtracking algoritmusokban a megoldások felépítése a gyökértől a levelek irányába történik. Így az ígéretes részfa leveleiben hirdetnek megoldást, pontosabban a megoldás-levelekben. Ezért is rajzoltuk a fát a megszokottól eltérően, a gyökerével alul, hiszen az a természetes, hogy felfele építkezzünk. Úgy is mondhatnánk, hogy a mélységi bejárás előremutató útszakaszain épít, a visszamutatókon pedig bont. Emlékezzünk most arra, miként jártuk körbe a fát a mélységi bejárás bemutatásakor a bevezető fejezetben. Többször is érintettük a csomópontokat, és attól függően, hogy az első vagy utolsó érintéskor *látogattuk* meg őket, beszéltünk *preorder,* illetve *postorder* mélységi bejárásról. Mivel a visszalépéses keresés a feladathoz rendelt fa csomópontjait első érintésükkor használja a megoldások építésében (ekkor kerülnek be a verembe, és majd utolsó érintésükkor lesznek onnan eltávolítva), ezért fogalmazhatunk úgy, hogy preorder mélységi bejárást alkalmaz.

Ezzel szemben a divide et impera algoritmusok a gyökértől a levelek irányába (fentről lefele) lebontják a feladatot egyre egyszerűbb részfeladatokra, majd a visszaúton (lentről felfele) a részfeladatok megoldásaiból felépítik az eredeti feladat megoldását. Egy részfeladat csak azután kerül megoldásra, miután a részfeladatai már meg lettek oldva. Milyen sorrendben lesznek hát megoldva a fa csomópontjai által képviselt részfeladatok? Posztorder mélységi bejárás szerinti sorrendben.

Vegyük észre azt a különbséget is, hogy a backtracking a megoldás-levelekbe leérkezve (felérkezve), a divide et impera pedig a gyökérbe visszaérkezve hirdet megoldást. Ez megmagyarázhatja azt, hogy a backtracking feladatoknak általában miért van több megoldásuk is, a divide et impera feladatoknak viszont csak egy (egy fának sok levele van, de csak egyetlen gyökere).

Az előbbi észrevételnek van egy másik következménye is, főleg optimalizálási feladatok esetében (bár egyik technikát sem elsősorban ilyen feladatok megoldására „találták ki"). Ilyenkor a feladat mögött egy döntési fa húzódik meg, és az optimális megoldást az optimális gyökér–levél út képviseli (lásd 1. fejezet, háromszögfeladat). Az optimális levélből nézve egyértelmű, melyik az optimális gyökér–levél út, viszont a gyökérből nézve egyáltalán nem. Ez a magyarázata annak, hogy optimalizálási feladatok esetén a backtracking kiválóan alkalmas mind az optimális megoldás (optimális gyökér–levél út), mind az ehhez tartozó optimumérték kiíratására, a divide et imperának viszont csak az utóbbi kézenfekvő. Persze felmerülhet bennünk az a gondolat, hogy nem lehetne-e minden csomópontban eltárolni ennek optimális fiát, hiszen ezen információ alapján később játszi könnyedséggel előállítható lenne az optimális gyökér–levél út is. Csakhogy ez nem jellemző a divide et impera stratégiára, amely hogy elkerülje magának a fának a megépítését, az egyes részfeladatok megoldásait csak addig tárolja el, amíg az apafeladat megoldását fel nem építi belőlük. A részfeladatok optimális megoldásainak eltárolása a dinamikus programozás egyik alapeleme.

Azért, hogy úgymond egymás mellett lássuk a két technikát, a következő optimalizálási feladatot mindkettővel megoldjuk.

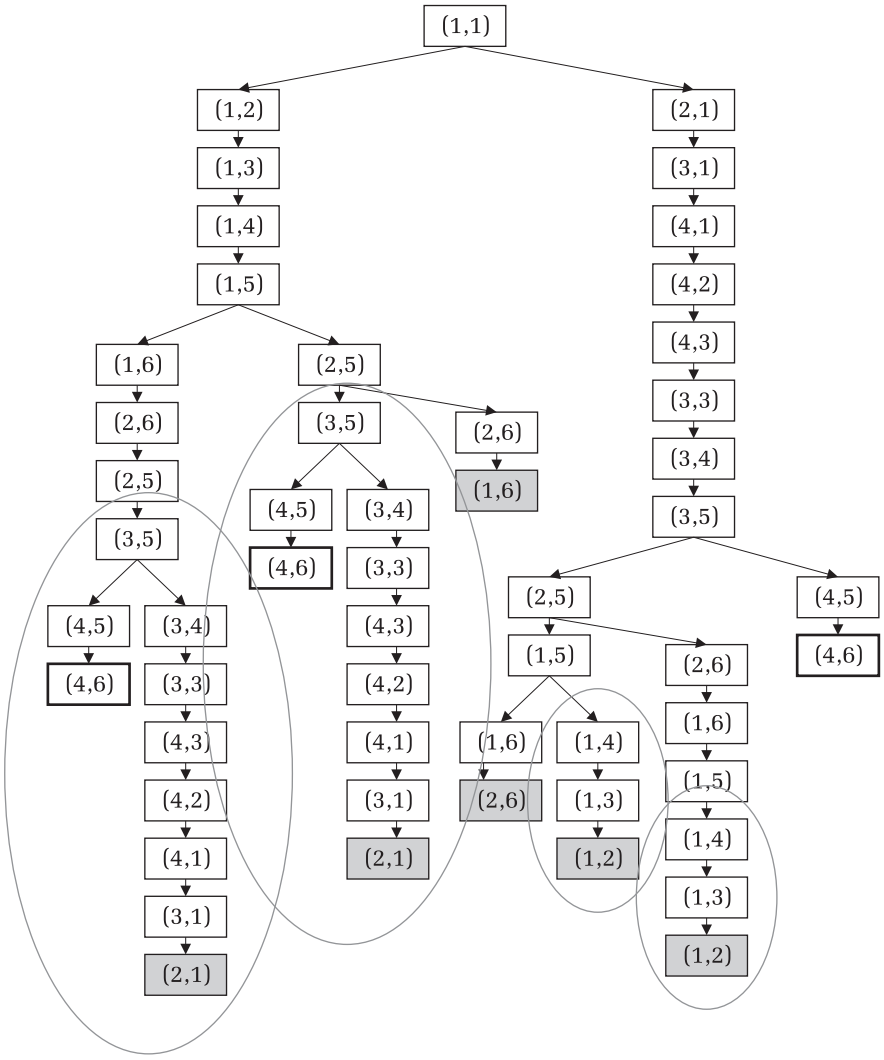
Egér–sajt feladat: Egy labirintusban, amelyet egy $n \times m$ ($n \leqslant 50$, $m \leqslant 50$) méretű, a nevű bináris mátrix ábrázol (a 0 érték utat jelöl, az 1-es falat), ismert egy egér (xe, ye) és egy darab sajt (xs, ys) pozíciója. Határozzuk meg az egértől a sajthoz vezető legrövidebb utat (a labirintusban négy irányban lehet közlekedni). Példa:

	1	2	3	4	5	6
1	**e**	0	0	0	0	0
2	0	1	1	1	0	0
3	0	1	0	0	0	1
4	0	0	0	0	1	**s**

4.1. ábra.

Az alábbiakban bemutatjuk az $(1, 1)$ pozícióból induló hurokmentes „egérutak" fáját, amely alapvetően egy döntési fa. A besatírozott levelekhez „zsákutcák" vezetnek. A megoldásleveleket (amelyeknek mindegyike a sajt pozícióját képviseli) megerősített kerettel rajzoltuk. A legmagasabban lévő $(4, 6)$ koordinátaértékű levélhez vezető gyökér–levél

út ábrázolja az optimális „egér–sajt” utat. Bejelöltük az identikus részfákat is.



4.2. ábra.

Megjegyzés: Bár egyik technika sem nyújtja a leghatékonyabb algoritmust a fenti feladatra, stratégiáik összehasonlítására kitűnően alkalmas (a feladathoz rendelhető fában azonos részfák is lehetnek, amelyeknek többszöri bejárását nem tudják elkerülni).

Backtracking stratégia: Generálja az összes hurokmentes utat, amely az egértől a sajthoz vezet, és kiválasztja közülük a legrövidebbet.

Divide et impera stratégia: Az egértől a sajthoz vezető legrövidebb út *hosszá*nak meghatározása visszavezethető az egérrel szomszédos szabad pozíciókból induló – sajthoz vezető – legrövidebb utak hosszának megtalálására. Miután ezek rendelkezésre állnak, a legrövidebbhez hozzáadva egyet, meg is van a keresett út hossza.

Figyeljük meg, hogy a backtracking több potenciális megoldás-*utat* is generál (az összes egér–sajt utat), a divide et impera viszont csak az optimális út *hosszát* építi fel.

Megjegyzések a `backtracking_egér` eljáráshoz:

- `út[]` – az aktuális utat tárolja,
- `útmin[]` – a legrövidebb utat tárolja,
- `kmin` – a legrövidebb út hosszát tárolja,
- `x, y` – az aktuális pozíció,
- `k` – hányadik állomás az aktuális pozíció az aktuális úton,
- az algoritmus:
  - regisztráljuk az (x, y) pozíciót mint az aktuális út k-adik állomását,
  - ellenőrizzük, hogy nem állunk-e éppen a sajton,
    - ha igen, akkor amennyiben jobb megoldást találtunk, mint az eddigi legjobb, regisztráljuk a `kmin` és az `útmin` változókban;
  - ha nem vagyunk még a sajtnál, akkor
    - mielőtt továbblépnénk, befalazzuk lábunk alatt a labirintust (ezzel elkerüljük, hogy ezen az ágon újra visszakerülhessünk ugyanebbe a pozícióba, és a kialakult hurokban az egeret a végtelenségig körbejárassuk; olyan ez, mintha nyomot hagynánk),
    - a szomszédos pozíciókra vonatkozó rekurzív hívások által sorra elmegyünk minden szabad irányba,
    - mielőtt visszalépnénk az (x, y) pozícióból, kifalazzuk lábunk alatt a labirintust (ezzel biztosítjuk, hogy más ágakon továbbra is elérhető legyen ezen pozíció; úgy is fogalmazhatnánk, hogy nyomtalanul lépünk vissza).
- Figyeljük meg, hogy akkor érkeztünk „zsákutcába”, ha az illető pozícióból egyetlen irányba sem tudunk továbblépni.

```
backtracking_egér(a[][],ut[],utmin[],kmin,xs,ys,x,y,k)
   út[k].x = x
   út[k].y = y
   ha x == xs és y == ys akkor // a sajthoz érkeztünk
      ha k < kmin akkor // rövidebb utat találtunk,
                        // mint az eddigi legjobb
         kmin = k
         minden i = 1, k végezd
            útmin[i] = út[i]
         vége minden
      vége ha
   különben
      a[x][y] = 1 // a hurkok elkerülése végett
      ha a[x-1][y] == 0 akkor
         backtracking_egér
            (a,ut,utmin,kmin,xs,ys,x-1,y,k+1)
      vége ha
      ha a[x][y+1] == 0 akkor
         backtracking_egér
            (a,ut,utmin,kmin,xs,ys,x,y+1,k+1)
      vége ha
      ha a[x+1][y] == 0 akkor
         backtracking_egér
            (a,ut,utmin,kmin,xs,ys,x+1,y,k+1)
      vége ha
      ha a[x][y-1] == 0 akkor
         backtracking_egér
            (a,ut,utmin,kmin,xs,ys,x,y-1,k+1)
      vége ha
      a[x][y] = 0 // visszalépés előtt
                  // visszaállítjuk a szabad utat
   vége ha
vége backtracking_egér
```

Megjegyzések a `divide_et_impera_egér` függvényhez:

- a függvény a legrövidebb út hosszát téríti vissza,
- az `(n·m)`-nél nagyobb vagy egyenlő hossz jelentése: nincs út,
- `x,y` – az aktuális pozíció.

```
divide_et_impera_egér (a[][],xs,ys,x, y)
   ha x == xs és y == ys akkor
      return 0 // a sajthoz érkeztünk
   vége ha
```

```
   h1 = n*m
   h2 = n*m
   h3 = n*m
   h4 = n*m
   a[x][y] = 1 // a hurkok elkerülése végett
   ha a[x-1][y] == 0 akkor
      h1=divide_et_impera_egér(a,xs,ys,x-1,y)
   vége ha
   ha a[x][y+1] == 0 akkor
      h2=divide_et_impera_egér(a,xs,ys,x,y+1)
   vége ha
   ha a[x+1][y] == 0 akkor
      h3=divide_et_impera_egér(a,xs,ys,x+1,y)
   vége ha
   ha a[x][y-1] == 0 akkor
      h4=divide_et_impera_egér(a,xs,ys,x,y-1)
   vége ha
   a[x][y] = 0 // visszalépés előtt
               // visszaállítjuk a szabad utat
   return minimum(h1,h2,h3,h4) +1
vége divide_et_impera_egér
```

Az alábbiakban közöljük a back_vagy_divide eljárást, amelynek keretén belül meghívjuk az imént megírt rekurzív eljárást és függvényt. Feltételeztük, hogy a feladatnak van megoldása.

```
back_vagy_divide(a[][],n,m,xe,ye,xs,ys})
   ki: "backtracking megoldás:"
   kmin = n*m
   backtracking_egér(a,ut,utmin,kmin,xs,ys,xe,ye,0)
   ki: "Az út hossza:",kmin
   ki: "Az út:"
   minden i = 1,kmin végezd
      ki: '(', utmin[i].x,',',utmin[i].y,')'
   vége minden
   ki: "divide et impera megoldás:"
   ki: "Az út hossza:",divide_et_impera_egér(a,xs,ys xe,ye)
vége back_vagy_divide
```

5. FEJEZET

GREEDY MÓDSZER
Mohó algoritmusok

A greedy módszert optimalizálási feladatok megoldására használjuk. Íme néhány példa greedy feladatra:

1. *(Felvonó)* Egy egyszemélyes felvonó előtt n személy áll, akikről ismert, hogy hányadik emeletre szeretnének feljutni (e_1, e_2, ..., e_n). Milyen sorrendben kellene használják a felvonót, ha azt szeretnék, hogy a várakozási idejük összege *minimális* legyen? Mennyi lesz ez az összidő, ha tudjuk, hogy a felvonó időegységekként egy emeletmagasságot tesz meg, és a kiszállás és beszállás „szempillantás alatt" történik?
2. *(Műsorok)* Adott n tévéműsor, amelyeknek ismert a kezdési és befejezési időpontjuk: (b_1, e_1), (b_2, e_2), ..., (b_n, e_n). Egy család, amelynek egy tévékészüléke van, úgy dönt, hogy a $[B, E]$ időintervallumban ($b_i \geqslant B$, $e_i \leqslant E$, $i = 1, n$) tévézni fog. Mely műsorokat válasszák (lehetnek átfedődő műsorok is, amelyeket természetesen más-más kanálison közvetítenek), ha azt szeretnék, hogy a *legtöbb* műsort lássák? *Legkevesebb* hány tévékészülékre lenne szükségük (és legalább hány tagú kellene hogy legyen a család) ahhoz, hogy minden műsort megnézhessen legalább egy családtag?
3. *(Telefonhálózat)* N számú város között telefonhálózatot szeretnének kiépíteni. Egy `d[1..m]` egydimenziós tömbben adott, hogy mely várospárok között építhető ki direkt telefonvonal, valamint, hogy ezek a kapcsolatok egyenként mennyibe kerülnének. Az i-edik közvetlen vonal végpontjait, valamint megépítési költségét a `d[i].x`, `d[i].y` és `d[i].k` mezők tárolják. Mely városok között építsék ki a közvetlen telefonvonalakat ahhoz, hogy összekapcsoljanak minden várost (legalább közvetve), és a költségek *minimálisak* legyenek?
4. *(Madarak)* Adott $n+1$ fa, amelyek 1-től $(n+1)$-ig vannak megszámozva. Az első n fa mindenikén elhelyezkedett egy-egy madár.

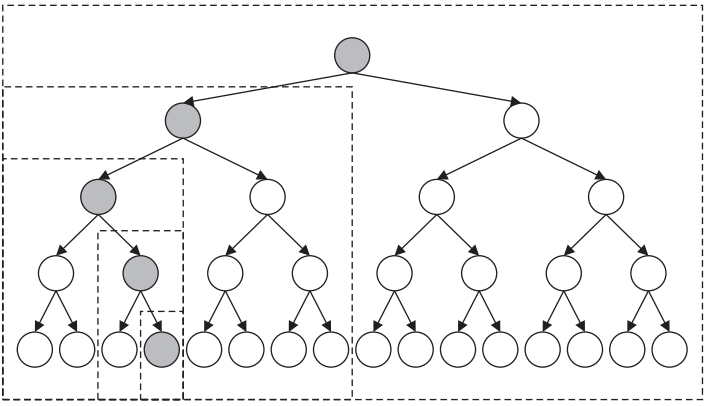
Ezeket is megszámozzuk 1-től n-ig. Az i-edik fára az i-edik madár szállt ($i = 1, n$). A madarak elkezdenek áthelyezkedni. Minden lépésben valamelyik madár átrepül az éppen üres fára (egyszerre csak egyetlen madár van a levegőben). Ismerve, hogy egy idő után melyik madár éppen melyik fára szállt, „repítsük vissza" a madarakat eredeti helyükre *minimális* számú repüléssel.

5. *(Legrövidebb utak)* Adott egy $n \times n$ méretű d mátrix, amely egy n várost összekötő úthálózatot ábrázol. A `d[i][j]` elem az i és j városok közti közvetlen út hosszát tárolja (ha két város közt nincs direkt út, a megfelelő mátrixelemek értéke ∞). Határozzuk meg az első várostól az összes többihez vezető *legrövidebb* utakat és ezeknek a hosszát.

A feladatok szövegében dőlt betűkkel írtuk azokat a szavakat, amelyek az optimalizálás gondolatát hordozzák.

A greedy feladatokban általában arról van szó, hogy egy döntéssorozattal kell meghatározni az optimális megoldást. Egyes esetekben egyszerűen az optimális sorrendet kell megtalálni, máskor egy halmaznak valamilyen szempontból vett optimális részhalmazát stb. Fontos megjegyezni azonban, hogy a második esetben is általában az vezet el az optimális részhalmazhoz, ha optimális sorrendben vizsgáljuk meg a halmaz elemeit.

Újra egy fa elevenedik meg előttünk, éspedig egy döntési fa, amely a feladathoz rendelhető.



5.1. ábra.

Minden döntéssel a feladat kisebb és kisebb méretű hasonló feladatokká redukálódik, amelyek az eredeti feladat egymásba ágyazott

részfeladatainak tekinthetők (ezt szemléltettük a szaggatott vonalú keretekkel). Az eredeti fa az első döntéssel leszűkül valamelyik fiúrészfájára (azzal, hogy választunk, mintha lemetszenénk a fáról a többi fiúrészfát), majd a következő döntéssel ennek valamelyik fiúrészfájára, és így tovább, míg már csak egy levél marad belőle.

Az optimális döntéssorozatot, amely elvezet az optimális megoldáshoz, a fa valamelyik gyökér–levél útja képviseli. Nevezzük el ezt az utat *optimális* útnak, az illető levelet pedig optimális levélnek. Ezek után úgy is megfogalmazható egy greedy feladat, hogy keressük meg a feladathoz rendelhető döntési fa optimális gyökér–levél útját.

5.1. A greedy módszer stratégiája

A greedy algoritmusok filozófiája nagyon egyszerű, és azokra az emberekre emlékeztet, akik a mának élnek. Neve jelentésével összhangban (mohó), mindig az adott lépésben optimálisnak látszó (legígéretesebb) döntést hozza, nem számolva az esetleges hosszú távú következményekkel. Úgy gondolkodik, hogy a lokális optimum majd globális optimumhoz vezet. Amennyiben ez nem igaz az illető feladatra, vagy nem talál megoldást, vagy nem találja meg az optimálisat (legfennebb véletlenül, bizonyos sajátos esetekben).

Megjegyzés: Mivel a greedy módszer alapgondolata, miszerint a lokális optimum globális optimumhoz vezet, általánosságban nem igaz, ezért a teljes megoldáshoz az is hozzátartozik, hogy bizonyítjuk, hogy az illető feladatra viszont igaz. Ennek ellenére – ha beérjük kevesebbel is, mint az optimális megoldás – célszerű lehet minden olyan esetben alkalmazni (még ha nem is bizonyítható a nyert algoritmus helyessége), amikor minden más megközelítés túl bonyolult vagy túl nagy időigényű algoritmust eredményezne.

Lássuk, mik lennének a mohó választások a fenti példákban:

1. Mivel annak a személynek a feljutási ideje, aki éppen használja a felvonót, bekerül a többi, még sorára váró lakó várakozási idejébe is, ezért mindig a *legalacsonyabban lakó* személy lesz a következő.
2. A *leghamarabb befejeződő* műsort választjuk ki elsőként, hogy a lehető leghosszabb *folytonos* időszakasz maradjon fenn további választásokra. Ha a soron következő legígéretesebb műsor átfedődik a már kiválasztottakkal, egyszerűen kihagyjuk.

3. Mindig a következő *legolcsóbb* szakaszon építjük ki a telefonvonalat, ügyelve arra, hogy ne építsünk közvetlen vonalat ott, ahol már létrejött közvetett kapcsolat.
4. Arra törekszünk, hogy mindenik madár a lehető *legkevesebb* repüléssel kerüljön a helyére. Tehát mindig az a madár fog repülni, amelyiknek a fája éppen üres. Ily módon az illető madár egy repüléssel a helyére kerül. Ha egy adott pillanatban az $(n+1)$-edik fa válik üressé, és nincs még minden madár a helyén, akkor ezek közül valamelyik elrepül az $(n+1)$-edik fára. Ez a madár végül két repülésből kerül a fájára.
5. Mindig a már elért városokhoz *legközelebb* eső város lesz a következő, amelyikhez meghatározzuk a legrövidebb utat. Ez a sorrend biztosítja majd, hogy minden városhoz a legrövidebb úton jussunk el.

Számos greedy feladat esetében az alábbi helyzet ismerhető fel. Létezik egy úgynevezett „kandidátusok halmaza” (az elemei arra kandidálnak, hogy az optimális megoldás részét képezzék), és egy bizonyos „célfüggvény”, amely a kandidátushalmaz részhalmazainak halmazán értelmeződik. A feladat abból áll, hogy a kandidátushalmaz adott tulajdonságú részhalmazai közül meg kell határozni azt a részhalmazt, amelyik optimalizálja a célfüggvényt. Íme a greedy algoritmus erre a helyzetre:

```
eljárás greedy(K)
   S ← Φ
   amíg nem megoldás(S) és K ≠ Φ
      x ← legígéretesebb(K)                    (1)
      K ← K \ {x}                              (2)
      ha bővíthető(S,x) akkor
         S ← S ∪ {x}                           (3)
      vége ha
   vége amíg
   ha megoldás(S) akkor
      kiír(S)
   különben
      ki: "Nincs megoldás"
   vége ha
vége greedy
```

Ahol:

– K – A kandidátusok halmaza.
– S – Ebben a halmazban építjük fel a megoldást.

- `legígéretesebb(K)` – A mohó döntés alapján kiválasztja a kandidátushalmaznak az adott pillanatban legígéretesebbnek tűnő elemét. Mivel ennek a függvénynek a szerepe a célfüggvény optimalizálása, ezért ez utóbbiból következtethető ki. A `legígéretesebb` függvény a lokális optimumot biztosítja, a célfüggvény pedig a globálist.
- `bővíthető(S,x)` – Ellenőrzi, hogy az `x` elemmel bővíthető-e az `S` halmaz, azzal a kilátással, hogy végül a kívánt tulajdonságú legyen. Tehát onnan vezethető le, hogy a kandidátushalmaz mely tulajdonságú részhalmazai között keressük az optimálisat. Ha a feladat azt kéri, hogy az eredeti halmazra határozzuk meg a célfüggvény optimumát (lásd az 1. példafeladatot), akkor a bővíthető függvény elveszíti szerepét.
- `megoldás(S)` – Ellenőrzi, hogy felépült-e a megoldás.
- `kiír(S)` – Kiírja a megoldást.

A greedy stratégia kulcsgondolata az eljárás (1), (2) és (3) soraiban testesül meg:

(1) – mindig az adott pillanatban legígéretesebbnek látszó kandidátust választjuk (a döntési fa aktuális csomópontjának legígéretesebb fiát);

(2) – a kiválasztott kandidátust „örökre” eltávolítjuk a kandidátushalmazból;

(3) – ha a megoldás halmaz bővíthető az illető elemmel, akkor végérvényesen beletesszük, ha nem, akkor végképp lemondunk róla.

Tehát egy greedy algoritmusban nincs visszalépés (*backtrack*). Amint mondani szokás, mindent feltesz egy lapra.

Hogyan állapítható meg, hogy egy feladat megoldható-e a greedy módszerrel? Nincs általános módszer, azonban van két alapelv, amelyek ha érvényesek az illető feladatra, akkor bizonyíthatóan alkalmazható rá a greedy stratégia. Szerencsére az esetek nagy többségében ez járható út.

5.1.1. A mohó-választás alapelve

Emlékezzünk, hogy a greedy stratégia szerint mindig az adott lépésben legjobbnak tűnő választást hajtjuk végre, bármelyik legyen is az, majd ezt *követően* megoldjuk azokat a részfeladatokat, amelyekre választásunk nyomán a feladat leszűkült. Ezzel összhangban az aktuális választás függhet (figyelembe tudja venni) az előző döntésektől, de nem

függhet a későbbiektől, hiszen mindez még a jövő titka. Úgy is fogalmaztunk, hogy a greedy algoritmusok fentről lefele haladva, egymást követő mohó döntések által a feladatot mind kisebb és kisebb méretű feladattá redukálják. Ez azt jelenti, hogy a döntési fának egyetlen gyökér–levél útját generálják, bízva abban, hogy pontosan ez lesz az optimális.

Ezek után a mohó-választás alapelve a következőképpen jelenthető ki:

a) a feladat *optimális* megoldása mohó-választással *kezdődik* (vagy módosítható úgy, hogy mohó-választással kezdődjön),

b) és ezen választás nyomán a feladat *hasonló* feladattá redukálódik.

Természetesen, ha a mohó-választás nyomán nyert feladat hasonló, akkor ennek az *optimális* megoldása is mohó-választással fog kezdődni (vagy módosítható, hogy azzal kezdődjön), és így tovább... A feladat optimális megoldásának ez a tulajdonsága szükséges feltétel annak bizonyításához, hogy a globális optimum felépíthető lokális optimumok (mohó-döntések) sorozataként. Mi hiányzik még a teljes bizonyításhoz?

Legyen $D_1, D_2, \ldots, D_n$ a feladat optimális megoldása. Pontosabban, az az *optimális* döntéssorozat, amely a globális optimumot biztosítja. Azt szeretnénk bebizonyítani, hogy ezen döntések mindenike egyenként mohó-választás, azaz lokális optimum. Mit biztosít ebből a mohó-választás alapelve? Azt, hogy D_1 biztosan mohó-választás. Mire van még szükség ahhoz, hogy a mohó-választás alapelvéből következzen a D_2 döntés mohó volta is? Ehhez a $D_2, D_3, \ldots, D_n$ döntéssorozatnak is *optimálisnak* kell lennie, annak a részfeladatnak az optimális megoldásának, amellyé a feladat az első mohó döntés nyomán redukálódott. Általánosan: a D_i $(i = 2, n)$ döntések mohó voltának bizonyításához további szükséges feltétel, hogy a $D_i, D_{i+1}, \ldots, D_n$ alakú részdöntéssorozatok ugyancsak optimálisak legyenek. Pontosan ezt mondja ki a következő alapelv.

5.1.2. Az optimalitás alapelve

A feladat optimális megoldása a részfeladatok (amelyekre a feladat a mohó-döntések nyomán redukálódik) optimális megoldásaiból épül fel. Ez azt jelenti, hogy ha $D_1, D_2, \ldots, D_n$ egy optimális döntéssorozat, akkor ebből következik, hogy a $D_i, D_{i+1}, \ldots, D_n$ alakú részdöntéssorozatok ugyancsak optimálisak az illető részfeladatokra nézve.

Végkövetkeztetésként leszögezhetjük, hogy ha egy feladat optimális megoldására érvényes a mohó-választás, valamint az optimalitás alapelve, akkor megoldható greedy módszerrel.

Az alábbiakban be fogjuk mutatni a második feladat kapcsán a greedy algoritmus helyességének bizonyítását. A többi megoldott feladat esetében csak az algoritmust közöljük.

A mohó stratégia helyességének bizonyítása a tévéműsor-kiválasztás feladatára:

Legyen $K = \{1, 2, \ldots, n\}$ a műsorok (kandidátusok) halmaza, és $S \subseteq K$ egy optimális részhalmaz (a legszámosabb, amely nem tartalmaz átfedődő műsorokat). Továbbá legyen p a legígéretesebb műsor, amely leghamarabb fejeződik be.

Először is bebizonyítjuk, hogy van olyan optimális megoldás, amely a mohó döntéssel kezdődik, azaz, hogy $p \in S$, vagy módosítható S úgy, hogy p eleme legyen. Tegyük fel, hogy $p \notin S$, és legyen q S-nek azon eleme, amelynek a legkisebb a befejezési ideje. Ha felcseréljük q-t p-vel (ezt megtehetjük, mert a p műsor hamarabb befejeződik, mint a q műsor), egy másik optimális megoldást kapunk. Tehát létezik olyan optimális részhalmaz, amely tartalmazza a legígéretesebb műsort.

Vegyük észre, hogy az első mohó döntés nyomán a feladat a következő részfeladattá redukálódott: *Legyen $K1 \subset K$ az a halmaz, amelyet úgy kapunk, hogy eltávolítjuk K-ból a p műsort, és a vele átfedődő műsorokat. Határozzuk meg $K1$-nek egy legszámosabb részhalmazát, amely nem tartalmaz átfedődő műsorokat.* Ez valóban az eredeti feladattal azonos, de kisebb méretű feladat.

Be fogjuk bizonyítani, hogy az $S - \{p\}$ részhalmaz optimális megoldása a fenti részfeladatnak. Mivel S nem tartalmaz átfedődő műsorokat, $S - \{p\} \subset K1$. Tegyük fel, hogy létezik $B \subset K1$ úgy, hogy B számosabb, mint $S - \{p\}$, és nem tartalmaz átfedődő műsorokat. Ez esetben viszont $B \cup \{p\}$ jobb megoldása lenne az eredeti feladatnak, mint S. Ez viszont ellentmond a feltételnek, miszerint S optimális megoldása a feladatnak. Tehát a feladatra érvényes az optimalitás alapelve is.

Mindezen elméleti fejtegetések után biztosan érdekli az olvasót, miként is közelíthet meg egy greedy feladatot? Sajnos (vagy éppen ez a szép benne) nincs recept ezt illetően. Talán ez a módszer az, amelyik, bár a legegyszerűbb, mégis a legtöbb leleményességet követeli. Különösen ahhoz kell leleményesség, hogy helyesen azonosítsuk a mohó-választást, amely a feladatot hasonló feladattá redukálja. Mindenképpen ez az első lépés.

Az vezethet nyomra ebben, hogy mit is kell optimalizálni. Ezért mondtuk, hogy a `legígéretesebb` függvény a célfüggvényből következtethető ki. Ha az előbbiekben bemutatott algoritmusváz alkalmazható a feladatra, akkor második lépésben a bővíthető függvényt kellene megírni. A többi függvény általában magától értetődő.

A fenti algoritmusvázzal azonban nem az volt a célunk, hogy sablont adjunk az olvasónak, inkább azt ajánljuk, hogy sajátítsa el a greedy gondolkodás szellemét, és sajátosan alkalmazza az egyes feladatokra.

Ebben segítenek a következő megoldott feladatok.

5.2. Megoldott feladatok

5.1. Felvonó: Egy egyszemélyes felvonó előtt n személy áll, akikről ismert, hogy hányadik emeletre szeretnének feljutni ($e_1, e_2, \ldots, e_n$). Milyen sorrendben kellene használják a felvonót, ha azt szeretnék, hogy a várakozási idejük összege *minimális* legyen? Mennyi lesz ez az összidő, ha tudjuk, hogy a felvonó időegységenként egy emeletmagasságot tesz meg, és a kiszállás és beszállás „szempillantás alatt" történik.

Megoldás: Rendezzük a személyeket emeleteik szerint növekvő sorrendbe. Ebben a sorrendben fogják használni a felvonót, hiszen ez a mohó sorrend. A rendezett sorrendbeli i-edik személy ($i = 2, n$) várakozási ideje: $e_1 + e_2 + \ldots + e_{i-1}$. Az első személynek nem kell várnia senkire.

```
felvonó(e[],n)
   rendezés(e,n)
   össz_vár_idő = 0
   akt_személy_vár_idő = 0
   minden i=2,n végezd
      akt_személy_vár_idő = akt_személy_vár_idő + e[i-1]
      össz_vár_idő = össz_vár_idő + akt_személy_vár_idő
   vége minden
   ki: össz_vár_idő
vége felvonó
```

A fenti algoritmus $O(n\lg n)$ bonyolultságú, mivel a rendezésnek ennyi az időigénye.

5.2. Műsorok: Adott n tévéműsor, amelyeknek ismert a kezdési és befejezési időpontjuk: (b_1, e_1), (b_2, e_2), ..., (b_n, e_n). Egy család, amelynek egy tévékészüléke van, úgy dönt, hogy a $[B, E]$ időintervallumban

($b_i \geqslant B, e_i \leqslant E, i = 1, n$) tévézni fog. Mely műsorokat válasszák (lehetnek átfedődő műsorok is, amelyeket természetesen más-más kanálison közvetítenek), ha azt szeretnék, hogy a *legtöbb* műsort lássák? *Legkevesebb* hány tévékészülékre lenne szükségük (és legalább hány tagú kellene hogy legyen a család) ahhoz, hogy minden műsort megnézhessen legalább egy családtag?

Megoldás: Mohó sorrendbe rendezzük a műsorokat. Amint láttuk, ez a befejezési időpontjaik szerinti növekvő sorrend. Ebben a sorrendben megpróbáljuk őket programra tűzni. Egy V változóban nyilvántartjuk az utolsó programra tűzött műsor befejezési időpontját. Ha a soron következő előadás ezt megelőzően kezdődik, kihagyjuk (nem bővíthető vele a már kiválasztott műsorok halmaza).

```
műsorok (e[],b[],n)
   rendezés(e,b,n) // az e és b tömbök párhuzamos rendezése
                   // a befejezési időpontok szerint
   ki: '(',b[1],',',e[1],')'
   V=e[1]
   minden i=2,n végezd
      ha b[i] > V akkor
         ki: '(',b[i],',',e[i],')'
         V = e[i]
      vége ha
   vége minden
vége műsorok
```

Az algoritmus időigénye – a rendezésből adódóan – $O(n \lg n)$.
A feladat második felének a megoldását az olvasóra hagyjuk.

5.3. Telefonhálózat: n város között telefonhálózatot szeretnének kiépíteni. Egy $d[1..m]$ bejegyzéstömbben adott, hogy mely várospárok között építhető ki direkt telefonvonal, valamint, hogy ezek a kapcsolatok egyenként mennyibe kerülnének. Az i-edik közvetlen vonal végpontjait, valamint megépítési költségét a $d[i].x$, $d[i].y$ és $d[i].k$ mezők tárolják. Mely városok között építsék ki a közvetlen telefonvonalakat ahhoz, hogy összekapcsoljanak minden várost (legalább közvetve), és a költségek *minimálisak* legyenek?

Megoldás: Ennél a feladatnál használjuk a fentiekben ajánlott sablont. A K kandidátushalmazt éppen a d tömb elemei képezik, hiszen minden potenciális közvetlen vonal kandidál arra, hogy a megépítettek között legyen. Ez utóbbiak fogják képezni a kandidátushalmaz megoldás-

részhalmazát (S). Egy $v[1..m]$ tömbben nyilvántartjuk a kandidátusok állapotát. Egy kandidátus vagy még a kandidátushalmazban van ($v[i] = 1$), vagy már a megoldáshalmaz része ($v[i] = 2$), vagy egyik halmazban sincs ($v[i] = 0$; el lett távolítva a kandidátushalmazból, de nem volt bővíthető vele a megoldáshalmaz). A $h[1..n]$ tömböt arra használjuk, hogy nyomonkövessük, mely városok között alakult már ki kapcsolat (közvetlen vagy közvetett). Ha $h[i] = h[j]$, akkor az i és j városok már egy hálózathoz tartoznak. Tehát a megoldáshalmaz akkor bővíthető a soron következő legígéretesebb kandidátussal, ha a végpontjainak megfelelő h tömbbeli értékek különböznek. Kezdetben mindenik város egy külön „hálózat" ($h[i] = i$, ahol $i = 1, n$). Valahányszor megépítünk egy közvetlen vonalat, egyesítjük azon hálózatokat (`egyesít` eljárás), amelyekhez a végpontjai tartoznak. Akkor jutottunk megoldáshoz, ha mindenik város egyetlen hálózathoz tartozik. Bebizonyítható, hogy ehhez pontosan $n - 1$ közvetlen vonal szükséges.

```
telefon_hálózat(d[],m,n)
   minden i=1,m végezd
      v[i] = 1
   vége minden
   minden i=1,n végezd
      h[i] = i
   vége minden
   p = 0 // számolja a megvizsgált kandidátusokat
   r = 0 // számolja a megépített direkt vonalakat
   összeg = 0
      // számolja a megépített
      // direkt vonalak összköltségét
   amíg r < n-1 és p < m végezd
      // amíg nem jutottunk megoldáshoz,
      // és még van kandidátus
      i = legígéretesebb(d,m)
      v[i] = 0
      p = p + 1
      ha h[d[i].x]≠h[d[i].y] akkor
         // a bővíthetőség feltétele
         egyesít(h,n,d[i].x,d[i].y)
         összeg = összeg + d[i].k
         v[i] = 2
         r = r + 1
      vége ha
   vége amíg
```

```
      ha r==n-1 akkor // megoldáshoz jutottunk
         ki: összeg
         minden i=1,m végezd
            ha v[i] == 2 akkor
               ki: '(',d[i].x,',',d[i].y,')'
            vége ha
         vége minden
      különben
         ki: "Nincs megoldás"
      vége ha
   vége telefon_hálózat
```

A kandidátushalmaz legígéretesebb közvetlen vonala az, amelynek a költsége legkisebb.

```
   legígéretesebb(d[],m)
      min_költség = ∞
      minden i=1,m végezd
         ha v[i] == 1 és d[i].k < min_költség akkor
            min_költség = d[i].k
            min_kandidátus = i
         vége ha
      vége minden
      return min_kandidátus
   vége legígéretesebb
```

Az `egyesít` eljárás egyesíti az a és b városok hálózatait azáltal, hogy az a várossal már összekapcsolt városok hálózati sorszámát (ugyanahhoz a hálózathoz tartozó városok hálózati sorszáma nyilván azonos) átírja a b város hálózati sorszámára ($h[b]$).

```
   egyesít(h[],n,a,b)
      minden i=1,n végezd
         ha h[i] == h[a] akkor
            h[i] = h[b]
         vége ha
      vége minden
   vége egyesít
```

A fenti algoritmus Kruskal nevéhez fűződik. Ha a d tömböt előre rendezzük ($O(m\lg m)$) és ha diszjunk részhálózatok egyesítését az eddig ismert leghatékonyabb algoritmussal végezzük ($O(m\lg m)$) [1], akkor a Kruskal-algoritmus időigénye $O(m\lg m)$.

5.4. Madarak: Adott $n+1$ fa, amelyek 1-től $(n+1)$-ig vannak megszámozva. Az első n fa mindenikén elhelyezkedett egy-egy madár. Ezeket is megszámozzuk 1-től n-ig. Az i-edik fára az i-edik madár szállt $(i = 1, n)$. A madarak elkezdenek áthelyezkedni. Minden lépésben valamelyik madár átrepül az éppen üres fára (egyszerre csak egyetlen madár van a levegőben). Ismerve, hogy egy idő után melyik madár éppen melyik fára szállt, „repítsük vissza" a madarakat eredeti helyükre *minimális* számú repüléssel.

Megoldás: Tegyük fel, hogy az `f[1..n+1]` tömbben tároljuk, hogy melyik madár éppen melyik fán tartózkodik: az i-edik fán az $f[i]$-edik madár van $(i = 1, n)$. Az `ü_f` változó az éppen üres fa sorszámát tárolja (`f[ü_f]=0`). Használunk egy `m[1..n]` tömböt is, amely azt tárolja, hogy az i-edik madár az `m[i]`-edik fán van $(i = 1, n)$. Az algoritmus kulcsgondolata a mohó választás módjában rejlik. Amint fentebb kifejtettük már, mindig az a madár fog repülni, amelyiknek a fája éppen üres. Ha egy adott pillanatban az $(n+1)$-edik fa válik üressé, és nincsen még mindenik madár a helyén, akkor ezek közül valamelyik elrepül az $(n+1)$-edik fára, felszabadítva egy másiknak a helyét. Az r változóban tároltuk, hogy melyik madár fog repülni. Ha az $(n+1)$-edik fa az üres, akkor a `keres_madár` függvény keres egy madarat, amelyik nincs még a helyén (ha nem talál ilyet, akkor nullát térít vissza). A mohó stratégiát az **amíg** végtelen ciklus valósítja meg, amelyből **return** utasítással lépünk ki (és vissza is térünk a függvényből), ha már minden madár a helyén van.

```
madarak(f[],n)
   minden i=1,n+1 végezd
      ha f[i]==0 akkor
         ü_f=i
      különben
         m[f[i]]=i
      vége ha
   vége minden
   amíg IGAZ végezd
      ha ü_f ≠ n+1 akkor
         r=ü_f
            // az üres fa sorszámával megegyező sorszámú
            // madár fog repülni
         ki: "az ",r,"-edik madár átrepül az ",
              m[r],"-edik fáról az ",ü_f,"-edik fára"
         f[m[r]]=0
            // a fa, ahonnan elrepült a madár, üres lesz
```

```
            f[ü_f]=r
               // a fán, ahova odarepült, az illető madár lesz
            ü_f=m[r]
               // az lesz az üres fa, ahonnan elrepült a madár
            m[r]=r // a madár a saját fájára került
         különben
            r=keres_madár(m,n)
               // keresünk egy madarat, amely az (n+1)-edik
               // fára kell repüljön
            ha r == 0 akkor
               return // minden madár a saját fáján van
            vége ha
            ki: "az ",r,"-edik madár átrepül az ",m[r],"-edik
                fáról az ",ü_f,"-edik fára"
            f[m[r]]=0
               // a fa, ahonnan elrepült a madár, üres lesz
            f[ü_f]=r
               //az (n+1)-edik fán, ahova odarepült,
               // az illető madár lesz
            ü_f=m[r]
               // az lesz az üres fa, ahonnan elrepült a madár
            m[r]=n+1
               // a madár az (n+1)-edik fára került
         vége ha
      vége amíg
   vége madarak

   keres_madár(m[],n)
      minden i=1,n végezd
         ha m[i] ≠ i akkor
            return i
         vége ha
      vége minden
      return 0
   vége keres_madár
```

Az algoritmus időigényének meghatározását az olvasóra bízzuk.

5.5. Legrövidebb utak: Adott egy $n \times n$ méretű d mátrix, amely egy n várost összekötő úthálózatot ábrázol. A `d[i][j]` elem az i és j városok közti közvetlen út hosszát tárolja (ha két város közt nincs direkt út, a megfelelő mátrixelemek értéke ∞). Határozzuk meg az első várostól az összes többihez vezető *legrövidebb* utakat és ezeknek a hosszát.

Megoldás: Ennek a feladatnak a megoldásánál kissé rejtettebb a mohó stratégia. Itt is a sablonra építünk. A kandidátusok a városok lesznek. A városok arra kandidálnak, hogy meg legyen határozva az első várostól hozzájuk vezető legrövidebb út. Mivel az első várostól önmagáig nyilván nulla hosszú a legrövidebb út, ezért ezt a várost már kezdetben áttehetjük a megoldáshalmazba. Végül mindenik városnak a megoldáshalmazba kell kerülnie (tehát a bővíthető függvény elveszti szerepét). A `v[1..n]` tömbben fogjuk nyilvántartani, hogy mely városok vannak még a kandidátushalmazban (`v[i] = 1`), és melyek kerültek már át a megoldáshalmazba (`v[i] = 0`).

Az `lru[1..n]` tömb `lru[i]` elemében ($i = 1, n$) az első várostól az i-edik városhoz – *a már megoldáshalmazban lévő városokon keresztül* – vezető legrövidebb út hosszát tároljuk. Mivel kezdetben csak az első város van a megoldáshalmazban, ezért az `lru` tömb kezdőértékként a közvetlen utak hosszát kapja. Majd, fokozatos finomítással, lépésről lépésre átalakítjuk, hogy végül a legrövidebb utak hosszát tartalmazza. Egy `ue[1..n]` tömböt is használunk, amelynek az `ue[i]` eleme az i-edik városhoz vezető legrövidebb út utolsó előtti állomását tartalmazza. Mivel a direkt utakból indulunk ki, ezért kezdetben az `ue` tömböt egyesekkel töltjük fel (mindenik úton pillanatnyilag az első város az utolsó előtti). Az `lru` tömb finomításával párhuzamosan az `ue` tömböt is finomítjuk. Az `ue` tömb alapján rekonstruáljuk majd magukat a legrövidebb utakat.

Az optimalitás alapelve a legrövidebb út problémájára a következőképpen jelenthető ki: *Egy legrövidebb út részútjai is legrövidebb utak.* Ebből adódik, hogy ha az első városból a v-edik városba tartó legrövidebb úton u az utolsó előtti állomás, akkor `lru[v]=lru[u]+d[u][v]`. Melyik városhoz határozzuk meg hamarabb a legrövidebb utat, az u-hoz, vagy a v-hez? Nyilván u-hoz, hiszen ez részét képezi a v-hez vezető legrövidebb útnak. Következtetésként kijelenthetjük, hogy a hosszabb „legrövidebb utak” felépíthetők a rövidebb „legrövidebb utakból”.

