

# Algoritmizálás

Horváth Gyula  
Szegedi Tudományegyetem  
Természettudományi és Informatikai Kar  
horvath@inf.u-szeged.hu

## 4. Dinamikus programozással megoldható feladatok

A dinamikus programozás elnevezés egy probléma-megoldási módszert jelöl. A módszer lényege, hogy a kiindulási problémát részproblémákra bontjuk és a részproblémák megoldásaival fejezzük ki a megoldást. Bár a megoldást rekurzívan fejezzük ki, azonban a tényleges kiszámítás nem rekurzív módon, hanem táblázat-kitöltéssel történik. Ez a módszer hatékony algoritmust eredményez számos fontos probléma megoldására.

### 4.1. Feladat: A pénzváltás probléma.

**Probléma:** Pénzváltás

**Bemenet:**  $P = \{p_1, \dots, p_n\}$  pozitív egészek halmaza, és  $E$  pozitív egész szám.

**Kimenet:** Olyan  $S \subseteq P$ , hogy  $\sum_{p \in S} p = E$ .

Megjegyzés: A pénzek tetszőleges címletek lehetnek, nem csak a szokásos 1, 2, 5, 10, 20, stb., és minden pénz csak egyszer használható a felváltásban.

### A Pénzváltás probléma; létezik-e megoldás

Először azt határozzuk meg, hogy van-e megoldás.

**A megoldás szerkezetének elemzése.**

Tegyük fel, hogy

$$E = p_{i_1} + \dots + p_{i_k}, \quad i_1 < \dots < i_k$$

egy megoldása a feladatnak. Ekkor

$$E - p_{i_k} = p_{i_1} + \dots + p_{i_{k-1}}$$

megoldása lesz annak a feladatnak, amelynek bemenete a felváltandó  $E - p_{i_k}$  érték, és a felváltáshoz legfeljebb a első  $i_k - 1$  ( $p_1, \dots, p_{i_k-1}$ ) pénzeket használhatjuk.

**Részproblémákra bontás.**

Bontsuk részproblémákra a kiindulási problémát: Minden  $(X, i) (1 \leq X \leq E, 1 \leq i \leq N)$  számpárra vegyük azt a részproblémát, hogy az  $X$  érték felváltható-e legfeljebb az első  $p_1, \dots, p_i$  pénzzel. Jelölje  $V(X, i)$  az  $(X, i)$  részprobléma megoldását, ami logikai érték;  $V(X, i) = \text{Igaz}$ , ha az  $X$  összeg előállítható legfeljebb az első  $i$  pénzzel, egyébként  $\text{Hamis}$ .

**Összefüggések a részproblémák és megoldásaik között.**

Nyilvánvaló, hogy az alábbi összefüggések teljesülnek a részproblémák megoldásaira:

1.  $V(X, i) = (X = p_i)$ , ha  $i = 1$
2.  $V(X, i) = V(X, i-1) \vee (X > p_i) \wedge V(X - p_i, i-1)$  ha  $i > 1$

$$V(X, i) \Leftrightarrow \begin{cases} X = p_i \vee \\ i > 1 \wedge V(X, i-1) \vee \\ i > 1 \wedge X > p_i \wedge V(X - p_i, i-1) \end{cases} \quad (1)$$

A kiindulási probléma megoldása:  $V(E, n)$

**Rekurzív megoldás.**

Mivel a megoldás kifejezhető egy  $V(X, i)$  logikai értékű függvénnyel, ezért a felírt összefüggések alapján azonnal tudunk adni egy rekurzív függvényeljárást, amely a pénzváltás probléma megoldását adja.

```

1 function V(X, i: word): boolean;
2 {Globális:P}
3 {Módszer: Rekurzív megoldás }
4 begin
5   V:=(X=P[i]) or
6     (i>1) and V(X, i-1) or
7     (i>1) and (X>P[i]) and V(X-P[i], i-1);
8 end{V};

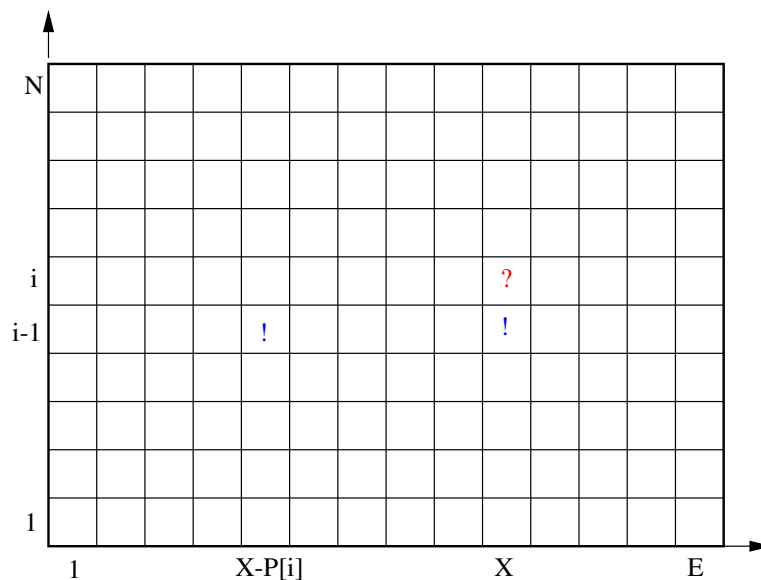
```

Elemezzük a rekurzív megoldás futási idejét. Számítsuk ki, hogy legrosszabb esetben hány eljárás-hívás történik; jelölje ezt  $F(E, n)$  adott  $E$  és  $n$  aktuális paraméterekre. Mivel  $F(E, 1) = 1$  és  $F(E, n) = F(E, n-1) + F(E - P[n], n-1)$  így  $F(E, n) = 2^{n-1}$ . Pl.  $n = 100$  esetén, feltéve, hogy olyan gyors gépünk van, amely 1 másodperc alatt  $2^{30}$  eljárás-hívást hajt végre, a futási idő legrosszabb esetben 17592 milliárd év lenne! Másképpen fogalmazva, a rekurzív megoldás a pénzek halmazának összes lehetséges részalmazát megvizsgálja, hogy az adott részalmaz összege megegyezik-e  $E$ -vel. De  $n$ -elemű halmaz összes részalmazainak száma  $2^n$ . Mi az oka annak, hogy a rekurzív megoldás ilyen lassú? Jóllehet csak  $E * n$  részprobléma van, de egy részprobléma megoldása (közvetve) sok másik részprobléma megoldásához kell és a rekurzív algoritmus ezeket mindig újra kiszámolja.

A gyorsítás tehát kézenfekvő: tároljuk a már kiszámított részproblémák megoldását.

### Megoldás a részproblémák megoldásainak táblázatos tárolásával.

Vegyünk egy  $VT$  táblázatot, amelyben minden lehetséges részprobléma megoldását tároljuk. Mivel minden részproblémát két érték határoz meg,  $X$  és  $i$ , ezért téglalap alakú táblázat kell.  $VT[X, i]$  az  $(X, i)$  részprobléma megoldását tartalmazza. **A részprob-**



1. ábra. A pénzváltás táblázata

### lélmák kiszámítási sorrendje.

Olyan kiszámítási sorrendet kell megállapítani, amelyre teljesül, hogy amikor az  $(X, i)$  részproblémát számítjuk, akkor ennek összetevőit már korábban kiszámítottuk. Mivel az  $(X, 1)$  részproblémáknak nincs összetevőjük, ezért közvetlenül kiszámíthatóak, azaz a táblázat első sorát számíthatjuk először. Ha  $i > 1$ , akkor az  $(X, i)$  részprobléma összetevői az  $(X, i-1)$  és  $(X - p_i, i-1)$ , ezért az  $i$ -edik sor bármely elemét ki tudjuk számítani, ha már kiszámítottuk az  $i-1$ -edik sor minden elemét. Tehát a táblázat-kitöltés sorrendje: soronként (alulról felfelé), balról-jobbra haladó lehet.

```

1 function V(P: Penzek; E, N: word): boolean;
2 { Pénzváltás négyzetes táblázat-kitöltéssel }
3 const
4   MaxE=30000; {a max. felváltandó összeg}
5   MaxN=200;  {a pénzek max. száma}
6 var
7   VT: array [1..MaxE, 1..MaxN] of boolean;
8   i, x: word;

```

```

9  begin{VT}
10 for x:=1 to E do
11   VT[x,1]:= false;           {az első sor, azaz V(x,1) számítása}
12 VT[P[1],1]:= P[1]<=E;
13 for i:=2 to N do           {az i-edik sor, azaz V(x,i) számítása}
14   for x:=1 to E do
15     VT[x,i]:= (P[i]=X) or
16               VT[x,i-1] or
17               (x>P[i]) and VT[x-P[i],i-1];
18   V:=VT[E,N];
19 end{V};

```

Az algoritmus futási ideje  $E * n$ -el arányos, mivel minden részprobléma (táblázatelem) kiszámítása konstans idejű. A memória igény  $E * n$  byte. Látható, hogy az  $i$ -edik sor kiszámításához csak az  $i - 1$ -edik sor kell. Ezért, ha csak arra kell válaszolni, hogy létezik-e megoldása a problémának, akkor elegendő lenne csak két egymást követő sort tárolni. Sőt, egyetlen sort is elég tárolni, ha a sorok kitöltését a felváltandó érték szerint csökkenő sorrendben (hátról előre haladva) végezzük. Az  $(X, i)$  és az  $(X, i - 1)$  részprobléma megoldását ugyanaz a táblázatelem tárolja, de nem írunk felül olyan részproblémát, amelyre később még szükség lenne. Tehát algoritmusunknak mind a tárigénye, mind a futási ideje függ az előállítandó értéktől, illetve felső korlátjától.