Tehát a mohó sorrend az lesz, hogy a legrövidebb utakat a hosszuk szerint növekvő sorrendben határozzuk meg. Bizonyítható, hogy egy adott lépésben az a következő legígéretesebb kandidáló város, amelyiknek az `lru` tömbbeli értéke a legkisebb. Ezt a kandidátusvárost határozza meg a `legígéretesebb` függvény. Valahányszor megtalálunk egy következő legrövidebb utat, frissítjük a még sorukra váró kandidátusokhoz vezető utak hosszát. A frissítés alapgondolata: mivel újabb város került a megoldáshalmazba, ellenőriznünk kell, hogy nem vezetnek-e rajta keresztül rövidebb utak a még kandidáló városokhoz.

Ez az algoritmus Dijkstra holland matematikus nevéhez fűződik. Időigénye $O(n^2)$.

```
legrövidebb_utak(d[][],n)
   minden i=1,n végezd
      v[i] = 1
      lru[i] = d[1][i]
      ue[i] = 1
   vége minden
   v[1] = 0
   minden i=2,n végezd
      x = legígéretesebb(lru,v,n)
      v[x] = 0
      frissít(x,lru,v,ue,n)
   vége minden
   minden i=2,n végezd
      ki: lru[i]
      ha lru[i] ≠ ∞ akkor
         kiír_legrövidebb_út(ue,i)
      vége ha
   vége minden
vége legrövidebb_utak

legígéretesebb(lru[],v[],n)
   min_távolság = ∞
   minden i=1,n végezd
      ha v[i] == 1 és lru[i] < min_távolság akkor
         min_távolság = lru[i]
         min_kandidátus = i
      vége ha
   vége minden
   return min_kandidátus
vége legígéretesebb

frissít(x,lru[],v[],ue[],n)
   minden i=1,n végezd
      ha v[i] == 1 és lru[x] + d[x][i] < lru[i] akkor
         lru[i] = lru[x] + d[x][i]
         ue[i] = x
      vége ha
   vége minden
vége frissít
```

A `kiír_legrövidebb_út` rekurzív eljárás kiindul az i-edik városból, utolsó előtti elemről utolsó előtti elemre szökdel, míg az egyes városba ér, majd a rekurzió visszaútján kiírja menetiránt a legrövidebb úton levő városokat.

```
kiír_legrövidebb_út(ue[],i)
   ha i ≠ 1 akkor
      kiír_legrövidebb_út(ue,ue[i])
   vége ha
   ki: i
vége kiír_legrövidebb_út
```

5.3. A mohó algoritmusok heurisztikája

A bevezetőben említettük, hogy bizonyos greedy algoritmusok nem mindig nyújtják az optimális megoldást, sőt egyes bemenetekre úgy találhatják, hogy nincs megoldás, holott a valóságban létezik. Az ilyen algoritmusokat heurisztikus megoldásoknak nevezzük. Az alábbiakban két olyan feladatot mutatunk be, amelyek esetében csak heurisztikus mohó algoritmusok ismertek. Mindkét esetben a „tökéletes megoldást" csak egy backtracking stratégia biztosítaná garantáltan. Mivel ezen algoritmusok időigénye nagy bemenetekre túl nagy lehet, úgy dönthetünk, hogy beérjük a mohó algoritmusok nyújtotta elég jó megoldásokkal.

5.6. Térképszínezés: Adott egy térkép, amely n országot tartalmaz, és ismert, melyik ország melyikkel szomszédos. Színezzük ki a térképet úgy, hogy ne kapjon két szomszédos ország azonos színt, és minimális legyen a használt színek száma.

Egy greedy megközelítés a következő lehetne: Veszem az első színt és kifestem vele a lehető legtöbb országot. A következő színekkel hasonlóképpen járok el. Bár bizonyítható, hogy bármely térkép kifesthető legfentebb négy színnel, az imént felvázolt algoritmus nem mindig fogja megtalálni a minimális színű megoldást.

5.7. Utazó kereskedő: Ismert egy ország úthálózata, azaz, hogy mely városok között vannak direkt utak, és mekkora ezek hossza. Határozzuk meg (egy utazó kereskedő részére) a legrövidebb olyan körutat, amely minden várost érint egyszer és csakis egyszer (kivéve az indulási várost).

A mohó gondolatmenet szerint az utazó kereskedő mindig az aktuális városból közvetlen elérhető legközelebbi – még nem érintett – várost választaná. Nem nehéz találni olyan példákat, amikor ez az algoritmus nem adná meg a legjobb megoldást, vagy egyszerűen nem találna megoldást (holott létezik).

Az 5.6. és 5.7. feladatokat megoldó heurisztikus algoritmusok implementálását az olvasóra hagyjuk.

5.4. Kitűzött feladatok

(A csillaggal megjelölt feladatok esetében heurisztikus algoritmusokat várunk.)

5.1. Többszörös összefésülés: Adott több növekvő számsorozat, amelyeknek hosszúságai $n_1, n_2, \ldots, n_n$. Fésüljük össze őket kettőnként egyetlen növekvő számsorozattá, *minimális* idő alatt. (Két számsorozat összefésüléséhez szükséges idő arányos az elemszámaik összegével.)

5.2. Ütemezés: Adott n egységidejű munka 1-től n-ig megszámozva, amelyeket egymást követve n időegység alatt szeretnénk elvégezni ugyanazzal a processzorral. Ismert, hogy az egyes munkákat $d_1, d_2, \ldots, d_n$ határidők előtt kellene elvégezni, különben a $w_1, w_2, \ldots, w_n$ penalizációkra kerül sor. Határozzuk meg a munkák azon ütemezését, amellyel a legkisebb összpenalizáció jár.

5.3. Huffman-kód: A Huffman-kódolás a következő gondolatmenetet követi: megszámoljuk egy szövegben az egyes karakterek előfordulási számát, majd a gyakoribb karaktereket rövidebb, a ritkábbakat pedig hosszabb bináris kódokkal helyettesítjük. Írjunk egy olyan algoritmust, amely megvalósít egy ilyen kódolást.

5.4. Optimális társítás: Legyenek az $A = \{a_1, a_2, \ldots, a_n\}$ és $B = \{b_1, b_2, \ldots, b_m\}$ $(n \leqslant m)$ halmazok, amelyek elemei nem nulla egészek. Határozzuk meg B egy olyan $B_1 = \{x_1, x_2, \ldots, x_n\}$ részhalmazát, amelyre az $E = a_1 \cdot x_1 + a_2 \cdot x_2 + \ldots + a_n \cdot x_n$ kifejezés értéke maximális.

5.5. Buszállomások: Legyen a várost átszelő főút mentén n buszállomás 1-től n-ig megszámozva. Az egymást követő állomások között pedig

a távolságok (méterben) legyenek a_1, a_2, ..., a_{n-1}. Mely állomásokban álljon meg a gyorsjárat, ha azt szeretnénk, hogy a megállók száma maximális legyen, és az egymást követő megállók között legyen legalább x méter távolság?

5.6. Részsorozatra bontás: Egy n elemű számsorozatot bontsunk minimális számú, nem föltétlen egymás utáni elemekből álló növekvő *részsorozatra.*

5.7. Turizmus: Egy turisztikai hivatal, amely n idegenvezetőt foglalkoztat, a k napos szezon bármely napján kész indítani m napos kirándulásokat. P igénylés érkezik. Mindenik csoport megjelöli, hogy a szezon melyik napján szeretne indulni ($a_1, a_2, \ldots, a_p$). Maximum hány kérést tud kielégíteni a hivatal?

5.8. Aranyzsákok: Adott egy n elemű számsorozat, amelynek elemei egy-egy zsákban található arany érmék számát ábrázolják. Minimális lépésből egyenlítsük ki az n zsák tartalmát (ha ez lehetséges). Egy lépésen azt értjük, hogy valamelyik zsákból valamennyi érmét átteszünk valamelyik másik zsákba.

5.9. Kiállítás*: Egy $n \times m$ méretű bináris mátrix 1-es elemei egy kiállítási csarnok azon pontjainak helyzetét ábrázolják, amelyekben képek találhatók, és ezért folyamatosan figyelni kell őket. Egy (i, j) pozíció fölé helyezett őrszem képes figyelni az i-edik sorban és a j-edik oszlopban található képeket, kivéve a pontosan alatta elhelyezkedőt (persze ez csak akkor jelent gondot, ha az (i, j) pozícióban van kép). Legkevesebb hány őrszemre van szükség a képtár felügyeletéhez?

5.10. Optimális sorrend*: Adott n pont a síkban a koordinátáik által. Milyen sorrendbe kössük egymás után a pontokat úgy, hogy ez minimális hosszúságú kábelt igényeljen?

5.11. Optimális partíció*: Adott egy n elemű halmaz. Határozzuk meg a halmaz azon k elemű partícióját, amelyben a legnagyobb összegű részhalmaz összege minimális.

5.12. Díjazás*: Egy verseny n győztesét egy-egy kirándulással szeretnék jutalmazni. Mind a győzteseket, mind a kirándulásokat az $1, 2, \ldots, n$

természetes számokkal azonosítjuk. Minden díjazott ad egy prioritáslistát azon kirándulásokkal, amelyeket elfogadna jutalomként (nem minden lista tartalmazza föltétlenül az összes kirándulást). Egy versenyző megelégedettségi fokát a következőképpen „súlyozzuk": Ha a kiutalt kirándulás a prioritáslistájában az i-edik volt, akkor a megelégedettségi fokának súlya $n-i+1$ lesz. Ha egy személy kirándulás nélkül marad, akkor megelégedettségi fokát nullára becsüljük. Határozzuk meg a legjobb elosztást, amikor a megelégedettségek összsúlya a lehető legnagyobb.

6. FEJEZET

BACKTRACKING ÉS GREEDY KÉZ A KÉZBEN

E fejezet keretén belül három megoldott feladat által bemutatjuk, miként növelhető a backtracking és greedy stratégiák hatékonysága a kombinálásuk révén.

6.1. Hátizsák: Egy üzletben n tárgy (áru) található, amelyeknek ismert az áruk és a súlyuk. Az árakat a t bejegyzés típusú tömb elemei a mezőiben, a súlyokat pedig az elemek g mezőiben tároljuk, ahol $t[i].g$ ($i = 1, n$) természetes számok. Állapítsuk meg, hogy mely tárgyakat fogja magával vinni egy tolvaj ahhoz, hogy a lehető legnagyobb nyereséggel távozzon (a hátizsákja legtöbb G súlyt bír meg).

A feladat szövege két változatban is ismert:

a) a tárgyak elvághatók,

b) a tárgyak nem vághatók el.

Példa:

Bemenet: $n = 4$, $G = 5$, $t[1..4].g = \{2, 1, 3, 1\}$, $t[1..4].a = \{5, 2, 4, 1\}$

Kimenet:

a) $(1, 1, 2/3, 0)$ – jelentése: az első és második tárgy egészében, a harmadiknak pedig $2/3$ része kerül a hátizsákba (a negyedik áru az üzletben marad). Ez $29/3$ egység nyereséget jelent.

b) $(1, 0, 1, 0)$ – jelentése: az első és harmadik tárgy kerül a hátizsákba (a második és negyedik áru az üzletben marad). Ez 9 egység nyereséget jelent.

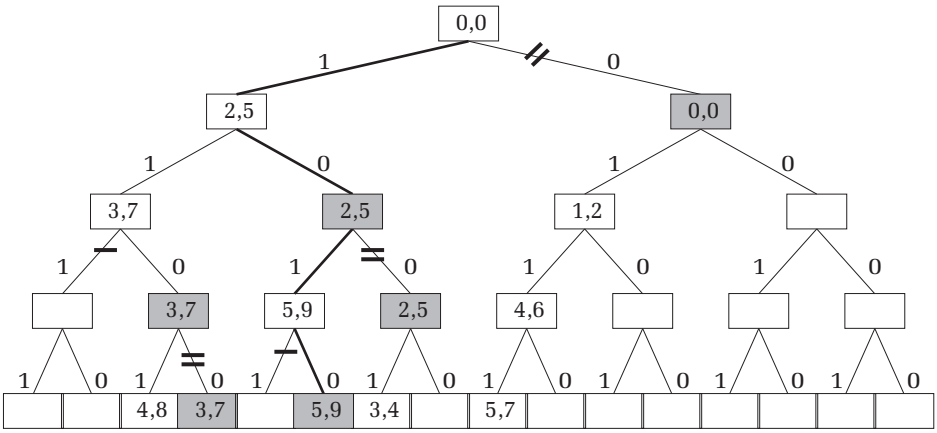
Megoldás: A feladat a) változata megoldható mohó stratégiával. A tárgyakat az ár/súly arány szerinti csökkenő sorrendben próbáljuk betenni a hátizsákba. Az első áruból, amelyik nem fér egészében a hátizsákba, levágunk annyit, hogy azzal teljesen megteljen. Bizonyítható, hogy ez a stratégia mindig az optimális megoldáshoz vezet.

Ha a feladat b) változatát próbáljuk mohó algoritmussal megoldani, a fenti megközelítés nem mindig vezet optimális megoldáshoz. A fenti példa esetében is a $(1, 1, 0, 1)$ kódú megoldást találnánk, holott az optimálisnak a kódja, amint láttuk, a $(1, 0, 1, 0)$.

Hogyan közelítené meg ezt a feladatot a backtracking stratégia? Mivel mind az n tárgy esetén két lehetőség közül választhatunk: vagy beletesszük a tárgyat a hátizsákba, vagy nem, a feladat megoldásainak tere egy n szintes bináris fa lesz. Ezt a fát látjuk lentebb, a példánkra felrajzolva. A fa gyökér–levél útjai a tárgyak halmazának részhalmazait ábrázolják. A már megszokott szóhasználat szerint nevezzük optimális gyökér–levél útnak azt, amelyik a feladat optimális megoldását képviseli, és optimális levélnek azt, amelyikhez ez az út vezet. Legprimitívebb változatában a backtracking generálhatná a tárgyak halmazának összes részhalmazát (bejárná mélységében a teljes bináris fát), kiválasztva közülük először azokat, amelyek belefférnek a hátizsákba, majd pedig azt, amelyik a legtöbb nyereséggel jár (maximumkeresést végzünk a potenciális megoldások között). Mivel az n elemű halmaz részhalmazainak száma 2^n (mindenik részhalmazhoz rendelhető egy n elemű bináris kód), ez a megoldás 2^n bonyolultságú algoritmust jelent. Hogyan lehetne optimalizálni ezt a backtracking algoritmust? Először is ügyelhetnénk arra, hogy csak olyan részhalmazokat generáljunk, amelyek befférnek a hátizsákba. Amint láttuk, ezek jelentik a feladat potenciális megoldásait. Az „ollót”, amely ebből a szempontból metszi meg a fát, vastagított szimpla vonalkával jelöltük. Egy másik optimalizálási lehetőség abban áll, hogy nyilvántarthatjuk, hogy minden csomópont képviselte állapotban mekkora összsúly van már a hátizsákban (ezt az értéket írtuk az egyes csomópontokba, a hozzájuk tartozó nyereséggel). Amennyiben a hátralevő tárgyak mind belefférnek a hátizsákba, akkor „gondolkozás nélkül” (anélkül, hogy bejárnánk az illető csomóponthoz tartozó részfát) az összest beletesszük (6.1. ábra).

Mindezen optimalizálások után is úgy találhatjuk, hogy míg a greedy stratégia nem volt kielégítő, a backtracking túl időigényes. Egy lehetséges (és jobb) megoldáshoz vezet a két módszer kombinálása (az „igazi” megoldást a dinamikus programozás módszere jelentené).

Hogyan lehetne még hatékonyabban optimalizálni a fenti backtracking algoritmust a mohó stratégia segítségével? Foglalkozzon a backtracking is mohó sorrendben (rendezzük a tárgyakat az értékük szerint csökkenő sorrendbe) a tárgyakkal, és először mindig azt a lehetőséget próbáljuk ki, hogy a tárgyat beletesszük (nyilván csak akkor, ha még belefér) a hátizsákba. Ez azt jelenti, hogy a bináris fában a bal ágakat kódoljuk 1-essel (beletesszük), és a jobb ágakat 0-val (nem tesszük bele). Ily módon a backtracking algoritmus is elsőnek éppen a mohó megoldást találja meg. Ezzel a megközelítéssel önmagában persze még nem



6.1. ábra.

növeltük az algoritmus hatékonyságát, csak azt biztosítjuk, hogy a potenciális megoldásokat egy bizonyos esélyességi sorrendben generáljuk. Mindez azonban előfeltétele a következő optimalizálásnak: mielőtt bejárnánk valamely csomópont jobb fiúrészfáját, először megvizsgáljuk, hogy van-e esély rá, hogy a megoldáslevél az illető részfában legyen. Ezt úgy tudjuk megtenni, hogy a szóban forgó részfában egy olyan mohó algoritmussal folytatjuk – vizsgálat céljából – a bejárást, amely elvághatja a tárgyakat. Ezt valósítja meg a `supremum` függvény, amely a részfa egyetlen gyökér–levél útján „szalad le”, tehát lineáris bonyolultságú. Az így kapott nyereség nagyobb lesz, mint a fa gyökeréből a megvizsgált részfa bármelyik leveléhez vezető út nyeresége. Az optimalizálás alapötlete a következő: ha ez a `supremum` érték sem nagyobb, mint a backtracking algoritmus folyamán addig talált legjobb megoldás, akkor az illető részfa biztosan nem tartalmazza az optimális levelet, tehát értelmetlen lenne bejárni (így teljesen le fogjuk metszeni a fáról). Az ábrán besatíroztuk azokat a csomópontokat, amelyekhez tartozó részfákra a példánk esetében meghívódik a `supremum` függvény. Vastagított dupla vonalkával jelöltük ezen optimalizálás „ollóját”. A „dupla ollóval” lemetszett részfákban bejelöltük a vizsgálati mohó utat. Az optimális gyökér–levél utat, amelynek kódja 1010, a vastagított vonal jelzi. Üresen hagytuk azokat a csomópontokat, amelyeknek bejárását sikerült elkerülni.

A `hátizsák` rekurzív backtracking eljárás k-adik szinti meghívása az `akt_g` és `akt_ny` paraméterekben megkapja, hogy az aktuális állapotban mekkora súly van már a hátizsákban, és mennyi nyereséggel.

Ezt a két értéket fogja *érték szerint* átadni a supremum függvénynek is, amely – amint már említettük – vizsgálat céljából mohó módon folytatja a $(k+2), \ldots, n$ tárgyak bepakolását (mivel jobb fiúról van szó, a $(k+1)$-edik tárgy nem kerül a hátizsákba). A max_ny és opt_x[] cím szerint átadott paraméterekben tárolódik a maximális nyereség és az optimális megoldás kódja.

```
supremum(t[],n,G,akt_g,akt_ny,k)
   minden i=k,n végezd
      ha akt_g+t[i].g <= G akkor
         akt_g=akt_g+t[i].g
         akt_ny=akt_ny+t[i].ár
      különben
         akt_ny=akt_ny+(G-akt_g)*t[i].ár/t[i].g
         return akt_ny
      vége ha
   vége minden
   return akt_ny
vége supremum

hátizsák(x[],t[],n,G,akt_g,akt_ny,k,max_ny,opt_x[])
   ha k == n akkor
      ha akt_ny > max_ny akkor
         max_ny=akt_ny
         opt_x[1..n]=x[1..n]
      vége ha
   különben
      ha akt_g+t[k+1].g <= G akkor
         x[k+1]=1
         hátizsák(x, t, n, G, akt_g+t[k+1].g,
                        akt_ny+t[k+1].ár, k+1, max_ny, opt_x)
      vége ha
      sup_ny=supremum(t,n,G,akt_g,akt_ny,k+2)
      ha sup_ny > max_ny akkor
         x[k+1]=0
         hátizsák(x, t, n, G, akt_g,
                                  akt_ny, k+1, max_ny, opt_x)
      vége ha
   vége ha
vége hátizsák
```

Az alábbiakban közöljük azt az algoritmusrészletet, amely tartalmazza a hátizsák eljárás meghívását és az optimális megoldás kiíratását:

```
...
   max_ny=0
   hátizsák(x,t,n,G,0,0,0,max_ny,opt_x)
   ki: max_ny,opt_x[1..n]
...
```

6.2. Alma/körte: Legyen $n+1$ szoba, amelyek vagonszerűen követik egymást. Az első n szoba mindenikében egy bizonyos mennyiségű alma és körte található (az $sz[1..n]$ bejegyzéstömb i-edik elemének $sz[i].a$, $sz[i].k$ mezői az i-edik szobában található alma, illetve körte mennyiségét tárolják). Egy elég nagy hátizsákkal rendelkező személynek a következőképpen kell végigmennie a szobákon, szobáról szobára haladva: megérkezve valamely i-edik szobába ($i = 1, n$), először kiüresíti a hátizsákját (a megfelelő gyümölcscsomóba), majd bepakolja a szobában található összes almát vagy összes körtét, és rakományát átviszi a következő szobába, az $(i+1)$-edikbe, ahol természetesen hasonlóan jár el (az első szobába üres hátizsákkal lép be). Állapítsuk meg, hogy az egyes szobákban mely típusú gyümölcsöket kell bepakolnia a személynek, ha azt szeretné, hogy minimális kalóriaveszteséggel érkezzen meg az $(n+1)$-edik szobába (két egymás utáni szoba között a szállított gyümölcsök számával egyenlő számú kalóriaegységet veszít a személy; a hátizsák kipakolása és megrakása nem jár kalóriaveszteséggel).

Megoldás: A feladat megoldásait az $x[1..n]$ bináris tömbben kódoljuk: az $x[i]$ tömbelem attól függően 0 vagy 1, hogy az i-edik szobában az almákat vagy a körtéket pakoljuk be a hátizsákba. A mohó megközelítés – miszerint mindig azzal a gyümölccsel megyünk tovább, amelyikből kevesebb van az illető szobában – nyilván nem kielégítő. Ezt szemlélteti az alábbi példa is:

$$n = 2, \quad sz[1..2] = \{(4,5),(3,6)\}.$$

Ha a mohó gondolatmenetet követjük, akkor az első szobából az almákat visszük tovább, hiszen ebből van kevesebb. Ez 4 kalória veszteséggel jár. Miután kipakoljuk a hátizsákunkat, a második szobában $4+3=7$ alma és 6 körte lesz. Itt a mohó algoritmus nyilván a körtéket pakolná fel, és vinné át 6 kalóriaveszteséggel a harmadik szobába. Tehát a mohó stratégia a $(0,1)$ kódú megoldást találja meg, ami 10 kalóriaveszteséggel jár. Létezik viszont ennél jobb megoldás is, az $(1,0)$ kódú (az első szobából a körtéket, a másodikból pedig az almákat visszük tovább) csak 8 kalóriát igényel.

A backtracking stratégia ennek a feladatnak a megoldására az lenne, hogy sorra generáljuk az összes potenciális megoldást (mind a 2^n darab n elemű bináris kódot), és kiválasztjuk közülük az optimálist, amelyik minimális kalóriaveszteséget okoz. A feladat megoldásterét egy n szintes teljes bináris fa képezi, amelynek mindenik gyökér–levél útja potenciális megoldásnak számít. Ez azt jelenti, hogy egy backtracking algoritmus a teljes fát be kellene járja (mélységében) ahhoz, hogy megtalálja az optimális megoldást képviselő optimális gyökér–levél utat, amelyik az optimális levélhez vezet. Hogyan lehetne, a mohó stratégia felhasználásával, céltudatosabbá tenni ezt az algoritmust? Feltételezzük, hogy a bináris fában a csomópontok bal fiai azt ábrázolják, hogy az almákat, a jobb fiak pedig azt, hogy a körtéket választottuk az illető csomópont által képviselt szobában. Minden csomópontban, mielőtt a mélységi bejárás továbblépne valamelyik fia irányába, megbecsüljük, hogy van-e esély rá, hogy az illető fiú részfa tartalmazza a megoldáslevelet. Tegyük fel, hogy a mélységi bejárás aktuális csomópontja a k-adik szobát képviseli (k-adik szinti csomópont). Meghatározzuk, hogy mekkora kalóriaveszteséggel járna úgy végigmenni a $k+1, \ldots, n$ sorszámú szobákon, hogy mindenik szobában csak azon gyümölcsrakások valamelyikét (mohó módon mindig a kisebbik rakást) pakoljuk be a hátizsákba, amelyek eredetileg ott voltak (amit az előző szobából hoztunk, azt kipakoljuk a szoba egy „sarkába” és otthagyjuk). Az így kapott érték biztosan kisebb lesz (infimum), mint bármely, az illető szobákon „szabályosan” (a gyümölcsöt, amelyet magunkkal hozunk az előző szobából, nem a „sarokba”, hanem a megfelelő típusú csomóba kell pakolni) áthaladó személy kalóriaveszteségé. A becsléseket az aktuális szobában (a k-adikban) a következőképpen végezzük el: ha `akt_kal` az eddigi kalóriaveszteség (amely az $1, \ldots, k$ sorszámú szobákon való végighaladásból adódott), `akt_a` és `akt_k` a k-adik szobában található alma, illetve körte mennyisége (ennyi kalóriaveszteséggel fog járni a k-adik szobából a $(k+1)$-edikbe való átjutás, attól függően, hogy az almákat vagy a körtéket választjuk), és `inf_kal[k+1]` a $k+1, \ldots, n$ sorszámú szobákon a fentebb leírtak szerinti végighaladással járó kalóriaveszteség, akkor

– a k-adik szobában csak abban az esetben választunk almát, ha

```
akt_kal+akt_a+inf_kal[k+1] < min_kal,
```

illetve

– a k-adik szobában csak abban az esetben választunk körtét, ha

```
akt_kal+akt_k+inf_kal[k+1] < min_kal
```

(`min_kal` tárolja az eddigi legjobb megoldás kalóriaveszteségét, az `opt_x[1..n]` tömb pedig az optimális megoldás kódját).

Úgy is megfogalmazhatnánk a megbecsülési feltételeket, hogy amennyiben az illető irányba egy ilyen alulbecslés sem kisebb, mint az eddig legjobbnak talált megoldás kalóriavesztesége, akkor biztosan „meddő” irányról van szó.

A fentebb közölt magyarázat után reméljük, hogy az alábbi optimalizált rekurzív backtracking eljárás megértése nem jelent gondot. Az eljárás azért kaphatja meg paraméterként az `inf_kal[1..n]` tömböt, mert ennek elemei előre meghatározhatók (lásd lentebb). A `min_kal` és `opt_x[]` paramétereket föltétlenül cím szerint kell átadnunk, hiszen ezekben téríti vissza az eljárás az eredményeket (memóriatakarékosság végett az `x[]`, `sz[]` és `inf_kal[]` tömbök is nyugodtan átadhatók cím szerint). Akkor jutottunk potenciális megoldáshoz, ha beléptünk az $(n+1)$-edik szobába. Ha a kapott potenciális megoldás kalóriavesztesége kisebb az addigi legjobb megoldás kalóriaveszteségénél, akkor az optimális megoldás adatait tároló változókat (`min_kal`, `opt_x[1..n]`) felülírjuk az aktuális megoldást azonosító adatokkal (`akt_kal`, `x[1..n]`).

```
alma_körte(x[],sz[],inf_kal[],n,min_kal,opt_x[],
                                    akt_a,akt_k,akt_kal,k)
   ha k == n+1 akkor
      ha akt_kal < min_kal akkor
         min_kal=akt_kal
         opt_x[1..n]=x[1..n]
      vége ha
   különben
      ha akt_kal+akt_a+inf_kal[k+1] < min_kal akkor
         x[k]=0
         alma_körte(x, sz, inf_kal, n, min_kal, opt_x,
         akt_a+sz[k+1].a, sz[k+1].k, akt_kal+akt_a, k+1)
      vége ha
      ha akt_kal+akt_k+inf_kal[k+1] < min_kal akkor
         x[k]=1
         alma_körte(x, sz, n, inf_kal, min_kal, opt_x,
         sz[k+1].a, akt_k+sz[k+1].k, akt_kal+akt_k, k+1)
      vége ha
   vége ha
vége alma_körte
```

Az alábbi algoritmusrészletben közöljük az `inf_kal[1..n]` tömb feltöltését megvalósító **minden** ciklust, meghívjuk az `alma_körte` eljárást, és kiírjuk az eredményeket.

```
...
   inf_kal[n+1]=0
   minden i=n,1 végezd
      inf_kal[i]=inf_kal[i+1]+min(sz[i].a,sz[i].k)
   vége minden
   min_kal=∞
   alma_körte(x,sz,inf_kal,n, min_kal,opt_x,
                                        sz[1].a,sz[1].k,0,1)
   ki: min_kal,opt_x[1..n]
...
```

6.3. Pénzösszeg: Adott egy S összeg és különböző értékű pénzérmék. A pénzérmék értékeit egy $a[1..n]$ tömbben tároljuk (minden pénzérmefajtából rendelkezésre áll akármennyi). Fizessük ki az S összeget, minimális számú pénzérmét felhasználva.

Megoldás: A feladat megoldását egy $db[1..n]$ tömbben lehetne kódolni, ahol $db[i]$ azt tárolná, hogy hány darabot használtunk az $a[i]$ értékű pénzérméből az S összeg kifizetéséhez. Ha a pénzérméink értékei k^0, k^1, k^2, ..., k^{n-1}, ($k \in \mathbb{N},\ k > 1$) alakúak, akkor a következő mohó stratégia bizonyíthatóan mindig megtalálja az optimális megoldást: a pénzérmékkel értékük szerint csökkenő sorrendben próbálunk fizetni, mindenfajta pénzérméből a lehető legtöbbet használva fel (ha a soron következő érme nagyobb értékű, mint a fennmaradt összeg, akkor ebből a fajtából nyilván egyet sem fogunk használni, úgymond átugorjuk). Ha a pénzérme értékei nem a fenti alakúak, akkor ez a megközelítés nem kielégítő. Ezt szemlélteti az alábbi példa is:

Példa:

$$S = 32, \quad n = 3, \quad a[1..3] = \{10, 5, 3\}.$$

A mohó algoritmus először felhasználna 3 darab 10-es pénzérmét, majd a fennmaradt összeget nem tudná kifizetni sem 5-ös, sem 3-as érmékkel. Ennek ellenére létezik megoldás: $db = (2, 0, 4)$ kódú: használunk két tízest, nem használunk ötöst, használunk négy hármast (összesen 6 pénzérmét használtunk).

Az alábbiakban bemutatjuk, hogyan lehet a backtracking és greedy stratégiák kombinálásával helyes algoritmust adni erre a feladatra.

Feltételezzük, hogy a pénzérmék értékeik szerint csökkenő sorrendbe vannak rendezve (ha nem így lenne, rendezzük őket). Kiindulunk a fent bemutatott mohó algoritmus megoldásából, a legesélyesebből. Ez, a példánkra, a $db = (3, 0, 0)$ kódú. Ha nem sikerült kifizetni a teljes összeget, akkor megkeressük a következő legesélyesebb megoldást. Hogyan? Visszalépünk (*back track*) a legkisebb értékű pénzérméhez, amit használtunk, csökkentjük a darabszámát eggyel, majd újraindítjuk a mohó algoritmust az ezt követő pénzérmétől a fennmaradt összegre... Ez az algoritmus elsőként garantáltan az optimális megoldást találja meg. Ha már a legnagyobb értékű pénzérmének is nullára csökkentettük a darabszámát, és még mindig nem találtunk megoldást, akkor ez azt jelenti, hogy az S összeg nem fizethető ki a megadott pénzérmetípusokkal.

Az alábbiakban felsoroljuk az ismételt mohó próbálkozások eredményeinek kódjait (kiemeljük dőlten azt az érmedarabszámot, amelyet az adott lépésben csökkentettünk eggyel; az ezt követő érmétől indítottuk újra a mohó algoritmust):

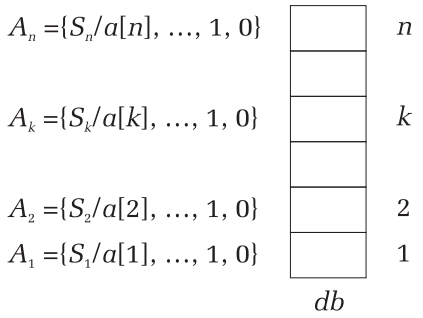
$$(\mathit{3},0,0) \quad (2,\mathit{2},0) \quad (2,1,\mathit{2}) \quad (2,1,\mathit{1}) \quad (2,\mathit{1},0) \quad (\mathbf{2},\mathbf{0},\mathbf{4})$$

```
fizetés(db[],a[],n,k,s)
   minden i=k,n végezd
      db[i]=s/a[i]
      s=s-db[i]*a[i]
   vége minden
   ha s == 0 akkor
      ki: db[1..n]
   különben
      i=n
      amíg i > 0 és db[i] ≠ 0 végezd
         i=i-1
      vége amíg
      ha i == 0 akkor
         ki: "Nincs megoldás"
      különben
         db[i]=db[i]-1
         fizetés(db,a,n,i+1,s+a[i])
      vége ha
   vége ha
vége fizetés
```

A `fizetés` eljárást a következőképpen fogjuk meghívni:

```
fizetés(db,a,n,1,S)
```

Erről a megközelítésről azt mondhatnánk, hogy „a backtracking segítette ki a greedy-t". De ez fordítva is felfogható. Tekinthetjük úgy is, mint olyan backtracking algoritmust, amelyben a greedy stratégia szerint állapítottuk meg, hogy milyen sorrendben foglalkozzon a pénzérmékkel (értékük szerint csökkenő sorrendben: $a[1] > a[2] > \ldots > a[n]$), és a greedy ajánlotta, hogy mindenik pénzérméből először a lehető legtöbbet próbáljon használni. Az első megoldás, amelyet a backtracking ebben a szellemben megtalál, megint csak az optimális lesz. Íme a backtracking alapábrája erre a helyzetre (lásd a backtracking technikával foglalkozó fejezetet):



6.2. ábra.

Az x tömb szerepét itt a db tömb tölti be. S_k-val jelöltük azt a pénzösszeget, amely még kifizetésre vár, amikor az algoritmus fellép a k-adik szintre. Tehát: $S_1 = S$, és $S_k = S - (db[1] \cdot a[1] + db[2] \cdot a[2] + \ldots + db[k-1]/a[k-1])$, ahol $k = 2, n$. Ezt a gondolatmenetet követi az alábbi tipikus backtracking algoritmus:

```
pénzérmék(db[],a[],n,k,f,stop)
   ha k == n akkor
      ha f == 0 akkor
         ki: db[1..n]
         stop=IGAZ
      vége ha
   különben
      minden db[k+1]=f/a[k+1],0,-1 végezd
         ha nem stop akkor
            pénzérmék(db,a,n,k+1,f-db[k+1]*a[k+1],stop)
```

```
            vége ha
          vége minden
       vége ha
    vége pénzérmék
```

Az f paraméterben az eljárásnak a k-adik szintre való meghívása azt kapja meg, hogy mekkora összeg maradt kifizetetlenül, amelyet a $(k+1), \ldots, n$ pénzérmékkel kell lefedni. A stop cím szerint átadott paraméter segít abban, hogy ahogy megkaptuk az első megoldást, leállíthassuk a rekurzív eljárást. Az eljárást az alábbi módon fogjuk meghívni:

```
Stop=HAMIS
pénzérmék(db,a,n,0,S,stop)
```

6.1. Kitűzött feladatok

6.1. Örökség: Két testvér n darab $a1, a2, .., an$ értékű pénzérmét örököl. A végrendeletben az áll, hogy csak akkor kaphatják meg örökségüket, ha el tudják osztani két egyenlő részre (különben oda kell adományozniuk örökségüket egy jótékonysági szervezetnek). Segítsünk az örökösökön, amennyiben ez lehetséges. Általánosítsuk a feladatot k örökösre.

6.2. Együttműködés: $2n$ informatikust, akik már együttműködtek a múltban egymással, meghívnak egy nagyméretű munka lebonyolítására. Az $a[1..2n][1..2n]$ mátrix $a[i][j]$ eleme akkor 1, ha az i informatikus kollaborált már a j-vel. Alkossunk n darab kéttagú csapatot úgy, hogy minimális legyen az olyan csapatok száma, amelyben nem ismerik még a tagok egymást.

7. FEJEZET

DINAMIKUS PROGRAMOZÁS

A dinamikus programozás nevet viselő programozási technika rendelkezik a legvonzóbb tulajdonságokkal a tárgyalt technikák közül. Számos olyan feladat van, amelyet bár megold „valahogy” a divide et impera is, a dinamikus programozás hatékonyan teszi ezt meg. Nem kevés azon optimalizálási feladatok száma sem, amelyek esetében, míg a greedy megközelítés nem kielégítő, a dinamikus programozás sikeresen alkalmazható.

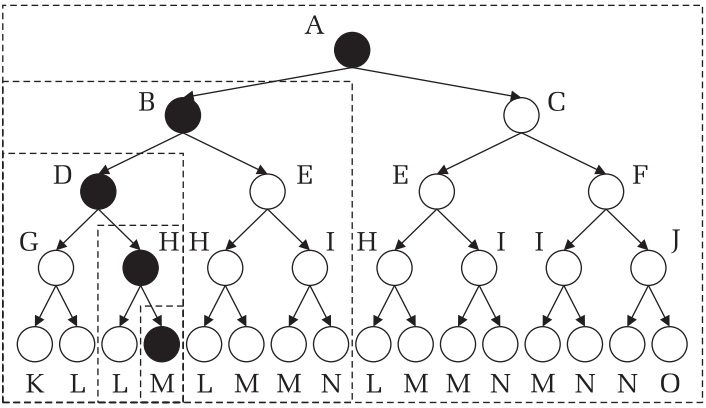
7.1. A döntési fa

A dinamikus programozást gyakran optimalizálási feladatok megoldására használjuk. Rendszerint arról van szó, hogy létezik egy célfüggvény, amelyet egy (optimális) döntéssorozat által optimalizálni kell. Minden optimalizálási feladathoz rendelhető egy gyökeres fa (fastruktúra), amelyet döntési fának fogunk nevezni. A gyökér a feladat kezdeti állapotát jelképezi, az első szinti csomópontok azokat az állapotokat, amelyekbe a feladat az első döntés nyomán kerülhet, a második szinten lévők azokat, amelyek a második döntésből adódhatnak stb. Egy csomópontnak annyi fia lesz, ahány lehetőség közül történik a választás az illető döntés alkalmával.

A 7.1. ábra egy olyan helyzetet mutat be, amikor négy döntés révén jutunk megoldáshoz. Minden döntés alkalmával két lehetőség közül választhatunk. A csomópontok címkéi a megfelelő állapotokat azonosítják.

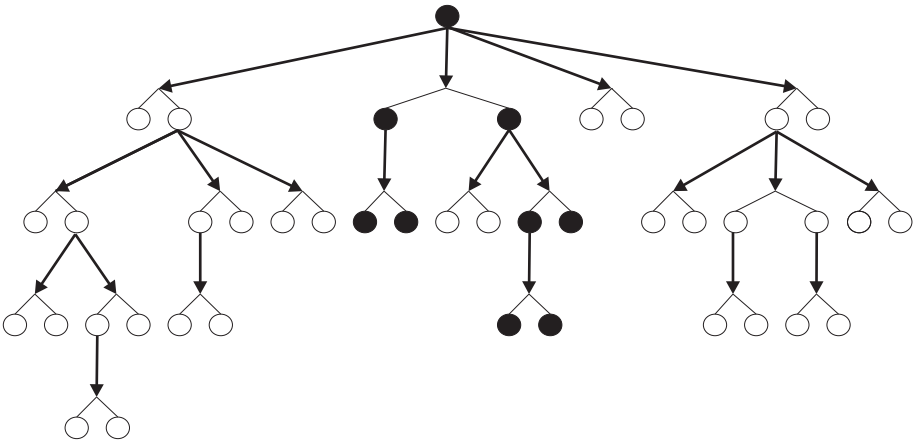
Két esetet különítünk el:

I. típusú döntési fa: *Minden döntéssel a feladat egy kisebb méretű hasonló feladattá redukálódik, amelyet az aktuális csomópont valamelyik részfája ábrázol.* Ilyenkor az optimális megoldást valamelyik gyökér–levél út fogja képviselni a döntési fában. Erre az esetre vonatkozik a 7.1. ábra is. A szaggatott vonalú téglalapokkal azt érzékeltettük, hogy – amennyiben a vastagított nyilakkal jelölt döntéssorozat az optimális – miként redukálódik általa a feladat egyre kisebb méretű részfeladataira.



7.1. ábra.

II. típusú döntési fa: *Minden döntéssel a feladat két vagy több kisebb méretű hasonló feladatra esik szét, amelyeket az aktuális csomópont megfelelő részfái ábrázolnak.* Az alábbi ábra egy olyan helyzetet mutat be, amikor minden döntéssel a feladat két részfeladatra bomlik. Feltételeztük, hogy a megvastagított bináris részfa ábrázolja az optimális részfeladatokra bontást. A dinamikus programozás ez esetre vonatkozó változatát az *„Optimális megosztás – optimális uralom”* névvel fogjuk illetni. Hogy miért kifejező ez a megnevezés, azt a későbbiekben tisztázzuk.



7.2. ábra.

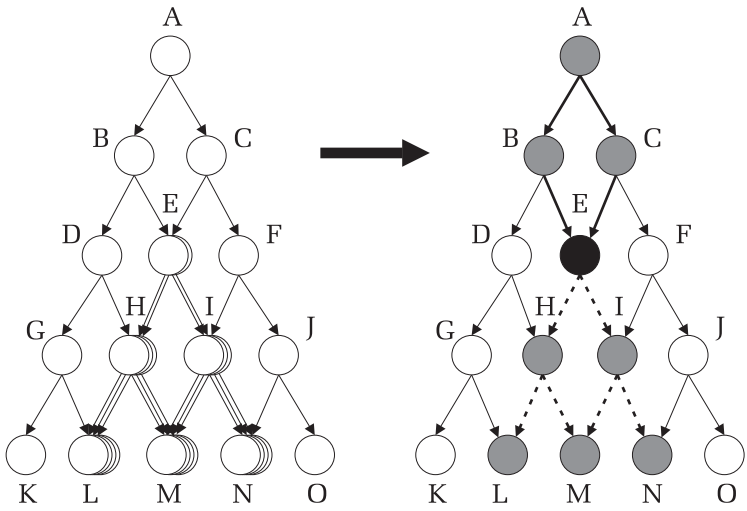
Dönteni nem jelent mást, mint választani. Természetesen attól függően, hogy miként választunk az egyes döntések alkalmával, más-más részfeladatokhoz jutunk. Kijelenthetjük hát, hogy egy döntési fa részfái azokat a részfeladatokat képviselik, amelyekké a feladat – az egyes döntéssorozatok által – redukálódhat (1. eset), illetve széteshet (2. eset).

Tekintettel arra, hogy a részfeladatok hasonlóak, beszélhetünk ezek általános alakjáról. Általános alakon mindig egy paraméteres alakot értünk. Átlátni egy feladat szerkezetét, többek között, az alábbiak tisztázását foglalja magába:

- Mi a részfeladatok általános alakja?
- Milyen paraméterek írják ezt le?
- Mely paraméterértékekre kapjuk az általános feladat határeseteiként az eredeti feladatot, illetve a triviális részfeladatokat?

7.2. Az összevont döntési fa

Bár a döntési fa csomópontjainak száma exponenciálisan függ az optimális döntéssorozat döntéseinek számától, gyakran megtörténik, hogy számos identikus csomópontot tartalmaz, amelyek nyilván azonos állapotokat ábrázolnak, és amelyeket természetesen ugyanazok a paraméterértékek jellemeznek.



7.3. ábra.

(Amint látni fogjuk később, minél nagyobb az identikus csomópontok száma, annál jobban tudja értékesíteni a dinamikus programozás az erősségeit.) Az első típusú döntési fák esetében ez a helyzet akkor alakul ki, ha különböző részdöntéssorozatok révén a feladat ugyanazon részfeladattá redukálódik. Egy ilyen helyzetet ábrázolt a 7.1. ábrán bemutatott döntési fa is. Csúsztassuk egymásra a fa identikus állapotait képviselő csomópontokat. Az így kapott adatszerkezetet *összevont döntési fának* fogjuk nevezni. Amint látható is, az összevont döntési fa már nem fastruktúra, hanem egy irányított gráf[1]. Az összevont fának pontosan annyi csomópontja lesz, ahány különböző állapotba kerülhet a feladat (ez a szám rendszerint csak polinom függvénye az optimális megoldáshoz vezető döntések számának). Bár az összevont döntési fa fogalmát az első esetre vonatkoztatva mutattuk be, amint látni fogjuk, kiterjeszthető a 2. típusú döntési fák esetére is.

7.3. Az optimalitás alapelve

A dinamikus programozás az optimalitás alapelvére épül. Úgy is fogalmazhatnánk, hogy ennek az alapelvnek az implementálása. Az optimalitás alapelve a következőképpen fogalmazható meg: *az optimális megoldás optimális részmegoldásokból épül fel.* Más szóval a feladat optimális megoldása felépíthető a részfeladatok optimális megoldásaiból. Ezt a stratégiát követi a dinamikus programozás: kiindul a triviális részfeladatok optimális megoldásaiból, és felépíti az egyre bonyolultabb részfeladatok optimális megoldásait, végül az eredeti feladatét. Ezért is szokás azt mondani, hogy az egyszerűtől halad a bonyolult fele, vagy hogy lentről felfele oldja meg a feladatot.

A dinamikus programozás egy másik jellemvonása, hogy nyilvántartja a már megoldott részfeladatok *optimális* megoldásait képviselő optimumértékeket (az optimalizálandó célfüggvény optimumértékeit az illető részfeladatokra vonatkozóan). Erre a célra rendszerint tömböt használunk, amelyet c-vel fogunk jelölni. Ez a tömb attól függően lesz egy-, két- vagy többdimenziós, hogy a részfeladatok általános alakját hány paraméter írja le. A tömb felhasznált elemeinek a száma

1 Valahányszor az összevont döntési fa gyökeréről, illetve leveleiről beszélünk, azokra a csomópontokra gondolunk, amelyek a döntési fában gyökér, illetve levél minőségben voltak jelen.

természetesen azonos az összevont döntési fa csomópontjainak a számával, azaz az egymástól különböző részfeladatok számával.