Tegyük fel, hogy a pénzekről és az előállítandó értékről nem tudunk semmit. Kérdés, hogy létezik-e olyan algoritmus, amelynek a futási ideje (legrosszabb esetben is) a pénzek  $n$  számának függvényében polinomiális? A választ nem tudjuk. Ez a számítástudomány legfontosabb nyitott problémája. Sőt, ha erre a feladatra találnánk a pénzek számában polinomiális idejű megoldást, akkor számos, fontos probléma megoldására is lenne hatékony algoritmus.

```

1  function VIT(const P:Penzek; E,N:longint):boolean;
2  { Pénzváltás megoldása lineáris táblázat-kitöltéssel }
3  const
4    MaxE=1000000; {a max. felváltandó összeg}
5  var T:array[0..MaxE] of boolean;
6    i,x:longint;
7  begin{VIT}
8    for x:=1 to E do {az első sor, azaz V(x,1) számítása}
9      T[x]:= false;
10   if P[1]<=E then T[P[1]]:= true;
11   for i:=2 to N do {az i-edik sor, azaz V(x,i) számítása}
12     for x:=E downto 1 do
13       T[x]:=T[x] or (x>=P[i]) and T[x-P[i]];
14   VIT:=T[E];
15 end{VIT};

```

## 4.2. A Pénzváltás probléma; egy felváltás előállítása

Ezidáig azt vizsgáltuk, hogy létezik-e megoldása a pénzváltás problémának. Most egy, de tetszőleges megoldást is meg kell határozni, amennyiben létezik megoldás.

**Bemenet:**  $P = \{p_1, \dots, p_n\}$  pozitív egészek halmaza, és  $E$  pozitív egész szám.

**Kimenet:** Olyan  $S \subseteq P$ , hogy  $\sum_{p \in S} p = E$

Akkor és csak akkor létezik megoldás, ha  $VT[E, N] = true$ .

Vegyük észre, hogy egy megoldás "kiolvasható" a  $VT$  táblázatból. Legyen  $i$  a legkisebb olyan index, hogy  $VT[E, i] = true$ , tehát  $VT[E, i - 1] = false$ . Ez azt jelenti, hogy  $E$  nem állítható elő az első  $i - 1$  pénzzel, de előállítható az első  $i$ -vel, tehát a  $P[i]$  pénz szerepel  $E$  valamely felváltásában. Tovább folytatva ezt a visszafejtést  $E - P[i]$  és  $i - 1$ -re, mindaddig, amíg a felváltandó érték 0 nem lesz, megkapunk egy megoldást.

```

1  X:=E; i:=N;
2  m:=0; {egy megoldásban a pénzek száma}
3  repeat
4    while (i>0) and VT[X,i] do Dec(i);
5    Inc(m);
6    S[m]:= i+1;
7    X:=X-P[i+1];

```

8 **until** X=0;  
 9 {az S[1], ..., S[m] indexű pénzek egy felváltása E-nek}

### 4.3. Feladat: Optimális pénzváltás

**Bemenet:**  $P = \{p_1, \dots, p_n\}$  pozitív egészek halmaza, és  $E$  pozitív egész szám.

**Kimenet:** Olyan  $S \subseteq P$ , hogy  $\sum_{p \in S} p = E$  és  $|S| \rightarrow$  minimális

Először is lássuk be, hogy az a mohó stratégia, amely mindig a lehető legnagyobb pénzt választja, nem vezet optimális megoldáshoz. Legyen  $E = 8$  és a pénzek halmaza legyen  $\{5, 4, 4, 1, 1, 1\}$ . A mohó módszer a  $8 = 5 + 1 + 1 + 1$  megoldást adja, míg az optimális a  $8 = 4 + 4$ .

**Az optimális megoldás szerkezetének elemzése.**

Tegyük fel, hogy

$$E = p_{i_1} + \dots + p_{i_k}, i_1 < \dots < i_k$$

egy optimális megoldása a feladatnak. Ekkor

$$E - p_{i_k} = p_{i_1} + \dots + p_{i_{k-1}}$$

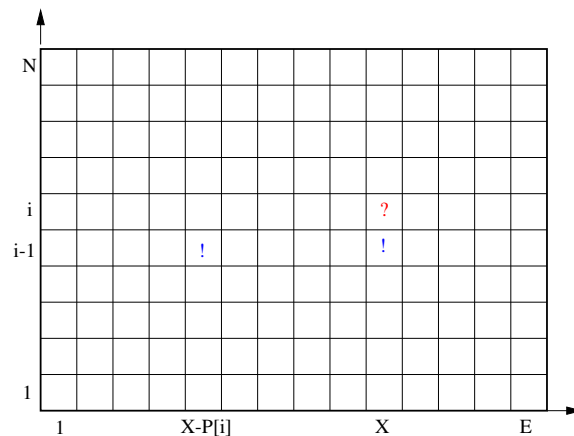
optimális megoldása lesz annak a feladatnak, amelynek bemenete a felváltandó  $E - p_{i_k}$  érték, és a felváltáshoz legfeljebb a első  $i_k - 1$  ( $p_1, \dots, p_{i_k-1}$ ) pénzeket használhatjuk. Ugyanis, ha lenne kevesebb pénzből álló felváltása  $E - p_{i_k}$ -nak, akkor  $E$ -nek is lenne  $k$ -nál kevesebb pénzből álló felváltása.

**Részproblémákra és összetevőkre bontás.**

A részproblémák legyenek ugyanazok, mint az előző esetben. Minden  $(X, i)$  ( $1 \leq X \leq E, 1 \leq i \leq N$ ) számpárra vegyük azt a részproblémát, hogy legkevesebb hány pénz összegeként lehet az  $X$  értéket előállítani legfeljebb az első  $i$   $\{p_1, \dots, p_i\}$  pénz felhasználásával. Ha nincs megoldás, akkor legyen ez az érték  $N + 1$ . Jelölje az  $(X, i)$  részprobléma optimális megoldásának értékét  $Opt(X, i)$ . Definiáljuk az optimális megoldás értékét  $X = 0$ -ra és  $i = 0$ -ra is, azaz legyen  $Opt(X, 0) = N + 1$  és  $Opt(0, i) = 0$ . Így  $Opt(X, i)$ -re az alábbi rekurzív összefüggés írható fel.

**A részproblémák optimális megoldásának kifejezése az összetevők megoldásaival.**

$$Opt(X, i) = \begin{cases} \infty & \text{ha } i = 0 \wedge X > 0 \\ 0 & \text{ha } X = 0 \\ Opt(X, i-1) & \text{ha } X < p_i \\ \min(Opt(X, i-1), 1 + Opt(X - p_i, i-1)) & \text{ha } X \geq p_i \end{cases} \quad (2)$$



2. ábra. A pénzváltás táblázata

```

1 procedure OptValto(const P :Penzek; E :word; N:word;
2                   var Db:word; var C:Megoldas );
3 const
4   MaxE=30000; MaxN=100;
5 var
6   Opt:array [0..MaxE] of 0..MaxN+1; {az opt. mego. értéke}

```

```

7  V: array [ 0..MaxE, 0..MaxN] of 0..MaxN+1;
8  i, x, Rop: word;
9  begin {OptValto}
10 for x:=1 to E do begin           { inicializálás }
11   Opt[x]:=N+1; V[x,0]:=N+1 end;
12 Opt[0]:=0;
13 for i:=1 to N do                 { táblázat-kitöltés }
14   for x:=E downto 1 do begin
15     if (x>=P[i]) then
16       Rop:=Opt[x-P[i]]+1        {x-P[i] opt. felváltása+1}
17     else
18       Rop:=N+1;
19     if Rop<Opt[x] then begin     {Opt[x]=1+Opt(x,i-1)}
20       Opt[x]:=Rop;              {az i. pénz szerepel x }
21       V[x,i]:=i;                {optimális felváltásában}
22     end else                     {Opt[x]=Opt(x,i-1)}
23       V[x,i]:=V[x,i-1];        {nincs jobb, mint i-1-re}
24   end {for x};

25 Db:=0; x:=E; i:=N;
26 if Opt[E]<=N then                 { van megoldás }
27   repeat                           { egy optimális megoldás előállítása }
28     i:=V[x,i];                     {i. szerepel x opt. felváltásában}
29     Inc(Db); C[Db]:=i;             {i bejegyzése a megoldásba}
30     x:=x-P[i];                     {x-P[i] opt. felváltását nézzük}
31     Dec(i);                         { az 1..i-1 pénzekkel}
32   until x=0;
33 end {OptValto};

```

Az OPTVALTO algoritmus tárigénye  $E * (N + 1) * 2$  byte. A futási idő szintén  $E * N$ -el arányos.

## A dinamikus programozás stratégiája.

A dinamikus programozás, mint probléma-megoldási stratégia az alábbi öt lépés végrehajtását jelenti.