Az optimalitás alapelve konkrétan egy rekurzív képletben testesül meg, amely leírja matematikailag, hogy miként épülnek fel az egyszerűbb részfeladatok optimális megoldásából az egyre bonyolultabbak optimális megoldásai. Nyilván egy olyan képletről van szó, amelybe bele van építve az optimális döntéshozás módja. A rekurzív képlet alapvetően a c tömb elemeire van megfogalmazva, tehát a részfeladatok optimumértékeivel dolgozik.

Hogyan tükröződik az optimalitás alapelvének implementálása a döntési fán, az összevont döntési fán, illetve a részfeladatok optimumértékeit tároló tömbben?

1. Ha a növekvő döntési fa koronáján identikus állapotokat képviselő csomópontok jelennek meg, a fa csak azon ág irányába fog tovább növekedni, amelyik az illető részfeladat optimális megoldását képviseli.
2. A rekurzív képlet diktálta sorrendben oly módon messük az összevont döntési fa csomópontjait, hogy mindenikhez egyetlen út vezessen, mégpedig az, amelyik az optimális megoldást ábrázolja.
3. Az algoritmus magva alapvetően a c tömb megfelelő elemeinek a rekurzív képletből adódó stratégia szerinti feltöltését jelenti. Bár a c tömb egy az egyben csak a részfeladatok optimumértékeit tárolja, elegendő információt tartalmaz az optimális döntéssorozat rekonstruálásához. Gyakran célszerű, hogy a c tömb feltöltésekor magukat az optimális választásokat is eltároljuk valamilyen módon. Ez megkönnyítheti, sőt felgyorsíthatja az optimális döntéssorozat rekonstruálását.

Ha egy feladatra érvényes az optimalitás alapelve, ez jelentősen lecsökkentheti az optimális megoldás megépítésének idejét, hiszen az építkezésben támaszkodhatunk kizárólag a részfeladatok *optimális* megoldásaira.

Egy fontos megjegyzés: Figyeljünk fel arra, hogy az optimalitás alapelve nem azt mondja ki, hogy a feladat megoldásának a részfeladatok optimális megoldásaiból való *bármely* felépítése optimális lesz. Tekintsük példakét azt a feladatot, amikor egy adott összeget minimális számú ismert értékű pénzérmékkel kell kifizetni (létezik 1 értékű pénzérme, és mindenikfajta pénzérméből bármennyi van). Ezt a feladatot vizsgáltuk már a 6. fejezetben, és visszatérünk rá a 9. fejezetben újra. A feladat e változatára igaz az optimalitás alapelve,

de nem érvényes a mohó választás alapelve, így hát a leghatékonyabb megoldást a dinamikus programozás nyújtja.

Szolgáljon példaként az az eset, amikor 13 eurót kell kifizetni és 1, 5 valamint 10 eurósok állnak rendelkezésünkre. A 6 euró optimális kifizetése $5 + 1$, a 7 euróé pedig $5 + 1 + 1$. Ennek ellenére a 13 euró optimális kifizetése nem $5 + 1 + 5 + 1 + 1 + 1$, hanem a $10 + 1 + 1 + 1$, amely viszont felfogható a 10 és 3, vagy a 11 és 2, valamint a 12 euró és az 1 euró optimális kifizetéseinek az összegekét.

7.4. Dinamikus programozás az I. típusú döntési fán

Képzeljük újra magunk elé az első típusú döntési fát. A gyökér a kezdeti állapotot ábrázolja, amikor még a teljes feladat megoldásra vár. A fa megoldáslevelei a feladat megoldott állapotait képviselik, a gyökér–megoldáslevél utak pedig a potenciális megoldásokat (ezek között található a keresett optimális megoldás). A fa csomópontjai ábrázolta közbeeső állapotokban a feladatnak van egy már megoldott része, és egy még megoldásra váró része. Nevezzük ezeket az illető állapot prefix, illetve szúfix részfeladatának.

A gyökérből egy adott csomóponthoz vezető út azt ábrázolja, hogy milyen döntéssorozat vezet az illető állapothoz. E döntéssorozat úgy tekinthető, mint a szóban forgó állapot prefix részfeladatának egy megoldása. Az összevont döntési fában egymásra került identikus csomópontokhoz vezető különböző döntéssorozatok a megfelelő prefix részfeladat különböző megoldásaiként foghatók fel.

Egy állapot szúfix részfeladatát (ez nem más, mint az a részfeladat, amellyé a feladat az illető állapotban redukálódott) – amint már utaltunk rá – a megfelelő csomópont részfája (az, amelyiknek a csomópont a gyökere) képviseli a döntési fában. A szúfix típusú részfeladatok megoldásait nyilván a megfelelő részfa gyökér–levél útjai ábrázolják. A 7.3. ábrán az E állapot prefix feladatának megoldásait megvastagítottuk, a hozzá tartozó szúfix feladatot képviselő részfa éleit pedig pontozottan rajzoltuk.

Az összevont döntési fában az ugyanazon állapothoz tartozó prefix, illetve szúfix feladatok optimális megoldásait az illető csomóponthoz kapcsolódó optimális gyökér–csomópont, illetve optimális csomópont–levél út fogja képviselni.

Két alesetet különböztetünk meg attól függően, hogy az összevont döntési fa, mint irányított gráf, körmentes-e vagy sem.

7.4.1. Ha az összevont döntési fa körmentes

Mit jelent az egyszerűtől a bonyolult fele (lentről felfele) haladni? A prefix részfeladatok szempontjából a gyökér–levelek irány jelenti a lentről felfelét. A szúfix feladatok viszont éppen fordítva, a levelektől a gyökér irányába növekednek. Ez a dualitás elvezet a dinamikus programozás két változatához:

1. Gyökér–levelek irányú dinamikus programozás (Előre-módszer)
2. Levelek–gyökér irányú dinamikus programozás (Hátra-módszer)

Legyen $D_1, D_2, \ldots, D_{i-1}, D_i, D_{i+1}, \ldots, D_n$ az optimális döntéssorozat (a döntési fában ez vezet végig az optimális megoldást ábrázoló gyökér–levél úton). Tegyük fel, hogy ez a döntéssorozat kiindulva az S_0 kezdeti állapotból (amelyet a fa gyökere képvisel) az $S_1, S_2, \ldots, S_i, \ldots, S_n$ állapotokon „halad át” (az S_n állapotot a fa „optimális levele” ábrázolja). A dinamikus programozásnak a fentebb említett két változata az optimalitás alapelvét a következő alakokban használja.

Ha feltesszük, hogy $D_1, D_2, \ldots, D_n$ a feladat megoldását jelentő optimális döntéssorozat, akkor

1. a $D_1, D_2, \ldots, D_i$ ($i = 1, 2, \ldots, n-1$) alakú részdöntéssorozatok is optimálisak;
2. a $D_i, D_{i+1}, \ldots, D_n$ ($i = 2, \ldots, n$) alakú részdöntéssorozatok is optimálisak.

Hogy jobban átlássuk e stratégiákat, alkalmazzuk őket az alábbi feladatra.

7.1. Háromszög: Egy n soros négyzetes mátrix főátlóján és főátló alatti háromszögében természetes számok találhatók. Feltételezzük, hogy a mátrix egy a nevű kétdimenziós tömbben van eltárolva. Határozzuk meg a „leghosszabb” csúcsból ($a[1][1]$ elem) alapra (n-edik sor) vezető utat, figyelembe véve a következőket:

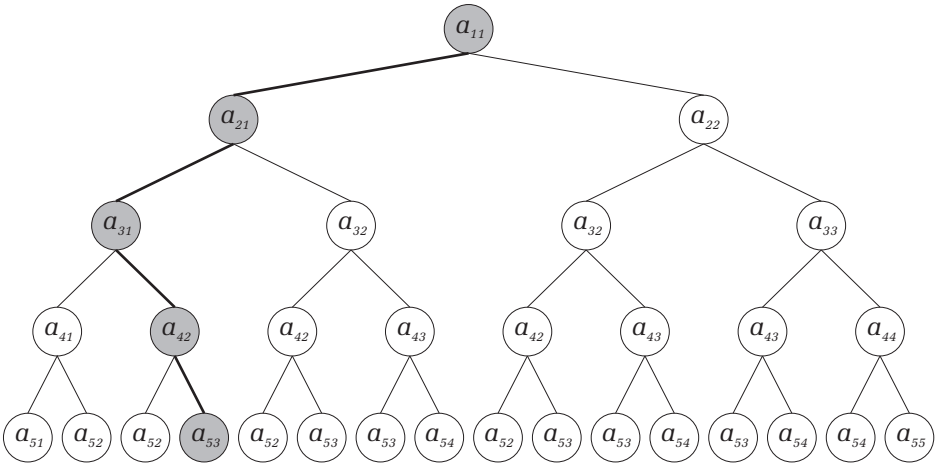
- egy úton az $a[i][j]$ elemet vagy az $a[i+1][j]$ (függőlegesen le), vagy az $a[i+1][j+1]$ elem (átlósan jobbra) követheti, ahol $1 \leqslant i < n$ és $1 \leqslant j < n$;
- egy út „hossza” alatt az út mentén található elemek összegét értjük.

Például, ha $n = 5$ és a mátrix az alábbi, akkor a „leghosszabb” olyan út, amely a csúcsból az alapra vezet, a besatírozott útvonal lesz, ennek hossza 37:

7				
5	9			
10	1	4		
2	7	3	1	
2	5	8	3	1

7.4. ábra.

A feladathoz a 7.5. ábrán látható döntési fa rendelhető.

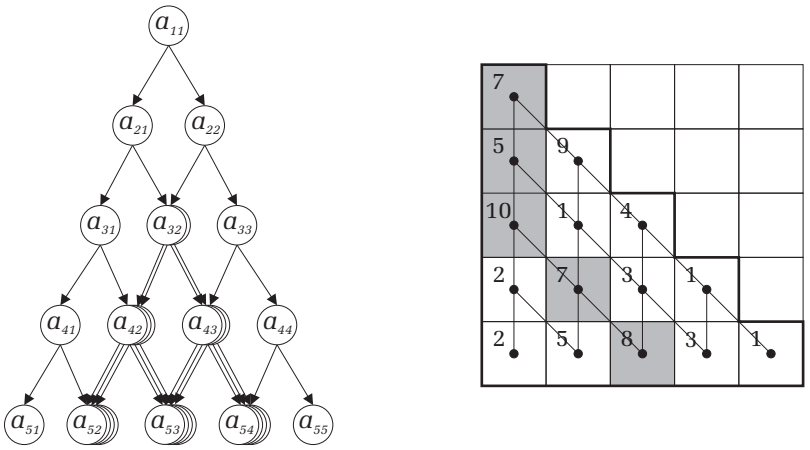


7.5. ábra.

Az általános prefix feladat: Határozzuk meg a csúcsból az (i, j) pozícióba vezető optimális utat!

Az általános szúfix feladat: Határozzuk meg az (i, j) pozícióból az alapra vezető optimális utat!

Tehát a feladat paraméterei i és j. Látható, hogy a döntési fának léteznek olyan csomópontjai, amelyeknek ugyanazok a paraméterértékek felelnek meg. Ezek a csomópontok identikus állapotokat képviselnek, amelyekhez ugyanaz a prefix, illetve szúfix feladat tartozik. Csúsztassuk egymásra az azonos állapotokat ábrázoló csomópontokat. A mellékelt összevont döntési fához jutunk. Ezt látjuk a tömbbe beépítve is.



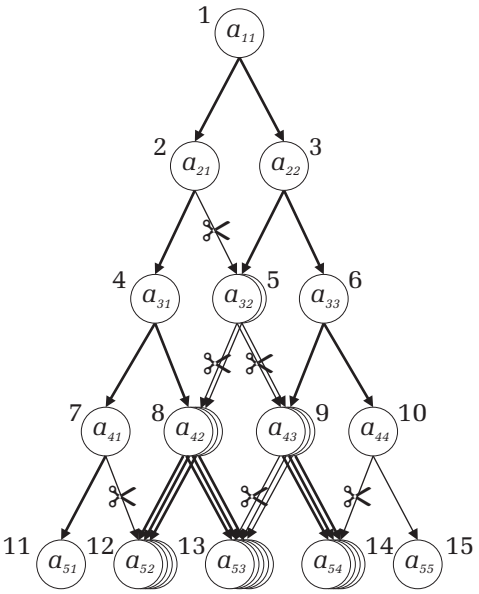
7.6. ábra.

Mivel a részfeladatokat két független paraméter határozza meg, az optimális megoldásaikhoz tartozó optimumértékek tárolásához kétdimenziós tömböt használunk (pontosabban a tömb főátlóján és a főátló alatt levő elemeit). Megfigyelhető, hogy az összevont döntési fa csomópontjainak száma megegyezik a tárolásra választott tömb felhasznált elemeinek számával.

Gyökér–levelek irányú változatban a $c[i][j]$ tömbelem az (i, j) állapothoz tartozó prefix feladat optimumértékét tárolja, azaz a csúcsból az (i, j) pozíciójú elemhez vezető optimális út hosszát. Ezzel szemben, a levelek–gyökér irányú változat esetén, az (i, j) pozícióból az alapra vezető legjobb út hossza kerül a $c[i][j]$ tömbelembe, ami a szúfix feladat optimumértékét jelenti.

7.4.1.1. Gyökér–levelek irányú dinamikus programozás

A gyökértől a levelek fele haladva megmetszünk minden csomópontot, csak a „legjobb” apaágat hagyva meg rajtuk, azt, amelyiken az illető csomópont prefix feladatának optimális megoldása halad át. Az ábrán bejelölt metszési sorrend csak egyike a lehetségeseknek. Az alapkövetelmény az, hogy amikor egy csomópont sorra kerül, az apacsomópontjai legyenek már megmetszve. Úgy is mondhatnánk, hogy a csomópontokkal topologikus sorrendben foglalkozunk. Ez a sorrend létezik, mivel az összevont döntési fa, mint irányított gráf, körmentes.

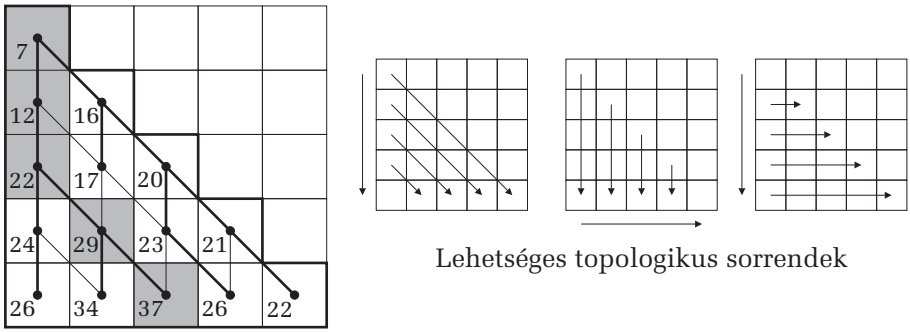


7.7. ábra.

A fentebb bemutatott algoritmus[2] implementálása nem jelent mást, mint a c tömb feltöltését – topologikus sorrendben – az alábbi rekurzív képlet szerint:

$$
\begin{aligned}
c[1][1] &= a[1][1] \\
c[i][1] &= a[i][1] + c[i-1][1], \quad 2 \leqslant i \leqslant n \\
c[i][i] &= a[i][i] + c[i-1][j-1], \quad 2 \leqslant i \leqslant n \\
c[i][j] &= a[i][j] + \max(c[i-1][j], c[i-1][j-1]), \quad 2 \leqslant i \leqslant n,\ 2 \leqslant j < i
\end{aligned}
$$

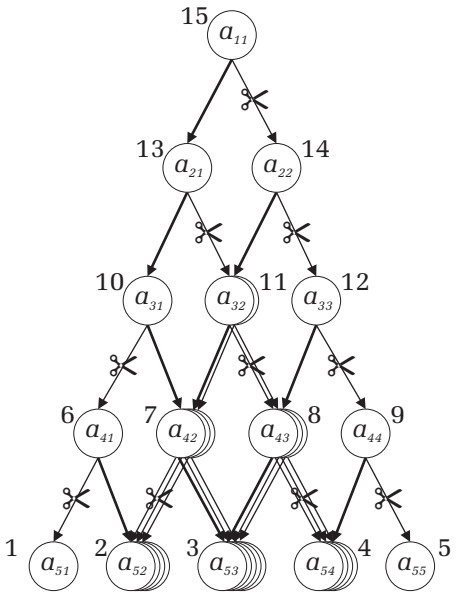
2 A c tömb n-edik sorának elemei a csúcsból az illető pozíciókba vezető legjobb utak hosszát tárolják. Nyilván ezek közül a legnagyobb jelenti a csúcsból az alapra vezető legjobb út hosszát. Ha szeretnénk magát az optimális utat is, a c tömb már elegendő információt tartalmaz ahhoz, hogy visszamenjünk az optimális levél–gyökér úton. Ha egy rekurzív eljárással „mászunk fel" az optimális úton, és a rekurzió visszaútján íratjuk ki az állomásokat, akkor a csúcstól az alap felé haladva – úgymond menetiránt – kapjuk meg az optimális megoldást.



7.8. ábra.

7.4.1.2. Levelek–gyökér irányú dinamikus programozás

A levelektől a gyökér fele haladva megmetszünk minden csomópontot, csak a „legjobb” fiút hagyva meg rajtuk, azt, amelyik az illető csomópont szúfix feladatát ábrázoló részfa legnagyobb optimumértékű fiú-részfájának a gyökere. Itt is a bejelölt metszési sorrend csak egyike a lehetséges sorrendeknek. Az alapkövetelmény ez esetben az, hogy

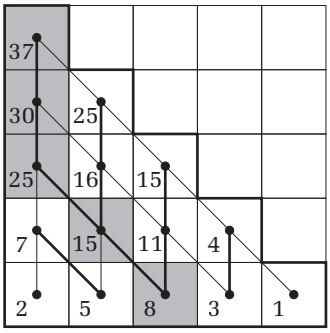


7.9. ábra.

amikor egy csomópont sorra kerül, a fiúcsomópontjai legyenek már megmetszve.

Ezen algoritmus[3] implementálása a c tömb feltöltése révén – fordított topologikus sorrendben – az alábbi rekurzív képlet szerint valósítható meg:

$$c[n][j] = a[n][j], \quad 1 \leqslant j \leqslant n$$
$$c[i][j] = a[i][j] + \max(c[i+1][j], c[i+1][j+1]), \quad n > i \geqslant 1,\ 1 \leqslant j \leqslant i$$



7.10. ábra.

7.4.2. Amikor az összevont döntési fa tartalmaz kört

Lássunk most egy olyan helyzetet, amikor az összevont döntési fa, mint irányított gráf, nem körmentes. Erre példa az alábbi feladat.

7.2. Irodaépület_1: Legyen az $a[1..n][1..m]$ mátrix, amelyet úgy fogunk fel, mint egy egyszintes, téglalap alakú irodaépületet. A mátrix elemei az irodákat képviselik, és azt tárolják, hogy mekkora illetéket kérnek el bárkitől, aki belépett az illető szobába. Bármely két szomszédos mátrixelem képviselte iroda között van ajtó. Belépni az épületbe csak az $(1,\ 1)$ pozíciójú irodánál lehet, kilépni pedig csak az (n, m) pozíciójúnál. Melyik az a minimális pénzveszteség, amellyel át lehet jutni az épületen?

3 Végül a $c[1][1]$ elem fogja tárolni az eredeti feladatnak mint legnagyobb szúfix feladatnak az optimumértékét. Mivel a c tömb tartalmazza az összes szúfix részfeladat optimumértékét, most már rendelkezünk kellő információval ahhoz (az a tömbben ez nem volt lehetséges), hogy egy mohó algoritmussal „leszaladjunk" a csúcsból alapra vezető optimális úton.

Példa: $n = 5, m = 4$

1	1	1	1
9	9	9	1
1	1	1	1
1	9	9	9
1	1	1	1

7.11. ábra.

A minimális pénzveszteség 14, amelyhez akkor jutunk, ha a besatírozott útvonalat követjük.

7.3. Irodaépület_2: Hasonló a feladat, csakhogy létezik néhány további kitétel:

- Léteznek olyan irodák is, amelyekben nem elvesznek pénzt, hanem adnak egy bizonyos összeget („negatív illeték").
- Létezhetnek egyirányú ajtók is (csak egyik oldalukon van kilincs). Ezt az információt a $b[1..n][1..m]$ tömbben tároljuk, amelynek elemei 4 elemű bináris karakterláncok, és azt kódolják, hogy ki lehet-e jutni az illető szobából fel, jobbra, le, illetve bal irányba.
- Feltételezzük, hogy nem létezik az épületben olyan iroda-körút, amelyet végigjárva gyarapodna a pénzünk.

Határozzuk meg az irodaépületen való legelőnyösebb átjutási módot.

Példa: $n = 5, m = 4$

Az a tömb

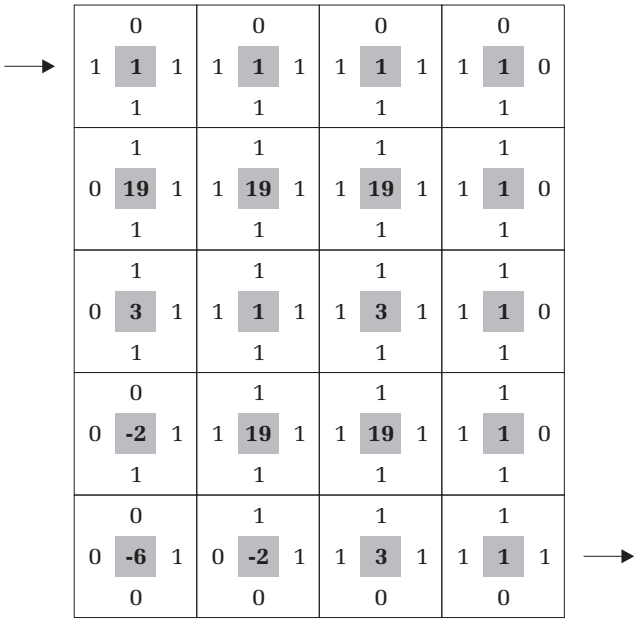
1	1	1	1
19	19	19	1
3	1	3	1
-2	19	19	1
-6	-2	3	1

A b tömb

0111	0111	0111	0011
1110	1111	1111	1011
1110	1111	1111	1011
0110	1111	1111	1011
0100	1100	1101	1101

7.12. ábra.

A legelőnyösebb út ez esetben is ugyanazon irodákon halad át és 7 pénznyereséggel jár (7.13. ábra).



7.13. ábra.

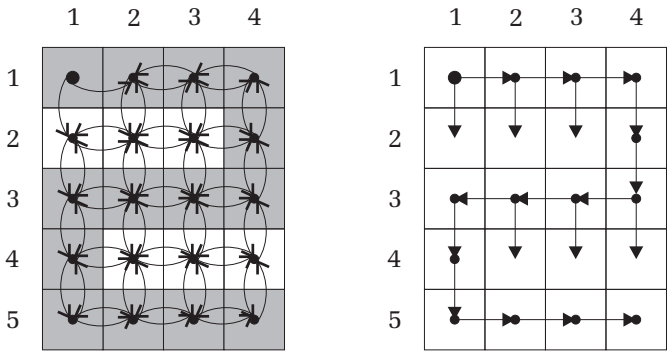
Egy általánosabb feladatot oldunk meg: meghatározzuk azokat a minimális költségű utakat, amelyek az $(1,1)$ koordinátájú szobától az összes többihez vezetnek.

Mivel a minimális költségű utat keressük, leszögezhetjük, hogy hurokmentesnek kell lennie. A feladathoz rendelhető fastruktúra (döntési fa) gyökere nyilván az $(1,1)$ pozíciójú irodát képviseli. Az egyes csomópontoknak annyi utóduk lesz, ahány irányba tovább lehet menni (anélkül, hogy hurok alakulna ki) az illető irodából. A levelek a zsákutcák végét ábrázolják. Az optimális megoldást a gyökérből az (n,m) pozíciójú irodát képviselő csomópontokhoz vezető utak közül a minimális költségű képviseli. Bár helyszűke miatt nem tudjuk felrajzolni az imént körvonalazott döntési fát, nem nehéz átlátni, hogy számos identikus csomópontja (a csomópontokat a képviselt irodák koordinátái azonosítják) van.

Csúsztassuk egymásra az identikus csomópontokat. Az így kapott összevont döntési fa (lásd lentebb) mint irányított gráf már nem körmentes. Ez abból adódik, hogy a döntési fában a csomópontok közti ős–utód viszony nem állandó. Például az $(1,2)$ csomópont bizonyos ágakon őse a $(2,2)$ csomópontnak ($(1,1)$, $(1,2)$, $(2,2)$, ...), más ágakon

pedig fordítva, a $(2,2)$ őse az $(1,2)$-nek ($(1,1)$, $(2,1)$, $(2,2)$, $(1,2)$, ...). Ebből adódóan az összevont döntési fában az $(1,2)$ és $(2,2)$ csomópontok között nyilván oda-vissza út lesz.

Ezek után a feladat úgy is megfogalmazható, hogy határozzuk meg az összevont döntési fa $(1,1)$ csomópontjától az (n,m) csomópontjához (általánosan az összes többi csomóponthoz) vezető minimális költségű utat. A bal oldali ábrán megvastagítottuk az optimális út éleit, illetve besatíroztuk az ennek megfelelő irodasort. A jobb oldali ábrán kiemeltük a döntési fából, a bal felső sarokban levő szobától, az összes többihez vezető optimális utakat képviselő részfát az irodaépület_1 feladatra vonatkoztatva.



7.14. ábra.

Általános részfeladatnak tekinthetjük a bal felső sarokból az (i,j) pozíciójú irodába vezető minimális költségű út meghatározását. Ez egyfajta gyökér–levelek irányú dinamikus programozást jelent. Az eredeti feladatot az $i=n$, $j=m$ értékekre, az egyetlen triviálist pedig az $i=1$, $j=1$ értékekre kapjuk. A $c[1..n][1..n]$ kétdimenziós tömb $c[i][j]$ elemében fogjuk eltárolni az (i,j) irodába vezető optimális út költségét.

7.4.2.1. Az optimalitás alapelvének ellenőrzése

Tegyük fel, hogy az (x,y) koordinátájú szobába az $(i_1=1, j_1=1)$, (i_2,j_2), ..., $(i_p=x, j_p=y)$ út a minimális költségű. Bebizonyítjuk, hogy az $(i_1=1, j_1=1)$, (i_2,j_2), ..., (i_k,j_k) alakú útszakaszok is mind optimálisak ($k=1,2,\ldots,p-1$).

Tegyük fel, hogy az (i_k,j_k) szobába a fentinél kisebb költségű útszakasz vezet. Legyen ez az $(u_1=1, v_1=1)$, (u_2,v_2), ..., $(u_r=i_k, v_r=j_k)$.

Két esetet különböztetünk meg. Ha ezen (i_k, j_k) szobába vezető jobb út nem keresztezi az (i_{k+1}, j_{k+1}), ..., $(i_p = x, j_p = y)$ útszakaszt, akkor az $(u_1 = 1, v_1 = 1)$, (u_2, v_2), ..., $(u_r = i_k, v_r = j_k)$, (i_{k+1}, j_{k+1}), ..., $(i_p = x, j_p = y)$ kisebb költségű, mint amiből kiindultunk ki. Ez viszont ellentmond a feltételnek, miszerint optimális útból indultunk ki. Továbbá, ha az $(u_1 = 1, v_1 = 1)$, (u_2, v_2), ..., $(u_r = i_k, v_r = j_k)$ út keresztezi az $(i_{k+1}, j_{k+1}), \ldots, (i_p = x, j_p = y)$ szakaszt, mondjuk az $(u_f = i_g, v_f = j_g)$ $(1 < f < r, k < g < p)$ szobában, akkor az $(u_1 = 1, v_1 = 1)$, (u_2, v_2), ..., $(u_f = i_g, v_f = j_g)$, (i_{g+1}, j_{g+1}), ..., $(i_p = x, j_p = y)$ út lesz kisebb költségű, mint az eredeti út, ami megint csak ellentmondás. Tehát az optimális utak optimális részutakból épülnek fel.

7.4.2.2. Az optimális megoldás meghatározása az 1. irodaépületben

Íme a rekurzív képlet, amely leírja az optimális megoldás szerkezetét:

$$c[1][1] = a[1][1]$$
$$c[i][j] = a[i][j] + \min\left(c[i-1][j], c[i][j+1], c[i+1][j], c[i][j-1]\right),$$

feltéve, hogy léteznek az illető pozíciójú szobák.

Amint látható, az általános képletet nem terheltük az i és j paraméterekre vonatkozó kitételekkel. Az ábrából jól érzékelhető, hogy az egyes szobákba milyen irányokból léphetünk be.

Az első irodaépület esetében – az optimalitás alapelvével összhangban – egy (i, j) irodához vezető minimális költségű út csakis olyan irodákon haladhat keresztül, amelyekhez a „legjobb" út kisebb költségű, mint az (i, j) irodához. Ez azt jelenti, hogy a $c[i][j]$ tömbelem meghatározásához effektív csak azokra a tömbelemértékekre van szükség, amelyek kisebb értékű minimális költségű utat képviselnek. Mindez arra utal, hogy a c tömb elemeit a képviselt irodákhoz vezető optimális utak optimumértékei szerinti növekvő sorrendben kell feltölteni.

Ez a sorrend biztosítható egy elsőbbségi sornak nevezett adatszerkezet segítségével. A sor elemei a c tömb elemei lesznek (emlékezzünk rá, hogy ezek, akárcsak az összevont döntési fa megfelelő csomópontjai, az illető irodákhoz vezető minimális költségű út meghatározásának részproblémáját képviselik), a bennük tárolt értékek növekvő sorrendjében.

Ezt szemlélteti az alábbi ábra. Feketével jelöltük azokat a tömbelemeket, amelyeket már eltávolítottunk a sorból. Ezek azokat az irodákat

képviselik, amelyekhez már meghatároztuk a minimális költségű út értékét. A sorban azok az irodák találhatók, amelyek elérhetők a „fekete irodákból". Ezeket szürkével jelöltük. A „szürke irodáknak" megfelelő tömbelemek azt tárolják, hogy milyen minimális költséggel tudunk eljutni hozzájuk, kizárólag fekete szobákon haladva át. A sor legnagyobb prioritású elemét, a sorelsőt, kiemeltük. Fehéren hagytuk azokat az irodákat, amelyek nem érhetők el a fekete irodákon keresztül.

Kezdetben a sorszerkezet az $(1, 1)$ pozíciójú irodát tartalmazza. Ez a tömbben úgy tükröződik, hogy a $c[1][1]$ elem szürke színt kap, és feltöltjük az $a[1][1]$ értékkel (ennyi illetéket kérnek el ebben az irodában). Ezt követően az algoritmus minden lépésben a következő műveleteket hajtja végre a sorelső irodán (legyenek ennek koordinátái (i_e, j_e)):

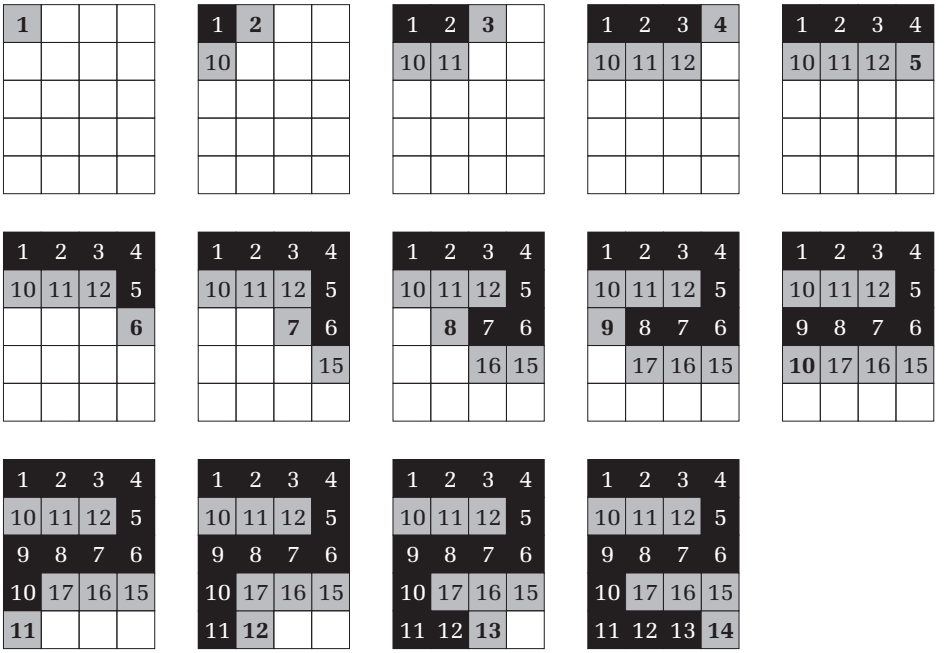
- A sorelső iroda színét feketére változtatja (ezzel törlődik a sorból mint amelyhez megtaláltuk a minimális költségű utat). Tudjuk, hogy a $c[i\_e][j\_e]$ érték minimális költségű utat képvisel, mivel az összes többi iroda (a szürkék és fehérek) vagy nem érhető el a már fekete irodákon keresztül (fehérek), vagy csak költségesebb úton érhető el (a többi szürke).
- Ha a sorelső irodának létezik olyan szürke szomszédja, amelyhez rajta keresztül kisebb költségű út vezet, mint amilyen az eddigi fekete irodákon keresztül vezetett, akkor frissítjük a megfelelő tömbelemet (ezzel az illető iroda nyilván előbbre kerül az elsőbbségi sorban). Ha az illető szürke szomszéd pozícióját (i_sz, j_sz)-szel jelöljük, akkor az alábbi műveletről lenne szó:

```
ha c[i_e][j_e]+a[i_sz][j_sz] > c[i_sz][j_sz] akkor
   c[i_sz][j_sz]=c[i_e][j_e]+a[i_sz][j_sz]
vége ha
```

  E feladat esetében speciálisan ez a helyzet nem alakulhat ki.
- A sorelső irodából elérhető „fehér irodákat" felvesszük a prioritás sorba (a megfelelő tömbelem színét szürkére változtatjuk, és feltöltjük). Ha egy ilyen fehér szomszéd koordinátái (i_f, j_f), akkor $c[i\_f][j\_f] = c[i\_e][j\_e] + a[i\_f][j\_f]$.

Mindezt addig ismételjük, míg az (n, m) pozíciójú iroda sorelső nem lesz, vagy ha minden irodához keressük a legrövidebb utat, akkor addig, míg ki nem ürül a sor.

Bizonyítható, hogy a fenti algoritmus tényleg az optimumértékek szerinti sorrendben fogja meghatározni az egyes irodákhoz vezető minimális költségű utakat.



7.15. ábra.

Az alábbiakban bemutatjuk a feladatot megoldó algoritmust. A `szín[1..n][1..n]` tömb az egyes szobák színét tárolja (0 – fehér, 1 – szürke, 2 – fekete). A `Q` egydimenziós, bejegyzés típusú tömbben az elsőbbségi sort szimuláljuk. A sorelemek a szobák koordinátáit (i és j mezők), valamint az ezeknek megfelelő aktuális c tömbbeli értéket (x mező) tárolják. Az `első` függvény visszatéríti a sorelső elemet, a `töröl_első` eljárás törli a sorelső elemet, a `beszúr_helyére` eljárás pedig beszúrja a rendezett sorba az illető szobának megfelelő elemet (feltöltve a megfelelő c tömbbeli elem indexeivel és értékével).

```
Irodaépület_1(a[][],n,m)
   szín[1..n][1..m]=0
   szín[1][1]=1
   c[1][1]=a[1][1]
   Q[1].i=1
   Q[1].j=1
   Q[1].x=c[1][1]
   e=1
   v=1
```

```
    amíg nem(első(Q).i==n és első(Q).j==m) végezd
       i_e= első(Q).i
       j_e= első(Q).j
       szín[i_e][j_e]=2
       töröl_első(Q)
       ha i_e>1 és szín[i_e-1][j_e]==0 akkor
          szín[i_e-1][j_e]=1
          c[i_e-1][j_e]=c[i_e][j_e]+a[i_e-1][j_e]
          beszúr_helyére(Q,i_e-1,j_e,c[i_e-1][j_e])
       vége ha
       ha j_e>1 és szín[i_e][j_e-1]==0 akkor
          szín[i_e][j_e-1]=1
          c[i_e][j_e-1]=c[i_e][j_e]+a[i_e][j_e-1]
          beszúr_helyére(Q,i_e,j_e-1,c[i_e][j_e-1])
       vége ha
       ha i_e<n és szín[i_e+1][j_e]==0 akkor
          szín[i_e+1][j_e]=1
          c[i_e+1][j_e]=c[i_e][j_e]+a[i_e+1][j_e]
          beszúr_helyére(Q,i_e+1,j_e,c[i_e+1][j_e])
       vége ha
       ha j_e<m és szín[i_e][j_e+1]==0 akkor
          szín[i_e][j_e+1]=1
          c[i_e][j_e+1]=c[i_e][j_e]+a[i_e][j_e+1]
          beszúr_helyére(Q,i_e,j_e+1, c[i_e][j_e+1])
       vége ha
    vége amíg
    ki: első(Q).x
vége Irodaépület_1
```

Ha minden irodáról eltároljuk, hogy melyik fekete iroda szomszédjaként került be a sorba, akkor könnyűszerrel meghatározhatjuk magát az irodasort is, amely optimálisan vezet az $(1,1)$ irodából az (n,m) irodába.

7.4.2.3. Az optimális megoldás meghatározása a 2. irodaépületben

Nem nehéz átlátni, hogy ebben az esetben a fenti algoritmus nem vezetne el az optimális megoldáshoz. A „negatív illetékek” jelenléte miatt a pillanatnyilag sorelső szürke irodához létezhet a kizárólag fekete irodákat érintő legkisebb költségű útnál kisebb költségű – „negatív illetékű” fehér irodákat érintő – út.

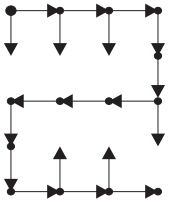
Milyen sorrendben töltsük fel ez esetben a c tömb elemeit? Mivel nincs lehetőség sem előre (nincs topologikus sorrend), sem „menet közben" (elsőbbségi sor segítségével) meghatározni a minimális költségű utak kiszámításának helyes sorrendjét, más módszerhez folyamodunk.

Első nekifutásból a c tömböt a sorfolytonos (fentről lefele, balról jobbra) bejárásából adódó értékekkel töltjük fel. Persze e bejárás alatt a $c[i][j]$ elem kitöltésekor csak a $c[i-1][j]$ és $c[i][j-1]$ szomszédokat tudjuk számításba venni (csak ezek lettek addigra már feltöltve), holott, ha az optimális utakon találhatók balra, illetve felfele mutató szakaszok (röviden: vissza-élek), akkor föltétlenül a $c[i][j+1]$ és $c[i+1][j]$ értékeket is figyelembe kellene vennünk. Például, a fenti példában $c[3][3]$ érték kiszámítása feltételezi a $c[3][4]$ érték ismeretét. Elmondható továbbá, hogy e feltöltés nyomán csak azokhoz a szobákhoz kapjuk meg a legrövidebb utakat, amelyeknek esetében ezek csak előre mutató (jobbra, illetve lefele) éleket tartalmaznak. Mi a megoldás? Újra és újra végigjárjuk a c tömböt, tovább finomítva rajta. E további átjárások alkalmával felülírjuk azokat a tömbelemeket, amelyekhez létezik előnyösebb út valamelyik szomszédos elemtől. Ezen alkalmakkal már mind a négy irányt figyelembe tudjuk venni. Ha egy újabb átjárás már nem eredményez semmilyen változtatást, azt jelenti, eljutottunk az optimális megoldáshoz. Megfigyelhető, hogy annyi finomító átjárásra van szükség, ahány vissza-él található az optimális utak részfájában (pontosabban a gyökér–levél utak maximális vissza-él száma). Ha $n \cdot m$ (az összevont döntési fa csomópontjainak száma) átjárás után még mindig tudunk finomítani, ez azt jelenti, hogy a feladat feltételeivel ellentétben, létezik az épületben olyan iroda-körút, amelynek körbejárásával gyarapodik a pénzünk.

Az alábbiakban közöljük a c tömb tartalmát a feltöltő, illetve finomító átjárások után. Kiemeltük azokat az elemeket, amelyek már optimális utakat képviselnek. A mellékelt ábrákon azt szerettük volna érzékeltetni, hogy miként épül az optimális megoldást képviselő részfa az optimalitás alapelvével összhangban lentről felfele, az egyszerűtől a bonyolult fele. Megvastagítva rajzoltuk azt a farészt, amely az adott lépésig már elkészült.

Az a tömb tartalma a példában:

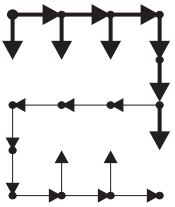
1	1	1	1
19	19	19	1
3	1	3	1
-2	19	19	1
-6	-2	3	1



7.16. ábra.

A c tömb tartalma a feltöltő átjárása után:

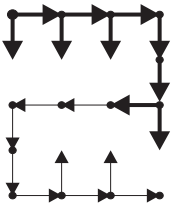
1	**2**	**3**	**4**
20	**21**	**22**	**5**
23	22	25	**6**
21	40	44	**7**
15	13	16	8



7.17. ábra.

A c tömb tartalma az első finomító átjárás után:

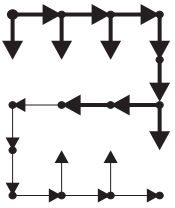
1	**2**	**3**	**4**
20	**21**	**22**	**5**
23	22	**9**	**6**
21	32	26	**7**
15	13	11	8



7.18. ábra.

A c tömb tartalma a második finomító átjárás után:

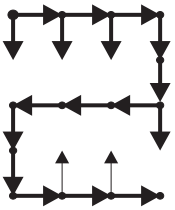
1	**2**	**3**	**4**
20	**21**	**22**	**5**
23	**10**	**9**	**6**
21	29	26	**7**
15	9	11	8



7.19. ábra.

A c tömb tartalma a harmadik finomító átjárás után:

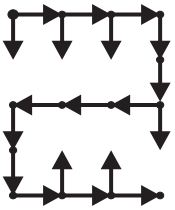
1	**2**	**3**	**4**
20	**21**	**22**	**5**
13	**10**	**9**	**6**
11	28	26	**7**
5	**3**	**6**	**7**



7.20. ábra.

A c tömb végleges alakjában (az ötödik átjárás után):

1	2	3	4
20	21	22	5
13	10	9	6
11	22	25	7
5	3	6	7



7.21. ábra.

Az `Irodaépület_2` eljárás implementálja a fentebb vázolt algoritmust. Az alábbi megvalósításban alkalmaztunk néhány programozói fogást. Azért, hogy ne kelljen rendhagyó módon kezeljük az első átjárást, a c tömböt feltöltöttük ∞ kezdőértékekkel. A szélső irodák kivételes kezelését azzal kerültük el, hogy szegélyeztük a c tömböt ∞ értékekkel. Azokat a finomításokat, amikor az aktuális szoba valamelyik szomszédjából nem lehet átjönni (nincs kilincs az illető ajtó túloldalán), úgy zártuk ki, hogy a megfelelő finomító kifejezéshez hozzáadtunk ∞-t. A `('1'-b[x][y][z])`$\cdot\infty$ kifejezés attól függően 0 vagy ∞, hogy az (x, y) irodának nyílik-e kifele a z irányú ajtója vagy sem. A t változót arra használjuk, hogy figyeljük, lett-e finomítva az aktuális `c[i][j]` tömbelem, a `kém`-et pedig arra, hogy ellenőrizzük, volt-e egyáltalán finomítás az aktuális átjárás alkalmával.

```
Irodaépület_2 (a[][], b[][], n, m)
   c[0..n+1][0..m+1]=∞
   c[1][1]=a[1][1]
   végezd
      kém=HAMIS
      minden i=1,n végezd
         minden j=1,m végezd
```

```
                t=c[i][j]
                c[i][j]=minimum(c[i][j],
                c[i-1][j]+a[i][j]+('1'-b[i-1][j][3])*∞,
                c[i][j+1]+a[i][j]+('1'-b[i][j+1][4])*∞,
                c[i+1][j]+a[i][j]+('1'-b[i+1][j][1])*∞,
                c[i][j-1]+a[i][j]+('1'-b[i][j-1][2])*∞)
                ha t≠c[i][j] akkor
                   kém=IGAZ
                vége ha
             vége minden
          vége minden
       amíg kém==IGAZ
       ki: c[n][m]
    vége Irodaépület_2
```

Ez a megközelítés nyilván az `Irodaépület_1` feladatot is megoldja, csakhogy nagyobb az időigénye.