1. Az [optimális] megoldás szerkezetének elemzése.
2. Részproblémákra és összetevőkre bontás úgy, hogy:
  - a) Az összetevőktől való függés körmentes legyen.
  - b) Minden részprobléma [optimális] megoldása kifejezhető legyen (rekurzívan) az összetevők [optimális] megoldásaival.
3. Részproblémák [optimális] megoldásának kifejezése (rekurzívan) az összetevők [optimális] megoldásaiból.
4. Részproblémák [optimális] megoldásának kiszámítása alulról-felfelé haladva:
  - a) A részproblémák kiszámítási sorrendjének meghatározása. Olyan sorba kell rakni a részproblémákat, hogy minden  $p$  részprobléma minden összetevője (ha van) előbb szerepeljen a felsorolásban, mint  $p$ .
  - b) A részproblémák kiszámítása alulról-felfelé haladva, azaz táblázat-kitöltéssel.
5. Egy [optimális] megoldás előállítása a 4. lépésben kiszámított (és tárolt) információkból.

### 4.4. Feladat: Testvéries osztozkodás (CEOI'95)

Két testvér ajándékokon osztozkodik. Minden egyes ajándékot pontosan ez egyik testvérnek kell adni. Minden ajándéknak pozitív egész számmal kifejezett értéke van. Jelölje  $A$  az egyik,  $B$  pedig a másik testvér által kapott ajándékok összértékét. A cél az, hogy az osztozkodás testvéries legyen, tehát  $A$  és  $B$  különbségének abszolútértéke minimális legyen.

Írjunk programot, amely kiszámítja a testvéries osztozkodás eredményeként keletkező  $A$  és  $B$  értékeket és megadja, hogy mely ajándékokat kapja a két testvér.

**Megoldás.**

Legyen  $\{e_1, \dots, e_n\}$  az ajándékok értékeinek egy felsorolása és jelölje  $E$  az összegüket. Feltehetjük, hogy  $A \leq B$ . Mivel  $A + B = E$ , ezért  $A$  a legnagyobb olyan szám, amelyre  $A \leq E/2$  és előállítható  $\{e_1, \dots, e_n\}$  egy részhalmazának összegeként.

Tehát a megoldás visszavezethető a pénzváltás probléma megoldására.

**4.5. Feladat: Igazságos osztozkodás**

Két testvér közösen kapott ajándékokat. Minden ajándéknak tudják a használati értékét, ami pozitív egész szám. Igazságosan el akarják osztani az ajándékokat, tehát úgy, hogy mind kettőjük ugyanannyi összértékűt kapjon. Észrevették, hogy ez nem feltétlenül teljesíthető, ezért elfogadnak olyan elosztást is, amely szerint a közösen is maradhat kinemosztott ajándék, de ragaszkodnak ahhoz, hogy mindketten azonos összértéket kapjanak, és a közösen maradt ajándékok összértéke a lehető legkisebb legyen.

Írjon programot, amely megad egy igazságos osztozkodást!

**Példa bemenet és kimenet**

**bemenet**

6  
10 3 12 5 15 6

**Megoldás**

**kimenet**

15  
6 3  
4 2 1

$$m = a_{i_1} + \dots + a_{i_u}$$

$$m = a_{j_1} + \dots + a_{j_v}$$

$$\{i_1, \dots, i_u\} \cap \{j_1, \dots, j_v\} = \emptyset$$

$$\sum_{i=1}^n a_i - 2m \rightarrow \text{minimális}$$

Nyilvánvalóan  $m \leq fel = \sum_{i=1}^n a_i / 2$ . Minden  $0 \leq x, y \leq fel$ -re és minden  $1 \leq n$ -re tekintjük azt a részproblémát, hogy előállítható-e legfeljebb az első  $i$  ajándék összegeként mind  $x$ , mind  $y$ , de minden szám legfeljebb egyik összegben szerepelhet. Legyen  $E(x, y, i)$  igaz, ha előállítható, egyébként hamis.

$E(0, 0, i) = igaz$  minden  $0 \leq i \leq n$ -re. Ha  $i > 0$ , akkor az alábbi rekurzív összefüggés adható:

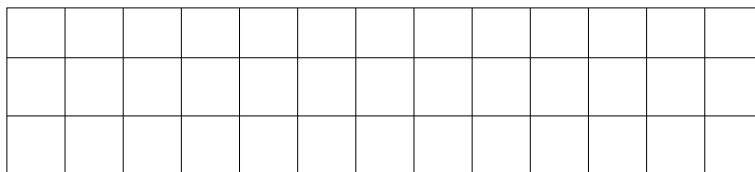
$$V(x, y, i) \Leftrightarrow \begin{cases} V(x, y, i - 1) \vee \\ a_i \leq x \wedge V(x - a_i, y, i - 1) \vee \\ a_i \leq y \wedge V(x, y - a_i, i - 1) \end{cases} \quad (3)$$

A megoldás értéke az a legnagyobb  $x$ , amelyre  $E(x, x, n) = igaz$ . Egy megoldás a pénzváltáshoz hasonlóan állítható elő.

**4.6. Feladat: Járdakövezés**

Számítsuk ki, hogy hányféleképpen lehet egy  $3 \times n$  egység méretű járdát kikövezni  $1 \times 2$  méretű lapokkal!