A fenti algoritmus másképp is felfogható. Ahelyett, hogy arra figyelnénk, hogy miként finomíthatók a c tömb aktuális elemei a szomszédai révén, összpontosítsunk arra, hogy hogyan lehet az aktuális elemmel finomítani a szomszédain. Ezt a megközelítést implementálja az alábbi eljárás az `Irodaépület_1` feladatra.

```
c[1..n][1..m]=0
c[1][1]=a[1][1]
végezd
   kém=HAMIS
   minden i=1,n végezd
      minden j=1,m végezd
         //fel
         ha i>1 és (c[i-1][j]==0 vagy
                              c[i-1][j]>c[i][j]+a[i-1][j])
         akkor
            c[i-1][j]=c[i][j]+a[i-1][j]
            kém=IGAZ
         vége ha
         //le
         ha i<n és (c[i+1][j]==0 vagy
                              c[i+1][j]>c[i][j]+a[i+1][j])
         akkor
            c[i+1][j]=c[i][j]+a[i+1][j]
            kém=IGAZ
         vége ha
```

```
            //balra
            ha j>1 és (c[i][j-1]==0 vagy
                                  c[i][j-1]>c[i][j]+a[i][j-1])
            akkor
               c[i][j-1]=c[i][j]+a[i][j-1]
               kém=IGAZ
            vége ha
            //jobbra
            ha j<m és (c[i][j+1]==0 vagy
                                  c[i][j+1]>c[i][j]+a[i][j+1])
            akkor
               c[i][j+1]=c[i][j]+a[i][j+1]
               kém=IGAZ
            vége ha
         vége minden
      vége minden
   amíg kém==IGAZ
   ki: c[n][m]
```

7.4.3. Gráfelméleti szempontok

A fentiekből kiderült, hogy amennyiben megelevenítjük a dinamikus programozással megoldandó feladat mögött meghúzódó döntési fát, majd az összevont döntési fát, ezzel visszavezetjük a feladatot egy gráfelméleti feladatra. Egészen pontosan egy súlyozott, irányított gráfban egy adott csúcsból (a gyökérből) az összes többihez vezető minimális súlyú utak meghatározásának problémájára. Hogyan történik e gráf tárolása? A részfeladatok optimális megoldásait tároló c tömb felhasznált elemei a gráf csomópontjait képviselik. A gráf élei, implicit módon, a feladat optimális részstruktúráját matematikailag leíró rekurzív képletből adódnak. Konkrétan: a gráfnak akkor létezik az (A, B) irányított éle, ha – a rekurzív képlet értelmében – az A csomópontnak megfelelő tömbelem közvetlenül függ a B csomópontot képviselő tömbelemtől. Például egy (i, j) csomópontba a háromszög feladat esetében az $(i-1, j-1)$ és $(i-1, j)$ csomópontokból, az irodaépület feladatban pedig az $(i-1, j)$, $(i, j+1)$, $(i+1, j)$ és $(i, j-1)$ csomópontokból vezettek élek (feltéve, ha az illető csomópontok léteztek). A élek súlyai a bemeneti adatokból általában azonnal adódnak. Mindkét példafeladatunk esetében az (i, j) csomópontba befutó élek mindegyike $a[i][j]$ súlyúnak tekinthető. Amint már többször is utaltunk rá, a minimális utak költségeit a c tömb elemei fogják tárolni.

Három esetet különböztethetünk meg, amelyek mindenikére közismert gráf-algoritmusok léteznek:

1. *Topologikus sorrend alapú dinamikus programozás:* Az összevont döntési fa körmentes. Ebben az esetben létezik a csomópontok topologikus sorrendjére alapozó, $O(N+M)$ futási idejű algoritmus a feladatra (N és M a gráf csomópontjainak, illetve éleinek száma). Ezt az algoritmust mutattuk be a háromszög feladat kapcsán.
2. *Dijkstra-algoritmus alapú dinamikus programozás:* Az összevont döntési fa tartalmaz ugyan kört, de nincs negatív éle. Erre az esetre vonatkozik *Dijkstra* algoritmusa, amely egy elsőbbségi sor segítségével az optimumértékek szerinti növekvő sorrendben határozza meg a minimális súlyú utakat. *Dijkstra* algoritmusának bonyolultsága $O(N^2)$, de javítható, ha az elsőbbségi sort bináris kupac ($O((N+M)\lg N)$), vagy Fibonacci-kupac ($N\lg N+M$) segítségével implementáljuk. (Irodaépület_1 feladat)
3. *Belmann–Ford-algoritmus alapú dinamikus programozás:* Az összevont döntési fának van negatív éle is, de nincs a gyökérből elérhető negatív összsúlyú köre. Ezt a feladatot oldja meg *Belmann–Ford* algoritmusa: ($O(NM)$). (Irodaépület_2 feladat)

7.5. „Optimális megosztás – optimális uralom”

Az eddigiekben arra az esetre alkalmaztuk a dinamikus programozást, amikor a feladat minden döntéssel *egy* hasonló, egyszerűbb feladattá *redukálódott.* A „redukálódott” kifejezéssel azt szerettük volna érzékeltetni, hogy minden döntéssel megoldunk valamennyit a feladatból. Ezért beszélhettünk arról, hogy minden lépésben van a feladatnak egy már megoldott és egy még megoldatlan része. Ezek voltak az illető állapothoz tartozó prefix, illetve szúfix részfeladatok. Úgy is mondhatnánk, hogy a D_1, D_2, ..., D_n döntéssorozat nyomán a feladat lépésről lépésre megoldódott. A kérdés csak az volt, miként hozzuk meg a döntéseket, hogy optimális megoldáshoz jussunk. Az ilyen típusú feladatok első ránézésre „mohó-feladatnak” tűnnek, csakhogy nem lévén érvényes rájuk a mohó választás alapelve, „kénytelenek vagyunk” a dinamikus programozáshoz folyamodni.

A dinamikus programozás egy másik területét olyan feladatok képezik, amelyek inkább az „oszd meg és uralkodj” technikához állnak

közelebb. Az „oszd meg és uralkodj” felosztja a feladatot két vagy több hasonló, egyszerűbb részfeladatra, és ezek megoldásaiból építi fel az eredeti feladat megoldását (a részfeladatok megoldásánál természetesen hasonlóképpen jár el, egészen addig, míg triviális részfeladatokhoz nem jut). Ha a feladat részfeladatokra bontása többféleképpen is elvégezhető, akkor felmerül az optimális felbontás kérdése! Ilyen esetben már többről van szó, mint egy egyszerű „oszd meg és uralkodj” feladatról. Ha minden lépésben meghatározható lenne mohó döntéssel, hogy hol van az optimális „vágás”, akkor azt mondjuk, hogy a mohó technika dolgozott az „oszd meg és uralkodj” keze alá. Ha viszont nincs elegendő információnk mohó vágásokhoz, de a feladat feldarabolására érvényes az optimalitás alapelve, akkor a dinamikus programozás segíthet az „oszd meg és uralkodj” technikának megtalálni az optimális megoldást.

Tehát egy olyan stratégiáról van szó, amely az „oszd meg és uralkodj” technika „oszd meg” fázisába beépíti a dinamikus programozást. (Ezt nevezi *Tudor Sorin vegyes módszer*nek.) Az optimalizálási feladatokra jellemző D_1, D_2, ..., D_n optimális döntéssorozat alapvetően a feladat optimális részfeladatokra bontását jelenti. Az aktuális feladat minden döntéssel *szétbomlik* hasonló, egyszerűbb részfeladatokra (ellentétben azzal az esettel, amikor egyetlen hasonló, egyszerűbb feladattá *redukálódott*). Amikor azt mondjuk, hogy optimális részfeladatokra bontás, arra gondolunk, hogy abból a szempontból optimális, hogy az „oszd meg és uralkodj” „uralkodj” fázisában az optimális megoldás felépülését vonja maga után. Mivel a D_1, D_2, ..., D_n döntéssorozat csak arról szól, hogy miként bomlik optimálisan részfeladataira a feladat, egy közbeeső S_i állapotban (amelybe a D_1, D_2, ..., D_i döntések nyomán jutottunk) nem beszélhetünk a feladat egy már megoldott részéről (prefix részfeladatról). Ebből adódóan, ilyen esetben a dinamikus programozásnak csak a levelek–gyökér irányú változata jöhet szóba. Ez összhangban van azzal a ténnyel, hogy az „oszd meg és uralkodj” technika csak azután old meg egy aparészfeladatot, ha a fiúrészfeladatait már megoldotta (a fentebb bevezetett terminológia szerint a részfeladaton szúfix típusú részfeladatokat értünk).

7.5.1. Az optimalitás alapelve a II. típusú döntési fán

Tegyük fel újra, hogy D_1, D_2, ..., D_n az a döntéssorozat, amelyik a feladatot optimálisan bontja részfeladataira. Az egyszerűség kedvéért feltételezzük, hogy minden döntéssel az aktuális részfeladat további *két*

részfeladatra bomlik (míg triviális részfeladatokhoz nem jutunk). Hogyan nyilvánul meg az optimalitás alapelve egy ilyen feladat esetén? Ha feltételezzük, hogy a D_1 döntéssel járó választás nyomán a feladat oly módon esik szét *két* részfeladatra, hogy ezek további felbontását a $D_2, \ldots, D_k$, illetve $D_{k+1}, \ldots, D_n$ részdöntéssorozatok biztosítják, akkor ezeknek egyenként ugyancsak optimálisaknak kell lenniük (abban az értelemben, hogy úgy bontják tovább az illető részfeladatokat, hogy az az optimális megoldásukhoz vezessen).

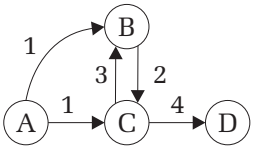
Úgy is fogalmazhatnánk, hogy amennyiben $D_1, D_2, \ldots, D_n$ a feladat optimális megoldásához vezető döntéssorozat, akkor valamely k-ra ($k = 2, n-1$) a $D_2, \ldots, D_k$ és $D_{k+1}, \ldots, D_n$ részdöntés-sorozatpár is optimális. Folytatva a gondolatmenetet, ez azt feltételezi, hogy valamely k_1-re ($k_1 = 3, \ldots, k-1$), illetve k_2-re ($k_2 = k+2, \ldots, n-1$) a $D_3, \ldots, D_{k1}$ és $D_{k1+1}, \ldots, D_k$, valamint a $D_{k+2}, \ldots, D_{k2}$ és $D_{k2+1}, \ldots, D_n$ részdöntés-sorozatpárok ugyancsak optimálisak. És így tovább...

Nem nehéz átlátni, hogy amennyiben ez a helyzet áll fenn, az általános részfeladat optimális megoldását egy D_i, D_{i+1}, ..., D_j alakú döntéssorozat-szakasz jelenti. Mit jelent ilyen esetben lentről felfele építkezni? Most is a megoldásukat jelentő döntéssorozat-szakaszok hossza szerinti növekvő sorrendben oldjuk meg a részfeladatokat, csakhogy ez alkalommal n egy hosszú, $n-1$ két hosszú ... részdöntéssorozatunk van.

Amíg abban az esetben, amikor minden döntéssel a feladat egyetlen hasonló egyszerűbb részfeladattá redukálódott, az optimalitás alapelvének érvényessége nyilvánvaló volt, ez nincs így ebben az esetben. Megtörténhet ugyanis, hogy a részfeladatoknak – amelyekre a feladatot felbontjuk – az optimalizálása konfliktushoz vezet. Legyen egy példa erre.

7.4. Leghosszabb út: Adott egy város utcahálozata a terek közti közvetlen *egyirányú* utcák hossza által. Határozzuk meg két tér között azt a *leghosszabb* utat, amely *nem érinti kétszer ugyanazt a teret.*

Példa:



7.22. ábra.

Az A és D terek között az $ABCD$ a leghosszabb út ($1+2+4=7$), de az A és B terek között van hosszabb út is, mint az optimális megoldásbeli AB közvetlent utca (1), nevezetesen az ACB ($1+3=4$) út.

7.5.2. Dinamikus programozás a II. típusú döntési fán

Egy megoldott feladaton keresztül tegyük a dinamikus programozás e változatát is megfoghatóbbá.

7.5: Adott egy karakterlánc. Bontsuk minimális számú tükörszóra.
Példa: Legyen a karakterlánc: ababb. Látható, hogy többféleképpen is tükörszavakra bontható:

```
(a)(b)(a)(b)(b), (a)(b)(a)(bb), (aba)(bb),
(a)(bab)(b),(aba)(b)(b)
```

Az optimális megoldást a harmadik változat jelenti.

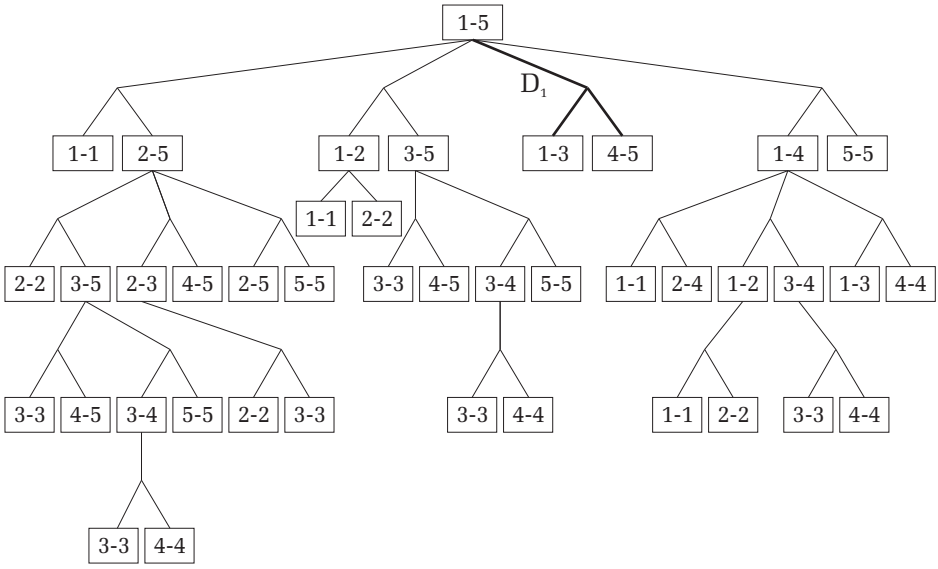
Az alapötlet az, hogy amennyiben a karakterlánc nem tükörszó önmagában, vágjuk kettőbe, visszavezetve ezáltal a feladatot két rövidebb karakterláncnak a tükörszavakra bontására. Ha ez megvan, akkor a két részoptimum összegeként kapjuk az eredeti feladat minimális tükörszószámát. Ez úgy hangzik, mint egy „oszd meg és uralkodj” algoritmus. Mégsem pusztán az, mert a vágások általában többféleképpen is elvégezhetők, és nincs lehetőség arra, hogy meghatározzuk (mohó döntéssel), melyik lenne az, amelyik az optimális megoldáshoz vezetne.

Az alábbiakban bemutatjuk a feladathoz rendelhető döntési fát a példafeladatra vonatkoztatva. A fa csomópontjai a megfelelő részkarakterláncnak a karakterláncon belüli kezdő és befejező indexét tartalmazzák. Például az (1–5) csomópont a teljes karakterláncot (ababb), a (3–5) pedig az abb részkarakterláncot képviseli.

A következő oldalon látható döntési fa eltér a megszokottól, hiszen azt mutatja be, hogy milyen módokon *bontható* részfeladatokra az eredeti feladat. Például, az eredeti feladat négyféleképpen vágható kettőbe: a|babb, ab|abb, aba|bb, abab|b. Az optimális vágás a harmadik (amely ez estben egyben megoldást is jelent). Ez a vágás az (1–3) és (4–5) részkarakterláncokhoz vezet, hiszen ezek már tükörszavak. Figyeljük meg, hogy a fa összes levele – és csak ezek – tükörszót ábrázol.

A fáról az is leolvasható, hogy a részfeladatok általános alakja: az (i-j) karakterlánc-szakasz optimális tükörszóra bontása. Mivel az általános alakban két független paraméter jelenik meg, és $i \leqslant j$, egy $c[1..n][1..n]$

(n – a karakterlánc karaktereinek száma) kétdimenziós tömb főátlóján és a főátlója felett elhelyezkedő elemeit fogjuk használni a részfeladatok optimumértékeinek a tárolására. A $c[i][j]$ tömbelem az (i-j) szakasz optimális felbontásának megfelelő minimális tükörszószámot fogja tárolni.

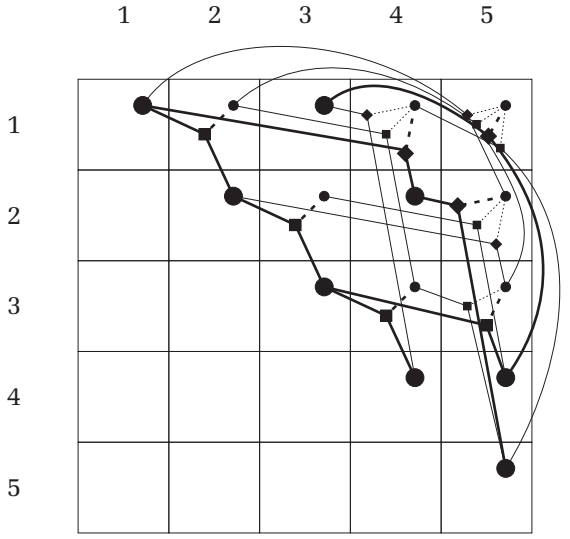


7.23. ábra.

A fentiekkel összhangban megfigyelhető, hogy amennyiben egymásra csúsztatjuk a döntési fa identikus csomópontjait, az így kapott „összevont fa" csomópontjai átrendezhetők úgy, hogy egy $n \times n$ méretű kétdimenziós tömb főátló feletti háromszögének megfelelő alakzatot alkossanak (a döntési fa egymástól különböző csomópontjainak száma: $n(n+1)/2$). Ezt láthatjuk a következő oldalon. A fa gyökere az 1. sor n-edik oszlopába kerül, ugyanúgy, ahogy az eredeti feladat optimumértékét a $c[1][n]$ tömbelem fogja tárolni. Mivel az egyelemű karakterláncok nyilván tükörszavak, ezért a főátló elemei mindenképpen triviális részfeladatok optimumértékeit tárolják, de természetesen lehetnek más tömbelemek is, amelyeknek megfelelő részfeladatok ugyancsak triviálisak, ha az illető részkarakterlánc önmagában tükörszó. Minden tükörszó-részkarakterláncot, mind a döntési fában, mind az összevont fában, levélcsomópontok képviselnek.

7.5.2.1. Az optimalitás alapelvének ellenőrzése

Mivel egy adott vágás nyomán két független részfeladat keletkezik, így az optimális módon való megoldásuk nem kerülhet konfliktusba, és a feladatra érvényes az optimalitás alapelve. Nevezetesen: ha az (i-j) szakasz az optimális feldarabolásakor első lépésben az (i-k) és ($k+1$, ..., j) szakaszpárra esik szét, akkor ezeknek az optimális feldarabolásbeli továbbdarabolása biztosan optimális rájuk nézve is. Ugyanis, ha valamelyiknek lenne egy jobb feldarabolása, mint ahogy az (i-j) szakasz optimális feldarabolásán belül feldarabolódik, akkor ezt választva az (i-j) szakaszra is egy jobb feldaraboláshoz jutunk. Ez viszont ellentmond annak, hogy az (i-j) szakasz optimális feldarabolásából indultunk ki. Tehát az eredeti karakterlánc optimális feldarabolása meghatározható a részkarakterláncok optimális feldarabolásaiból.



7.24. ábra.

7.5.2.2. Az optimalitás alapelve az „összevont döntési fában”

A levelektől a gyökér felé haladva mindenik apacsomóponton azt a fiúcsomópontpárt hagyjuk meg, amelynek megfelelő részfeladatpár optimumértékének összege minimális (a többit levágjuk a fáról). Az összevont fán megvastagítottuk az optimális választásokat.

7.5.2.3. Az optimalitás alapelve az optimumértékeket tároló tömbben

A c tömb főátlón levő, illetve főátló feletti elemeit az alábbi rekurzív képlet szerint töltjük fel:

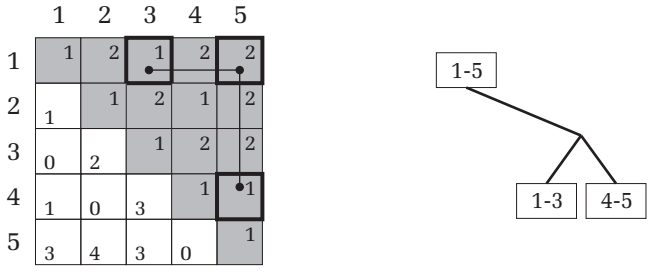
$$\begin{aligned} c[i][j] &= 1, \quad \text{ha az } (i\text{-}j) \text{ karakterláncszakasz tükörszó} \\ &= \min\{c[i][k] + c[k+1][j]\},\ i \leqslant k < j, \quad \text{ellenkező esetben} \end{aligned}$$

Ahogy a képlet is sugallja, az elemeket olyan sorrendben kell kitölteni, hogy amikor sorra kerül a $c[i][j]$ feltöltése, a $c[i][k]$ és $c[k+1][j]$ elempárok ($i \leqslant k < j$) legyenek már feltöltve. Egy ilyen sorrend az, amely kiindul a főátló menti elemektől, és átlóról átlóra halad a $c[1][n]$ elemig.

Ha meg szeretnénk adni magát az optimális tükörszóra bontást is (nemcsak a minimális tükörszószámot), akkor a c tömb most már kellő információt tartalmaz ahhoz, hogy a gyökérből kiindulva mohó vágásokkal, egyszerű „oszd meg és uralkodj” feladatként tükörszavakra daraboljuk karakterláncunkat. Hogy ne kelljen a `min` függvény segítségével újra kiválasztani az optimális vágásokat, a c tömb feltöltésekor célszerű, ha a $c[j][i]$ tömbelembe (a főátló alatti háromszöget különben sem használtuk ki) eltároljuk az (i-j) szakasz optimális vágásához tartozó optimális k értéket. Ha ezt megtesszük, akkor a megoldás rekonstruálásakor az (i-j) karakterláncszakasz optimálisan az $(i, c[j][i])$ és $(c[j][i]+1, j)$ karakterláncpárra bomlik. Ha az $i < j$ esetben az (i-j) szakasz önmagában tükörszó, akkor ezt a $c[j][i]$ elemben jelezze a 0 érték.

Az optimális megoldást a döntési fában[4] ez esetben nem valamelyik gyökér–levél út, hanem az optimális feldarabolásnak megfelelő bináris (mert minden vágással két részfeladatot kapunk) részfa ábrázolja. Ezt látjuk lentebb a döntési fából kiemelve (a példaesetre az optimális megoldás bináris fája csak gyökérből és első szinti fiakból áll), és ezt rajzoltuk be a c tömbbe is, amely – amint már megtárgyaltuk – a főátlón és a főátló felett a részfeladatok optimumértékeit, a főátló alatt pedig az optimális k értékeket tartalmazza.

4 A szerző részletesen tárgyalja az „Optimális megosztás – optimális uralom” típusú dinamikus programozásos feladatok gráfelméleti hátterét a „Dinamikus programozás és d-gráfok” című cikkében.



7.25. ábra.

7.6. Összefoglalás

7.6.1. A dinamikus programozás stratégiájának főbb jellegzetességei

1. A módszer a feladatot lentről felfele oldja meg. Ez azt jelenti, hogy kiindulva a triviális részfeladatok *optimális* megoldásaiból, lépésről lépésre felépíti az egyre bonyolultabb részfeladatok *optimális* megoldásait.
2. Miután egy részfeladatot megoldott, az eredményét (a célfüggvény optimumértékét az illető részfeladatra vonatozóan) eltárolja (rendszerint egy tömbben), hogy rendelkezésre álljon a későbbiekben mint építőelem – amennyiben szükség lesz rá – a nagyobb méretű részfeladatok megoldásában. Ezzel az eljárással a dinamikus programozás elkerüli az egymásra tevődő részfeladatok többszöri megoldását.
3. Bár egy részfeladatnak több megoldása is lehet, csak az optimálisat tárolja el. Miért? *Az optimalitás alapelve garantálja, hogy az optimális részmegoldásokból felépíthető az eredeti feladat optimális megoldása.*

Megjegyzés: A fejezet elején azt mondtuk, hogy a dinamikus programozást elsősorban optimalizálási feladatok megoldásához használjuk. Tekintettel azonban a fenti első két tulajdonságra, célszerű megpróbálni alkalmazni minden olyan esetben, amikor sok az egymásra tevődő részfeladat. Példaként tekintsük az n-edik Fibonacci-szám meghatározásának feladatát. A klasszikus algoritmus erre az, hogy az első két Fibonacci-szám összegeként meghatározzuk a harmadikat, majd a második és harmadik összegeként a negyediket, és így tovább, míg végül

az $(n-2)$-edik és $(n-1)$-edik összegeként ki nem számítjuk az n-ediket. Mindez feltételezi, hogy minden lépésben eltároljuk legalább az utolsó két kiszámított értéket, hiszen ezekből épül majd fel a következő. Bár nem optimalizálási feladatról van szó, a bemutatott elemzés felszínre hozta a dinamikus programozás néhány alapvonását: a részfeladatokat az egyszerűtől a bonyolult fele haladva oldottuk meg, és mindig eltároltuk azoknak a részfeladatoknak a megoldásait, amelyekre később szükségünk volt az építkezésben. Ez a megoldás összehasonlíthatatlanul hatékonyabb, mint a divide et impera megközelítés (az n-edik Fibonacci-szám kiszámítását rekurzívan visszavezetjük az $(n-2)$-edik és az $(n-1)$-edik elem egymástól független meghatározására), ami azonos részfeladatok többszöri megoldásához vezetne.

7.6.2. Milyen esetben folyamodjunk a dinamikus programozáshoz?

Nincs egyértelmű válasz erre a kérdésre, mégis van néhány nyomra vezető jel.

1. A feladatra érvényes kell hogy legyen az optimalitás alapelve, miszerint: az optimális megoldás optimális részmegoldásokból épül fel.
   Ez a kritérium önmagában még nem utal egyértelműen dinamikus programozásra. Megtörténhet például, hogy elegendő a greedy megközelítés, ami sokkal egyszerűbb és hatékonyabb algoritmust eredményez.
2. Ne legyen érvényes a mohó kiválasztás alapelve. Ez azt jelenti, hogy a globálisan optimális döntéssorozatban nem föltétlenül lokális optimum minden egyes döntés.
   Ez a feltétel kizárja a greedy megoldást, de még mindig versenyben marad a divide et impera stratégia, amely ugyancsak egyszerűbb, mint a dinamikus programozás.
3. A részfeladatok között, amelyekre a feladat lebomlik, sok az azonos.
   Ilyen esetben a divide et impera algoritmus nem hatékony. Mivel fentről lefele haladva egymástól függetlenül oldaná meg a részfeladatokat, rengeteget dolgozna fölöslegesen.

Ha a fenti három feltétel egyidőben teljesül egy adott feladatra, akkor ez arra mutat, hogy a feladat nagy valószínűséggel dinamikus programozással oldható meg hatékonyan.

7.6.3. Hogyan közelítsünk meg egy dinamikus programozás feladatot?

Az alábbi lépéssorozatot ajánljuk.

1. Meghatározzuk a részfeladatok általános alakját.

Hogyan bontható le a feladat kisebb méretű hasonló részfeladatokra oly módon, hogy érvényes legyen rá az optimalitás alapelve? Az optimalitás alapelvének ellenőrzése azért is szükséges, mert ezáltal alapozzuk meg matematikailag, hogy a dinamikus programozás tényleg az optimális megoldást nyújtja. Ennél a lépésnél megállapítjuk az általános részfeladat paramétereit, és azt, hogy mely paraméterértékekre kapjuk az eredeti feladatot, illetve a triviális részfeladatokat. Ugyancsak itt tisztázzuk, hogy a részfeladatok növekedése a paraméterek növekedésével vagy csökkenésével jár kéz a kézben.

2. Eldöntjük, hol fogjuk eltárolni az egyes részfeladatok optimális megoldásait jellemző optimumértékeket.

Általában *annyi*-dimenziós tömböt használunk, ahány független paramétere van az általános részfeladatnak. Mely tömbrekeszekben kerül eltárolásra a fő feladat optimumértéke, illetve a triviális részfeladatok optimumértékei? Érdekes megfigyelni, hogy a felhasznált tömbrekeszek száma azonos a különböző részfeladatok számával.

Megjegyzés: Ha a feladat nem kéri magát az optimális megoldást is, csak az ehhez tartozó optimumértéket, akkor szükségtelen lehet a teljes tömb eltárolása. Például a „háromszög” feladat esetében használhattunk volna egydimenziós c tömböt is, amelyet állandóan felülírunk a kétdimenziós tömb aktuális sorával. Ez azért lehetséges, mert a következő sor kizárólag az előtte lévőből áll elő.

3. Megkeressük azt a rekurzív képletet, amely matematikailag leírja, miként épül fel az általános részfeladat optimális megoldását jellemző optimumérték a részfeladatok optimumértékéből. Segíthet ennél a lépésnél, ha behatároltuk, hogy melyik formájában igaz a feladatra az optimalitás alapelve.

4. A rekurzív képlet alapján – az egyszerűtől haladva a bonyolult fele – feltöltjük a tömböt a részfeladatok optimumértékeivel. Tehát a képletet nem rekurzívan alkalmazzuk, hanem lentről felfele építkezünk a segítségével. A tömböt oly módon kell bejárnunk, hogy egy adott részfeladat optimumértékének kiszámításakor rendelkezésre álljanak már azon optimumértékek, amelyekből – a képlet alapján – ez meghatározható.

5. A 4. lépésben csak a megoldás optimumértéke kerül kiszámításra. Ezek után az optimális döntéssorozat meghatározása megvalósítható lineáris időben, és aszerint történik, hogy melyik formájában volt érvényes az optimalitás alapelve a feladatra.

A levelek–gyökér irányú változat esetén csak arra van szükség, hogy az eredeti feladat optimumértékét tartalmazó tömbrekesztől (ez a döntési fa gyökerét képviseli) indulva, menjünk vissza az optimális úton az optimális levelet ábrázoló tömbrekeszig (ez valamelyik triviális részfeladat optimális megoldását tárolja), rekonstruálva ezáltal az optimális döntéssorozatot.

Ha a gyökér–levelek irányú változatot alkalmaztuk, akkor az optimális levelet képviselő tömbrekesztől indulva, egy rekurzív eljárás által először felmegyünk az optimális úton, és a rekurzió visszaútján írjuk ki az optimális döntéssorozatot.

Amikor az „optimális megosztás – optimális uralom" változatot alkalmazzuk, az optimális felbontás a döntési fa optimális részfájának (általában bináris) bejárását feltételezi.

Ahhoz, hogy ezen optimális megoldás-meghatározás tényleg lineáris időben történjen, egyes esetekben szükség lehet arra, hogy az optimális részmegoldások optimumértékeinek építésekor (a 4. lépésben) eltároljuk az optimális választásokat.

7.7. Megoldott feladatok

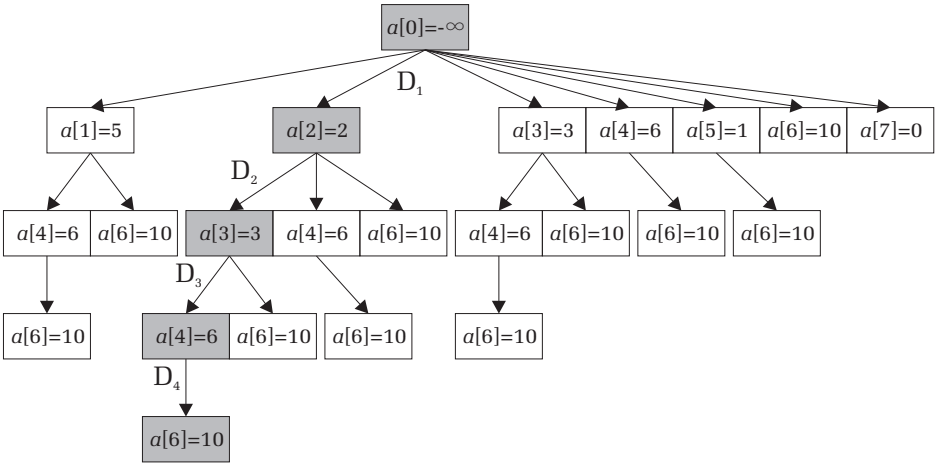
Az első négy feladatot részletesebben tárgyaljuk, megelevenítve a feladat mögött meghúzódó döntési fát is. A többi feladat megoldásakor csak végigjárjuk a javasolt öt lépést.

7.6. Leghosszabb növekvő részsorozat: Adott egy n elemű számsorozat az $a[1..n]$ tömbben. Határozzuk meg a leghosszabb – nem föltétlenül összefüggő – növekvő részsorozatát.

Megoldás (levelek–gyökér irányú dinamikus programozás):

I. Példa gyanánt legyen a következő 7 elemű számsorozat: 5, **2**, **3**, **6**, 1, **10**, 0.

Ha kiegészítjük a sorozatot egy $a[0] = -\infty$ elemmel, akkor a feladat megoldása az $a[0]$ elemmel kezdődő leghosszabb szigorúan növekvő részsorozat lesz. Mellékelve (7.26. ábra) láthatjuk a feladathoz rendelhető döntési fát (ha nem vezettük volna be az $a[0] = -\infty$ elemet, akkor



7.26. ábra.

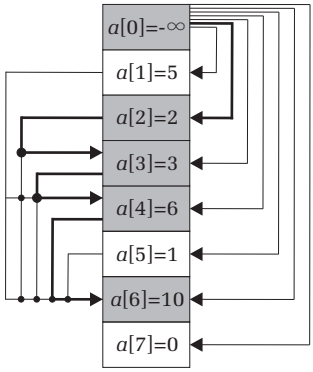
a fának egy virtuális gyökere lett volna). A megoldást a leghosszabb gyökér–levél út képviseli. A részfeladatokat, amelyekre a feladat lebontható, a részfák ábrázolják (a fent bevezetett terminológiával összhangban ezek szúfix részfeladatok). Az általános részfeladat: meghatározni az i-edik elemmel kezdődő leghosszabb szigorúan növekvő részsorozatot. Jól látható az is, hogy a lebontás alkalmával számtalan azonos részfeladat jelenik meg, ami kizárja a divide et impera technika alkalmazásának lehetőségét.

Tehát a dinamikus programozás levelek–gyökér változatát fogjuk alkalmazni. Természetesen választhatnánk a gyökér–levelek alakot is, és ez esetben az általános (prefix) részfeladat az i-edik elemmel *végződő* leghosszabb részsorozat meghatározásának problémája lenne.

Ha egymásra csúsztatjuk az identikus csomópontokat, a 7.27. ábrán látható összevont döntési „fához” jutunk.

II. *Az optimalitás alapelvének ellenőrzése:* Legyen $(a[i1 = 0], a[i_2], \ldots, a[i_k])$ a leghosszabb szigorúan növekvő részsorozat. Megfigyelhető, hogy ez azonos az $a[i_1]$ elemmel kezdődő leghosszabb szigorúan növekvő részsorozattal.

Az optimalitás alapelve a következő alakban érvényes (összhangban a levelek–gyökér változattal): ha az $a[i_1]$ elemmel kezdődő leghosszabb szigorúan növekvő részsorozat az $(a[i_1], a[i_2], \ldots, a[i_k])$, akkor az $a[i_2]$ elemmel kezdődő leghosszabbnak föltétlenül az $(a[i_2], a[i_3], \ldots, a[i_k])$



7.27. ábra.

részsorozatnak, az $a[i_3]$-mal kezdődő leghosszabbnak pedig föltétlenül az $(a[i_3], a[i_4], \ldots, a[i_k])$ részsorozatnak kell lennie stb. Általánosan: az $(a[i_1], a[i_2], \ldots, a[i_k])$ megoldás optimalitásából következik az $(a[i_p], a[i_{p+1}], \ldots, a[i_k])$ $(p = 2, \ldots, k)$ részmegoldások optimalitása.

Bizonyítás: Indirekt bizonyítást használunk. Kiindulunk abból, hogy a feladat optimális megoldása, az $a[i_1]$ elemmel kezdődő $(a[i_1], a[i_2], \ldots, a[i_k])$, szigorúan növekvő részsorozat. Továbbá feltételezzük, hogy létezik legalább egy p $(p = 2, k)$, amelyre a vele kezdődő leghosszabb szigorúan növekvő részsorozat nem az $(a[i_p], a[i_{p+1}], \ldots, a[i_k])$. Ez azt jelenti, hogy létezik egy $a[i_p]$-vel kezdődő, hosszabb szigorúan növekvő részsorozat. Legyen ez az $(a[j_1 = i_p], a[j_2], \ldots, a[j_q])$, ahol $q > k - p + 1$. Viszont, ha így áll a dolog, akkor létezik az $a[i_1]$-gyel kezdődő szigorúan növekvő részsorozatok között a k hosszú $(a[i_1], a[i_2], \ldots, a[i_k])$ részsorozatnál hosszabb részsorozat is, az $(a[i_1], a[i_2], \ldots, a[i_{p-1}], a[i_p = j_1], a[j_2], \ldots, a[j_q])$, hiszen ennek hossza $q + p - 1 > k$. Ez azonban ellentmond a feltételnek, miszerint optimális megoldásból indultunk ki.

Tehát az optimális megoldás optimális részmegoldásokból épül fel.

III. Mivel az általános részfeladatnak egyetlen független paramétere van (i), a részfeladatok optimális megoldásainak optimumértékét egy $c[0..n]$ egydimenziós tömbben fogjuk tárolni (az összevont döntési fa is erre enged következtetni). Pontosabban, a $c[i]$ elembe az $a[i]$ elemmel kezdődő leghosszabb szigorúan növekvő részsorozat hossza fog kerülni. Végül, az optimális megoldás optimumértékét a $c[0]$ tömbelem fogja tartalmazni.

IV. Tekintettel arra, hogy levelek–gyökér irányú dinamikus programozást alkalmazunk, a $c[i]$ érték kiszámításakor már rendelkezésre

állnak a $c[i+1]$, $c[i+2]$, ..., $c[n]$ értékek. Hogyan építhető fel ezekből a $c[i]$ érték? Ezt jól szemlélteti a feladat döntési fája. A szúfix részfeladatok szempontjából a lentről felfele (az egyszerűtől a bonyolult fele) irány azt jelenti, hogy egy apacsomópont képviselte feladat optimumértékét a fiúcsomópontjai képviselte feladatok optimumértékeiből kell kiszámítanunk. Az alapgondolat az, hogy miközben jövünk hátulról előre a számsorozaton, minden $a[i]$ elem az őt követő – nála nagyobb – elemekkel kezdődő leghosszabb szigorúan növekvő részsorozatok közül a legesleghosszabb elé fog belépni. Más szóval, a $c[i]$ optimumérték eggyel több lesz, mint a hozzá tartozó csomópont fiúcsomópontjainak megfelelő c-tömbbeli optimumértékek közül a legnagyobb. Ezt írja le matematikailag az alábbi rekurzív képlet:

$$c[n] = 1$$
$$c[i] = 1 + \max\{c[j], \text{ ahol } a[j] > a[i]\},$$
$$\text{minden } 0 \leqslant i < n, j = i+1, 2, \ldots, n$$

V. A c tömb elemeit természetesen hátulról előre sorrendben fogjuk feltölteni, a fenti képlet alapján.

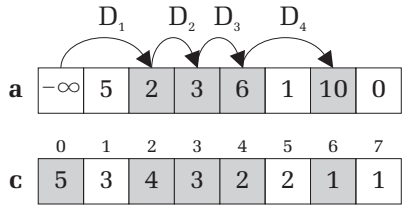
a	$-\infty$	5	2	3	6	1	10	0

	0	1	2	3	4	5	6	7
c	5	3	4	3	2	2	1	1

7.28. ábra.

```
a[0]=-∞
c[n]=1
minden i=n-1,0,-1 végezd
   c[i]=0
   minden j=i+1,n végezd
      ha a[j] > a[i] és c[j] > c[i] akkor
         c[i]=c[j]
      vége ha
   vége minden
   c[i]= c[i]+1
vége minden
ki: "A leghosszabb szigorúan növekvő részsorozat
                                          hossza: ", c[0]
```

VI. Az a és a c tömb alapján most már könnyűszerrel meghatározható maga a leghosszabb szigorúan növekvő részsorozat is. Az alábbi algoritmusrészlet alapjában véve végigmegy a döntési fa optimális gyökér–levél útján, és kiírja az ennek megfelelő optimális megoldást. Az u változó az utolsó kiírt elem indexét tárolja (az $a[0]$ elemet nem íratjuk ki).



7.29. ábra.

```
u=0
minden i=1,n végezd
   ha a[i]>a[u] és c[i]==c[u]-1 akkor
      kiír: a[i]
      u=i
   vége ha
vége minden
```

7.7. Leghosszabb közös részsorozat: Adott két számsorozat az $a[1..n]$ és a $b[1..m]$ tömbökben. Határozzuk meg a leghosszabb közös részsorozatukat.

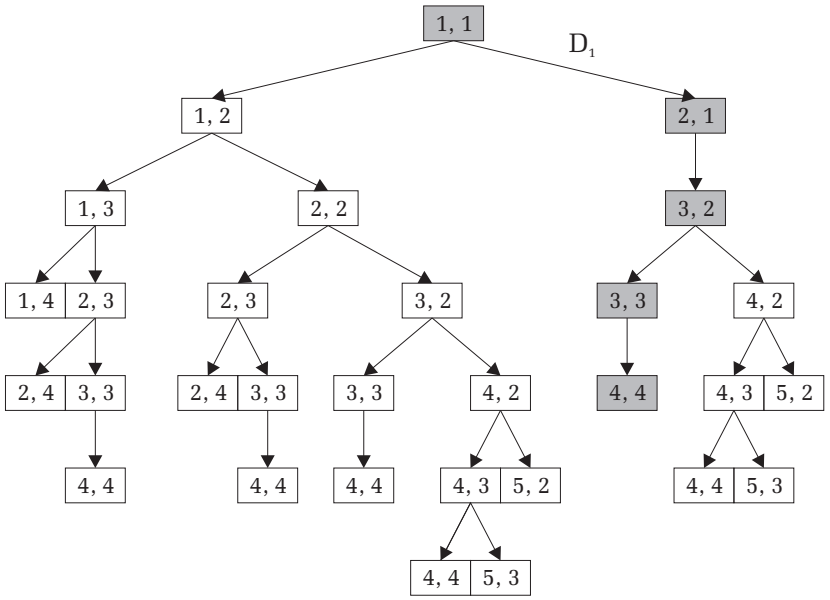
Megoldás (levelek–gyökér irányú dinamikus programozás):

I. Legyen a következő példa: $a = (2, \mathbf{9}, \mathbf{5}, 7)$; $b = (\mathbf{9}, 3, \mathbf{5})$.

Hogyan bontható le a feladat hasonló, de egyszerűbb részfeladatokra? Ha $a[1] = b[1]$, akkor ez az elem természetesen részét fogja képezni a leghosszabb közös részsorozatnak és a feladat leredukálódik az $a[2..n]$ és $b[2..m]$ tömbszakaszok leghosszabb közös részsorozatainak meghatározására (mindkét tömbben lépünk). Ha $a[1]$ különbözik $b[1]$-től, akkor vagy az a tömbben lépünk, vagy a b-ben. Tehát ebben az esetben két lehetőség közül kell választanunk: az $a[1..n]$ és $b[2..m]$, vagy az $a[2..n]$ és $b[1..m]$ tömbszakaszok leghosszabb közös részsorozatainak meghatározására redukáljuk a feladatot. Folytatva a gondolatmenetet, nyilvánvaló, hogy a részfeladatok általános alakja: meghatározni az $a[i..n]$ és $b[j..m]$ tömbszakaszok leghosszabb közös részsorozatát. Mivel két független paraméterünk van, az alábbiakban néha úgy fogunk utalni az általános

(szúfix) részfeladatra mint „(i, j) feladat”-ra. Az eredeti feladatot $i = 1$ és $j = 1$ értékekre ($(1, 1)$ feladat), a triviális részfeladatokat pedig $i = n + 1$ vagy $j = m + 1$ (legalább egyik tömbszakasz üres) értékekre kapjuk. A triviális részfeladatok optimumértékei nyilván nullák, hiszen ha valamelyik résztömb üres, akkor a közös részsorozat nem létezvén, nulla hosszúnak tekinthető.

Következzen a feladat döntési fája a megadott példára. A lebontásból adódó részfeladatokat ábrázoló részfák gyökereiben az illető részfeladat paraméterei láthatók. Az ábrán jól megfigyelhető az is, hogy a lebontás különböző ágain azonos részfeladatok jelennek meg. Az optimális megoldást az a gyökér–levél út képviseli, amelynek mentén a legtöbbször léptünk egyszerre mindkét tömbben (minden ilyen alkalom egy újabb elemet jelentett a közös részsorozatban).

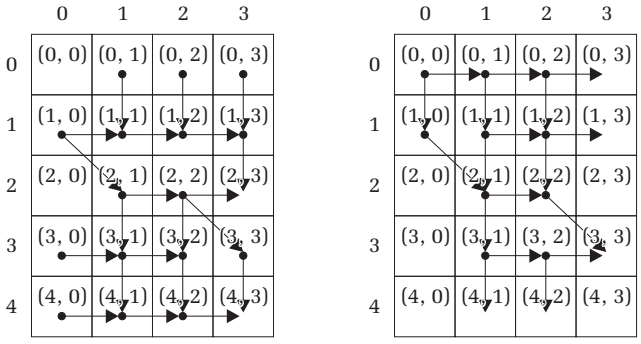


7.30. ábra.

Az olvasó biztosan észrevette, hogy ezt a feladatot is úgy fogtuk fel, mint levelek–gyökér irányú dinamikus programozással megoldható feladatot. Megint csak megtehettük volna, hogy „(i, j) feladatnak” az $a[1..i]$ és $b[1..j]$ tömbszakaszok leghosszabb közös részsorozata meghatározása (prefix) részfeladatát tekintjük. Ez esetben az eredeti feladatot az $i = n$ és

$j = m$, a triviális részfeladatokat pedig az $i = 0$ vagy $j = 0$ paraméterértékekre kapjuk. Mindez természetesen gyökér–levelek irányú dinamikus programozáshoz vezetett volna.

Az alábbiakban bemutatjuk az összevont döntési fát a dinamikus programozás mindkét változatához (csak a levelek–gyökér változatra folytatjuk tovább a feladat megoldását; jobb oldali ábra).



7.31. ábra.

II. *Az optimalitás alapelvének ellenőrzése:* Legyen $(a[i_1] = b[j_1], a[i_2] = b[j_2], \ldots, a[i_k] = b[j_k])$ a két számsorozat leghosszabb közös részsorozata. Megfigyelhető, hogy ez azonos az $a[i_1..n]$ és $b[j_1..m]$ tömbszakaszok leghosszabb közös részsorozatával.

Az optimalitás alapelve a levelek–gyökér változatnak megfelelő alakban érvényes: ha az $a[i_1..n]$ és $b[j_1..m]$ tömbszakaszok leghosszabb közös részsorozata az $(a[i_1] = b[j_1], a[i_2] = b[j_2], \ldots, a[i_k] = b[j_k])$, akkor az $a[i_2..n]$ és $b[j_2..m]$ tömbszakaszokban föltétlenül az $(a[i_2] = b[j_2], a[i_3] = b[j_3], \ldots, a[i_k] = b[j_k])$ sorozatnak, az $a[i_3..n]$ és $b[j_3..m]$ tömbszakaszokban pedig föltétlenül az $(a[i_3] = b[j_3], a[i_4] = b[j_4], \ldots, a[i_k] = b[j_k])$ sorozatnak kell a leghosszabb közös részsorozatnak lennie, és így tovább. Általánosan: az $(a[i_1] = b[j_1], a[i_2] = b[j_2], \ldots, a[i_k] = b[j_k])$ megoldás optimalitásából következik az $(a[i_p] = b[j_p], a[i_{p+1}] = b[j_{p+1}], \ldots, a[i_k] = [j_k])$ $(p = 2, \ldots, k)$ részmegoldások optimalitása.

Bizonyítás: Tekintettel arra, hogy a bizonyítás ugyanazt a gondolatmenetet követi, mint az előző feladat esetében, az olvasóra bízzuk.

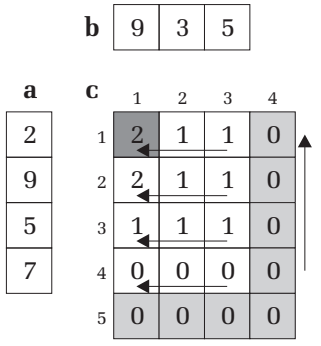
III. Mivel az általános részfeladatnak két független paramétere van, az optimális részmegoldások optimumértékét egy $c[1\,..\,n{+}1][1\,..\,m{+}1]$ kétdimenziós tömbben lehet tárolni. A $c[i][j]$ tömbelem az $a[i..n]$ és $b[j..m]$ tömbszakaszok leghosszabb közös részsorozatának a hosszát fogja tartalmazni. Az eredeti feladat megoldása a $c[1][1]$ tömbelembe kerül.

IV. Tekintettel a levelek–gyökér irányú dinamikus programozásra, a $c[i][j]$ tömbelem értékét, vagyis az (i, j) feladat optimumértékét vagy az $(i, j+1)$ és $(i+1, j)$ feladatok optimumértékeiből ($c[i][j+1]$ és $c[i+1][j]$), vagy az $(i+1, j+1)$ feladat optimumértékéből ($c[i+1][j+1]$) kell kiszámítani. Ponosabban: ha $a[i] = b[j]$, akkor ez az elem természetesen részét fogja képezni az $a[i..n]$ és $b[j..m]$ tömbszakaszok leghosszabb közös részsorozatának (mindkét tömbben lépünk). Ez esetben az (i, j) feladat optimumértéke eggyel több lesz, mint az $(i+1, j+1)$ feladaté. Ha viszont $a[i] \neq b[j]$, akkor az (i, j) feladat optimumértéke az $(i, j+1)$ és $(i+1, j)$ feladatok optimumértékei közül a nagyobbikkal lesz egyenlő (csak az egyik tömbben lépünk, anélkül hogy csatolnánk elemet a közös részsorozathoz). A rekurzív képlet, amely mindezt matematikailag leírja, a következő:

$$
\begin{aligned}
c[i][m+1] &= 0, && \text{minden } i = 1, 2, \ldots, n+1 \\
c[n+1][j] &= 0, && \text{minden } j = 1, 2, \ldots, m+1 \\
c[j][j] &= 1 + c[i+1][j+1], && \text{ha } a[i] = b[j] \\
&= \max(c[i][j+1], c[i+1][j]), && \text{ha } a[i] \neq b[j], \\
& && \text{minden } 1 \leqslant i \leqslant n, 1 \leqslant j \leqslant m \text{ estén.}
\end{aligned}
$$

Határozza meg az olvasó, milyen volna a rekurzív képlet gyökér–levél irányú dinamikus programozás esetén!

V. Azért, hogy a $c[i][j]$ tömbelem kitöltésekor rendelkezésre álljanak már a $c[i+1][j]$, $c[i][j+1]$ és $c[i+1][j+1]$ tömbelemek, a tömb sorait lentről felfele sorrendben töltjük fel, minden sort jobbról balra. Ezt szemlélteti a megadott példára a mellékelt ábra.

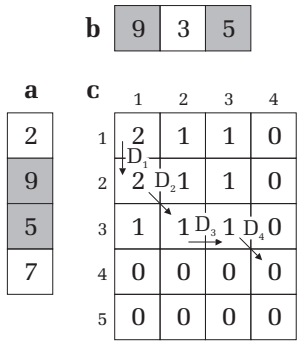


7.32. ábra.

```
minden i = 1,n+1 végezd
   c[i][m+1]=0
vége minden
minden j = 1,m+1 végezd
   c[n+1][j]=0
vége minden

minden i = n,1,-1 végezd
   minden j = m,1,-1 végezd
      ha a[i]==b[j] akkor
         c[i][j] = c[i+1][j+1] + 1
      különben
         ha c[i][j+1] > c[i+1][j] akkor
            c[i][j]=c[i][j+1]
         különben
            c[i][j]=c[i+1][j]
         vége ha
      vége ha
   vége minden
vége minden
ki: "A leghosszabb közös részsorozat hossza: ", c[1][1]
```

Hogyan kellene feltölteni a c tömböt, és hogyan módosulna az algoritmus gyökér–levél irányú dinamikus programozás esetén? (Gyakorlat az olvasónak.)

VI. A c tömb nyújtotta plusz információ birtokában végigmegyünk az optimális döntéssorozatnak megfelelő gyökér–levél úton, és kiírjuk a leghosszabb közös részsorozatot.



7.33. ábra.

```
i=1
j=1
amíg i <= n és j <= m végezd
   ha a[i] == b[j] akkor
      ki: a[i]
      i=i+1
      j=j+1
   különben
      ha c[i][j] == c[i][j+1] akkor
         j=j+1
      különben
         i=i+1
      vége ha
   vége ha
vége amíg
```

Hogyan történne a leghosszabb közös részsorozatnak a kiíratása gyökér–levél irányú dinamikus programozás esetén? (Gyakorlat az olvasónak.)

7.8. Mátrixszorzás: Tegyük fel, hogy össze szeretnénk szorozni n mátrixot $(A_1, A_2, \ldots, A_n)$, amelyeknek adottak a méretei: $d[0] \times d[1], d[1] \times d[2], \ldots, d[n-1] \times d[n]$. A mátrixszorzás asszociativitásából adódóan ezt sokféleképpen megtehetjük. Határozzuk meg a mátrixsorozat egy olyan zárójelezését, amelynek megfelelő mátrixszorzássorrend minimális számú elemi szorzást feltételez. (Megjegyezzük, hogy egy $n \times m$ méretű mátrixnak egy $m \times p$ méretűvel való összeszorzása $n \cdot m \cdot p$ darab elemű szorzással jár.)

Megoldás („optimális megosztás – optimális uralom”):

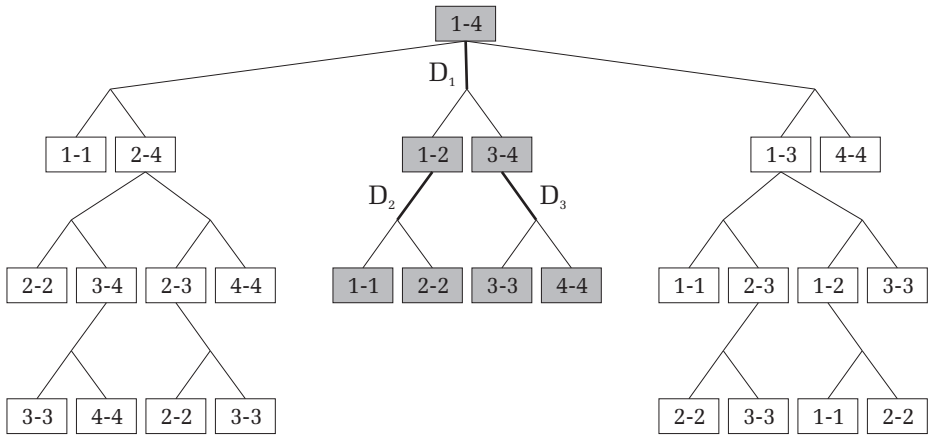
I. Legyen a következő példa:

$$n = 4, A_1(1 \times 10), A_2(10 \times 1), A_3(1 \times 10), A_4(10 \times 1).$$

Az optimális megoldás erre a példára: $((A_1 \cdot A_2) \cdot (A_3 \cdot A_4))$, amely 21 elemi szorzást feltételez.

Egy legrosszabb megoldás 210 szorzást feltételezne: $((A_1 \cdot (A_2 \cdot A_3)) \cdot A_4)$. Tehát egyáltalán nem mindegy, milyen sorrendben szorozzuk össze a szomszédos mátrixokat.

Hogyan bontható le a feladat hasonló, de egyszerűbb feladatokra? Ha kettőbe osztjuk az $A[1..n]$ mátrixsorozatot úgy, hogy az $A[i]$ és $A[i+1]$ mátrixok között végezzük el a vágást, akkor visszavezettük a feladatot



7.34. ábra.

az $A[1..i]$ és $A[i+1\,..\,n]$ szakaszok mátrixainak az optimális összeszorzására, és ezt követően a két eredménymátrix szorzatára. Ezen első vágást $(n-1)$-féleképpen tudjuk elvégezni ($i = 1, 2, \ldots, n-1$). Jelöljük D_1-gyel az első vágás optimális helyének eldöntését. Természetesen folytatjuk a "darabolást" – először az $A[1..i]$, majd pedig az $A[i+1\,..\,n]$ szakaszon – egészen addig, míg egyelemű szakaszokhoz jutunk. Az $A[1..i]$ szakasz feldarabolása további $i-1$ döntést ($D_2, \ldots, D_i$), az $A[i+1\,..\,n]$ szakaszé pedig további $n-i-1$ döntést ($D_{i+1}, \ldots, D_{n-1}$) feltételez. Összesen $n-1$ döntésre (vágásra) lesz szükség, amelyeket a meghozásuk sorrendje szerint sorszámoztunk. Figyeljük meg, hogy az előző feladatoktól eltérően minden döntéssel a feladat további *két* részfeladatra bomlik. Ha minden lépésben meg tudnánk határozni mohó módra az optimális vágáshelyet, akkor a feladat megoldható lenne divide et impera technikával. Mivel ez nem lehetséges, a dinamikus programozás „optimális megosztás – optimális uralom" változatára apellálunk. A dinamikus programozás szükségességére utal az is – ahogy a feladathoz rendelhető fa is mutatja –, hogy a feladat lebontásakor különböző ágakon azonos részfeladatok jelennek meg.

A részfeladatok általános alakja: az $A[i..j]$ tömbszakasz mátrixainak optimális összeszorzása. Néha (ahogy a fán is) „$i-j$ feladatként" fogunk utalni rá. Az eredeti feladatot $i = 1$ és $j = n$ értékekre, a triviálisakat pedig $i = j$ értékekre kapjuk. Mivel $i \leqslant j$, összesen $n(n+1)/2$ különböző részfeladat fog megjelenni a lebontás alkalmával, amiből n lesz triviális.

II. *Az optimalitás alapelvének ellenőrzése:* Tegyük fel, hogy az $A[i..j]$ tömbszakasz mátrixainak optimális összeszorzásához a $D_i, \ldots, D_{j-1}$ döntéssorozat vezet. Feltételezzük továbbá, hogy a D_i döntéssel az „$i-j$ feladat" az $A[i..k]$ és $A[k+1..j]$ szakaszokra bomlik, amelyeket a $D_{i+1}, \ldots, D_k$, illetve a $D_{k+1}, \ldots, D_{j-1}$ részdöntéssorozatok oldanak meg. Ezeknek ugyancsak optimálisan kell megoldaniuk a megfelelő részfeladatokat, hiszen ellenkező esetben, ha lenne jobb megoldásuk is ennél, akkor ezt választva, az $A[i..j]$ szakaszra is jobb megoldást kapnánk, ami viszont ellentmond azon kiindulási feltételnek, hogy a $D_i, \ldots, D_{j-1}$ döntéssorozat nyújtja az optimális megoldást. Függetlenek voltak a részfeladatok, és ezért optimalizálásuk során nem kerültek konfliktusba.

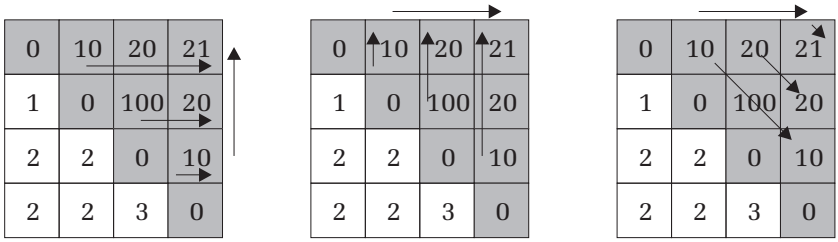
III. Mivel a részfeladatok általános alakjának két független paramétere van, a részfeladatok optimális megoldásait képviselő optimumértékeket egy kétdimenziós tömbben fogjuk eltárolni. Egészen pontosan a $c[i][j]$ érték az $A[i..j]$ szakasz mátrixainak optimális összeszorzásához szükséges elemi szorzások számát fogja tárolni. Mivel $i \leqslant j$, csak a főátlón és a főátló feletti háromszögben lévő elemeket fogjuk használni. A triviális feladatok optimumértékei a főátlóra, az eredeti feladaté pedig a $c[1][n]$ tömbelembe fognak kerülni.

IV. A rekurzív képlet, amely leírja, miként épülnek fel az egyre bonyolultabb részfeladatok optimális megoldásai az egyszerűbbek optimális megoldásaiból, az alábbi:

$$\begin{aligned} c[i][j] &= 0, \text{ ha } i = j \\ &= \min\{c[i][k] + c[k+1][j] + d[i-1]d[k]d[j]\}, \text{ ha } i > j,\ i \leqslant k < j. \end{aligned}$$

V. Mivel az egyszerűtől haladunk a bonyolult fele, a fenti képlet szerint a $c[i][j]$ elem kitöltésekor rendelkezésre kell álljanak már a $c[i][k]$ és $c[k+1][j]$ elemek értékei, ahol $i \leqslant k < j$. A 7.35. ábra olyan feltöltési sorrendeket mutat be, amelyek biztosítják ezt. Az algoritmust az első ábra szerint írtuk meg. Az utolsó ábra szerinti megoldás az alábbi gondolatmenetet követi: kiindulva az egy hosszú szakaszok optimális megoldásaiból, átlóról átlóra haladva, felépítjük először a kettő hosszú, azután a három hosszú, ..., és végül az n hosszú mátrixsoroknak megfelelő feladatok optimális megoldásait.

```
minden i=1,n végezd
   c[i][i]=0
vége minden
```

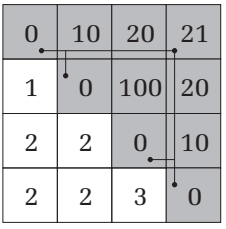


7.35. ábra.

```
minden i=n-1,1,-1 végezd
   minden j=i+1,n végezd
      c[i][j]=∞
      minden k=i,j-1 végezd
         ha c[i][k] + c[k+1][j] + d[i-1]*d[k]*d[j] < c[i][j]
         akkor
            c[i][j] = c[i][k] + c[k+1][j] + d[i-1]*d[k]*d[j]
            c[j][i]=k
         vége ha
      vége minden
   vége minden
vége minden
ki: c[1][n]
```

Megjegyzés: A $c[j][i]$ tömbelemben eltároltuk az $A[i..j]$ szakasz optimális vágásának a helyét (az optimális k értéket). Ez segít majd az optimális megoldás felépítésében.

VI. Mivel most már rendelkezünk mindenik „$i-j$ részfeladat" esetében az optimális vágások helyével ($c[j][i]$), a mátrixsor optimális zárójelezése egy egyszerű divide et impera feladatra redukálódott. Az alábbi eljárás nem más, mint a feladathoz rendelt fa optimális bináris részfájának a mélységi bejárása.



7.36. ábra.

```
zárójelezés(i,j)
   ha i==j akkor
      kiír 'A',i
   különben
      ki: '('
      zárójelezés(i,c[j][i])
      ki: '*'
      zárójelezés(c[j][i]+1,j)
      ki: ')'
   vége ha
vége zárójelezés
```

A fenti rekurzív eljárás a megadott példára a `zárójelezés(1,n)` hívás nyomán az alábbi megoldást írja ki: `((A1*A2)*(A3*A4))`

7.9. Könyvespolc: Adott egy H cm hosszú polc és n könyv, amelyeknek ismert a vastagságuk cm-ben. Legyen az i-edik könyv vastagsága v_i, ahol $i = 1, \ldots, n$. Mely könyveket tegyük fel a polcra, ha úgy szeretnénk teljesen megtölteni a polcot, hogy a lehető legtöbb könyvet helyezzük el?

Megoldás (gyökér–levelek irányú dinamikus programozás):

I. Legyen a következő példa: $n = 5$, $H = 10$, $v = (\mathbf{3}, 7, \mathbf{3}, \mathbf{4}, 3)$

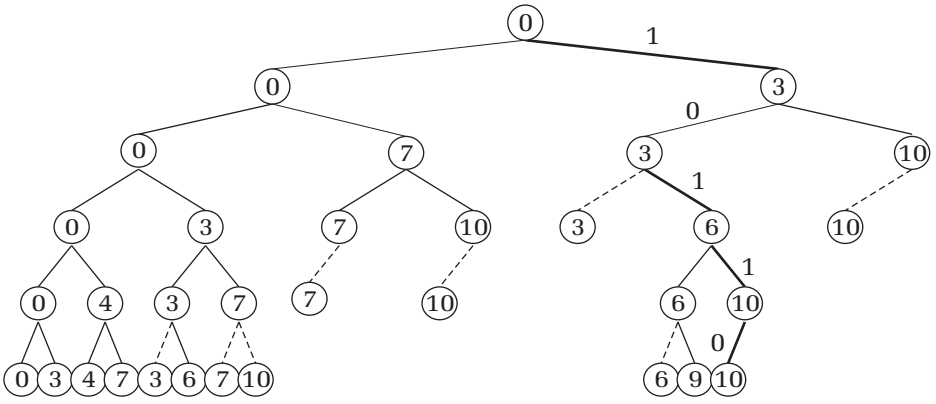
Az egyik optimális megoldást az jelentené, ha az első, harmadik és negyedik könyvet tennénk fel a polcra. Világos, hogy a mohó megközelítés, miszerint a könyveket vastagságuk szerint növekvő sorrendben választjuk ki, nem kielégítő.

Tömören fogalmazva, a következőképpen közelíthetnénk meg a feladatot: sorra meghatározzuk, milyen polcszélességek miként rakhatók ki optimálisan, ha az első, ha az első kettő, és végül, ha mind az n könyvből válogatunk. Ebben a megközelítésben a részfeladatok általános alakja: egy h ($0 \leqslant h \leqslant H$) polcszélesség optimális kirakása az első i ($0 \leqslant i \leqslant n$) könyv segítségével. Ezért a későbbiekben az általános feladatra néha úgy utalunk mint (h, i) feladatra. Az eredeti feladatot a $h = H$ és $i = n$ értékekre kapjuk. Optimális alatt természetesen azt értjük, hogy az illető kirakási mód a felhasználható könyvek közül a lehető legtöbbet tartalmazza. Más szóval, az optimalizálandó érték a polcra felkerülő könyvek száma.

Legyen a kezdeti állapot az, hogy a polc üres (ez így is természetes). Az első könyvet vagy feltesszük a polcra vagy nem (első döntés). Az első könyv felőli döntésünk után két állapot alakulhat ki: üres polc (0), a polcon az első könyv (v_1). A zárójelben az illető állapothoz tartozó

kirakott polcszélességet tüntettük fel. A második könyvről hasonlóképpen el kell döntenünk (második döntés), hogy használjuk vagy sem, ami azt jelenti, hogy a következő állapotok alakulhatnak ki: üres polc (0), a polcon az első könyv (v_1), a polcon a második könyv (v_2), a polcon az első két könyv ($v_1 + v_2$). Figyeljük meg, hogy minden lépésben az állapottér megduplázódik. Egy lépés alatt egy újabb könyv – a következő – bevonását értjük. Mivel mindig dönthetünk úgy, hogy nem használjuk a soron következő (i-edik) könyvet, elmondható, hogy az első i könyv segítségével kirakható polcszélességek magukba foglalják az első $(i-1)$ könyvvel kirakhatókat is (ahol $i = 1, 2, \ldots, n$). Viszont, ha használjuk az i-edik könyvet, akkor minden $(i-1)$-edik lépésbeli kirakott polcszélességből adódik egy újabb (például a h értékűből a $h + v_i$ értékű). Tehát milyen új kirakott polcszélességértékek jelennek meg az i-edik lépésben? Azok, amelyeket úgy kapunk, hogy minden $(i-1)$-edik lépésbeli értékhez hozzáadjuk az i-edik könyv vastagságát (v_i).

Az állapottér növekedését nyilván egy bináris fával szemléltethetnénk (lásd az ábrát). Biztosan felismerte az olvasó, hogy úgy közelítettük meg a feladatot, mint gyökér–levelek irányú dinamikus programozási problémát. Részfeladat alatt a csomópontok képviselte állapotokhoz tartozó prefixfeladatot értettük. Egy adott részfeladatban kirakott polcszélességet a csomópont értéke, a felhasználható könyvek számát a csomópont szintje, a megoldást pedig a gyökérből az illető csomóponthoz vezető út képviseli a döntési fában. Konkrétabban: az i-edik szinti csomópontok, a gyökérből odavezető úttal együtt, azt ábrázolják, hogy



7.37. ábra.

az első i könyv felhasználásával milyen polcszélességek milyen módokon rakhatók ki (építhetők fel).

Bár az i-edik szinti csomópontok száma elméletileg 2^i, ezek nem képviselnek mind különböző polcszélességeket. Sőt, megjelenhetnek H-nál nagyobb polcszélességek is, amelyeket természetesen figyelmen kívül hagyhatunk. Fontos észrevétel, hogy bár a fa exponenciálisan nő, egy adott szinten az egymástól különböző értékű csomópontok száma nem haladja meg H-t. Ha több azonos szintű csomópontnak ugyanaz az értéke, akkor nyilván ugyanazt a részfeladatot képviselik (ugyanaz a (h, i) értékpár felel meg nekik). Ez azért történhet meg, mert ugyanarra a (prefix) részfeladatra több megoldás is létezhet: egy adott h polcszélességet az első i könyv segítségével többféleképpen is kirakhatunk.

A részfeladatok megoldásai bináris sorozatok által kódolhatók. Például, a „h polcszélesség első i könyvvel való kirakása" általános részfeladat megoldásai olyan $b_1 b_2..b_i$ alakú bináris sorozatok lesznek (b_j, ahol $j = 1, 2, \ldots, i$, attól függően 1 vagy 0, hogy a j-edik könyv szerepel vagy sem az illető megoldásban), amelyekre $\Sigma b_j v_j = h$ (a megoldásban szereplő könyvek vastagságainak összege h). Így a megadott példa esetén az optimális megoldás kódja az összes könyv felhasználása után 10110 lesz (az első, harmadik és negyedik könyvet használjuk). Ha több egyforma hosszúságú, de különböző elemű kódra a $\Sigma b_j v_j$ érték azonos, akkor az illető kódok ugyanannak a részfeladatnak különböző megoldásait képviselik. Az optimális megoldás az lesz, amelyiknek a kódja a legtöbb 1-es számjegyet tartalmazza. A fában a kódolást úgy tüntettük fel, hogy a balra leágazó élekre 0-st (nem használjuk az illető könyvet), a jobbra leágazókra pedig 1-est (használjuk az illető könyvet) írtunk.

Ezen kódolást figyelembe véve, úgy is megfogalmazhatjuk a feladatot, hogy keressük azt a lehető legtöbb 1-es számjegyet tartalmazó, n elemű bináris sorozatot, amelyre igaz, hogy $\Sigma b_i v_i = H$, ahol $i = 1, \ldots, n$.

II. *Az optimalitás alapelvének ellenőrzése:* Tegyük fel, hogy D_1, D_2, $\ldots$, D_i, a (h, i) feladat, $b_1 b_2 \ldots b_i$ kódú optimális megoldáshoz vezető optimális döntéssorozat. Ez azt jelenti, hogy nem létezik olyan i elemű bináris sorozat, amely $b_1 b_2 \ldots b_i$-nél több 1-est tartalmaz, és érvényes rá a $\Sigma b_j v_j = h$ összefüggés ($j = 1, \ldots, i$). Két esetet különböztethetünk meg: $b_i = 1$ vagy $b_i = 0$, attól függően, hogy használtuk vagy sem az i-edik könyvet. Be fogjuk bizonyítani, hogy ezen feltételek mellett a $b_1 b_2 \ldots b_{i-1}$ kódú megoldás ugyancsak optimálisan oldja meg az első esetben a $(h - v_i, i - 1)$, a másodikban pedig a $(h, i - 1)$ részfeladatot. Ez nyilvánvaló, hiszen ha létezne valamelyik esetben egy olyan $a_1 a_2 \ldots a_{i-1}$

kódú megoldás, amely az első $i-1$ könyvből többet használ fel, mint a $b_1b_2\ldots b_{i-1}$ kódú, akkor az $a_1a_2\ldots a_{i-1}b_i\ldots b_n$ kódú megoldás is optimálisabban oldaná meg az eredeti feladatot, mint a $b_1b_2\ldots b_n$ kódú, ami viszont ellentmond a feltételnek. Tehát a D_1, D_2, ..., D_i döntéssorozat optimalitásából következik a D_1, D_2, ..., D_{i-1} részdöntéssorozat optimális volta. Általánosítva: a D_1, D_2, ..., D_n döntéssorozat optimalitása feltételezi minden D_1, D_2, ..., D_j ($j = 1, 2, \ldots, n-1$) alakú részdöntéssorozat optimalitását. Amint várható, az optimalitás alapelvének ezen alakja a dinamikus programozás gyökér–levelek irányú változatának felel meg. A példafeladatra alkalmazva: az 1, 10, 101, 1011, illetve 10110 kódú megoldások mind optimálisak, az első 1, 2, 3, 4, illetve 5 könyv használata mellett, a 3, 3, 6, 10, illetve 10 polcszélességek kirakására.

Egy nagyon fontos következtetést vonhatunk le az optimalitás alapelvének ellenőrzése nyomán: *az első i könyv felhasználásával kirakható polcszélességek optimális megoldásai meghatározhatók az első $i-1$ könyv segítségével optimálisan kirakott polcszélességek megoldásaiból, ahol $i = 1, \ldots, n$.* Ez azt jelenti, hogy minden lépésben elég csak az addig felhasznált könyvekkel kirakható polcszélességeknek az optimális megoldásait eltárolni.

Hogyan tükröződik az optimalitás alapelve a fentebb bemutatott döntési fában? Ha egy adott i-edik lépésben a fa i-edik szintjén több azonos értékű csomópont jelenik meg (a H-nál nagyobb értékű csomópontokat be se rajzoltuk), akkor az $(i+1)$-edik lépésben csak azon ág irányába fog tovább növekedni, amelyik az illető állapot prefix feladatának az optimális megoldását képviseli. Pontozott vonallal jelöltük azokat az „elhalt” ágakat, amelyek nem nőttek tovább. Végül, az n-edik szinti levélértékek adják meg, hogy milyen polcszélességek rakhatók ki az összes könyv felhasználásával. Ha létezik H értékű levél közöttük, akkor a hozzá vezető út ábrázolja a feladat optimális megoldását.

III. Tekintettel arra, hogy az általános feladatnak két független paramétere van, a részmegoldások optimumértékei egy $c[0..i][0..H]$ kétdimenziós tömbben tárolhatók el. Pontosabban, a $c[i][h]$ tömbelem azt fogja tárolni, hogy hány könyv szerepel a h polcszélesség első i könyv felhasználásával történő optimális kirakásában. Ha egy h polcszélesség nem rakható ki az első i könyv segítségével, azt a $c[i][h] = -1$ érték fogja jelezni.

IV. Nulla könyvvel nyilván csak a 0 értékű polcszélesség rakható ki. Általánosan: milyen esetekben rakható ki a h polcszélesség az első i könyv segítségével? Ha az első $(i-1)$ könyv segítségével vagy a h, vagy

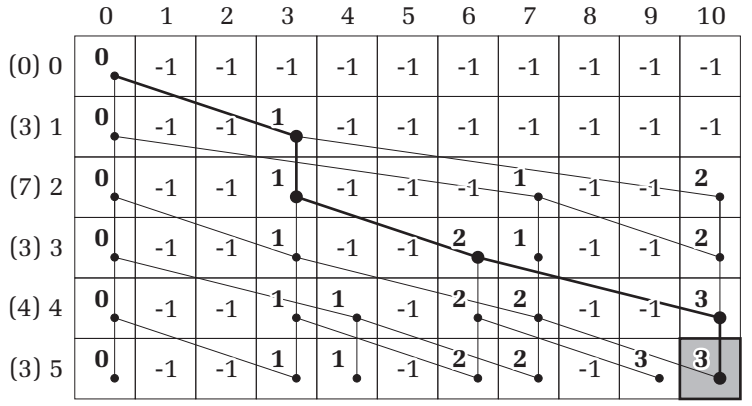
a $h - v_i$ polcszélességeket kiraktuk már, ugyanis csakis ezekből az értékekből hozható ki az i-edik könyv segítségével a h polcszélesség, azáltal, hogy használjuk vagy nem. Természetesen, amennyiben mindkét lehetőség fennáll, csak az optimális megoldást fogjuk eltárolni, azt, amelyik több könyvet használ.

$c[0][0] = 0$
$c[0][h] = -1$, ahol $1 \leqslant h \leqslant H$

Minden $1 \leqslant i \leqslant n$ és $0 \leqslant h \leqslant H$ esetén,

ha $h \geqslant v_i$ és $c[i-1][h-v_i] > -1$,
akkor $c[i][h] = \max\{c[i-1][h], c[i-1][h-v_i]+1\}$,
különben $c[i][h] = c[i-1][h]$.

V. Mivel a c tömb i-edik sorának elemei az $(i-1)$-edik sor elemei segítségével számíthatók ki, a tömb feltöltése soronként történik fentről lefele irányba. A sorindexek elé beírtuk zárójelbe az illető könyv vastagságát. Berajzoltuk a tömbbe az összevont döntési fát, megvastagítva az egyik optimális megoldást képviselő gyökér–levél utat. Emlékezzünk rá, hogy az optimalitás alapelve abban tükröződik az összevont döntési fában gyökér–levelek irányú dinamikus programozás esetén, hogy minden csomópontnak csak az „optimális” apacsomópontját hagyjuk meg (a többit levágjuk róla). A tömb feltöltésekor ez a `max` függvény általi választás révén valósul meg.



7.38. ábra.

```
c[0][0]=0
minden h=1,H végezd
   c[0][h]=-1
vége minden
minden i=1,n végezd
   minden h=0,H végezd
      c[i][h]=c[i-1][h]
      ha h >= v[i] és c[i-1][h-v[i]] > -1
                              és c[i-1][h-v[i]]+1 > c[i][h]
      akkor
         c[i][h]= c[i-1][h-v[i]]+1
      vége ha
   vége minden
vége minden
ki: "Legtöbb ",c[n][H]," könyv felhasználásával rakható ki
                                        a ",H," széles polc."
```

VI. Hogyan olvasható ki az optimális megoldás a c tömbből? Indulva a $c[n][H]$ tömbelemtől, most már optimális döntésről optimális döntésre lépegetve, visszamehetünk a kezdeti állapotot képviselő $c[0][0]$ tömbelemhez. A fán ez azt jelenti, hogy a megoldáslevélből felmegyünk a gyökérbe. Azért, hogy ne fordított sorrendbe írjuk ki a felhasznált könyvek sorszámait, egy rekurzív eljárás által előbb „felmászunk” az optimális levél–gyökér úton, és a rekurzió visszaútján írjuk ki az optimális döntéssorozatnak megfelelő könyveket.

```
könyvek(i,h)
   ha nem (i==0 és h==0) akkor
      ha h >= v[i] és c[i-1][h-v[i]] > -1
                             és c[i-1][h-v[i]]+1 == c[i][h]
      akkor
         könyvek(i-1,h-v[i])
         kiír: i
      különben
         könyvek (i-1,h)
      vége ha
   vége ha
vége könyvek
```

Fel lehetett volna fogni az optimális polckirakás feladatát is mint levelek–gyökér irányú dinamikus programozási problémát? Minden bizonnyal! A (h, i) prefix részfeladat komplementer szúfix részfeladata a $H - h$ polcszélességnek az $i + 1, \ldots, n$ könyvekkel való optimális

kirakása. Nem nehéz átlátni, hogy ez esetben a két változat csak abban különbözne, hogy fordított sorrendben foglalkoznánk a könyvekkel.

Egy másik megközelítése a feladatnak az lenne, hogy ne könyvről könyvre, hanem polcszélességről polcszélességre $(0, 1, 2, \ldots, H)$ lépegessünk. Egy adott h polcszélesség akkor rakható ki, ha valamelyik $h - v_i$ $(i = 1, \ldots, n)$ polcszélesség már ki lett rakva az i-edik könyv felhasználása nélkül. Az illető kirakott polcszélességhez hozzávesszük az i könyvet, és máris van egy megoldásunk a h polcszélesség kirakására. Több megoldás esetén nyilván azt fogjuk eltárolni, amelyik a legtöbb könyvet használja.

Hogyan fogjuk eltárolni ez esetben a részfeladatok optimális megoldásait? Használhatnánk egy $c[0..H]$ egydimenziós tömböt, amelyben a $c[h]$ $(h = 0, H)$ tömbelem a h polcszélesség optimális kirakásában felhasznált könyvek számát tárolja (ha egy polcszélesség nem kirakható, akkor a megfelelő tömbelem értéke legyen -1). Emellett szükséges egy $K[0..H][1..n]$ kétdimenziós tömb is, amelyikben eltároljuk, hogy mely könyveket használtuk fel az egyes polcszélességek optimális kirakásánál. Konkrétabban, a K tömb h-adik sora a h polcszélesség optimális kirakásában felhasznált könyveket tárolja: ha $K[h][i] = 1$, akkor használtuk az i-edik könyvet, ha viszont 0, akkor nem.

Az algoritmus implementálása maradjon gyakorlat az olvasónak.

Bízva abban, hogy az eddig bemutatott megoldott feladatok révén már magunkévá tettük a dinamikus programozás szellemét, az alábbi feladatok megoldására csak rávezetjük az olvasót.

7.10. Zsetonrakás: Legyen a következő kétszemélyes játék. Adott egy n ($n \leqslant 1000$) zsetonból álló rakás. A zsetonok kétszínűek, és sorszámozva vannak 1-től n-ig (az első a legfelső). A zsetonok színét az $a[1..n]$ tömb tárolja. A játékosok felváltva lépnek. Lépésen azt értjük, hogy a soron következő játékos elvesz a rakás tetejéről legalább egy azonos színű zsetont. Az lesz a nyertes, aki megkaparintja az utolsó zsetont. Írjunk programot, amely eldönti, hogy adott zsetonrakás mellett létezik-e biztos nyerő stratégiája annak, aki kezd. Ha a válasz igen, akkor a program szimuláljon egy játékot a számítógép és a felhasználó között. A győztes stratégiát a számítógépnek implementáljuk.

Megoldás (levelek–gyökér irányú dinamikus programozás):

I. Képzeljük el, hogy elkezdődik a játék. A játékosok lépései nyomán milyen alakú (szúfix) részfeladatokká redukálódik az eredeti feladat? Nem nehéz belátni, hogy a részfeladatok általános alakja: meghatározni, hogy létezik-e biztos nyerő stratégia egy $i..n$ rakás esetén az éppen soron

következő játékosnak (nyilván, ha neki van, akkor ellenfelének nincs). Az eredeti feladatot $i = 1$, az egyetlen triviális részfeladatot pedig $i = n$ paraméterértékek mellett kapjuk.

II. Lévén az általános alaknak egyetlen független paramétere, a részfeladatok „optimumértékeit" a $c[1..n]$ tömbben fogjuk eltárolni. Pontosabban a $c[i]$ elem attól függően lesz 1 vagy 0, hogy a soron következő játékosnak van vagy nincs biztos nyerő stratégiája az $i..n$ zsetonrakásban. Az eredeti feladat megoldása a $c[1]$, a triviális részfeladaté pedig a $c[n]$ tömbelembe fog kerülni.

III. Mivel a részfeladatokat „lentről felfele" irányban oldjuk meg, a $c[i]$ tömbelem kiszámításakor már rendelkezésre állnak a $c[i+1]$, $c[i+2]$, $\ldots$, $c[n]$ értékek.

Ha $a[i] \neq a[i+1]$ (az i-edik és $(i+1)$-edik zsetonok különböző színűek), akkor a soron következő játékosnak nincs más választása, mint elvenni az i-edik zsetont, és így az „i-feladat" egyértelműen az „$(i+1)$-feladatra" vezetődik vissza. Ez azt jelenti, hogy ha az $(i+1)..n$ rakásban volt biztos nyerő stratégia, akkor az $i..n$ rakásban nincs, és fordítva, ha az $(i+1)..n$ rakásban nem volt, akkor az $i..n$ rakásban van.

Ha az i-edik és $(i+1)$-edik zsetonok azonos színűek ($a[i] = a[i+1]$), akkor a soron következő játékosnak mindenképpen biztos nyerő stratégiája van. Miért? Ha az $(i+1)..n$ rakásban nincs biztos nyerő stratégia, akkor csak a legfelső zsetont veszi el (otthagyva társának a vesztes rakást), ellenkező esetben az i-edik zsetonnal együtt elveszi ellenfele elől az $(i+1)..n$ rakás tetejéről azokat a zsetonokat is, amelyek ebben a rakásban a biztos győzelemhez vezető első lépést jelentik.

Íme a rekurzív képlet:

$$\begin{aligned} c[n] &= 1, \\ c[i] &= 1, \text{ ha } a[i] = a[i+1] \\ &= 1 - c[i+1], \text{ ha } a[i] \neq a[i+1] \\ &\qquad\qquad \text{bármely } i = (n-1), (n-2), \ldots, 1 \text{ esetén} \end{aligned}$$

IV., V. A c tömb feltöltését megvalósító eljárásnak a megírását, illetve a játék szimulálásának az implementálását az olvasóra hagyjuk.

7.11. Karakterláncok egymásba alakítása: Adott két karakterlánc $x[1..n]$ és $y[1..m]$, amelyek számjegyeket tartalmaznak. Karakterről karakterre haladva x-ben, állítsuk elő x karaktereiből az y karaktereit (sorban, egyenként). Tegyük fel, hogy az $x[1\ .\ .\ i-1]$ szakasz felhasználásával

már előállítottuk az $y[1\,..\,j-1]$ szakaszt. Tehát x-ben az i-edik, y-ban pedig a j-edik az aktuális karakter. Az alábbi műveletek megengedettek (a zárójelekben az illető műveletek kódját látjuk):

1. hozzáad számjegyet (+): ha $y[j] \geqslant x[i]$, akkor $y[j]$ előállítható az $x[i]$ + <*számjegy*> művelet révén.
2. kivon számjegyet (–): ha $y[j] \leqslant x[i]$, akkor $y[j]$ előállítható az $x[i]$ – <*számjegy*> művelet révén.
3. másol (=): ha $y[j] = x[i]$, akkor $y[j]$ előállítható az $x[i]$ egyszerű átmásolásával.
4. beszúr (*): $y[j]$-t nem $x[i]$-ből állítjuk elő, hanem egyszerűen beszúrjuk az előállítás alatt lévő karakterlánc végére, vagyis a j-edik pozícióba. Ezt a műveletet követően x-ben továbbra is $x[i]$ marad az aktuális karakter.
5. átlép (/): átlépjük az $x[i]$ karaktert. Ilyenkor y-ba nem kerül karakter, és így továbbra is $y[j]$ lesz a következő előállítandó karakter.
6. végére ugrik (#): átugorjuk az összes karaktert, ami még megmaradt x-ben (a teljes $x[i..n]$ szakaszt). Természetesen ezzel a művelettel sem kerül y-ba karakter.

Tudva azt, hogy az 1., 2. és 4. műveletek költsége kettő, a többieké pedig egy, végezzük el az átalakítást minimális költséggel.

Példa: Ha $x = 372$ és $y = 78$, akkor az optimális megoldás költsége 4, és a következő műveletsorral valósítható meg: /=+6 (a 3-ast átlépjük, a 7-est átmásoljuk, a 2-eshez hozzáadunk 6-ot).

Megoldás (gyökér–levelek irányú dinamikus programozás):

I. Ahogy a feladat szövege is utal rá, a (prefix) részfeladatok általános alakja: az $x[1..i]$ részkarakterlánc optimális átalakítása az $y[1..j]$ részkarakterlánccá. Az eredeti feladatot az $i = n$ és $j = m$ paraméterértékekre kapjuk. Triviális részfeladatnak azt fogjuk tekinteni, ha valamelyik karakterlánc üres ($i = 0$ vagy $j = 0$).

II. A részfeladatok optimális megoldásának optimumértékét a $c[0..n][0..m]$ kétdimenziós tömbben fogjuk eltárolni. A $c[i][j]$ tömbelem az $x[1..i]$ részkarakterláncnak az $y[1..j]$ részkarakterlánccá való átalakításához szükséges minimális költséget fogja tartalmazni. A nagy feladat optimumértéke természetesen a $c[n][m]$ rekeszbe fog kerülni. A tömb 0-dik sorában és oszlopában levő elemek a triviális részfeladatok optimumértékeit kapják kezdőértékként.

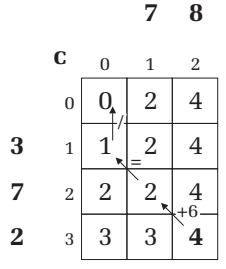
III. Tekintettel arra, hogy gyökér–levelek irányú dinamikus programozást választottunk, a $c[i][j]$ tömbelem kitöltésekor rendelkezésre

állnak már a $c[i-1][j]$, $c[i][j-1]$, $c[i-1][j-1]$, ... értékek. Hogyan határozható meg ezen optimumértékekből a $c[i][j]$ optimumérték?

- Ha x üres ($i = 0$), akkor bármely $y[1..j]$ ($j > 0$) karakterláncszakasz csakis j darab beszúrási művelettel állítható elő, amelyeknek összköltsége $2 \cdot j$.
- Ha y üres ($j = 0$), akkor bármely $x[1..i]$ ($0 < i < n$) karakterláncszakasz esetén i átlépésműveletet kell végrehajtanunk, amelynek költsége i.
- Ha $i = n$, akkor $c[n][j]$ ($0 \leq i \leq n$) felépíthető bármely $c[k][j]$-ből ($k = 0, n-1$) egyetlen végére ugrással.
- Általános esetben $c[i][j]$ felépíthető $c[i-1][j]$-ből átlépéssel, $c[i][j-1]$-ből beszúrással, illetve $c[i-1][j-1]$-ből vagy hozzáadással, vagy kivonással, vagy másolással.

```
c[0][0] = 0
c[0][j] = 2*j, minden j=1, 2, ..., m
c[i][0] = i, minden i=1, 2, ..., n
c[n][j] = 1 + min{c[k][j]/ ahol k=0, 2, ..., n-1}
          minden j=1, 2, ..., m

ha 1<=i<n és 1<=j<=m, akkor
   c[i][j] = min{ 1+c[i-1][j], // átlépés
                  2+c[i][j-1], // beszúrás
                  2+c[i-1][j-1], // hozzáadás, ha y[j]>x[i]
                  2+c[i-1][j-1], // kivonás, ha y[j]<x[i]
                  1+c[i-1][j-1], // másolás, ha y[j]=x[i]
                  }
```

IV. Miután inicializáltuk a c tömb 0-dik sorát, illetve oszlopát, a $c[1..n][1..m]$ tömbterület feltöltése történhet soronként (fentről lefele, balról jobbra). Az algoritmus megírása maradjon gyakorlat.



7.39. ábra.

V. A rekurzív eljárásnak a megírását, amely indul a $c[n][m]$ tömbelemtől, „visszamászik” optimális úton (döntésről döntésre) a c[0][0] tömbelemig, majd pedig a rekurzió visszaútján kiírja az optimális megoldáshoz vezető műveletsort, ugyancsak az olvasóra hagyjuk. Segítségképpen közöljük a c tömb tartalmát a magadott példára (7.39. ábra).

7.12. Virágüzlet: Legyen n különböző fajtájú virágcsokor 1-től n-ig megszámozva, és m különböző típusú váza 1-től m-ig megszámozva ($n \leqslant m$). Mind a vázák, mind a virágcsokrok *sorszámuk szerint növekvő sorrendben* vannak elhelyezve (például balról jobbra irányban) a virágüzlet kirakatában. Minden vázába egy csokrot teszünk, és ha több váza van, mint csokor, akkor egyesek üresen maradnak. Továbbá minden virágcsokornak föltétlenül a kirakatba kell kerülnie.

Egy $a[1..n][1..m]$ mátrixban adott, hogy az egyes csokrok, különböző vázákban, milyen mértékű esztétikai hatást keltenek. Pontosabban az $a[i][j]$ elem értéke annak fokmérője, hogy hogyan néz ki az i-edik virág a j-edik vázában. Egy üresen maradt váza esztétikai értéke 0.

Határozzuk meg azt a kompozíciót, amelynek összesztétikai hatása maximális.