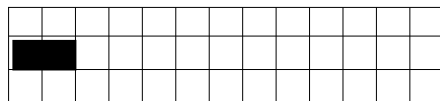
**Megoldás**



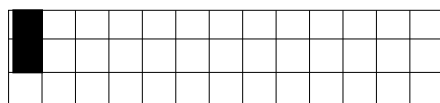
3. ábra.

Jelölje  $A(n)$  a megoldás értékét  $3 \times n$  egység méretű járda esetén.

Az első oszlop középső négyzete háromféleképpen fedhető le.



4. ábra. 1. eset



5. ábra. 2. eset

Az egyes esetek csak az alábbi módon folytathatók:

Jelölje  $B(n)$  azt, hogy hányféleképpen fedhető le egy  $3 \times n$  egység méretű járda, amelynek a bal alsó sarka már le van fedve. Szimmetria miatt a jobb felső sarok lefedettsége esetén is  $B(n)$ -féle lefedés van.

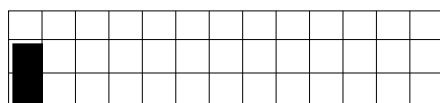
$$A(n) = \begin{cases} 0 & \text{ha } n = 1 \\ 3 & \text{ha } n = 2 \\ A(n-2) + 2B(n-1) & \text{ha } n > 2 \end{cases} \quad (4)$$

$$B(n) = \begin{cases} 1 & \text{ha } n = 1 \\ 0 & \text{ha } n = 2 \\ A(n-1) + B(n-2) & \text{ha } n > 2 \end{cases} \quad (5)$$

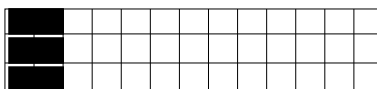
```

1 program járda;
2 function B(n: integer): longint; forward;
3 function A(n: integer): longint;
4 begin
5   if (n=1) then
6     A := 0
7   else if (n=2) then
8     A := 3
9   else
10    A := A(n-2)+2*B(n-1);
11 end{A};
12
13 function B(n: integer): longint;
14 begin
15   if (n=1) then
16     B := 1
17   else if (n=2) then
18     B := 0
19   else
20     B := A(n-1)+B(n-2);
21 end{B};
22 begin

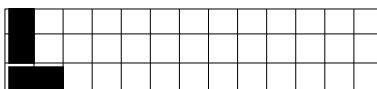
```



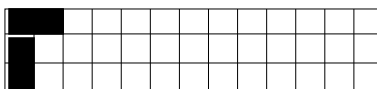
6. ábra. 3. eset



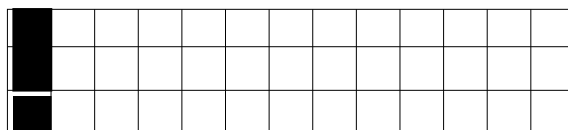
7. ábra. Az 1. eset csak így folytatható



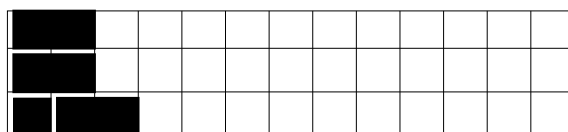
8. ábra. A 2. eset csak így folytatható



9. ábra. A 3. eset csak így folytatható



10. ábra. Az 1. eset csak így folytatható



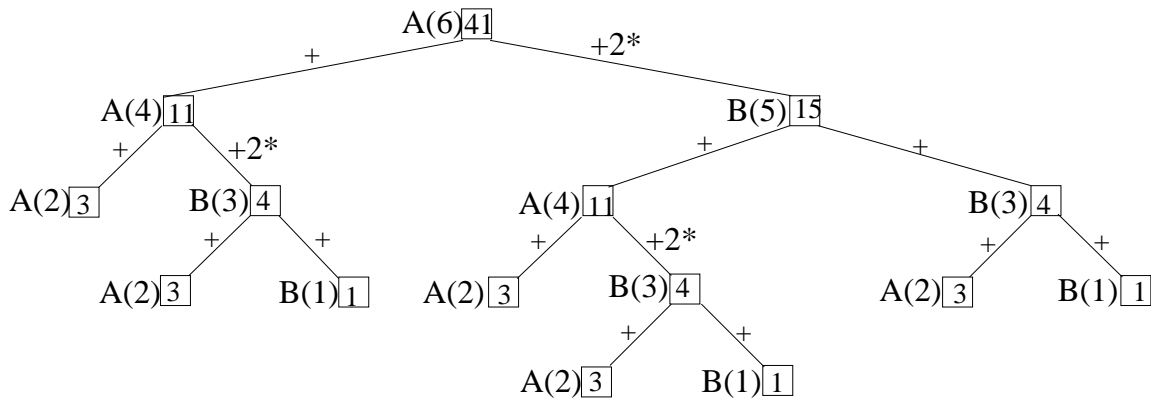
11. ábra. Az 2. eset csak így folytatható



```

23  writeln(A(32));
24  end.