Példa: Ha $n = 3$, $m = 5$ és az a mátrix az alábbi,

$$\begin{array}{rrrrr} 7 & 23 & -5 & -24 & 16 \\ 5 & 21 & -4 & 10 & 23 \\ -21 & 5 & -4 & -20 & 20, \end{array}$$

akkor a kihozható maximális összesztétikai hatás 53. Az optimális megoldás kódja pedig 2, 4, 5 (az 1. csokrot a 2. vázába, a 2. csokrot a 4. vázába, a 3. csokrot az 5. vázába tesszük).

Megoldás (gyökér–levelek irányú dinamikus programozás):

I. Ha elkezdjük elhelyezi a virágokat a vázákba, rögtön látszik, hogy a (prefix) részfeladatok általános alakja : az első i virág optimális elhelyezése az első j vázába ($i = 1..n$, $j = i \,..\, m-n+i$). A j index határai abból adódnak, hogy az első i virág elhelyezéséhez szükség van legalább az első i vázára. Továbbá legfennebb az első $m-n+i$ vázát használhatjuk fel az első i virág elhelyezésénél, hogy a fennmaradt $n-i$ virág részére is maradjon legalább $m-(m-n+i) = n-i$ váza. Az eredeti feladatot nyilván az $i = n$ és $j = m$ paraméterértékekre kapjuk. Triviális részfeladatnak tekinthetjük azokat az eseteket, amikor a virágcsokorszám nulla ($i = 0$).

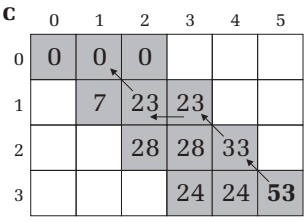
II. A részfeladatok optimumértékeit egy $c[0..n][0..m]$ kétdimenziós tömbben fogjuk eltárolni. A $c[i][j]$ tömbelembe az a maximális esztétikai

hatás kerül, amely kihozható az első i virág esetében, az első j vázába való elhelyezésükkor. A triviális részfeladatok optimumértékeivel a tömb 0-dik sorát inicializáljuk. Az eredeti feladat optimumértéke a $c[n][m]$ tömbelembe kerül.

III. Ha az i-edik virágot a j-edik vázába tesszük, akkor ez azt jelenti, hogy a $c[i][j]$ értékét a $c[i-1][j-1]$ értékre (az első $i-1$ virág optimális elhelyezése az első $j-1$ vázában) alapozva határozzuk meg. Ha $j > i$, akkor még van az a lehetőség is, hogy a j-edik vázát üresen hagyjuk. Ez esetben a $c[i][j]$ értékét a $c[i][j-1]$ értékre (az első i virág optimális elhelyezése az első $j-1$ vázában) támaszkodva számítjuk ki. Természetesen a két variáns közül azt fogjuk eltárolni $c[i][j]$-ben, amelyik a nagyobb összesztétikai hatást biztosítja.

$$c[0][j] = 0, \text{ ahol } j = 0, m-n$$
$$c[i][j] = c[i-1][j-1] + a[i][j], \text{ ha } i = j$$
$$c[i][j] = \max\{c[i-1][j-1] + a[i][j], c[i][j-1]\}, \text{ ha } i < j \leqslant m-n+i$$

IV., V. Az alábbi ábra a megadott példán szemlélteti, hogy miként kell feltöltenünk a c tömböt:



7.40. ábra.

A c tömb feltöltését megvalósító eljárásnak, illetve az optimális csokorelhelyezés kódjának a kiírását biztosító rekurzív eljárásnak a megírása maradjon gyakorlat.

7.13. Remi: Adott n remi-rakás az $r[1..n]$ tömb segítségével ($r[i]$ azt tárolja, hogy hány remi van az i-edik rakásban). Csoportosítsuk k csoportba a rakásokat úgy, hogy szomszédos rakásokat teszünk egymásra. Az összremiszám osztható k-val, tehát elméletileg lehetséges lenne egyforma csoportok kialakítása, de a kért módszerrel természetesen ez nem mindig lehetséges. Egy csoport penalizációja egyenlő azzal az értékkel, amennyivel abszolút értékben eltér a $(\Sigma r[i])/k$ középértéktől. Határozzuk meg azt a csoportosítást, amelynek összpenalizációja minimális.

Példa: Legyen $n = 7$, és az egyes rakásokban legyen $12, 9, 2, 11, 15, 5$ és 2 remi. Továbbá a kért csoportszám legyen 4. Ez esetben az optimális csoportosítás: $(12, 9)$, $(2, 11)$, (15), $(5, 2)$. Az egyes csoportok penalizációja a 14-es középértékhez viszonyítva: $|21 - 14| = 7$, $|13 - 14| = 1$, $|15 - 14| = 1$ és $|7 - 14| = 7$. Ez összesen $7 + 1 + 1 + 7 = 16$ penalizációt jelent.

A kiíratás formátuma legyen: `1-2 3-4 5 6-7`.

Megoldás („optimális megosztás – optimális uralom"):

I. Nem nehéz belátni, hogy a részfeladatok általános alakja: egy $r[i..j]$ remirakás-szakasz optimális csoportosítása p csoportba, ahol $1 \leqslant i \leqslant j \leqslant n$, $1 \leqslant p \leqslant (j - i + 1)$ és $1 \leqslant p \leqslant k$. Az eredeti feladatot az $i = 1$, $j = n$ és $p = k$ paraméterértékekre kapjuk. Egy részfeladat akkor triviális, ha $p = 1$. Ha $i = j$, akkor p kötelezően 1.

II. Lévén három független paramétere a részfeladatok általános alakjának, az optimumértékek eltárolására a $c[1..n][1..n][1..k]$ háromdimenziós tömb bizonyos elemeit fogjuk felhasználni. Konkrétabban a $c[i][j][p]$ tömbelem azt az összpenalizáció-értéket tárolja, amely az $r[i..j]$ rakás-szakasz p csoportba való optimális csoportosításához tartozik. Az eredeti feladat optimumértéke a $c[1][n][k]$ tömbelembe, a triviális részfeladatokéi pedig a $p = 1$ értéknek megfelelő tömblapnak a főátló és főátló feletti elemeibe kerülnek.

III. Az i-edik rakástól a j-edik rakásig tartó szakasz $(j-i)$-féleképpen bontható két részszakaszra. Ezen részszakaszpárok alakja: $(r[i..f], r[f + 1\,..\,j])$, ahol $f = i, j - 1$. Továbbá mindenik részszakaszpár tagjai között a p csoportszámot is kétfele kell osztani. Általában ezt is többféleképpen lehet megvalósítani (pontosabban $\min\{f-i+1, j-f, p-1\}$-féleképpen). Ez az érték onnan jött ki, hogy az $r[i..f]$ szakasz legfennebb $f - i + 1$, az $r[f+1\,..\,j]$ szakasz legfennebb $j - f$, továbbá mindkét szakasz legfennebb $p - 1$ csoportba csoportosítható. Természetesen, ha rögzítjük, hogy egy adott részszakaszpár első tagját g csoportba fogjuk csoportosítani, akkor ebből egyértelműen adódik, hogy a másodikat $p - g$ csoportba kell csoportosítanunk. Íme a rekurzív képlet, amely matematikailag leírja, miként történik ez esetben az optimalitás alapelvére támaszkodó lentről felfele építkezés (jelöljük H-val a $(\Sigma r[i])/k$ középértéket):

$$c[i][j][1] = |(r[i] + \ldots + r[j]) - H|,\ 1 \leqslant i \leqslant j \leqslant n$$
$$c[i][j][p] = \min_f\{\min_g\{c[i][f][g] + c[f + 1][j][p - g]\}\}$$

Az áttekinthetőség kedvéért az f és g paraméterek határait nem írtuk újra be a képletbe, hiszen részletes leírást adtunk róluk.

IV. Ez esetben könnyebb a fenti képlet alapján megírni a c tömb feltöltését biztosító algoritmusrészletet, mint szavakkal elmagyarázni, milyen sorrendben kerülnek feltöltésre a tömbelemek, vagy szemléletesen lerajzolni ezt. Az $a[0..n]$ tömböt úgy töltjük fel, hogy $a[0] = 0$ legyen, és az $a[i]$ elem az $r[1] + \ldots + r[i]$ összeget tartalmazza ($i = 1, 2, \ldots, n$). Ez azért előnyös, mert az $a[j] - a[i-1]$ különbség konstans időben megadja az $r[i] + \ldots + r[j]$ összeget.

Azért, hogy rekonstruálni tudjuk az optimális csoportosítást, a c tömb feltöltésével párhuzamosan minden (i, j, p) paraméterértékhármasra eltároljuk egy $d[1..n][1..n][1..k]$ tömb $d[i][j][p]$ és $d[j][i][p]$ elemeibe az optimális f és g értékeket.

```
a[0]=0
minden i=1,n végezd
   a[i]=a[i-1]+r[i]
vége minden
minden i=n,1,-1 végezd
   minden j=i,n végezd
      minden p=1,k végezd
         ha p==1 akkor
            c[i][j][1] = |a[j]-a[i-1]-H|
         különben
            c[i][j][p]=∞
            minden f=i,j-1 végezd
               minden g=1,p-1 végezd
                  ha g < f+i-1 és p-g < j-f és
                     c[i][f][g] + c[f+1][j][p-g] < c[i][j][p]
                  akkor
                     c[i][j][p]=c[i][f][g]+c[f+1][j][p-g]
                     d[i][j][p]=f
                     d[j][i][p]=g
                  vége ha
               vége minden
            vége minden
         vége ha
      vége minden
   vége minden
vége minden
ki: c[1][n][k]
```

V. A következő rekurzív eljárás mélységében járja be a feladathoz rendelhető döntési fa optimális bináris részfáját, és közben kiírja az optimális csoportosítást.

```
csoportosítás(i,j,p)
   ha p==1 akkor
      ha i==j akkor ki: i,' '
      különben ki: i,'-',j,' '
      vége ha
   különben
      csoportosítás(i,d[i][j][p],d[j][i][p])
      csoportosítás(d[i][j][p]+1,j,p-d[j][i][p])
   vége ha
vége csoportosítás
```

Az eljárást a következőképpen hívjuk meg: `csoportosítás(1,n,k)`

7.14. Mérleg: Adott egy különleges kétkarú mérleg. A karok elhanyagolható súlyúak és $h = 15$ hosszúak. A karokon ismert pozíciókban elhanyagolható súlyú horgok vannak, összesen n horog. A horgok pozícióit a $p[1..n]$ tömbben tároljuk ($-h \leqslant p[i] \leqslant h$, ahol $i = 1, \ldots, n$). Adott továbbá m ($1 \leqslant m \leqslant 20$) különböző értékű súly a $g[1..m]$ tömbben ($1 \leqslant g[i] \leqslant 25$, ahol $i = 1, 2, \ldots, m$). Határozzuk meg, hányféleképpen egyenlíthető ki a mérleg az összes súly felhasználásával. Egy mérleg akkor kiegyensúlyozott, ha a ráaggatott súlyok forgatónyomatékainak (súly szorozva erőkarral) összege nulla.

Példa: Ha $n = 2$ és $m = 4$, továbbá, ha a horgok a -2 és 3 pozíciókban vannak, és a súlyok rendre $3, 4, 5$ és 8 értékűek, akkor a mérleg kétféleképpen egyenlíthető ki.

Megoldás (gyökér–levelek irányú dinamikus programozás):

I. A feladat a könyvespolc nevű megoldott feladatra emlékeztet. Nevezzük a mérleg értékének az éppen ráaggatott súlyok és a hozzájuk tartozó erőkarok (az erőkart a súly horgának a pozíciója adja meg) szorzatainak (a nyomatékoknak) összegét. A mérleg akkor kiegyensúlyozott, ha nullaértékű. Súlyról súlyra haladva, mindenik súlyt sorban mindenik horogra ráakasztva meghatározzuk, hogy milyen értékű mérlegek hozhatók ki. Az általános (prefix) részfeladat: hányféleképpen hozható ki az első i súly felhasználásával a j mérlegérték. Triviális részfeladat: nulla darab súllyal ($i = 0$) csakis a nullaértékű mérleg ($j = 0$) hozható ki. Az eredeti feladatot az $i = m$ és $j = 0$ értékekre kapjuk.

II. A $c[0..m][-max..max]$ kétdimenziós tömb $c[i][j]$ eleme azt fogja tárolni, hogy az első i súly felhasználásával a j értékű mérleg hányféleképpen hozható ki. A $-max$, illetve $+max$ mérlegértékeket akkor kapnánk, ha mindenik súlyt vagy a $-h$, vagy a $+h$ pozíciójú horogra akasztanánk (feltételezve, hogy vannak ezekben a pozíciókban horgok). Tehát $max = h\Sigma g[i]$, ahol $i = 1, 2, \ldots, m$. Ha a j mérlegérték nem rakható ki az első i súly segítségével, akkor a $c[i][j]$ elem a 0 értéket fogja kapni. Az eredeti feladat eredménye nyilván a $c[m][0]$ rekeszbe fog kerülni.

III. Amikor sorra kerül a c tömb i-edik sorának feltöltése, az $(i-1)$-edik sora már ki lett töltve és azt tartalmazza, hogy milyen mérlegértékek hányféleképpen rakhatók ki az első $(i-1)$ súly segítségével. Egy j mérlegérték az első i súly figyelembevételével akkor hozható ki, ha létezik egy olyan, az első $(i-1)$ súly segítségével már kirakott mérlegérték, amelyikhez ha hozzáadjuk az i-edik súlynak valamelyik horogpozícióra vonatkozó nyomatékát, a j értéket kapjuk. Természetesen annak megfelelően, hogy hányféleképpen volt kirakható az illető $(i-1)$-edik szinti mérlegérték, annyiféleképpen lesz kirakható belőle az i-edik szinti j mérlegérték. Persze megtörténhet, hogy létezik több olyan $(i-1)$-edik szinti mérlegérték is, amelyikhez találunk az i-edik súlynak olyan horgot, hogy ugyanahhoz a j mérlegértékhez jussunk. Ilyenkor nyilván összeadódik a kirakási módok száma.

$c[0][0] = 1$
$c[0][j] = 0$, ha $-max \leqslant j \leqslant max$ és $j \neq 0$

Minden $0 < i \leqslant m$ és $-max \leqslant j \leqslant max$ esetén

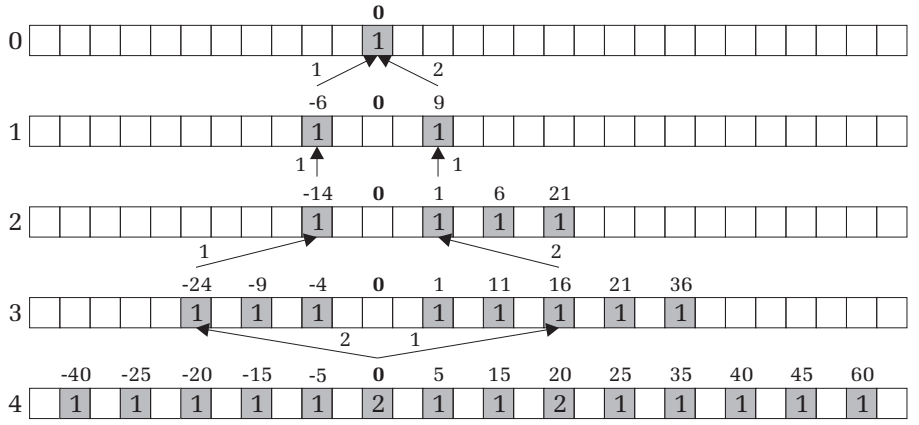
$c[i][j] = \Sigma c[i-1][k]$ minden olyan $k = -max..max$ esetén, amelyre $c[i-1][k] > 0$ és létezik olyan $r = 1, 2, \ldots, n$, hogy $k + g[i] \cdot p[r] = j$.

IV. Az eljárást, amely feltölti a c tömböt, úgy írjuk meg, hogy sorról sorra haladva, az aktuális sorból mindig előállítjuk a következőt. Az i-edik sor mindenik kirakott mérlegértékéből előállítjuk az összes mérlegértéket, amely abból adódhat, hogy az $(i+1)$-edik súlyt végighordozzuk az összes horgon.

```
minden i=0,m végezd
   minden j=-max,max végezd
      c[i][j]=0
   vége minden
vége minden
```

```
c[0][0]=1
minden i=0,m-1 végezd
   minden j=-max,max végezd
      ha c[i][j]≠0 akkor
         minden r=1,n végezd
            c[i+1][j+g[i+1]*p[r]] += c[i][j]
         vége minden
      vége ha
   vége minden
vége minden
```

Az alábbi ábra a c tömb sorait tartalmazza a példafeladatra. Minden sorban csak azokat az elemeket jelöltük be, amelyek kirakható mérlegértékeknek felelnek meg.



7.41. ábra.

V. A feladat nem kéri a kiegyensúlyozás módját is (csak ezek számát). Ebből kifolyólag nem lett volna kötelező a c tömböt kétdimenziósnak választani. Mivel az i-edik sort kizárólag az $(i-1)$-edikből építjük fel, ezért az $(i-1)$-edik sorra csak addig van szükségünk, amíg előállítjuk belőle az i-ediket. Így minden lépésben az újonnan generált sort nyugodtan ráírhattuk volna az azelőttire. Buzdítjuk az olvasót, hogy ha már rendelkezik a kétdimenziós c tömb tartalmával, írassa ki a mérleg $c[m][0]$-féle kiegyensúlyozási módját is. Ennek érdekében célszerű lenne a c tömb feltöltésekor azt is tárolni, hogy egy i-edik lépésbeli j mérlegérték kirakási száma az i-edik súlynak mely horgokra való ráakasztásából adódott (ezt az információt írtuk a nyilakra a fenti ábrán). A két kiegyensúlyozási mód kódja:

$1,1,1,2$ – az első három súlyt az első horogra, az utolsót pedig a másodikra tesszük: $(-2)\cdot 3+(-2)\cdot 4+(-2)\cdot 5+3\cdot 8=0$

$2,1,2,1$ – az első és harmadik súlyt a második horogra, a másodikat és a negyediket pedig az elsőre tesszük: $3{\cdot}3+(-2){\cdot}4+3{\cdot}5+(-2){\cdot}8=0$

7.15. Alma-körte: Legyen $n+1$ szoba, amelyek vagonszerűen követik egymást. Az első n szoba mindenikében egy bizonyos mennyiségű alma és körte található (az $a[1..n]$ és $k[1..n]$ tömbök $a[i]$ és $k[i]$ elemei, az i-edik szobában található almák, illetve körték számát tárolják). Egy elég nagy hátizsákkal rendelkező személynek a következőképpen kell végigmennie a szobákon, szobáról szobára haladva: megérkezve valamely i-edik szobába ($i=1,2,\ldots,n$), először kiüresíti a hátizsákját (a megfelelő gyümölcscsomóba), majd belepakolja a szobában található összes almát vagy összes körtét, és rakományát átviszi a következő szobába, az $(i+1)$-edikbe, ahol természetesen hasonlóan jár el (az első szobába természetesen üres hátizsákkal lép be). A kérdés az, hogy melyik gyümölcsöt pakolja fel az illető személy az egyes szobákban, ha azt szeretné, hogy minimális kalóriaveszteséggel érkezzen meg az $(n+1)$-edik szobába (két egymás utáni szoba között a szállított gyümölcsök számával egyenlő számú kalóriaegységet veszít az utas; a hátizsák kipakolása és megrakása nem jár kalóriaveszteséggel).

Megoldás (levelek–gyökér irányú dinamikus programozás): Emlékezzünk rá, hogy ezt a feladatot már megoldottuk egyszer a greedy és backtracking stratégiák kombinálásával.

A tiszta mohó megközelítés, miszerint mindig azzal a gyümölccsel megyünk tovább, amelyikből kevesebb van az illető szobában, nyilván nem kielégítő. Ezt szemlélteti az alábbi példa is:

$$n=2, a[1..2]=(4,3), k[1..2]=(5,6).$$

Ha a mohó gondolatmenetet követjük, akkor az első szobából az almákat visszük tovább, hiszen ebből van kevesebb. Ez 4 kalória veszteséggel jár. Miután kipakoljuk a hátizsákunkat, a második szobában $4+3=7$ alma és 6 körte lesz. Itt a mohó algoritmus nyilván a körtéket pakolná fel, és vinné át 6 kalória veszteséggel a harmadik szobába. Tehát a mohó stratégia szerinti megoldás 10 kalória veszteséggel jár. Létezik viszont ennél jobb megoldás is, amely csak 8 kalóriát igényel (az első szobából a körtéket, a másodikból pedig az almákat visszük tovább). Ezt a megoldást találja meg polinomiális időben az alábbi dinamikus programozás stratégia.

I. Ha elindulunk, néhány lépés után jól látszik, hogy döntéseink nyomán a feladat a következő alakú (szúfix) részfeladatokká redukálódik: határozzuk meg az $i..n$ szobákon való minimális kalóriaveszteségű áthaladást, ha az i-edik szobából indulnánk. Két esetet kell azonban megkülönböztetnünk: ha almával, illetve, ha körtével indulunk.

II. A részfeladatok optimumértékeinek tárolására két egydimenziós tömböt fogunk használni: $ca[1..n]$ és $ck[1..n]$. Egészen pontosan a $ca[i]$, illetve a $ck[i]$ tömbelemek azokat a minimális kalóriamennyiségeket tárolják, amelyek akkor szükségesek az $i..n$ szobákon való áthaladáshoz, ha az i-edik szobából almával, illetve körtével indulunk. Az eredeti feladat optimumát a $\min(ca[1], ck[1])$ érték jelenti majd.

III. Tekintettel a levelek–gyökér irányú módszerre, amikor sorra kerül a $ca[i]$, illetve a $ck[i]$ tömbelemek kitöltése, már rendelkezünk $(ca[i+1], ck[i+1]), \ldots, (ca[n], ck[n])$ értékpárokkal.

Az első esetben, amikor almával indulunk ($ca[i]$), a következő lehetőségek közül kell a legelőnyösebbet választanunk:

- Az i-edik szobából, mint első szobából, almával ($a[i]$) indulunk, de az $(i+1)$-edikből az ott található eredeti körtemennyiséggel megyünk tovább ($ck[i+1]$).
- Az i-edik szobából, mint első szobából, almával ($a[i]$) indulunk, az $(i+1)$-edikben megint az almákat választjuk ($a[i] + a[i+1]$), de az $(i+2)$-edikből az ott található eredeti körtemennyiséggel megyünk tovább ($ck[i+2]$).
- ...
- Egészen az $(n-1)$-edik szobáig mindenikben az almát választjuk, de az n-edikből az ott található eredeti körtemennyiséggel megyünk tovább ($ck[n]$).
- Mindenik szobában az almát választjuk.

A második esetben, amikor körtével indulunk, hasonló lehetőségek közül kell választanunk. Következzenek a rekurzív képletek:

```
ca[n] = a[n]
ck[n] = k[n]
minden i = (n - 1), (n - 2), ..., 1 esetén
   ca[i] = min{a[i] + ck[i+1],
               a[i] + (a[i] + a[i+1]) + ck[i+2],
               ...
               a[i] + (a[i] + a[i+1]) + ... +
                           + (a[i] +...+ a[n - 1]) + ck[n],
```

```
                a[i] + (a[i] + a[i+1]) + ... +
                                        + (a[i] +...+ a[n])
        }
        ck[i] = min{k[i] + ca[i+1],
                a[i] + (a[i] + a[i+1]) + ca[i+2],
                ...
                k[i] + (k[i] + k[i+1]) + ... +
                             + (k[i] +...+ k[n - 1]) + ca[n],
                k[i] + (k[i] + k[i+1]) + ... +
                                        + (k[i] +...+ k[n])
        }
```

IV., V. A *ca* és *ck* tömbök feltöltését, valamint ezek alapján a megoldás kiírását az olvasóra hagyjuk.

7.8. Kitűzött feladatok

7.1. Türelmetlen informatikus: Tegyük fel, hogy aplikáció n csomagból áll. A csomagokat egymást követően töltjük le az internetről, és egymást követően installáljuk is. Ismert mindenik csomag letöltési, valamint telepítési ideje. A letöltés elkezdése után leghamarabb mikor kezdődhet el az applikáció párhuzamos installálása, ha tudjuk, hogy egy csomag installálására nem kerülhet sor, csak ennek teljes letöltése után?

7.2. Diszjunkt szakaszok: Határozzuk meg egy n elemű számsorozat k darab, maximum m elemű diszjunkt szakaszát, amelyekhez tartozó elemek összege maximális.

Példa: $n = 7$, $m = 2$, $k = 3$, $a[1..7] = \{35, 40, 50, 10, 30, 45, 60\}$

A szakaszok: 1–2, 3–4, 6–7.

A maximális összeg: 240.

7.3. Bináris művelet: Adott az $A = \{1, 2, \ldots, n\}$ halmaz, amelyen definiáljuk az $o : A \times A \to A$ asszociatív, de nem kommutatív bináris műveletet az $op[1..n][1..n]$ tömb segítségével. Továbbá egy $c[1..n][1..n]$ tömb által minden művelethez rendelünk egy súlyt. A $c[i][j]$ tömbelem az ioj művelet költségét tárolja. Adva lévén az $x_1, x_2, \ldots, x_p$ értékek ($x_i \in A$), határozzuk az $x_1 o x_2 o \ldots o x_p$ műveletsor egy olyan zárójelezését, amely nyomán a műveletsor minimális költséggel végezhető el.

7.4. Számfeldarabolás: Adott n és k, valamint egy legalább n és legfennebb $n \cdot k$ számjegyű természetes szám. Daraboljuk fel a számot n darab legalább egy és legfennebb k számjegyű szakaszra úgy, hogy a szakaszok képezte számok összege maximális legyen.

7.5. Sáska: Adott egy $n \times m$ méretű mátrix, amelyen – a bal felső elemtől a jobb alsó elemig – egy sáska a következőképpen halad át: *ugrik* egyet vízszintesen (bármelyik elemhez az aktuális sorból), majd *lép* egyet függőlegesen (az aktuális elem alatti elemhez), és így tovább. Ha úgy fogjuk fel a mátrix elemeit, mint táplálékmennyiségeket, melyik a sáska legtáplálóbb útvonala?

7.6. Szövegtördelés: Legyen egy n szavas szöveg, amelyet m karakter hosszú sorokba szeretnénk tördelni úgy, hogy a szavak nem választhatók el, és minden két szó közé (amelyek ugyanabban a sorban egymást követik) pontosan egy szóköz kell kerüljön. Jelölje $s_1, s_2, \ldots, s_p$ az egyes sorok végén maradt szóközök számát (kivéve az utolsó sort). Adva lévén a szöveg és az m értéke, határozzuk meg azt a tördelést, amely esetén az $s_1^3 + s_2^3 + \ldots + s_p^3$ összeg minimális.

7.7. Közös ős: Nevezzük egy szám őseinek azokat a számokat, amelyeket úgy nyerünk belőle, hogy bizonyos számjegyeit töröljük (például 75 őse 7145-nek). Határozzuk meg két nagy természetes szám legnagyobb értékű közös ősét.

7.8. Lépcső: Egy lépcsőnek n foka van. k lépcsőfokon egy-egy palackban víz és p lépcsőfokon egy-egy palckban energizáló ital van. Ismert, mely lépcsőfokokon mennyi víz, illetve mely lépcsőfokokon mennyi energizáló ital van. A víz ingyen van, de az energizáló ital pénzbe kerül (decilitere q euro). G úr fel szeretne jutni a lépcsőn minimális számú lépésből. Normális körülmények között egy lépésből egy lépcsőfokhoz van ereje, de ha az aktuális fokon megiszik x deciliter vizet, akkor képes *egy* x fokos lépésre. Sőt, x deciliter energizáló ital *egy* $2 \cdot x$ fokos lépést tesz lehetővé. Ha egy lépcsőfokon víz is és energizáló ital is lenne, nincs értelme mindkettőt felhajtani, mert hatásuk nem adódik össze. Melyik a minimális lépésszámú legolcsóbb feljutási lehetőség a lépcsőn?

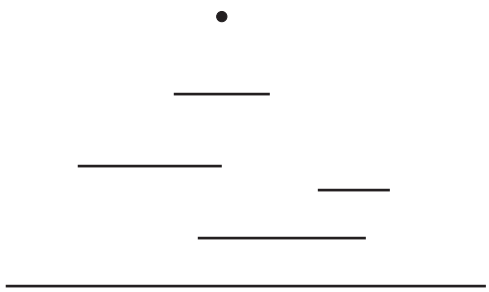
7.9. Rendőrök: Két rendőr, akik egyben barátok is, T időegységet vannak szolgálatban. Ez időszak alatt n szabálytalanság történik. Adott

minden egyes törvénysértés időpontja és típusa. Háromféle szabálytalanság létezik, 1-es, 2-es és 3-as típusú, és ismert az ezekért járó büntetések értéke, valamint az, hogy hány időegységet igényel az alkalmazásuk. Az 1-es és 2-es típusú törvénysértéseket kezelheti egy-egy rendőr is, de a 3-as típusúak mindkét rendőrt igénybe veszik. Mely szabálytalanságokat kezeljék a rendőrök, hogy „összbevételük" maximális legyen?

7.10. Teljes feldarabolás: Legyen egy $n \times m$ méretű mátrix, amelyet vízszintes, illetve függőleges irányú teljes vágások (az elvágott darab ketté esik) sorozata révén elemeire szeretnénk darabolni. Egy vágás költségét a vele szomszédos elemek maximumaként definiáljuk. Határozzuk meg a minimális költségű feldarabolás összköltségét.

7.11. Kódolás: Egy karakterláncot, amely csak az angol ábécé kisbetűit tartalmazza, úgy lehet kódolni, hogy az egymás után következő ismétlődő részkarakterláncokat helyettesítjük magával a részkarakterlánccal és az ismétlődése számával. Határozzuk meg az optimális kódolást, amely a legrövidebb kódot eredményezi.
Példa: `"aaacaaacbbdefdef"` → `"aaac2b2def2"`

7.12. Szabadesés: Képzeljük el az alábbi szerkezetet:



7.42. ábra.

Ismertek az n vízszintes platform magasságai, valamint két végük x koordinátái (a padló 0 magasságban található). Adott továbbá egy pontszerű labda kezdeti pozíciója, valamint az, hogy legfennebb mennyit eshet folyamatosan. A labda szabadon esik 1 m/s sebességgel, és ha ráesik valamelyik vízszintes platformra, elgurul valamelyik irányba ugyancsak 1 m/s sebességgel. Ha a labda valamelyik platformnak éppen a végére

esik, úgy tekintjük, hogy ráesett. A platformok elhanyagolható vastagságúak és nincsenek közös pontjaik. Minden hosszmérték méterben van kifejezve és ezek egész számok.

A labda a lehető legrövidebb idő alatt ért padlót. Mely platformokat érintette, hányadik időpillanatokban, és melyik irányba gurult el az egyes platformokon?

7.13. Jó bor: Egy borospince fölé épült üzletbe n kliens érkezik, akik egymást követve $x_1, x_2, \ldots, x_n$ liter bort szeretnének vásárolni. Az eladó a következőképpen szolgálja ki őket:

- Ha nem áll rendelkezésre fent az üzletben az igényelt teljes mennyiség, akkor visszautasítja a vevőt.
- Ha rendelkezik a kért bormennyiséggel fent az üzletben, vagy kiszolgálja a vevőt, vagy elküldi.
- Valahányszor elküld egy klienst, a pincében található végtelen nagy hordóból pontosan annyi bort hoz fel, amennyit az elküldött kliens igényelt.

Mely vevőket szolgálta ki az eladó, ha tudjuk, hogy sikerült a lehető legtöbb bort eladnia? Kezdetben egy csepp bor sincs fent az üzletben.

7.14. Összes legrövidebb út: Adott egy élsúlyozott irányított gráf. Határozzuk meg bármely két csomópontja között a legrövidebb utat.

7.15. Ember a labirintusban: Adott egy labirintus egy bináris mátrix által, valamint ismert egy ember pozíciója a labirintusban. Határozzuk meg a legrövidebb utat, amelyen kijuthat az ember a labirintusból.

7.16. Mozaik: Egy $a[1..n][1..m]$ egész elemű tömböt úgy fogunk fel, mint egy mozaikképet. Az $a[i][j]$ tömbelem a kép (i, j) pozíciójú képeleme színkódját tárolja $(0..255)$. A képet egy golyósorozat éri, és darabokra törik az alábbi szabályok szerint:

- A golyók a képet (később a képdarabokat) a képelemek sarkainál találják el.
- Minden lövés nyomán az eltalált képdarab tovább darabolódik, ugyanis elhasad a találati ponton áthaladó vízszintes és függőleges egyenesek mentén.

Végül azt találjuk, hogy mindenik képdarab a középpontjára nézve szimmetrikus színösszetételű (a középpontra szimmetrikus pozíciójú képelemek azonos színűek).

Legkevesebb hány golyó érte a képet, és mennyi darabra esett szét? Több megoldás esetén egy olyat írjunk ki, amely esetén minimális a szimmetrikus darabok száma.

(Ezen utolsó feladatot az ACM feladatokra jellemző stílusban szövegeztük meg.)

7.17. Gazdálkodó informatikus: Egy nyugalmazott infótanár úgy dönt, hogy kiköltözik a szüleitől örökölt birtokára és gazdálkodó lesz. A birtok része egy téglalap alakú ($N \times M$ egység méretű) kaszáló is, amelyet el szeretne osztani a lehető legigazságosabban k fia között. Minden fiúnak egy téglalap alakú (egész számú hosszúságú, illetve szélességű) részt szán, minthogy ő is ilyen alakút örökölt. Mivel nem egyenletes a fűhozam, mint megrögzött infós, úgy fogja fel a kaszálót, mint egy $a[1..N][1..M]$ pozitív egész elemű mátrixot, amelynek az $a[i][j]$ eleme a kaszálló (i, j) pozíciójában található egységnyi területen nőtt fűmennyiséget tárolja. Az apa arra is odafigyel, hogy mindenik fia örökségének legyen kijárata a kaszáló szélére (amikor elmennek a bugyláik után, ne kelljen áthajtsanak a szekérrel egymás területén). Jelöljük S-sel a széna összmennyiségét, s_f-fel azt, amennyit az egyes fiúk kapnak ($f = 1..k$). Bár tudjuk, hogy S osztható k-val, a földterület nem biztos, hogy feldarabolható úgy (a feladat feltételei mellett), hogy mindenik fiú pontosan ugyanannyi mennyiségű szénát kapjon. Határozzuk meg a „lehető legigazságosabb" elosztást, azt, amelyikre a $e = \Sigma|S/k - s_f|$ kifejezés értéke minimális (e_{min}), miközben $f = 1..k$.

A bemeneti állomány T darab tesztadatsort tartalmaz az alábbi szerkezet szerint (az utolsó sort követi egy újsor karakter):

```
<T, a tesztek száma>
<az 1. teszt N, M és k értéke
                              egy-egy szóközzel elválasztva>
<az 1. teszt a mátrixa 1. sorának elemei
                              egy-egy szóközzel elválasztva>
...
<az 1. teszt a mátrixa N. sorának elemei
                              egy-egy szóközzel elválasztva>
...
<a T. teszt N, M és k értéke
                              egy-egy szóközzel elválasztva>
<a T. teszt a mátrixa 1. sorának elemei
                              egy-egy szóközzel elválasztva>
```

```
...
<a T. teszt a mátrixa N. sorának elemei
                                egy-egy szóközzel elválasztva>
```

A kimenetnek az első bemeneti tesztre tartalmaznia kell az S, az S/k és az e_{min} értékeket egy-egy szóközzel elválasztva, majd az ezt követő k sorban egy-egy fiú örökségét (a kapott s_f terület méretét, az átlagtól való eltérés mértékét és a bal felső, illetve a jobb alsó sarok koordinátáit egy-egy szóközzel elválasztva). A következő tesztadatok esetében csak az e_{min} értéket kérjük. Lásd a példát!

Példa bemenet:

```
2
2 2 4
7 7
7 11
2 2 2
7 6
8 9
```

Példa kimenet:

```
32 8 6
7 1 1 1 1 1
7 1 1 2 1 2
7 1 2 1 2 1
11 3 2 2 2 2
0
```

8. FEJEZET

DIVIDE ET IMPERA VAGY DINAMIKUS PROGRAMOZÁS

Különösen a dinamikus programozás „levelek–gyökér irányú” változata (amikor az összevont döntési fa körmentes) mutat hasonlóságot a divide et impera technikával. A két technikát bemutató fejezetek rávilágítottak arra, hogy mindkét stratégia úgy fogja fel a feladatot, mint ami kisebb méretű hasonló részfeladatokra bontható, vagy hogy ezekből épül fel. Ez a szerkezet mindkét esetben fastruktúra. A fa csomópontjai, illetve a hozzájuk tartozó részfák ábrázolják az egyes részfeladatokat. Megtörténhet, hogy a feladat lebontásakor különböző ágakon azonos részfeladatokhoz jutunk, ami azt jelenti, hogy a fának lesznek identikus részfái.

Ezen emlékeztetők után lássuk a hasonlóságokat és a különbségeket.

1. A divide et impera akkor nem hatékony, ha a feladat lebontásakor identikus részfeladatok jelennek meg, hiszen ezeket többször is megoldja. A dinamikus programozás viszont annál hatékonyabb, minél több az azonos részfeladat, mivel képes elkerülni ezek ismételt megoldását. Ez a különbség a stratégiáikból adódik.
2. Egy divide et impera algoritmus először lebontja a feladatot (preorder mélységi bejárás szerint), majd a rekurzió visszaútján posztorder sorrendben megold *minden* részfeladatot, amellyel a lebontás alkalmával találkozott. Mivel minden részfeladat megoldását a közvetlen fiú-részfeladatai megoldásaiból építi fel, ezért csak ezeket tárolja el, és ezeket is csak ideiglenesen (amíg az apacsomópont megoldása megépül). Más szóval, nem vezet nyilvántartást a már megoldott részfeladatokról és azok megoldásairól. Ez a magyarázata annak, hogy az azonos részfeladatokkal – anélkül, hogy tudomása lenne róla – többször is találkozik, újra és újra megoldva őket.

   Ezzel szembe a dinamikus programozás lentről (az egyszerűtől a bonyolult fele haladva) kezd neki a feladatnak, és minden részfeladatot csak egyszer old meg. Nyilvántartást vezet (általában egy tömbben) a már megoldott részfeladatok optimális megoldásairól,

hogy amennyiben valamelyikre szükség lenne a későbbiekben, ne kelljen újra megoldania.

3. Optimalizálási feladatok esetében a dinamikus programozás a részfeladatok optimumértékeiről végzett nyilvántartásából utólag elő tudja állítani magát az optimális megoldást is. Ezzel szemben a divide et impera csak az optimális megoldáshoz tartozó optimumértékkel tud szolgálni.
4. Mi történne, ha a divide et impera kölcsönvenné a dinamikus programozástól a már megoldott részfeladatok nyilvántartásának ötletét? Úgy értjük ezt, hogy amikor először találkozik egy részfeladattal, a dinamikus programozáshoz hasonlóan, tárolja el a megoldását egy tömbbe, hogy valahányszor újra találkozik vele, egyszerűen csak elő kelljen vegye a megoldását. E feljavítás esetén a két technika ugyanolyan komplexitású algoritmust fog nyújtani. Ez esetben a részfeladatok megoldásának lentről felfele iránya a postorder mélységi bejárási sorrendnek megfelelő fordított topologikus sorrend lesz. Ezt a stratégiát inkább a dinamikus programozás rekurzív változatának nevezhetnénk, mint divide et imperának.
5. Egy másik hasonlóság a divide et impera és a dinamikus programozás „levelek–gyökér” változata között, hogy mindkét esetben a gyökérben hirdetünk megoldást: a divide et impera a rekurzióból visszaérkezve ide, a dinamikus programozás iteratívan felérkezve ide. Tehát míg az egyik algoritmus alapvetően rekurzív, a másik alapvetően iteratív.

8.1. Dinamikus programozás rekurzívan

Amint már fentebb is utaltunk rá, e megközelítésnek két fő jellegzetessége van:

1. Rekurzívan használja a lentről felfele irányú építkezés képletét, akárcsak a divide et impera.
2. A már megoldott részfeladatok optimális megoldásait eltárolja, hogy amennyiben később szüksége lenne rájuk, egyszerűen csak elő kelljen vegye azokat (akárcsak a dinamikus programozás).

Mivel a második tulajdonság a meghatározó, ezért e stratégia inkább dinamikus programozás, mint divide et impera. Ennek ellenére jól felhasználható, mint ötvözet, a két technika összehasonlítására.

Az alábbiakban közöljük az n-edik Fibonacci-szám, valamint az N elem K-adrendű kombináció ($K \leqslant N$) száma feladatok rekurzív dinamikus programozásos megoldását. Ezt követően adunk egy rekurzív megoldást a dinamikus programozás bemutatására használt „háromszög" feladatra is.

8.1. Fibonacci-számok: Az `f[0..n]` tömb `f[i]` eleme, az i-edik Fibonacci-szám értékét tárolja. Kezdetben a tömböt -1 értékekkel töltjük fel. Ha egy tömbelem értéke -1, azt jelenti, hogy az illető Fibonacci-szám még nincs meghatározva.

```
f[i] = 0, ha i=0
     = 1, ha i=1
     = f[i-1] + f[i-2], ha i>1

fibonacci(f[],i)
   ha f[i]≠-1 akkor return f[i]
   különben
      ha i==0 vagy i==1 akkor f[i]=i
      különben f[i]=fibonacci(f,i-1)+fibonacci(f,i-2)
      vége ha
      return f[i]
   vége ha
vége fibonacci
```

A függvényt a következő módon hívjuk meg: `fibonacci(f,n)`

8.2. Kombinációk száma: A `c[0..N][0..N]` tömb `c[n][k]` ($0 \leqslant k \leqslant n \leqslant N$) eleme az n elem k-adrendű kombinációi számát tárolja. Kezdetben a tömböt -1 értékekkel töltjük fel. Ha egy tömbelem értéke -1, azt jelenti, hogy az illető kombinációszám még nincs meghatározva.

```
c[n][k] = 0, ha k=0 vagy n=k
        = c[n][k-1]+c[n-1][k-1], ha 0<k<n<=N

c_n_k(c[][],n,k)
   ha c[n][k]≠-1 akkor return c[n][k]
   különben
      ha k==0 vagy n==k akkor c[n][k]=1
      különben c[n][k]=c_n_k(c,n,k-1)+c_n_k(c,n-1,k-1)
      vége ha
      return c[n][k]
```

```
    vége ha
vége c_n_k
```

A függvényt a következőképpen hívjuk meg: `c_n_k(c,N,K)`

8.3. Háromszög: Egy n soros négyzetes mátrix főátlóján és főátló alatti háromszögében természetes számok találhatók. Feltételezzük, hogy a mátrix egy a nevű kétdimenziós tömbben van tárolva. Határozzuk meg azt a „leghosszabb" utat, amely az $a[1][1]$ elemtől indul és az n-edik sorig vezet, figyelembe véve a következőket:

- egy úton az $a[i][j]$ elemet az $a[i+1][j]$ elem (függőlegesen le) vagy az $a[i+1][j+1]$ elem (átlósan jobbra le) követheti, ahol $1 \leqslant i < n$ és $1 \leqslant j < n$;
- egy út „hossza" alatt az út mentén található elemek összegét értjük.

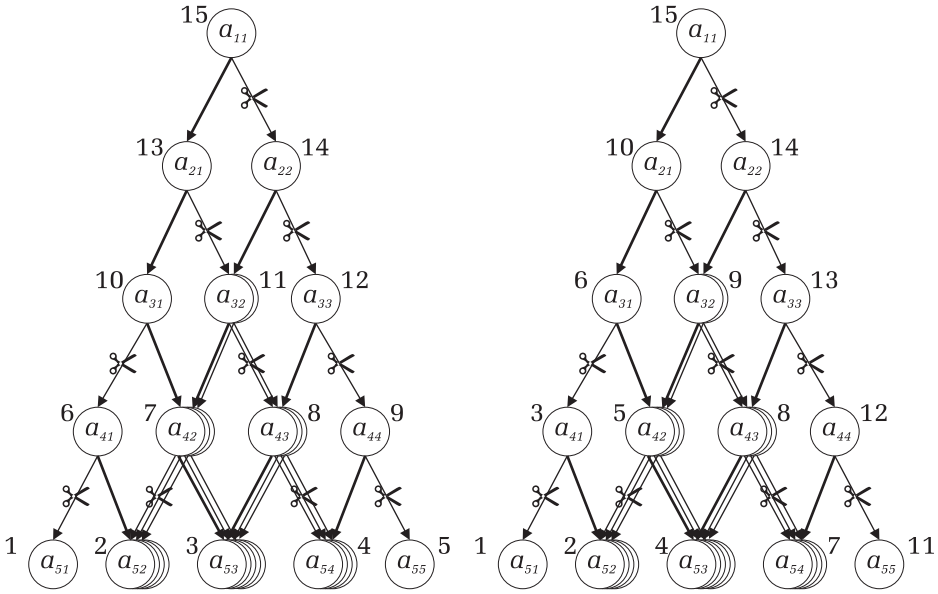
A háromszög függvényben ugyanazokat a jelöléseket használtuk, mint a dinamikus programozást bemutató fejezetben.

```
háromszög(c[][],a[][],n,i,j)
   ha c[i][j]≠-1 akkor return c[i][j]
   különben
      ha i==n akkor c[i][j]=a[i][j]
      különben
         c[i][j]=a[i][j]+max(háromszög(c,a,n,i+1,j),
                                 háromszög(c,a,n,i+1,j+1))
      vége ha
      return c[i][j]
   vége ha
vége háromszög
```

A függvény meghívása: `háromszög(c,a,n,1,1)`

Mellékelve bemutatjuk a „háromszög feladat" összevont döntési fáján, hogy milyen sorrendben oldja meg a részfeladatokat a dinamikus programozás klasszikus levelek–gyökér irányú változata (bal oldali ábra), illetve a rekurzív megvalósítás (jobb oldali ábra). Természetesen mindkét sorrend egy-egy fordított topologikus sorrend (8.1. ábra).

8.4. Egér–sajt-feladat: Egy labirintusban, amelyet egy $n \times m$ ($n \leqslant 50$, $m \leqslant 50$) méretű, a nevű bináris mátrix ábrázol (a 0 érték utat jelöl, az 1-es falat), ismert egy egér (xe, ye) és egy darab sajt (xs, ys) pozíciója. Határozzuk meg az egértől a sajthoz vezető legrövidebb utat (a labirintusban négy irányban lehet közlekedni).



8.1. ábra.

Példa:

	1	2	3	4	5	6
1	**e**	0	0	0	0	0
2	0	1	1	1	0	0
3	0	1	0	0	0	1
4	0	0	0	0	1	**s**

8.2. ábra.

E feladat divide et impera megoldását már kifejtettük a 4. fejezetben, ahol a divide et impera módszert a backtrackinggel hasonlítottuk össze. Ott azt is tisztáztuk, hogy miért alkalmas a divide et impera csak a legrövidebb út *hosszának* meghatározására. Ugyancsak a 4. fejezetben utaltunk rá, hogy az egér–sajt-feladat hatékony megoldása dinamikus programozással történik.

A feladat mögött meghúzódó összevont döntési fa, mint irányított gráf, számtalan kört tartalmaz, de nem tartalmaz negatív súlyú élet (minden él súlya 1). Ezért e feladatot a dinamikus programozás Dijkstra-algoritmus alapú változata oldja meg, amely alapvetően gyökér–levelek

irányú típusú (az általános feladat az (xe, ye) pozíciótól az (i, j) pozícióig vezető legrövidebb út meghatározása). A divide et impera stratégia viszont a levelek–gyökér irányú dinamikus programozáshoz áll közelebb. Milyen akadályba ütköznénk, ha levelek–gyökér irányú stratégiával próbálnánk megközelíteni az egér–sajt-feladatot?

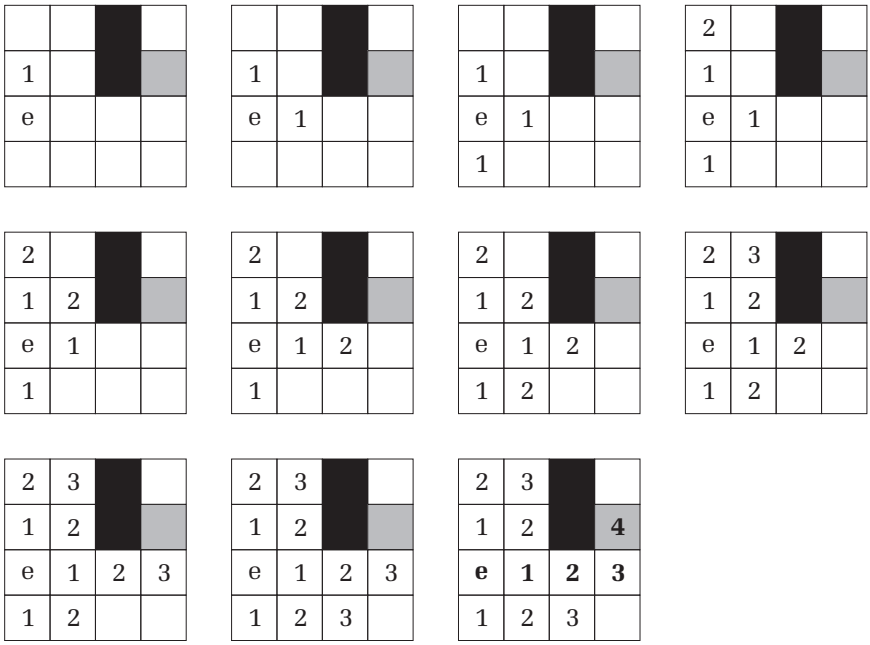
Ez esetben az általános feladat az (i, j) pozícióból az (xs,ys) pozícióba vezető legrövidebb út meghatározásának problémája kellene hogy legyen, ami azt jelenti látszólag, hogy két független paramétere van az általános alaknak. Csakhogy van egy bökkenő. Az egérnek először el kell jutnia az (i, j) pozícióba, és csak azután beszélhetünk az innen a sajthoz vezető legrövidebb útról. Továbbá az egér – hogy elkerülje a hurkokat – be kell hogy falazza a maga mögött hagyott útszakaszt, ami a labirintus ideiglenes átkonfigurálását jelenti. Mivel az (i, j) pozícióba az egér különböző utakon is eljuthat, az (i, j) pozíciótól a sajthoz vezető legrövidebb út keresése más-más konfigurációjú labirintusban történik. Ezzel magyarázható, hogy a 4. fejezetben felvázolt döntési fa identikus csomópontjaihoz különböző részfák tartoznak. Tehát az i és j paraméterek nem azonosítják egyértelműen az (i, j) pozíciótól a sajthoz vezető legrövidebb út problémáját.

Tekintettel arra, hogy a feladat mögött meghúzódó összevont döntési fának, mint irányított gráfnak, az élei mind egységnyi hosszúak, a leghatékonyabb algoritmus egy egyszerű szélességi bejáráson alapszik, amely Lee-algoritmusként ismert, és a Dijkstra-algoritmus alapú dinamikus programozás egy egyszerűsített változata.

Az alábbi példára végigvisszük az algoritmust lépésről lépésre a c tömbön. Az egeret e-vel, a sajtot szürkével és a falat feketével jelöltük (8.3. ábra). Egy sorszerkezetet (Q) használunk, mint ahogyan azt már a bevezető fejezetben bemutattuk a fastruktúrák szélességi bejárásánál. A sor elemei az egyes cellák indexpárjait tartalmazzák. A sornak mindig az elejéről veszünk ki (`kivesz_sorból` eljárás) és a végére szúrunk be (`betesz_sorba` eljárás). Mivel minden él hossza 1, nincs szükség elsőbbségi sorra. A `sor_első` eljárás a második, illetve harmadik cím szerinti paramétereiben visszaadja a sorelső elem indexeit. Az `üres` függvény azt ellenőrzi, hogy kiürült-e a sor, az `első_e` függvény pedig azt, hogy a paraméterként kapott tömbindexek a sorelső elem indexei-e.

```
minden i=1,n végezd
   minden j=1,m végezd
      ha a[i][j]==0 akkor c[i][j]=-1
      vége ha
```

```
   vége minden
vége minden
c[xe][ye]=0

betesz_sorba (Q, xe, ye)
amíg nem üres (Q) és nem első_e (Q, xs, ys) végezd
   sor_első (Q, i_e, j_e)
   kivesz_sorból (Q)
   minden (i,j) szomszédjára (i_e,j_e)-nek végezd
      ha a[i][j]==0 és c[i][j]==-1 akkor
         c[i][j]=c[i_e][j_e]+1
         betesz_sorba (Q, i, j)
      vége ha
   vége minden
vége amíg

ha üres (Q) akkor ki: "Nincs megoldás"
különben ki: c[xs][ys]
vége ha
```



8.3. ábra.

Ha az **amíg** ciklus azért ér véget, mert kiürült a sor, akkor nem lehet az egértől a sajtig eljutni. Különben, amelyik pillanatban először elérjük a sajtot, meg is van a legrövidebb út. Ha szeretnénk magát az utat is, akkor ez már egyértelmű a c tömb alapján a sajttól visszafele.

Ahogy a szélességi bejárás mindig a legrövidebb úton éri el egy gráf csomópontjait, úgy bizonyítható, hogy a fenti algoritmus is a legrövidebb egér–sajt utat találja meg.

9. FEJEZET

MOHÓN VAGY DINAMIKUSAN?

Ez a fejezet a mohó és a dinamikus programozási technikákat hasonlítja össze. Ha tisztán látjuk a két stratégia közötti alapvető hasonlóságokat és különbségeket, akkor ez segíteni fog abban, hogy felismerjük, mikor célszerű alkalmazni őket, és el fogjuk kerülni az alábbi tévedéseket is:

- Dinamikus programozást alkalmazunk, bár a mohó megközelítés is kielégítő lenne.
- Mohó algoritmust használunk ott, ahol dinamikus programozásra lenne szükség.

A mohó és dinamikus programozási stratégiák váll váll mellett:

1. Általában mindkét technikát optimalizálási feladatok megoldására használjuk.
2. Mind a mohó, mind a dinamikus programozási stratégia esetében a megoldást egy optimális döntéssorozat jelenti.
3. Míg az első technika egyetlen döntéssorozatot állít elő (bízva abban, hogy ez lesz az optimális), addig a második több (optimális) részdöntéssorozatot is generál, amelyekből majd felépíti az eredeti feladatot megoldó (optimális) döntéssorozatot. Ez a különbség abból adódik, hogy a mohó algoritmus mohó döntések sorozata által, a dinamikus programozás pedig az optimalitás alapelve szerint építkezve oldja meg a feladatot.
4. A fenti megállapítással összhangban, a mohó stratégia fentről lefele, a dinamikus programozás pedig lentről felfele oldja meg a feladatot. A mohó algoritmusok mindig a döntési fa gyökerétől a levelei fele haladnak, és minden mohó választással a feladatot kisebb méretű hasonló feladattá redukálják, míg triviálissá nem válik. Amint láthattuk, a dinamikus programozás esetében a lentről felfele jelenthet mind levelek–gyökér, mind gyökér–levelek irányt. Az első esetben a triviális (szúfix) részfeladatokat ábrázoló levelektől indulva, felépítettük az egyre bonyolultabb (szúfix) részfeladatok optimális megoldásait, végül pedig – felérkezve a gyökérbe – az eredeti feladatnak mint legnagyobb (szúfix)

feladatnak az optimális megoldását. A második esetben a gyökér képviselte kezdeti állapothoz tartozó triviális (prefix) részfeladatból indultunk. Nem rendelkezvén kellő információval ahhoz, hogy mohó döntést hozzunk, regisztráltuk a terebélyesedő fa koronáján megjelenő összes – egymástól különböző – csomóponthoz (amelyek egymástól különböző állapotokat képviseltek) vezető optimális döntéssorozatot mint az illető csomóponthoz tartozó (prefix) részfeladat optimális megoldását. Egyre több és egyre bonyolultabb (prefix) részfeladatot oldva meg, végül „felérkeztünk" a döntési fa leveleibe. Miután kiválasztottuk az optimális levelet, az ide vezető gyökér–levél út képviselte az eredeti feladatnak mint legnagyobb (prefix) részfeladatnak az optimális megoldását.

5. A két technika alkalmazása más-más komplexitású algoritmust eredményez. Tegyük fel, hogy az illető feladathoz rendelhető fa magassága n. Döntési fáról lévén szó, a fa összcsomópontjainak száma nyilván exponenciálisan függ n-től. A mohó-stratégia alkalmazása lineáris algoritmust ($O(n)$) eredményez, hiszen egyetlen gyökér–levél utat jár be. Bár a dinamikus programozás általában nem tudja elérni ezt a komplexitást, ha az egymástól különböző részfeladatok száma polinom függvény szerint függ n-től, akkor algoritmusa polinomiális lesz.
   Megjegyzés: Itt a stratégiából adódó komplexitást vizsgáltuk. Ez a komplexitás nőhet még, attól függően, hogy az egyes döntések meghozatala milyen komplexitású plusz feladattal jár.
6. Mindkét technika valamilyen mértékben az optimalitás alapelvére támaszkodik. A dinamikus programozás algoritmusok teljesen erre az alapelvre épülnek. A mohó technika esetében viszont csak szükséges feltétele annak, hogy a feladat megoldható legyen mohó döntéssorozat által. A mohó algoritmusok a mohó-választás alapelvére épülnek. Tehát míg a dinamikus programozás teljesen kihasználja az optimalitás alapelvét, a mohó-technika csak részlegesen (a módszer helyességének bizonyításában).

Az alábbiakban a közismert „hátizsák feladat" segítségével állítjuk egymás mellé a két technikát. (Már találkoztunk ezzel a feladattal abban a fejezetben, amely azt fejtegette, hogy miként képesek kiegészíteni egymást a backtracking és greedy technikák).

9.1. Hátizsák: Egy üzletben n tárgy (áru) található, és ezek mindenikének ismert az ára ($a[1..n]$) és a súlya ($g[1..n]$, minden $g[i]$ ($i = 1, n$)

természetes szám). Mely tárgyakat vigye magával a tolvaj, hogy a lehető legnagyobb nyereséggel távozzon? (A hátizsákja legtöbb G súlyt bír meg.)

A feladat két változatban ismert:

a) a tárgyak elvághatók,

b) a tárgyak nem vághatók el.

Amint látni fogjuk, a feladatra – mindkét változatában – érvényes az optimalitás alapelve. Ezzel szemben a mohó-választás alapelve csak az első estben igaz. Tehát a) változatában mohó-technikával, b) változatában pedig dinamikus programozással fogjuk megoldani a feladatot.

Az optimalitás alapelvének ellenőrzése

a) Tekintsük a tárgyakat az $a[i]/g[i]$ hányados szerint csökkenő sorrendben. Ez a mohó-választási sorrend. Tegyük fel, hogy a rendezett sor első m tárgya fér bele a hátizsákba: az első $(m-1)$ tárgy teljes mértékben, az m-ediknek pedig egy bizonyos százaléka. Legyen ezen optimális megoldásnak a kódja $(k_1, k_2, \ldots, k_{m-1}, k_m, k_{m+1}, \ldots, k_n)$, ahol $k_i = 1$ minden $i = 1, m-1$, $0 < k_m <= 1$, $k_i = 0$ minden $i = m+1, n$, és $\Sigma(k_i \cdot g[i]) = G$. Az első mohó döntés nyomán (a sor elején lévő, legnagyobb ár per súly hányadosú tárgynak a hátizsákba való helyezése) a feladat a $G - g[1]$ méretű hátizsáknak a 2., 3., ..., n. tárgyak felhasználásával való optimális megtöltése feladatává redukálódik. Elmondható, hogy ennek a feladatnak a $(k_2, \ldots, k_{m-1}, k_m, k_m, k_{m+1}, \ldots, k_n)$ kódú megoldás jelenti az optimális megoldását? Minden bizonnyal, hiszen ha lenne erre a részfeladatra egy jobb megoldás, akkor mellé véve az első tárgyat, az eredeti feladatra is jobb megoldást kapnánk, mint amelyikből kiindultunk. Ez viszont ellentmondana a feltételnek, miszerint optimális megoldásból indultunk ki. Tehát a feladat optimális megoldása a részfeladatok optimális megoldásaiból épül fel.

b) Ebben az esetben a tárgyak lehetnek bármilyen sorrendben. Legyen ezen optimális megoldásnak a kódja $(k_1, k_2, \ldots, k_n)$, ahol k_i $(i = 1, n)$ egyenlő 1-gyel vagy 0-val, attól függően, hogy az illető tárgyat bele tettük vagy sem a hátizsákba. Ez alkalommal a hátizsák telítettségét leíró összefüggés a $\Sigma(k_i \cdot g[i]) <= G$ alakban áll fenn (a hátizsák nem biztos, hogy pontosan megtelik). Az első tárgy felől való döntés nyomán a következő részfeladatokká redukálódhat a feladat:

– a $G - g[1]$ méretű hátizsák optimális megtöltése 2., 3., ..., n. tárgyak segítségével (beletettük az első tárgyat a hátizsákba);

– a G méretű hátizsák optimális megtöltése 2., 3., ..., n. tárgyak segítségével (nem tettük bele az első tárgyat a hátizsákba).

Mindkét esetben a részfeladatok optimális megoldásának föltétlenül a $(k_2, k_3, \ldots, k_n)$ kódú megoldásnak kell lennie. Tegyük fel például, hogy az első esetben a $G - g[1]$ méretű hátizsák megtöltésére létezik egy, az előbbinél jobb megoldás, a $(p_2, p_3, \ldots, p_n)$ kódú ($\Sigma(p_i \cdot g[i]) <= G - g[1]$, $i = 2, n$). Ha azonban ez így lenne, akkor az $(1, p_2, p_3, \ldots, p_n)$ kódú megoldás az eredeti feladatra nagyobb nyereséget biztosítana, mint amiből kiindultunk. Ez viszont ellentmond annak, hogy a $(k_1, k_2, \ldots, k_n)$ kódú megoldás optimálisan oldja meg a feladatot. A második esetben még nyilvánvalóbb a bizonyítás. Következésképpen a feladat b) változata esetén is elmondható, hogy a részfeladatok optimális megoldásaiból felépíthető az eredeti feladat optimális megoldása.

A mohó-választás alapelvének ellenőrzése

a) Be fogjuk bizonyítani, hogy a mohó-választásból adódó tárgy (ha a tárgyak rendezve vannak az $a[i]/g[i]$ hányados szerint csökkenő sorrendben, akkor ez az első tárgy) mindenképpen részét képezi az optimális megoldásnak, tehát az optimális megoldás föltétlenül vele kezdődik. Tegyük fel, hogy létezik olyan optimális megoldás, amely nem tartalmazza a legnagyobb ár per súly hányadosú tárgyat. Vegyünk ki a hátizsákban levő tárgyakból $g[1]$ súly mértékben (ezt meg tudjuk tenni, mert a tárgyak elvághatók), és helyettesítsük az 1. tárggyal. Mivel az első tárgyból a legértékesebb egy egységnyi súlyú mennyiség, az előbbi cserével egy jobb megoldáshoz jutottunk, mint amiből kiindultunk. Ez viszont ellentmond a feltételnek.

b) Egy ellenpéldával bizonyítjuk, hogy létezik olyan eset, amikor az optimális megoldás nem tartalmazza a mohó-választásból adódó tárgyat.

Legyen $n = 3$, $g[1..3] = (10, 20, 30)$, $a[1..3] = (60, 100, 120)$ és $G = 50$. Az ár per súly hányados szerint a legígéretesebb tárgy az első. Ezzel szemben az optimális megoldás a $(0, 1, 1)$ kódú, az, amikor a 2. és 3. tárgyakat tesszük a hátizsákba.

A mohó döntés alapelve azért nem érvényes ez esetben, mert a mohóválasztás odavezethet, hogy nem tudjuk teljesen megtölteni a hátizsákot (amikor a tárgyak elvághatók, ez nem áll fenn). Ez olyan, mintha arra kényszerülnénk, hogy egy bizonyos mennyiségű nullaértékű tárgyat

(levegőt) tegyünk a hátizsákba. Lokálisan döntünk olyan kérdésben, amelynek hosszú távú következménye lehet.

Mohó-algoritmus az a) esetre

Feltételezzük, hogy a tárgyak már rendezve vannak ár per súly szerint csökkenő sorrendben. A $k[1..n]$ tömbben fogjuk kialakítani a megoldás kódját.

```
minden i = 1,n végezd
   k[i] = 0
vége minden
s = 0 // a hátizsákba került tárgyak összsúlya
e = 0 // a bepakolt áruk összára
i = 1
// az előrendezésből adódóan mohó döntésről
// mohó döntésre haladunk
amíg (s+g[i] <= G) végezd
   // a következő tárgy teljesen belefér a zsákba
   s += g[i]
   e += a[i]
   k[i] = 1
   i += 1
vége amíg
ha s<G akkor
   k[i] = (G-s)/g[i]
   e += a[i]*k[i]
vége ha
ki: e, k[1..n]
```

Dinamikus programozás algoritmus a b) esetre

A feladat nagymértékben hasonlít a könyvespolc feladatra, amit részletesen tárgyaltunk a dinamikus programozást bemutató fejezetben. Ott megemlítettünk egy második megközelítési módot is a feladatra. Itt ezt fogjuk alkalmazni, mert talán jobban talál ezen fejezet témájához.

Részfeladatnak tekinthetjük egy s hátizsákméret optimális megpakolását vagy kirakását ($0 <= s <= G$). Összhangban azzal, hogy a dinamikus programozás az egyszerűtől halad a bonyolult fele, megpróbáljuk sorba kirakni (optimálisan) a $0, 1, 2, \ldots, G$ hátizsákméreteket. Persze egyáltalán nem biztos, hogy a teljes értéksor kirakható (a tárgyak

halmazának nincs egyetlen olyan részhalmaza sem, amelyre a súlyok összege az illető hátizsákméret legyen), sőt, amint láthattuk, lehet, hogy maga a G hátizsákméret sem rakható ki. Ilyenkor a feladat megoldása a G-hez legközelebbi kirakott érték lesz. Tegyük fel, hogy az s hátizsákméretet szeretnénk kirakni. Ekkorra már nyilván kiraktuk a $0, 1, \ldots, s-1$ értékeket (amennyiben lehetséges volt). Mikor és hogyan építhető fel ezen optimális részmegoldásokból az s-méretű feladat optimális megoldása? Nyilván ennek az a feltétele, hogy valamely $i = 1, n$ esetén az $s-g[i]$ hátizsákméretet ki tudtuk volt rakni az i-edik tárgy felhasználása nélkül (az optimális megoldásban). Ha így áll a dolog, akkor az $s-g[i]$ optimális kirakásához hozzávéve az i-edik tárgyat, találtunk egy kirakási módot az s hátizsákméretre. Ez nem föltétlenül az optimális kirakás, hiszen létezhet jobb is. Ha több kirakási mód is van egy adott s értékre, akkor nyilván az az optimális, amelyik a legtöbb nyereséget hozza a tolvajnak. Vegyük figyelembe, hogy az optimalitás alapelve nem azt mondja ki, hogy minden optimális részmegoldásra épülő megoldás ugyancsak optimális, hanem hogy az optimális megoldás *valamelyik* optimális részmegoldásra épül. Ez viszont elég, hogy mindig csak az optimális megoldást kelljen megőriznünk, hiszen szavatolja, hogy az optimális megoldás felépíthető kizárólag az optimális részmegoldásokból.

Legyenek a $c[0..G]$ és $d[0..G][1..n]$ tömbök azok, amelyekben eltároljuk az egyes hátizsákméretek optimális kirakásait. A $c[s]$ elem az s érték optimális kirakásából adódó nyereséget, a d tömb s-edik sora pedig az optimális kirakásban felhasznált tárgyakat tárolja, ($d[s][i]$ érték attól függően lesz 1 vagy 0, hogy használtuk-e az i-edik tárgyat). Ha egy hátizsákméret még nincs kirakva, vagy nem kirakható, akkor ezt (-1)-gyel jelezzük a c tömb megfelelő elemében. Íme az algoritmus (feltételezzük, hogy a d tömb elemei kezdetben le vannak nullázva, a c tömb elemei pedig (-1)-gyel vannak feltöltve):

```
c[0]=0
minden s=1,G végezd
   minden i=1,n végezd
      ha s>=g[i] és c[s-g[i]]≠-1
         és d[s-g[i]][i]==0 és c[s-g[i]]+a[i]>c[s]
      akkor
         c[s]=c[s-g[i]]+a[i]
         k=i
      vége ha
   vége minden
```

```
      ha c[s]≠-1 akkor
         d[s][1..n]=d[s-g[k]][1..n]
         d[s][k]=1
      vége ha
      // az s-g[k] hátizsákméret optimális
      // megoldásából építjük fel az s
      // hátizsákméret optimális megoldását
   vége minden
   s=G
   amíg c[s]==-1 végezd
      s-=1
   vége amíg
   ki: c[s], d[s][1..n]
```

A k változóban őrizzük meg annak a tárgynak a sorszámát, amelyikre az aktuális hátizsákméret optimális megoldását kapjuk.

Összehasonlítva a két algoritmust, láthatjuk, hogy az első mohó döntésről mohó döntésre haladva építi fel az optimális megoldást, a második pedig a részfeladatok optimális megoldásaiból építkezik.

9.1. Visszapillantás az eddig bemutatott négy technikára

Tekintsük újra a pénzérmés feladatot, és használjuk fel arra, hogy felelevenítsük elménkben az első kilenc fejezet néhány gondolatát. Ezen eszmefuttatás előkészítheti a talajt a 10. fejezethez, amelynek be kell majd építenie a branch and bound technikát az előbbiekről kialakult képbe.

9.2. Pénzösszeg: Adott egy $S >= 1$ összeg és különböző értékű pénzérmék. A pénzérmék értékeit egy $a[1..n]$ tömbben tároljuk (minden pénzérmefajtából rendelkezésre áll akármennyi). Fizessük ki az S összeget, minimális számú pénzérmét felhasználva.

Gondolkodjunk el az alábbiakon:

- Mi annak szükséges és elégséges feltétele, hogy a feladatnak legyen megoldása minden $S >= 1$ esetén? Bizonyítsuk be válaszunkat!
- Mi lenne egy *backtracking* stratégia a feladatra? Implementáljuk az algoritmust!

- Az S összeg kifizetése visszavezethető az i és $S-i$ összegek kifizetésére, ahol $i = 1 \ldots S-1$. Építsünk fel egy *divide et impera* algoritmust erre az észrevételre!
- Mutassuk ki, hogy amennyiben a pénzérméink értékei k^0, k^1, k^2, $\ldots$, k^{n-1}, ($k \in \mathbb{N}$, $k > 1$) alakúak, akkor alkalmazható a mohóstratégia a feladatra! Írjuk meg a *greedy* algoritmust!
- Bizonyítsuk be, hogy a feladatra érvényes az optimalítás alapelve! Implementáljuk a *dinamikus programozás* algoritmust, miszerint sorra meghatározzuk az $1, 2, 3, \ldots, S-1$ és végül az S összeg optimális kifizetési módjait.
- Mekkora lesz a bonyolultsága a négy algoritmusnak?

Végkövetkeztetésként elmondaható, hogy mind a hátizsák, mind a pénzösszeg feladat esetében az alapvető kérdés nem más, mint: *mohón vagy dinamikusan?*

10. FEJEZET

A BRANCH AND BOUND STRATÉGIA BEÁGYAZVA A TECHNIKÁKRÓL KÖRVONALAZOTT KÉPBE

A branch and bound technika sokkal összetettebb, hogy itt a teljesség igényével bemutathatnánk. Célunk csak ízelítőt adni belőle, egy jellegzetesen branch and bound, az úgynevezett A^* algoritmus révén.

Foglaljuk össze az eddigiekben bemutatott négy technika stratégiáját a hatáskörükbe tartozó feladatokhoz rendelhető faszerkezeten, és az így kapott képbe ágyazzuk bele a branch and bound stratégiát.

A backtracking és divide et impera technikák mélységében járják be a „feladat fáját". A backtracking preorder sorrendbe építi be a csomópontokat a megoldásba, a divide et impera pedig postorder sorrendben oldja meg a csomópontokhoz tartozó részfeladatokat.

A mohó stratégia a döntési fa egyetlen gyökér–levél útján „szalad" le, mélységében.

Milyen sorrendben oldja meg az összevont döntési fa csomópontjaihoz tartozó részfeladatokat a dinamikus programozás? (Főleg a gyökér–levelek irányú változatát vizsgáljuk most.) Az alapelv az, hogy az egyszerűtől halad a bonyolult fele. Még konkrétabb sorrendet szab a rekurzív képlet, amely matematikailag leírja, hogy miként épülnek fel az egyszerűbb részfeladatok optimális megoldásaiból az egyre bonyolultabb részfeladatok optimális megoldásai. Láthattuk, hogy körmentes összevont döntési fa esetén ez a sorrend biztosítható egy topologikus sorrend szerinti bejárással. Ha az összevont döntési fa, mint súlyozott irányított gráf, tartalmazott kört (de nem tartalmazott negatív élet), akkor elsőbbségi sor segítségével (Dijkstra-algoritmus) határoztuk meg a részfeladatok megoldásának helyes sorrendjét. Ez utóbbi stratégia a szélességi bejáráshoz áll közelebb (például ugyanúgy sorszerkezetet használ), bár a klasszikus szélességi bejárás úgy tekinti a gráfot mint súlyozatlant (vagy mintha az élei mind egysúlyúak lennének). Egy másik lényeges különbség az, hogy míg a klasszikus szélességi bejárás egyből a legrövidebb úton (az élek számát tekintve) éri el az egyes csomópontokat,

a dinamikus programozás e változata általában többszöri finomítás révén állítja elő az optimális utakat.

A következő feladat, amelyet 1990-ben adtak a Nemzetközi Informatika Olimpiászon, egy olyan helyzetet mutat be, amelyben egyik bemutatott technika sem igazán hatékony.

10.1. Tili-toli: Adott egy 4×4 méretű mátrix, amely 0-tól 15-ig tartalmazza a természetes számokat (lásd lennebb). A nulla pozíciója üres helynek számít. Egy lépésen azt értjük, hogy valamelyik nullával szomszédos (fel, le, jobbra, balra irányban) számot az üres helyre húzzuk (a valóságban az illető szám helyet cserél a nullával). Keverjük össze a mátrix elemeit, véletlenszerű lépéseket téve.

Adva lévén egy összekevert állapot, állítsuk vissza a kezdeti állapotot minimális számú lépésből.

Példa:

10.1. táblázat.

Kezdeti állapot				Összekevert állapot			
1	2	3	4	8	2	5	11
5	6	7	8	1	6	9	10
9	10	11	12	3	15	0	7
13	14	15	0	12	14	13	4

Megjegyzések:

- Egy lépést úgy is felfoghatunk, mintha a nulla lépne, helyet cserélve az elmozdított számmal.
- A feladathoz rendelhető döntési fában a gyökér a kezdeti állapotot (az összekevert konfigurációt) ábrázolja, az első szinti csomópontok azokat a konfigurációkat, amelyekhez egy lépésből juthatunk, a második szintiek, amelyek két lépésből adódhatnak, és így tovább.
- A feladat optimálisan megoldott állapotát a döntési fában legmagasabban található – kirakott konfigurációnak megfelelő – csomópont képviseli.
- A sarkokból két irányba, az oldallapokról három irányba, egyébként négy irányba „léphet a nulla”. Ezzel összhangban, és figyelembe véve, hogy nem lépünk vissza oda, ahonnan ide léptünk,

a fa csomópontjai fiainak száma 1, 2 vagy 3 lesz (kivéve a gyökeret, amelynek 2, 3 vagy 4 fiú-konfigurációja lehet, hiszen esetében nem volt előző lépés).

– Amint már utaltunk rá, a feladat csomópontok képviselte állapotait a 16 szám konfigurációja határozza meg, nevezetesen az, hogy éppen hol találhatók a (4×4)-es mátrixban. Ez 16 független paramétert jelent.

10.1. Hogyan közelítenék meg a feladatot a bemutatott technikák?

A döntési fa mélysége alapvetően végtelen. Ezért egy mélységi bejárás végzetes lenne. Igaz, korlátozhatnánk a fát azzal, hogy nem fogadunk el ugyanazon az ágon ismétlődő állapotokat, hiszen ez végtelen ciklust (végtelen rekurziót) jelentene. De még így is a fa egyes ágai nagyon mélyek lehetnek (16! különböző konfiguráció létezik). Egyértelmű, hogy a döntési fa számtalan identikus csomópontot tartalmaz. Ha ezeket egymásra csúsztatjuk, egy olyan súlyozatlan irányított gráfhoz jutunk, amely számos kört tartalmaz. Mivel a minimális lépésszámú megoldást keressük, az összevont döntési fát úgy is tekinthetjük, mint amelynek élei egysúlyúak.

A backtracking stratégia egy másik hátránya (azon túl, hogy mélységében járja be a fát), hogy nincs benne céltudatosság. A mélységi bejárás közben széltében veszi az ágakat, általában balról jobbra sorrendben. Ha netalántán az optimális megoldáslevél a jobb oldali ágon található, csak a teljes fa bejárása után fogja megtalálni. Ez exponenciális bonyolultságú algoritmust jelent. Sőt, ha meg is találná hamarabb, nem tudná eldönteni, hogy a legmagasabban levő megoldáslevélről van-e szó, és folytatnia kellene a keresést. Ezért az se oldaná meg a problémát, ha az aktuális csomópont fiú részfáit valamilyen módon ígéretesség szerinti sorrendbe állítanánk, és ebben a sorrendben járnánk be őket.

Egy divide et impera algoritmus egyik problémája megint csak az lenne, hogy mélységében járja be a fát. Továbbá, mivel a fában ugyanaz a konfiguráció különböző ágakon többször is megjelenhet, sok egymásra tevődő részfeladat többször is megoldásra kerülne.

Mi a helyzet a mohó-stratégiával? Definiáljuk egy adott konfiguráció „távolságát” a megoldás (a kirakott) konfigurációtól úgy, mint a számok

aktuális és végső pozíciói közötti távolságok összegét. A mátrix $(i1, j1)$ és $(i2, j2)$ pozíciói közti távolság alatt az $|i2 - i1| + |j2 - j1|$ értéket értjük (legkevesebb hány lépésből lehetne az egyik pozícióból a másikba eljutni). Egy másik, egyszerűbb lehetőség távolság függvényre, ha tekintjük azon számok számát, amelyek még nincsenek a helyükön. Egy mohó-algoritmus az lehetne, hogy mindig azt a lépést tekintjük legígéretesebbnek, amelyik a megoldás állapotához „legközelebbi" konfigurációhoz vezet. Sajnos bizonyítható, hogy ez a megközelítés nem kielégítő. Sőt, nem ismert egyetlen mohó-algoritmus sem erre a feladatra.

Az eddigi megközelítések mind mélységi bejárás alapúak voltak. Viszont tekintettel arra, hogy a legmagasabb szinten található megoldáslevelet keressük, egyértelműen egy szélességi-szerű bejárásra alapuló algoritmus ajánlja magát. Valóban, kijelenthető, hogy az optimális megoldást az első – szélességi bejárás találta – megoldáslevél jelenti, a gyökérből hozzá vezető úttal.

Amint láthattuk, a dinamikus programozás gyökér–levelek irányú változata szélességi-szerű bejárást alkalmaz, ha az összevont döntési fa kört tartalmaz. Csakhogy lévén e gráf súlyozatlan, nincs mód rá, hogy hamarabb megtalálja az optimális megoldást, mint a klasszikus szélességi bejárás. Továbbá, hány dimenziós kellene hogy legyen a részfeladatok optimumértékeit tároló tömb? Ahány független paraméter jellemzi a (prefix) részfeladatok általános alakját, nevezetesen 16. Már ez is elég ok a dinamikus programozás kizárására.

Térjünk vissza a sorszerkezet segítségével megvalósítható klasszikus szélességi bejárás gondolatához. A fenti fejtegetés alapján eddig ez a megközelítés bizonyult a legígéretesebbnek. Hátrányként azonban továbbra is fennáll az, hogy a szélességi bejárás is exponenciális bonyolultságú algoritmust jelent: ha az optimális megoldáslevél a döntési fa n-edik szintjén található, akkor a megtalálásához szükséges az exponenciálisan növekvő fa első n szintjének a bejárása. Lehetne-e még javítani ezen, még ha nem is tudjuk teljesen felszámolni az exponenciális bonyolultságot? A megoldás egy branch and bound stratégiában rejlik.

10.2. Egy branch and bound stratégia

A szélességi bejárást úgy is fel lehet fogni, mintha párhuzamosan a fa mindenik ágának irányába elindulnánk, ugyanazzal a sebességgel haladva minden levél felé. Egy adott pillanatban a sorszerkezetben a növekvő

fa koronáján lévő, éppen meglátogatásra váró csomópontok találhatók (a sorszerkezetből eltávolított csomópontok már meg lettek látogatva). Hogyan lehetne több céltudatosságot vinni ebbe az algoritmusba?

Súlyozzuk a csomópontokat aszerint, hogy mennyire vannak „távol" a gyökér képviselte kezdeti konfigurációtól, és mennyire vannak „közel" a megoldás konfigurációhoz. Egy csomópont súlya e két „távolság" összege lesz. Az első távolság adott a csomópont fabeli szintjének értéke által (ennyi lépés lett megtéve az illető konfigurációig az illető ágon). A második távolságot csak megbecsülni tudjuk. Használjuk ehhez a fentebb (a mohó stratégia vizsgálatánál) bemutatott második távolságfüggvényt, miszerint egy konfiguráció „távolsága" a megoldás állapottól arányos azon számok számával, amelyek még nem kerültek a helyükre. A megoldás konfiguráció esetében a második távolság nyilván nulla lesz. Egy konfiguráció annál ígéretesebb, minél kisebb a súlya.

A hagyományos szélességi bejárásban a sor elején lévő konfiguráció fiúkonfigurációit mindig a sor végére tennénk be (a bal fiútól haladva a jobb fiú felé). Tehát a csomópontok fabeli pozíciója határozná meg, hogy milyen sorrendben kerülnek be a sorszerkezetbe. Változtassunk ezen! A sorszerkezetben a csomópontok legyenek súlyuk szerint növekvő sorrendben (elsőbbségi sor). Ez azt jelenti, hogy amikor eltávolítjuk a sor elejéről a sor legígéretesebb csomópontját, a fiúcsomópontjait nem a sor végére, hanem a rendezett sorbeli helyükre szúrjuk be. Az algoritmus akkor ér véget, ha megtaláltuk a megoldás konfigurációt, vagy ha kiürült a sor (nincs megoldás). Mivel azokat a konfigurációkat, amelyekkel találkoztunk már, nem szúrjuk be újra a sorba, ha nincs megoldás, a sor előbb-utóbb biztosan kiürül. Bizonyítható, hogy ez esetben is – akárcsak a klasszikus szélességi bejárás esetén – elsőnek az optimális megoldás konfigurációt képviselő levelet találjuk meg (csakhogy rövidebb idő alatt), azt, amelyik a legmagasabban található a fában. Az imént leírt algoritmus alapvetően egy branch and bound stratégia. Fő jellegzetessége a csomópontok súlyozásában jelen lévő heurisztika.

Szemléletesen ezt a stratégiát úgy lehet elképzelni, hogy ahelyett, hogy minden levél irányába ugyanazzal a sebességgel haladnánk, az ígéretesebbnek tűnő irányokat előnybe helyezzük. Olyan, mintha a szélességi bejárásnak adtunk volna egy olyan mélységi jelleget, amelyben céltudatosság van. De a pillanatnyilag kevésbé ígéretes irányokról se mondunk le. Ott vannak a sor végén, arra az eshetőségre, ha netalántán melléfogtunk. Ne feledjük, hogy az ígéretesség meghatározása csak megbecsülésen alapult.

Összehasonlítva az előbbi technikákkal, elmondható, hogy az imént bemutatott branch and bound stratégia céltudatosabb, mint a backtracking, és óvatosabb, mint a mohó-algoritmusok. A branch and bound stratégia is és a dinamikus programozás is elsőbbségi sor által implementált szélességi-szerű bejárást alkalmaz, és mindkettő elkerüli az identikus csomópontokkal való többszöri foglalkozást. A lényeges különbség viszont abban áll, ahogy a csomópontok súlyozását végzik. A dinamikus programozásban a jelen kizárólag a múltban gyökerezik (a sorban lévő csomópontok aktuális súlya a *bejárt részfa* legjobb gyökér-csomópont útjának súlya). Ezzel szemben a branch and bound stratégiák esetében a csomópontok súlyozásában, egy bizonyos értelemben, jelen van a jövő is, azon heurisztika révén, hogy megbecsüljük a megoldáslevéltől való távolságukat. További különbség köztük például az is, hogy a branch and bound technika – a klasszikus szélességi bejáráshoz hasonlóan – minden csomópontot éppen a legrövidebb úton ér el először, míg a dinamikus programozás e változata általában többszöri finomítás révén állítja elő az optimális utakat.

Foglaljuk össze ezen branch and bound stratégia alapelemeit:

- Szélességi bejáráson alapszik, amelyet sorszerkezet segítségével valósítunk meg.
- A feladathoz rendelhető döntési fa csomópontjait súlyozzuk. A súlyfüggvénybe általában a csomópont (legyen x) gyökértől „mért távolságát” (legyen $F(x)$) és az x gyökerű részfa legmagasabb megoldás csomópontjának „becsült közelségét” (legyen $g(x)$) építjük be. A g függvényt célszerű úgy megválasztani, hogy megoldáslevélre az értéke nulla legyen. *Alapvetően a súlyfüggvény határozza meg, mennyire lesz hatékony a „szélességi” keresésünk.*
- A sorszerkezetben a csomópontok súlyuk szerinti ígéretességi sorrendben találhatók.
- A sorszerkezetbe egy csomópontot csak akkor szúrunk be, ha az általa képviselt állapottal identikus állapotot képviselő csomópont nincs már benne, és nem is volt benne. A második feltétel ellenőrizhetősége érdekébe csak logikailag töröljük a sorból eltávolított elemeket.
- Az algoritmus akkor ér véget, ha megoldás csomópontot találunk (a sorelső elemre a g függvény értéke nulla), vagy ha kiürül a sor (nincs megoldás).

- Bizonyítható, hogy az optimális megoldáscsomópont az elsőnek megtalált megoldáscsomópont lesz.
- A branch and bound stratégiát gyakran olyan feladatoknál alkalmazzuk, amelyek minimális számú lépésben kérik a megoldást. Ilyen feladatok esetén a teljes megoldásnak tartalmaznia kell annak bizonyítását, hogy az elsőnek megtalált megoldáscsomópont garantáltan az optimális megoldáscsomópont. Még ha ezt nem is tudjuk bizonyítani, egy jó súlyfüggvénnyel kielégítő megoldáshoz juthatunk.

A következőkben közöljük a tili-toli feladatot megoldó branch and bound algoritmust. Igyekeztünk, amennyire csak lehetett, általánosan megírni, hogy sablonszerűen lehessen használni más, hasonló feladatok megoldásában.

A sorszerkezet szimulálásához láncolt listát fogunk használni. A lista elejére a FEJ pointer fog mutatni. A sorelső elem logikai törlését a sorból a sor elejére mutató pointer (ELSŐ) léptetésével oldjuk meg. A FEJ-jel kezdődő lista tartalmazza az összes konfigurációt, amellyel találkoztunk már. Az ELSŐ-vel kezdődő listaszakasz jelenti magát a sorszerkezetet. A lista elemei nyilván bejegyzés típusúak lesznek, és a következő mezőket tartalmazzák:

- a – állapotmező (bejegyzés típusú): a csomópont képviselte állapotot leíró paramétereket tartalmazza,
- F – a csomópont távolsága a gyökértől,
- apa – az apacsomópontnak a listabeli címét tartalmazza,
- köv – a lista következő elemére mutat.

Az állapotmező (a) almezői:

- M[1..4][1..4] – a konfiguráció mátrixa,
- i0,j0 – a nulla pozíciója a mátrixban.