```



12. ábra. Rekurziós fa

```

1  program jarda2;
2  {megoldás din. prog. módszerrel (táblázat-kitöltéssel)}
3  const
4    maxN=100;
5  var
6    A,B: array [1..maxN] of Int64;
7    n,i: integer;
8  begin
9    n:=64;
10   A[1]:=0; A[2]:=3;
11   B[1]:=1; B[2]:=0;
12   for i:= 3 to n do begin
13     A[i]:=A[i-2]+2*B[i-1];
14     B[i]:=A[i-1]+B[i-2];
15   end;
16   writeln('A(',n,')_=_',A[n]);
17 end.

```

A(64) = 1582048049556775361

#### 4.7. Feladat: Tükörszó (IOI'2000)

Egy karaktorsorozatot tükörszónak nevezünk, ha balról-jobbra, valamint jobbról-balra olvasva megegyezik. Például görög, egészsége, mesélésem.

Írjunk olyan programot, amely kiszámítja, hogy egy adott szóból minimálisan hány betűt kell törölni, hogy tükörszót kapjunk.

##### Bemenet

A tukorszo.be szöveges állomány első és egyetlen sora egy  $S$  szót tartalmaz, amelynek hossza legfeljebb 5000, és  $S$  minden  $c$  karakterére: ' $d' \leq c \leq 'z'$ ' és ' $A' \leq c \leq 'Z'$ '.

##### Kimenet

A tukorszo.ki szöveges állomány első és egyetlen sora egy  $m$  nemnegatív egész számot tartalmazzon, ami a minimális törölendő karakterek száma, amellyel a bemeneti  $S$  szó tükörszóvá tehető.

## Megoldás

Vegyük észre, hogy egy  $S$  szó akkor és csak akkor tükörszó, ha vagy üres szó, vagy egybetűs, vagy az első és utolsó betűje megegyezik és ezeket elhagyva ismét tükörszót kapunk.

### Az optimális megoldás szerkezetének elemzése.

Minden  $S$  szóra jelölje  $T(S)$  a probléma egy megoldását, tehát olyan tükörszót, amely  $S$ -ből a lehető legkevesebb betű törlésével kapható. Ilyen biztosan létezik, hiszen egy kivételével minden betűt törölve tükörszót kapunk. Ha  $S$  egy betűből áll, akkor maga is tükörszó,  $T(S) = S$ . Legyen  $S = xRy$ , ahol  $x$  az első,  $y$  pedig az utolsó betűje  $S$ -nek ( $R$  lehet üres szó is). Ha  $x = y$ , akkor  $T(S) = T(R)$ . Ha  $x \neq y$ , akkor vagy az  $x$ , vagy az  $y$  betűt biztosan törölni kell, tehát a megoldás vagy  $T(xR)$ , vagy  $T(Ry)$ . Az optimális megoldás szerkezete azt sugallja, hogy minden  $(i, j)$ ,  $1 \leq i \leq j \leq n$  indexpárra tekintsük azt a részproblémát, hogy az  $S[i..j] = S[i]...S[j]$  szó legkevesebb hány betű törlésével tehető tükörszóvá. Jelölje az  $(i, j)$  részprobléma megoldását  $M(i, j)$ . Tehát a kitűzött feladat megoldása  $M(1, n)$ .

### A részproblémák megoldásának kifejezése az összetevők megoldásaival.

Ha  $i > j$  esetre  $M(i, j)$  értékét 0-ként értelmezzük, akkor a részproblémák megoldásai között az alábbi rekurzív összefüggést lehet felírni.

$$M(i, j) = \begin{cases} 0, & \text{ha } i \geq j \\ M(i+1, j-1), & \text{ha } i < j \text{ és } S[i] = S[j] \\ 1 + \min(M(i+1, j), M(i, j-1)), & \text{ha } i < j \text{ és } S[i] \neq S[j] \end{cases}$$

Tehát az  $(i, j)$  részprobléma összetevői:  $(i+1, j-1)$ ,  $(i+1, j)$  és  $(i, j-1)$ .

### A részproblémák kiszámítási sorrendje, táblázat-kitöltés.

Tároljuk a részproblémák megoldását táblázatban, az  $(i, j)$  megoldását az  $M[i, j]$  táblázatelemben. Mivel az  $(i, j)$  részprobléma megoldása legfeljebb az  $(i+1, j-1)$ ,  $(i+1, j)$  és  $(i, j-1)$  megoldásaitól függ, ezért a táblázat-kitöltés sorrendje lehet alulról felfelé, soronként pedig jobbról-balra haladó.