A TILI-TOLI eljárás az algoritmus fő eljárása, amely létrehoz egy új listaelemet (ráállítja a FEJ pointert), beleolvassa a kezdeti konfigurációt, szimulálja a szélességi keresést, majd pedig kiírja azt a konfigurációsort, amelyeik lépésről lépésre elvezet a kezdeti konfigurációtól a megoldás konfigurációig (amennyiben találtunk megoldást).

```
TILI-TOLI()
   FEJ=újelem()
   FEJ->apa=0
   FEJ->köv=0
   FEJ->F=0;
   beolvas_állapot(FEJ)
```

```
   ELSŐ=FEJ;
   amíg ELSŐ≠0 és nem MEGOLDÁS(ELSŐ) végezd
      GENERÁL_ÉS_BESZÚR_FIAK(ELSŐ,4)
      ELSŐ=ELSŐ->köv // logikailag törli a sor első elemét
   vége amíg
   ha ELSŐ==0 akkor kiírás "Nincs megoldás"
   különben KIÍRATÁS(ELSŐ)
   vége ha
vége TILI-TOLI
```

A beolvas_állapot eljárás feltölti a p címen lévő listaelem a állapotmezője paramétereit.

```
beolvas_állapot(p)
   minden i=1,4 végezd
      minden j=1,4 végezd
         be: p->a.M_b[i][j]
         ha p->a.M_b[i][j]==0 akkor
            p->a.i0=i
            p->a.j0=j
         vége ha
      vége minden
   vége minden
vége beolvas_állapot
```

A KIÍRATÁS rekurzív eljárás, indulva a megtalált megoldáslevéltől – apáról apára szökdelve a listában –, felmegy a gyökérig, majd a rekurzió visszaútján kiírja az optimális megoldást jelentő lépéssort.

```
KIÍRATÁS(p)
   ha p->apa≠0 akkor KIÍRATÁS(p->apa)
   vége ha
   kiír_állapot(p)
vége KIÍRATÁS

kiír_állapot(p)
   minden i=1,4 végezd
      minden j=1,4 végezd
         ki: p->a.M_b[i][j]
      vége minden
   vége minden
vége kiír_állapot
```

A MEGOLDÁS függvény ellenőrzi, hogy a sor elején lévő csomópont képviselte konfigurációhoz viszonyítva, a megoldás-konfiguráció

„becsült közelsége” nulla-e. A g_becsült_közelség függvény egy p címen lévő elemben tárolt konfiguráció távolságát adja meg a megoldás konfigurációtól azáltal, hogy megszámolja, hány szám „nincs a helyén”.

```
MEGOLDÁS (ELSŐ)
   ha g_becsült_közelség(ELSŐ)==0 akkor return 1
   különben return 0
   vége ha
vége MEGOLDÁS

g_becsült_közelség(p)
   k=0
   minden i=1,4 végezd
      minden j=1,4 végezd
         ha p->a.M_b[i][j]==0 akkor
            ha nem (i==4 és j==4) akkor k+=1
            vége ha
         különben
            ha p->a.M_b[i][j]≠(i-1)*4+j akkor k+=1
            vége ha
         vége ha
      vége minden
   vége minden
   return k
vége g_becsült_közelség
```

A GENERÁL_ÉS_BESZÚR_FIAK kulcseljárás előállítja a sor elején lévő konfigurációból az ebből az egy lépésből adódó fiúkonfigurációkat, és amennyiben még nem találkoztunk velük, beszúrja azokat a súly szerint rendezett sorba (BESZÚR_RENDEZVE eljárás). A fiúkonfigurációk előállítását úgy oldottuk meg, hogy sorra készítettünk n másolatot (az n paraméterben kapja meg az eljárás a feladatra jellemző lépéstípusszámot) az apakonfigurációról, és ezeket rendre átalakítottuk a valós lépéseknek megfelelő fiúkonfigurációkká (például a jobb felső sarokból nem léphetünk jobbra, illetve fel irányba). Az átalakít eljárás 1-et vagy 0-t térít vissza, attól függően, hogy létezik-e vagy sem az illető irányba fiúkonfiguráció (ez az érték kerül a *jó* változóba). Az EGYEDI eljárás végigszalad a teljes listán – azokon az elemeken is, amelyek logikailag már törölve lettek a sorból –, és ellenőrzi, hogy találkoztunk-e már az illető fiúkonfigurációval. Ha egy előállított fiúkonfiguráció nem megfelelő, vagy nem egyedi, akkor a számára létrehozott elemet felszámoljuk (felszámol eljárás).

```
GENERÁL_ÉS_BESZÚR_FIAK(ELSŐ,n)
   minden i=1,n végezd
      fiú=újelem()
      másol(fiú,ELSŐ) // átmásolja az ELSŐ című bejegyzést a fiú címűre
      jó=átalakít( fiú,i)
      fiú->apa=ELSŐ
      fiú->F=(ELSŐ->F)+1
      ha jó==1 és EGYEDI(fiú,FEJ) akkor
         BESZÚR_RENDEZVE(ELSŐ,fiú)
      különben
         felszámol(fiú)
      vége ha
   vége minden
vége GENERÁL_ÉS_BESZÚR_FIAK
```

Az átalakítás annyit jelent, hogy megtesszük az i-edik irányba a megfelelő lépést (a nulla helyet cserél az illető szomszédjával), amennyiben ez lehetséges, majd aktualizáljuk a nulla pozícióját $(i0, j0)$. A négy irány: felfele ($i = 1$), lefele ($i = 2$), balra ($i = 3$), jobbra ($i = 4$). Ha létezik az illető fiúkonfiguráció, 1-gyel térünk vissza, különben 0-val.

```
átalakít(fiú,i)
   ha i==1 és fiú->a.i0>1 akkor
      fiú->a.M_b[fiú->a.i0][fiú->a.j0]=
                     fiú->a.M_b[(fiú->a.i0)-1][fiú->a.j0]
      fiú->a.i0--
      fiú->a.M_b[fiú->a.i0][fiú->a.j0]=0
      return 1
   vége ha
   ha i==2 és fiú->a.i0<4 akkor
      fiú->a.M_b[fiú->a.i0][fiú->a.j0]=
                     fiú->a.M_b[(fiú->a.i0)+1][fiú->a.j0]
      fiú->a.i0++
      fiú->a.M_b[fiú->a.i0][fiú->a.j0]=0
      return 1
   vége ha
   ha i==3 és fiú->a.j0>1 akkor
      fiú->a.M_b[fiú->a.i0][fiú->a.j0]=
                     fiú->a.M_b[fiú->a.i0][(fiú->a.j0)-1]
      fiú->a.j0--
      fiú->a.M_b[fiú->a.i0][fiú->a.j0]=0
      return 1
   vége ha
```

```
      ha i==4 és fiú->a.j0<4 akkor
         fiú->a.M_b[fiú->a.i0][fiú->a.j0]=
                          fiú->a.M_b[fiú->a.i0][(fiú->a.j0)+1]
         fiú->a.j0++
         fiú->a.M_b[fiú->a.i0][fiú->a.j0]=0
         return 1
      vége ha
      return 0
   vége átalakít

   EGYEDI(fiú,FEJ)
      amíg FEJ≠0 végezd
         ha FEJ->a==fiú->a akkor return 0
            FEJ=FEJ->köv
         vége ha
      vége amíg
      return 1
   vége EGYEDI
```

A rendezett sorba való beszúrás két lépést feltételez: először meg kell keresni a helyét a sorban, majd be kell kötni a láncolt listába.

```
   BESZÚR_RENDEZVE(ELSŐ,fiú)
      p=ELSŐ
      amíg p->köv≠0 és súly(p->köv) < súly(fiú) végezd
         p=p->köv
      vége amíg
      fiú->köv=p->köv
      p->köv=fiú
   vége BESZÚR_RENDEZVE
```

Emlékezzünk, hogy egy csomópont súlya a gyökértől mért és a legközelebbi megoldás-konfigurációig becsült „távolságok” összege.

```
   súly(fiú)
      return fiú->F + g_becsült_közelség(fiú)
   vége súly
```

A branch and bound stratégia kulcselemeit a nagybetűs nevű eljárások hordozzák. Ezek alapvetően nem változnak feladattól feladatig. A listaelemeket leíró bejegyzésről is elmondható, hogy csak az a állapotmezője feladatfüggő. Ezért a következő megoldott feladata esetében effektíve csak a kisbetűs eljárásokat írjuk meg.

10.2. Misszionárius – kannibál: Egy folyó egyik partján van K kannibál, M misszionárius és egy kétszemélyes csónak. Hogyan tudnak átjutni a kannibálok és a misszionáriusok a túlsó partra, anélkül hogy mészárlás történne. Erre akkor kerülne sor, ha valamelyik parton többségben maradnának a kannibálok (és van misszionárius is az illető parton).

Példa: Legyen $K = 3$ és $M = 3$.

10.2. táblázat.

	Bal part (K_b, M_b)	Jobb part (K_j, M_j)
	3 3	**0 0**
Átmegy 2 kannibál (4)	1 3	2 0
Visszajön 1 kannibál (3)	2 3	1 0
Átmegy 2 kannibál (4)	0 3	3 0
Visszajön 1 kannibál (3)	1 3	2 0
Átmegy 2 misszionárius (2)	1 1	2 2
Visszajön 1 kannibál és 1 misszionárius (5)	2 2	1 1
Átmegy 2 misszionárius (2)	2 0	1 3
Visszajön 1 kannibál (3)	3 0	0 3
Átmegy 2 kannibál (4)	1 0	2 3
Visszajön 1 kannibál (3)	2 0	1 3
Átmegy 2 kannibál (4)	**0 0**	**3 3**

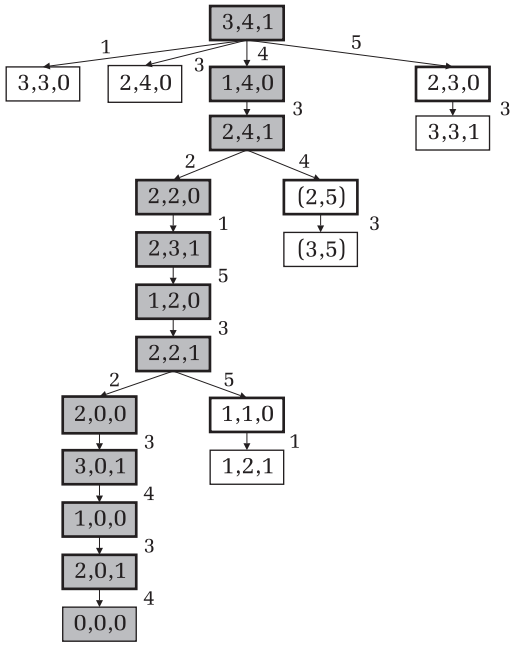
Megoldás: Észrevehető, hogy a feladat egy adott állapotát egyértelműen meghatározza a kannibál és misszionárius szám a bal parton (ebből egyértelműen adódik azok száma, akik a jobb parton vannak), valamint a csónak helyzete. Ezért a listaelemek a állapotmezőjének almezői a (K_b, M_b, P) paraméterek lesznek. A partot képviselő P paraméter 1 vagy 0 értékeket vehet fel, attól függően, hogy a bal vagy a jobb parton van a csónak.

A feladat egy adott $x(K\_b, M\_b, P)$ állapotának a megoldás állapothoz való közelségét a $g(x) = K\_b + M\_b + P$ függvénnyel fogjuk jellemezni. A távolságfüggvény megválasztásában figyelembe vettük, hogy a megoldásállapotra nulla legyen az értéke ($K\_b = 0$, $M\_b = 0$, $P = 0$). Mivel a feladat nem kéri a minimális lépésszámú megoldást, gyorsabban célba jutunk, ha az imént értelmezett távolságfüggvénnyel súlyozzuk a csomópontokat (anélkül, hogy figyelembe vennénk a csomópont gyökértől mért távolságát).

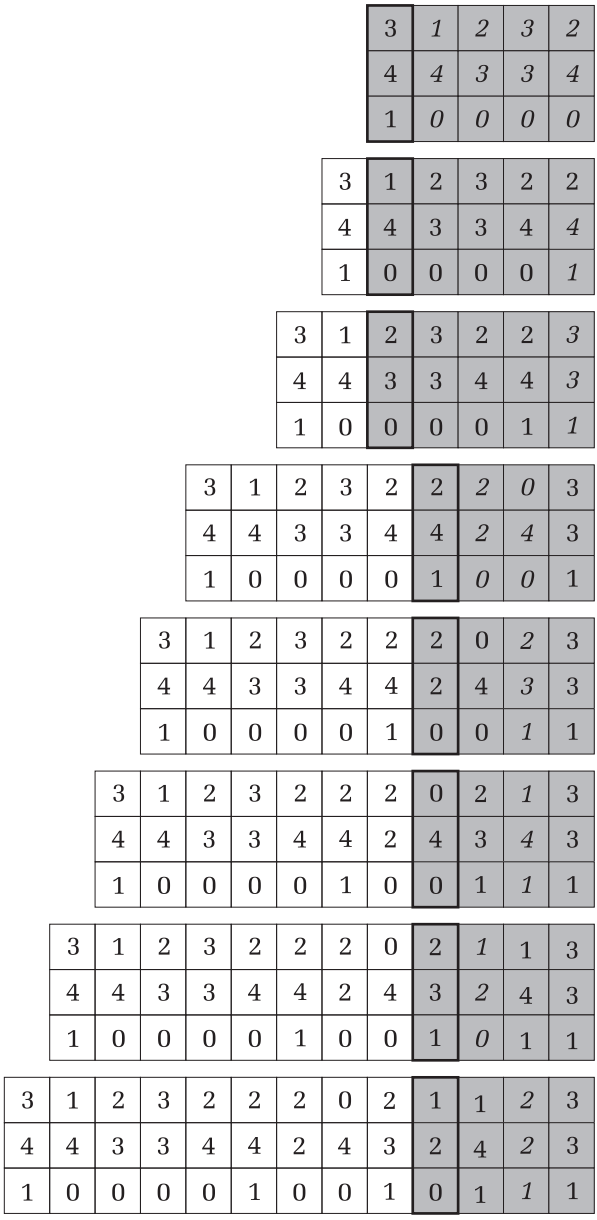
Megfigyelhető az is, hogy alapvetően ötféle „lépés” létezik (a lépéstípust bejelöltük zárójelben a példafeladatban is):

1. átmegy 1 misszionárius
2. átmegy 2 misszionárius
3. átmegy 1 kannibál
4. átmegy 2 kannibál
5. átmegy 1 misszionárius és egy kannibál

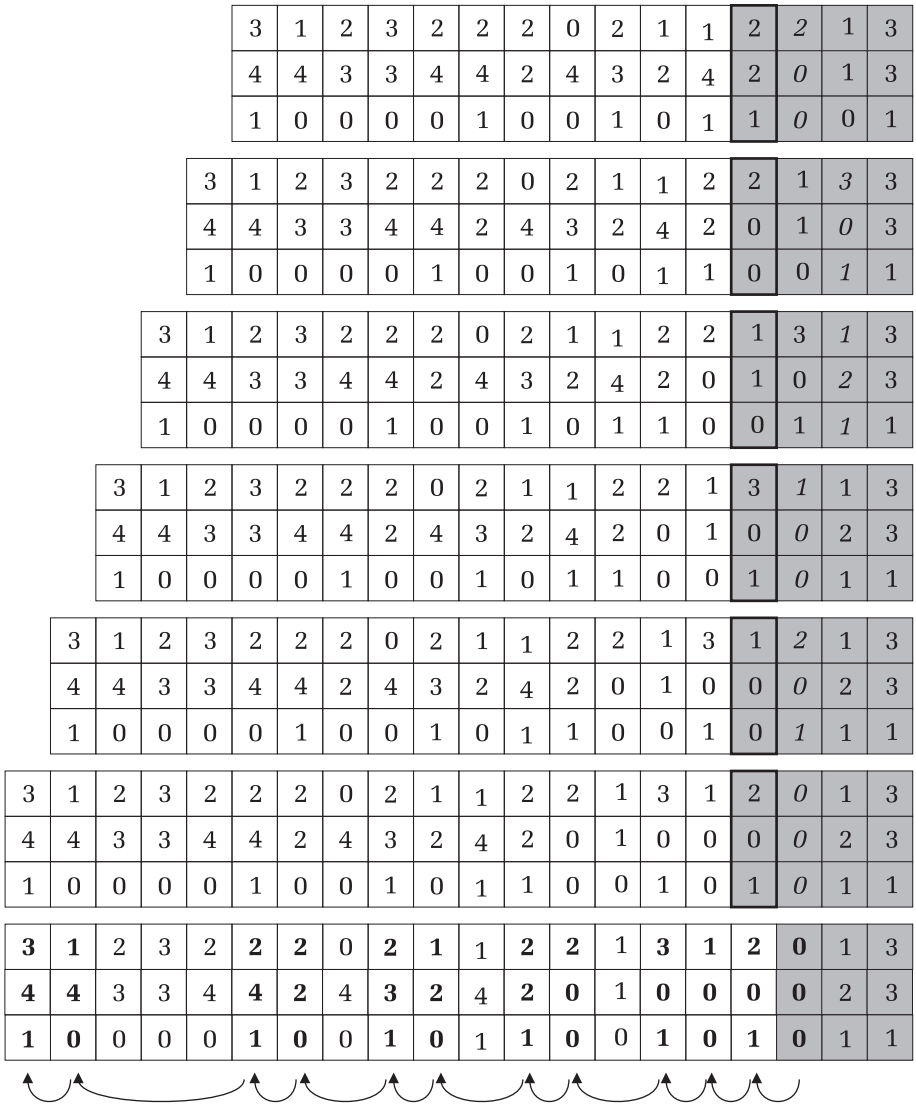
A 10.1. ábra a feladat döntési fájának a branch and bound algoritmus által megépített részfáját mutatja be a $K = 3$, $M = 4$ esetre. A csomópontokra az egyes állapotok paramétereit, a nyilakra pedig a lépéstípust írtuk. A megoldást képviselő gyökér–levél út csomópontjait besatíroztuk. A 10.2–10.3. ábrán mellékeltük a láncolt listát is, besatírozva azon elemeit, amelyek a sorszerkezetben vannak. A listát azokban a pillanatokban örökítettük meg, amikor a sorelső elemnek voltak beszúrandó fiai. Az ilyen csomópontok keretét megvastagítottuk mind a fában, mind a listában. Az adott pillanatban beszúrt fiúállapotokat dőlt karakterekkel jelöltük meg a listában. Figyeljük meg, hogy a fiúelemek mindig a rendezett sorbeli helyükre kerülnek beszúrásra.



10.1. ábra.



10.2. ábra.


3 1 2 3 2 2 2 0 2 1 1 2 2 1 3
4 4 3 3 4 4 2 4 3 2 4 2 0 1 3
1 0 0 0 0 1 0 0 1 0 1 1 0 0 1
3 1 2 3 2 2 2 0 2 1 1 2 2 1 3 1 3
4 4 3 3 4 4 2 4 3 2 4 2 0 1 0 3
1 0 0 0 0 1 0 0 1 0 1 1 0 0 1 1
3 1 2 3 2 2 2 0 2 1 1 2 2 1 3 1 3
4 4 3 3 4 4 2 4 3 2 4 2 0 1 0 2 3
1 0 0 0 0 1 0 0 1 0 1 1 0 0 1 1 1
3 1 2 3 2 2 2 0 2 1 1 2 2 1 3 1 2 1 3
4 4 3 3 4 4 2 4 3 2 4 2 0 1 0 0 0 2 3
1 0 0 0 0 1 0 0 1 0 1 1 0 0 1 0 1 1 1
3 1 2 3 2 2 2 0 2 1 1 2 2 1 3 1 2 0 1 3
4 4 3 3 4 4 2 4 3 2 4 2 0 1 0 0 0 0 2 3
1 0 0 0 0 1 0 0 1 0 1 1 0 0 1 0 1 0 1 1
3 1 2 3 2 2 2 0 2 1 1 2 2 1 3 1 2 0 1 3
4 4 3 3 4 4 2 4 3 2 4 2 0 1 0 0 0 0 2 3
1 0 0 0 0 1 0 0 1 0 1 1 0 0 1 0 1 0 1 1

10.3. ábra.

A lista végső állapotát bemutató ábrán nyilakkal bejelöltük, hogy a KIÍRATÁS rekurzív eljárás miként „mászik fel" a megoldáslevéltől gyökérhez vezető úton, apáról apára szökdelve.

Az alábbiakban közöljük a feladatot megoldó branch and bound algoritmus azon eljárásait, amelyek eltérnek a tili_toli algoritmus megfelelő eljárásaitól.

A misszionárius_kannibál főeljárás alapvetően csak a nevében fog különbözni a tili_toli eljárástól. Egy további minimális eltérés, hogy a GENERÁL_ÉS_BESZÚR_FIAK eljárást $n = 5$ fiúszámra hívjuk meg.

```
beolvas_állapot(p)
   p->a.P = 1
   be: p->a.K_b, p->a.M_b
vége beolvas_állapot

kiír_állapot(p)
   ki: p->a.K_b, p->a.M_b
vége kiír_állapot

g_becsült_közelség(p)
   return p->a.K_b + p->a.M_b + p->a.P
vége g_becsült_közelség

súly(fiú)
   return g_becsült_közelség(fiú)
vége súly
```

A megjegyzések a bal partról nézve vannak megfogalmazva. Először megtesszük a lépést, majd ellenőrizzük, hogy lehetséges, illetve helyes lépésről van-e szó.

```
átalakít(fiú,i)
   ha i==1 akkor
      ha fiú->a.P==1 akkor
         fiú->a.M_b-- // átmegy egy misszionárius
      különben
         fiú->a.M_b++ // átjön egy misszionárius
      vége ha
   vége ha
```

```
ha i==2 akkor
   ha fiú->a.P==1 akkor
      fiú->a.M_b-=2 // átmegy két misszionárius
   különben
      fiú->a.M_b+=2 // átjön két misszionárius
   vége ha
vége ha
ha i==3 akkor
   ha fiú->a.P==1 akkor
      fiú->a.K_b-- // átmegy egy kannibál
   különben
      fiú->a.K_b++ // átjön egy kannibál
   vége ha
vége ha
ha i==4 akkor
   ha fiú->a.P==1 akkor
      fiú->a.K_b-=2 // átmegy két kannibál
   különben
      fiú->a.K_b+=2 // átjön két kannibál
   vége ha
vége ha
ha i==5 akkor
   ha fiú->a.P==1 akkor
      fiú->a.M_b-- // átmegy egy misszionárius
      fiú->a.K_b-- // és egy kannibál
   különben
      fiú->a.M_b++ // átjön egy misszionárius
      fiú->a.K_b++ // és egy kannibál
   vége ha
vége ha
fiú->a.P = 1-fiú->a.P // a csónak átkerül a másik partra,
   // ha megtett lépés nyomán a bal parton a kannibál
   // vagy misszionárius szám negatív lett, vagy ha
   // meghaladta összkannibál, illetve összmisszionárius
   // számot, akkor ez a lépés nem lehetséges. Továbbá,
   // ha valamelyik parton, ahol vannak misszionáriusok,
   // többségbe kerültek a kannibálok, akkor helytelen
   // lépésről van szó.
ha fiú->a.K_b < 0 vagy fiú->a.M_b < 0 vagy
fiú->a.K_b > K vagy fiú->a.M_b > M vagy
(fiú->a.M_b > 0 és fiú->a.K_b > fiú->a.M_b) vagy
(fiú->a.M_b < M és fiú->a.K_b < fiú->a.M_b)
akkor return 0
```

```
        különben return 1
    vége átalakít
```

Ha az olvasó még inkább egymás mellett szeretné látni a backtracking és branch and bound stratégiákat, javasoljuk, hogy írjon a misszionáriusok és kannibálok feladatra egy backtracking algoritmust is. Hasonló gondolatmenetet lehet követni, mint a backtracking fejezet „békák" elnevezésű megoldott feladatában.

10.3. Kitűzött feladatok

10.1. Tili-toli-2: Oldjuk meg a tili-toli feladatott arra az esetre, ha két üres hely (két nullás szám) van mátrixban.

10.2. Tili-toli-backtrack: Oldjuk meg a tili-toli feladatokat backtracking stratégiával.

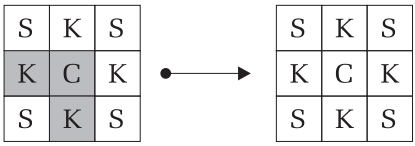
10.3. QUAD játék: A QUAD játékot egy 9 mezőből álló 3×3-as táblán játsszák. A tábla négy sarokmezőjét S-sel, az oldalak középső mezőjét K-val, a középpontban lévő mezőt pedig C-vel jelöljük. Mindenik mező lehet fehér vagy fekete.

Az alábbi játékszabályok léteznek:

- A fekete mezők lenyomásának nincs semmilyen következménye.
- Egy fehér sarok (S) lenyomása megváltoztatja a szomszédos közepek (K), a centrum (C), valamint a saját színét.
- Egy fehér közép (K) lenyomása megváltoztatja a szomszédos sarkok (S), valamint a saját színét.
- Egy fehér centrum (C) lenyomása megváltoztatja a szomszédos közepek (K), valamint a saját színét.

A játék célja olyan lépéssorozat megtalálása, amely elvezet egy adott kezdeti konfigurációtól egy végső konfigurációig. Adjunk branch and bound algoritmust a feladatra.

Példa:



10.4. ábra.

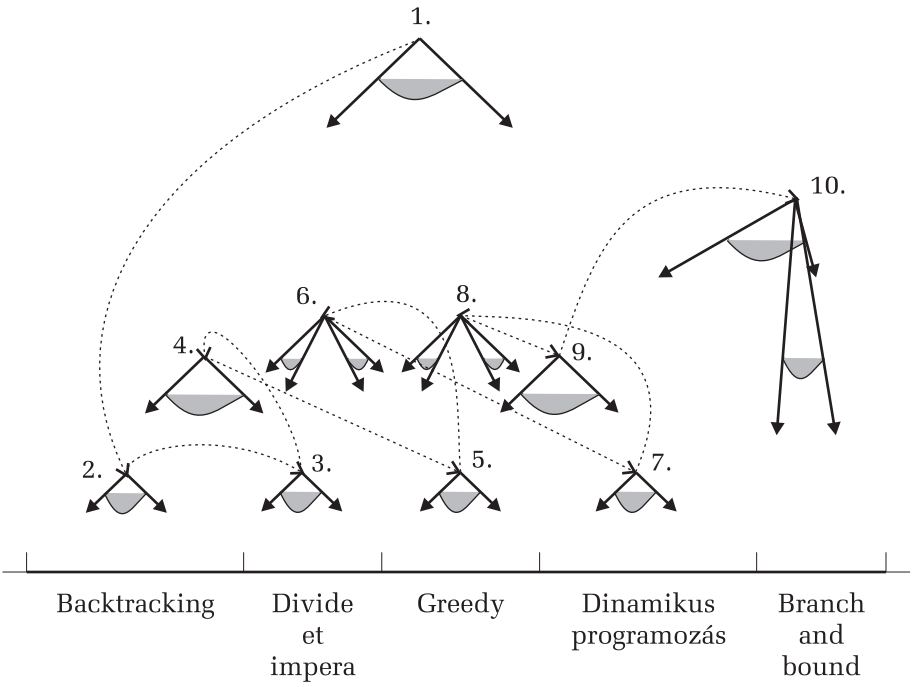
10.4. Quad-backtrack: Oldjuk meg a QUAD feladatot backtracking stratégiával.

10.5. Utazó kereskedő: Ismert egy ország úthálózata, azaz, hogy mely városok között vannak direkt utak, és mennyi ezek hossza. Határozzuk meg (egy utazó kereskedő részére) a legrövidebb olyan körutat, amely minden várost érint egyszer és csakis egyszer (kivéve az indulási várost).

11. FEJEZET

HATÁRÁTKELŐK A PROGRAMOZÁSI TECHNIKÁK VILÁGÁBAN

E könyv, amelyet a kedves olvasó a kezében tart, és reméljük, továbbra is keze ügyében marad, egy jelképes világot, az algoritmustervezési stratégiák világát igyekezett bemutatni. Tekintsük át újra e világot, felülnézetből, egy operatőr kameráján keresztül (az ábra a képzeletbeli kameránk mozgását ábrázolja).



11.1. ábra.

Az első fejezet madártávlatból adott átfogó képet az öt bemutatásra kerülő technikáról, egyelőre csak körvonalazva stratégiáikat. A második felejzetben a backtracking feladatok „országára” fókuszáltunk, hogy

részletekbe menő képet kapjunk róla. A harmadik fejezetben képzeletbeli kameránk fókusza átkerült az „oszd meg és uralkodj” algoritmusokra, feltárva ezen „ország” titkait. A negyedeik fejezet nyújtotta az első igazi felülnézetet, az első két technika párhuzamba állításával. Itt tárulhattak fel előttünk a backtracking és divide et impera közötti alapvető, illetve árnyalati hasonlóságok és különbségek. Az ötödik fejezet a mohóalgoritmusokat állította a figyelem középpontjába, majd a hatodik fejezetben kameránk újra felemelkedet, hogy párhuzamba állítva láthassuk a greedy és backtracking stratégiákat. A hetedik fejezetben valósággal átkutattuk a dinamikus programozás felségterületét. Mélyreható vizsgálódásunkat további két felülnézet-fejezet egészítette ki, amikor is először a divide et imperával, majd pedig a greedy stratégiával került összehasonlításra a dinamikus programozás. A tizedik fejezetben más filmezési technikát alkalmaztunk. Miközben kameránkkal újra végigpásztáztunk a négy már bemutatott technikán, figyelmünkbe az optimalizálási feladatok világának egy olyan része ötlött, amely egyik addigi országhoz sem tartozott igazán. Odafókuszálva, úgy azonosíthattuk e titokzatos területet mint a branch and bound feladatok külön világának csücskét.

11.1. Határmenti algoritmusok

Ahogy az olvasó megfigyelhette, a felülnézetet gyakran azáltal valósítottuk meg, hogy ugyanazt a feladatot többféle technikával is megoldottuk. Ezt tettük például a 4. fejezetben az „egér–sajt”, illetve a 9. fejezetben a hátizsák feladat kapcsán. Ez esetekben ugyanarra a feladatra két különböző algoritmust adtunk.

Egy másik módja a stratégiák együttes tanulmányozásának a „határmenti algoritmusok” tanulmányozása. Ahogy a határmenti városokban keverednek a kultúrák, ugyanígy léteznek olyan algoritmusok is, amelyekben két vagy több technika jellegzetes vonásai is fellelhetők. Mintegy vizsgálódásunk utójátékaként lássunk két ilyen algoritmust.

11.1.1. Bináris keresés

Elevenítsük fel a bináris keresés algoritmust, illetve a feladatot, amelyet megold:

Adott egy szigorúan növekvő számsorozat, amelyet az $a[1..n]$ *tömbben tároltunk el, valamint egy* x *szám. Írjunk* hatékony *algoritmust* x

megkeresésére a számsorozatban. Ha megtalálható, térítsük vissza a sorszámát, különben nullát.

Az algoritmus: Megcélozzuk az $a[1..n]$ tömbszakasz középső elemét, az $a[n/2]$ elemet. Ha ez éppen a keresett elem, a keresést befejeztük. Ellenkező esetben, attól függően, hogy x kisebb vagy nagyobb, mint ezen középső elem, a keresést vagy a számsorozat alsó ($a[1\,..\,n/2-1]$), vagy a felső ($a[n/2+1\,..\,n]$) felében folytatjuk, ahol természetesen hasonlóan járunk el. Az algoritmus akkor ér véget, ha megtaláltuk a keresett elemet, vagy ha üres tömbszakaszhoz jutottunk.

Ezt az algoritmust hagyományosan divide et impera stratégiának tekintik, és ezt nem is szeretnénk vitatni, hiszen minden lépésben a feladatot valóban kettőbe osztja. Mégis rá fogunk mutatni, hogy egy határmenti algoritmusról van szó, „Mohóország" határához közel. Mielőtt rámutatnánk a mohó jegyekre, elevenítsük meg a feladat mögött meghúzódó bináris fastruktúrát.

A fa gyökere a teljes számsorozatot képviseli, a többi csomópont pedig a felezésekből nyert tömbszakaszokat. A fa levelei a feldarabolás nyomán adódó üres tömbszakaszokat ábrázolják, kivéve egyet, amennyiben x megtalálható a számsorozatban. Ez esetben a szóban forgó levél egy olyan tömbszakaszt képvisel, amelynek x pontosan a középső eleme.

A mohó-algoritmusokat optimalizálási feladatok megoldására használjuk. Fellelhető-e e feladat esetében az optimalizálási jelleg? Egy bizonyos értelemben igen! A feladatot lineáris kereséssel is meg lehetne oldani (sorban megvizsgáljuk az összes elemet, anélkül hogy kihasználnánk a sorozat rendezettségét), de tőlünk hatékony algoritmust kérnek. A keresésére nézve a legrosszabb eset az, ha a keresett szám nem található meg a számsorozatban. Ilyenkor a lineáris keresés $O(n)$ időigényű, a bináris viszont csak $O(\lg n)$. Tehát a feladat úgy is felfogható, hogy írjunk olyan kereső algoritmust, amely esetében az összehasonlítások száma a legrosszabb esetben minimális. Ebből a szempontból a középső elem megcélozása helyi optimumnak tekinthető, hiszen csakis középen történő vágásokkal jutunk $\lg n$ magas fához. Tehát a mohó döntés nem abban áll, hogy a számsorozat melyik felében folytatjuk a keresését (hiszen ez egyértelmű), hanem abban, hogy minden lépésben éppen a középső elemet vizsgáljuk meg. Bármely más választás a legrosszabb esetben $\lg n$-nél több összehasonlítást vonna maga után.

Más szóval, a sorozat rendezettségéből adódóan egyetlen összehasonlítással – amennyiben nem találtuk el a keresett elemet – kizárhatók a további keresésből vagy a megvizsgált elemet megelőző, vagy az őt

követő elemek szakasza. A *legrosszabb eset* nyilván az, ha mindig a nagyobbik szakaszban *kell folytatnunk* a keresést (mert potenciálisan ott található meg a keresett elem). Ha az éppen vizsgálat alatt lévő részsorozat m hosszú, akkor a vágásból adódó nagyobbik rész az $\lceil (m-1)/2 \rceil .. (m-1)$ intervallumban mozoghat. A bináris keresés a középső elem választásával ezen intervallumnak éppen az alsó, a lineáris keresés pedig (mivel mindig az első elemet vizsgálja) a felső korlátját használja.

Az alábbiakban összefoglaljuk a divide et impera stratégia azon vonásait, amelyek hiányoznak a bináris keresés algoritmusból, és felsorolunk további mohó jegyeket, amelyek viszont jelen vannak benne:

- Bár az algoritmus a feladatot minden lépésben kettéosztja, nem oldja meg mindkét részfeladatot, és nem ezek megoldásaiból építi fel az apafeladat megoldását, mint ahogy a klasszikus divide et impera stratégia esetében szokásos.
- A feladat minden lépésben egy hasonló és egyszerűbb feladattá redukálódik, ami az algoritmus greedy jellegét hangsúlyozza.
- Az algoritmus már leérkezve a fa valamelyik levelébe, megoldást tud hirdetni, nemcsak amikor visszaér a gyökérbe (rekurzív implementálás esetén). Ezért is van az, hogy klasszikus változatában a bináris keresés iteratív algoritmus.

11.1.2. Dijkstra-algoritmus

Ezzel az algoritmussal a mohó stratégiával foglalkozó fejezetben találkoztunk, de amint látni fogjuk, a dinamikus programozás számos jellegzetességét is magán viseli. Az „legrövidebb utak” feladatot megoldó algoritmusról van szó.

Adott egy úthálózat – amely n várost köt össze – egy $n \times n$ méretű d mátrixban. A $d[i][j]$ elem az i és j városok közti direkt út hosszát tárolja (ha két város közt nincs direkt út, a megfelelő mátrixelemek értéke ∞). Határozzuk meg az első várostól az összes többihez vezető legrövidebb utakat és ezek hosszát.

Az algoritmus: Minden lépésben a következő legközelebbi városhoz határozzuk meg a legrövidebb utat, és ezt a nála közelebb eső városokhoz vezető, már rendelkezésre álló legrövidebb utak alapján tesszük meg.

Mi adja az algoritmus mohó jellegét? Az, hogy mohó sorrendben határozzuk meg a legrövidebb utakat. Ebben a megközelítésben a feladat mögött meghúzódó fa gyökere az eredeti feladatot képviseli, azaz az összes legrövidebb út meghatározásának problémáját. Azzal, hogy

minden lépésben meghatározzuk a következő legközelebbi városhoz vezető legrövidebb utat, a feladat mind egyszerűbb és egyszerűbb részfeladatokká redukálódik.

Az érem másik oldala viszont az, hogy általános részfeladatnak tekinthetjük egy városhoz vezető legrövidebb út meghatározásának problémáját is. A triviális részfeladat a legközelebb eső városhoz vezető legrövidebb út meghatározása, hiszen ide biztosan a direkt út lesz a legrövidebb. A legnagyobb méretű részfeladat nem az eredeti feladat, hiszen ez az összes utat kéri, hanem a legtávolabbi városhoz vezető legrövidebb út problémája. Ebből a szemszögből a Dijkstra-algoritmus útmeghatározási sorrendje már nem egy mohó sorrend, hanem az optimalitás alapelve diktálta egyszerűtől a bonyolult fele való haladás sorrendje, ami viszont a dinamikus programozást juttatja eszünkbe.

E két technika kapcsolatát még érdekesebbé teszi az a tény, hogy számos *dinamikus programozási* feladat visszavezethető egy olyan optimális út problémájára (a feladathoz rendelhető összevont döntési fában mint irányított gráfban), amelyet Dijkstra imént bemutatott *mohó-algoritmusa* old meg.

SZAKIRODALOM

ANDONE, R.–GARBACEA, I.
1995 *Algoritmi fundamentali.* O perspectivă C++. Cluj-Napoca, Libris. 185–187, 219–221.

CERCHEZ, E.
2002 *Informatica* (Culegere de probleme pentru liceu). Iaşi, Editura Polirom

CERCHEZ, E.–ŞERBAN, M.
2005 *Programarea în limbajul C/C++ pentru liceu* (Metode şi tehnici de programare). Iaşi, Editura Polirom

COMENIUS, J. A.
1653 *Orbis sensualium pictus.*

CORMEN, T. H.–LEIRSERSON, C. E.–RIVES, R. L.
1990 *Introduction to Algorithms.* The Massachusetts Institute of Technology. 266–270, 287–289.

HRINCIUC LOGOFĂTU, D.
2001 *Probleme rezolvate şi algoritmi (C++).* Bucureşti, Editura Polirom

IONESCU, C.–BĂLAN, A.
2004 *Informatică pentru grupele de performanță.* Cluj-Napoca, Editura Dacia

IONESCU K.
2005 *Bevezetés az algoritmikába.* Kolozsvár, Egyetemi kiadó

KÁTAI Z.
2004 *Programozás C nyelven.* Kolozsvár, Scientia Kiadó

KÁTAI Z.
2005 „Upperview" algorithm design in teaching computer science in high schools. *Teaching Mathematics and Computer Science* 3.

KÁTAI Z.
2006 *Dynamic programming and d-graphs.* Kolozsvár, Studia Universitatis Babeş–Bolyai – Series Informatica

KÁTAI Z.
2007 *Dynamic programming strategies on the decision tree hidden behind the optimizing problems.* Lithuania, Informatics in Education, Institute of Mathematics and Informatics

KOSTER, C. H. A.
1988 *Programozás felülnézetből.* Budapest, Műszaki Könyvkiadó
LÉNÁRD F.
1963 *A problémamegoldó gondolkodás.* Budapest, Akadémiai Kiadó
ODAGESCU, I.–COPOS, C.–LUCA, D.–FURTUNA, F.–SMEUREANU, I.
1994 *Metode şi tehnici de programare.* Bucureşti, Intact. 95–108.
PÓLYA Gy.
1979 *A problémamegoldás iskolája.* Budapest, Tankönyvkiadó
REVÁKNÉ MARKÓCZI I.–MÁTHÉ J.
2002 A természettudományos problémamegoldó gondolkodás fejlesztése a középiskolában. *Új Pedagógiai Szemle.* 10.
TUDOR, S.
1996 *Tehnici de programare.* Bucureşti, Editura L & S Infomat

http://www.cis.upenn.edu/∼matuszek/cit594-2004/Lectures/44-dynamic-programming.ppt

ABSTRACT

"To teach means scarcely anything more than to show how things differ from one another in their different purposes, forms and origins... Therefore, he who differentiates well teaches well." In this book we are going to present a teaching- learning method and suggest a syllabus that help the high school students look at the algorithm design strategies from a so called "upper view": greedy, backtracking, divide and conquer, dynamic programming, branch and bound. The goal of the suggested syllabus is, beyond the presentation of the techniques, to offer the students a view that reveals them the basic and even the slight differences and similarities between the strategies. In consensus with the Comenius principle this is essential, if we want to master this field of programming.

The method we are presenting makes possible to discuss uniformly all the above-mentioned techniques. We tried to establish such an "upper view" where each technique can be seen in the same time next to each other. By this means it becomes possible to integrate all the four techniques into a frame that forms a whole. If the students recognize the position of certain techniques related to the others, then the so called "more difficult" strategies become available for them.

In chapter 7 we also present a study and classification of the dynamic programming strategies. By presenting the characteristics of certain dynamic programming strategies on the decision tree hidden behind the optimizing problems, we offer a clear tool for their study and classification, which can help in the comprehension of the essence of this programming technique.

REZUMAT

Cartea pe care cititorul ține în mână prezintă o sinteză a tehnicilor de programare cum ar fi: backtracking, divide et impera, greedy, programarea dinamică şi branch and bound. Scopul didactic a cărții este de a transmite cititorului o vedere de ansamblu, şi să ajute în discernarea asemănărilor şi a diferențelor – chiar şi de noanță – între tehnicile de programare menționate.

Capitolul 7 extinde metoda asupra metoda programării dinamice, făcând posibilă o clasificare originală a strategiilor de sub cupola acestei tehnici. Clasificarea prezentată creşte semnificativ accesibilitatea acestei metode dificile.

A SZERZŐRŐL

Kátai Zoltán 1968. március 13-án született Nagyváradon. Középiskolai tanulmányait a marosvásárhelyi Bolyai Farkas Elméleti Líceumban végezte 1982–1986 között, egyetemi tanulmányait pedig a Kolozsvári Műszaki Egyetem Automatizálás és Számítógépek Karán 1987–1992 között.

1992–2005 között informatika tanár a marosvásárhelyi Bolyai Farkas Elméleti Líceumban, 1996–2000 között kvalifikált oktató a Gábor Dénes Főiskola marosvásárhelyi karán, 1995–1997 között pedig óraadó tanár a marosvásárhelyi Petru Maior Egyetemen. 2002-től a Sapientia Erdélyi Magyar Tudományegyetem Műszaki és Humántudományok Kara Matematika–Informatika Tanszékének adjunktusa.

Fő kutatási területe a programozási technikák, dinamikus programozás, valamint az érzékszervek párhuzamos bevonása a tanítás-tanulás folyamatába.

JEGYZETEK

A SAPIENTIA –
ERDÉLYI MAGYAR TUDOMÁNYEGYETEM JEGYZETEI

Megjelent:

BEGE ANTAL
Számelméleti feladatgyűjtemény. Marosvásárhely, Műszaki és Humán Tudományok Kar, Matematika–Informatika Tanszék. 2002.

BEGE ANTAL
Számelmélet. Bevezetés a számelméletbe. Marosvásárhely, Műszaki és Humán Tudományok Kar, Matematika–Informatika Tanszék. 2002.

VOFKORI LÁSZLÓ
Gazdasági földrajz. Csíkszereda, Csíkszeredai Kar, Gazdaságtan Tanszék. 2002.

TŐKÉS BÉLA–DÓNÁTH-NAGY GABRIELLA
Kémiai előadások és laboratóriumi gyakorlatok. Marosvásárhely, Műszaki és Humán Tudományok Kar, Gépészmérnöki Tanszék. 2002.

IRIMIAŞ, GEORGE
Noţiuni de fonetică şi fonologie. Csíkszereda, Csíkszeredai Kar, Humán Tudományok Tanszék. 2002.

SZILÁGYI JÓZSEF
Mezőgazdasági termékek áruismerete. Csíkszereda, Csíkszeredai Kar, Gazdaságtan Tanszék. 2002.

NAGY IMOLA KATALIN
A Practical Course in English. Marosvásárhely, Műszaki és Humán Tudományok Kar, Humán Tudományok Tanszék. 2002.

BALÁZS LAJOS
Folclor. Noţiuni generale de folclor şi poetică populară. Csíkszereda, Csíkszeredai Kar, Humán Tudományok Tanszék. 2003

POPA-MÜLLER IZOLDA
Műszaki rajz. Marosvásárhely, Műszaki és Humán Tudományok Kar, Gépészmérnöki Tanszék. 2004.

Fodorpataki László–Szigyártó Lídia–Bartha Csaba
Növénytani ismeretek. Kolozsvár, Természettudományi és Művészeti Kar, Környezettudományi Tanszék. 2004.

Marcuş Andrei–Szántó Csaba–Tóth László
Logika és halmazelmélet. Marosvásárhely, Műszaki és Humán Tudományok Kar, Matematika–Informatika Tanszék. 2004.

Kakucs András
Műszaki hőtan. Marosvásárhely, Műszaki és Humán Tudományok Kar, Gépészmérnöki Tanszék. 2004.

Biró Béla
Drámaelmélet. Csíkszereda, Gazdasági és Humántudományi Kar, Humántudományi Tanszék. 2004.

Biró Béla
Narratológia. Csíkszereda, Gazdasági és Humántudományi Kar, Humántudományi Tanszék. 2004.

Márkos Zoltán
Anyagtechnológia. Marosvásárhely. Műszaki és Humán Tudományok Kar, Gépészmérnöki Tanszék. 2004.

Grecu Victor
Istoria limbii române. Csíkszereda, Gazdasági és Humántudományi Kar, Humántudományi Tanszék. 2004.

Varga Ibolya
Adatbázis-kezelő rendszerek elméleti alapjai. Marosvásárhely, Műszaki és Humántudományok Kar, Matematika–Informatika Tanszék. 2004.

Csapó János
Biokémia. Csíkszereda, Műszaki és Társadalomtudományi Kar, Műszaki és Természettudományi Tanszék. 2004.

Csapó János–Csapóné Kiss Zsuzsanna
Élelmiszer-kémia. Csíkszereda, Műszaki és Társadalomtudományi Kar, Műszaki és Természettudományi Tanszék. 2004.

Kátai Zoltán
Programozás C nyelven. Marosvásárhely, Műszaki és Humántudományok Kar, Matematika–Informatika Tanszék. 2004.

Weszely Tibor

Analitikus geometria és differenciálgeometria. Marosvásárhely, Műszaki és Humántudományok Kar, Matematika–Informatika Tanszék. 2005.

Györfi Jenő

A matematikai analízis elemei. Marosvásárhely, Gazdaság- és Humántudományok Kar, Matematika–Informatika Tanszék. 2005.

Finta Béla–Kiss Elemér–Bartha Zsolt

Algebrai struktúrák feladatgyűjtemény. Marosvásárhely, Műszaki és Humántudományok Kar, Matematika–Informatika Tanszék. 2006.

Antal Margit

Fejlett programozási technikák. Marosvásárhely, Műszaki és Humántudományok Kar, Matematika–Informatika Tanszék. 2006.

Csapó János–Salamon Rózália

Tejipari technológia és minőségellenőrzés. Csíkszereda, Műszaki és Társadalomtudományi Kar, Élelmiszer-tudományi Tanszék. 2006.

Oláh-Gál Róbert

Az informatika alapjai közgazdász- és mérnökhallgatóknak. Csíkszereda, Gazdaság- és Humántudományok Kar, Matematika–Informatika Tanszék. 2006.

A PARTIUMI KERESZTÉNY EGYETEM JEGYZETEI

Megjelent:

KOVÁCS ADALBERT
Alkalmazott matematika a közgazdaságtanban. Lineáris algebra. Nagyvárad, Alkalmazott Tudományok Kar, Közgazdaságtan Tanszék. 2002.

HORVÁTH GIZELLA
A vitatechnika alapjai. Nagyvárad, Bölcsészettudományi Kar, Filozófia Tanszék. 2002.

ANGI ISTVÁN
Zeneesztétikai előadások. I. Nagyvárad, Alkalmazott Tudományok Kar, Zenepedagógiai Tanszék. 2003.

PÉTER GYÖRGY–KINTER TÜNDE–PAJZOS CSABA
Makroökonómia. Feladatok. Nagyvárad, Alkalmazott Tudományok és Művészetek Kar, Közgazdaságtan Tanszék. 2003.

ANGI ISTVÁN
Zeneesztétikai előadások. II. Nagyvárad, Alkalmazott Tudományok Kar, Zenepedagógiai Tanszék. 2005.

TONK MÁRON
Bevezetés a középkori filozófia történetébe. Nagyvárad, Bölcsészettudományi Kar, Filozófiai Tanszék. 2005.

Scientia Kiadó
400112 Kolozsvár (Cluj-Napoca)
Mátyás király (Matei Corvin) u. 4. sz.
Tel./fax: +40-264-593694
E-mail: kpi@kpi.sapientia.ro

Korrektúra:
Jancsik Pál

Műszaki szerkesztés:
Mikó Csilla, Lineart kft.

Tipográfia:
Könczey Elemér

Készült a T3 Kiadó nyomdájában
150 példányban, 16,5 nyomdai ív terjedelemben
520085 Sepsiszentgyörgy (Sf. Gheorghe)
Sport u. 8/A., tel.: +40-267-351684
Felelős vezető: Bács Attila