A négyzetes táblázat tárigénye  $5000 * 5000 * 2 = 50000000$  byte, ami túl sok. Látható azonban, hogy elegendő lenne csak két egymást követő sort tárolni. Sőt, egy sort is elég tárolni, ha megoldjuk, hogy ne írjuk felül azt a  $T[i] = M(i, j)$  kitöltésekor az  $M(i, j-1)$  értéket, amit szintén  $T[i]$  tárol.

n									0
								0	
							0		
j		?	x				0		
j-1		x	x			0			
					0				
				0					
			0						
		0							
1	0								
	1	i	i+1						n

13. ábra. Táblázat-kitöltési sorrend: soronként alulról felfelé, jobbról balra haladva.

```
1 function Megold(const S:Sorozat; N:word):word;
2 {Megoldás lineáris táblázat-kitöltéssel}
3 var
4   T:array[1..MaxN] of word;
5   i,j:integer;
6   Ment, Menti:word;
7 begin
8   T[1]:=0;
9   for j:=2 to N do begin
10    T[j]:=0; Menti:=0; {Minti:=M(i+1,j-1)}
```

```

11   for i:=j-1 downto 1 do begin
12     {M(i,j) szamitasa es tarolasa T[i]-ben}
13     Ment:=T[i];
14     if S[i]=S[j] then
15       T[i]:=Menti {Menti=M(i+1,j-1)}
16     else {M(i,j)=1+Min (M(i,j-1)+M(i+1,j))}
17       T[i]:=1+Min(T[i], T[i+1]);
18     Ment:=Menti;
19   end{for i};
20 end{for j};
21 Megold:=T[1]; {=M(1,N)}
22 end{Szamit};

```

Az algoritmus futási ideje  $\Theta(n^2)$ , tárigénye  $\Theta(n)$ .

Ha egy megoldást is elő kell állítani, akkor minden  $(i, j)$  részproblémára tarolni kell azt az információt, hogy melyik összetevőre kapjuk az optimális megoldást ha  $S[i] \neq S[j]$ .

Vagy minden  $(i, j)$  részproblémára taroljuk  $M(i, j)$  értékét, és ekkor  $M(i+1, j) < M(i, j-1)$  összehasonlítással megadható, hogy az  $i$ -edik (első), avagy a  $j$ -edik (utolsó) betűt kell-e törölni, vagy egy külön  $L$  tömbben tároljuk, hogy a részsorozat melyik végéről kell törölni az optimális megoldáshoz. Ezt nevezzük az optimális megoldás visszafejtésének.

Ekkor algoritmus futási ideje is és tárigényre is  $\Theta(n^2)$ .

```

1  procedure KiIr;
2  {Global: T, S, N}
3  var
4    Bal, Jobb: word;
5  begin
6    Bal:=1; Jobb:=N;
7    while Bal<Jobb do begin
8      if S[Bal]=S[Jobb] then begin
9        Inc(Bal); Dec(Jobb)
10     end else if T[Bal+1,Jobb] < T[Bal, Jobb-1] then begin
11       Write(KiF, Bal, ' ');
12       Inc(Bal);
13     end else begin
14       Write(KiF, Jobb, ' ');
15       Dec(Jobb);
16     end;
17   end{while}
18
19 end{KiIr};

```

#### 4.8. Feladat: Számjáték (IOI'96)

Tekintsük a következő kétszemélyes játékot. A játéktábla pozitív egész számok sorozata. A két játékos felváltva lép. Egy lépés azt jelenti, hogy a játékos a sorozat bal, avagy jobb végéről levesz egy számot. Az levett szám hozzáadódik a pontszámához. A játék akkor ér véget, ha a számok elfogytak. Az első játékos nyer, ha az általa választott számok összege legalább annyi, mint a második játékos által választottak összege. A második játékos a lehető legjobban játszik. A játékot az első játékos kezdi. Ha kezdetben a táblán levő számok száma páros, akkor az első játékosnak van nyerő stratégiája.

Írjunk olyan programot, amely az első játékos szerepét játssza és megnyeri játékot! A második játékos lépéseit egy már adott számítógépes program szolgáltatja. A két játékos a rendelkezésedre bocsátott `Play` modul három eljárásán keresztül kommunikál egymással.

**StartGame** Az első játékos a játszmat a paraméter nélküli `StartGame` eljárás végrehajtásával indítja.

**MyMove** Ha az első játékos a bal oldalról választ számot, akkor a `MyMove ('L')` eljárást hívja. Hasonlóképpen a `MyMove ('R')` hívással közli a második játékosal, hogy a jobb oldalról választott.

**YourMove** A második játékos (tehát a gép) azonnal lép. Az első játékos a lépést a `YourMove (C)` utasítással tudhatja meg, ahol `C` egy karakter típusú változó. (`C/C++` nyelven `YourMove (&C)`). A `C` változó értéke `'L'` vagy `'R'` lesz attól függően, hogy a második játékos a bal vagy a jobb oldalról választott.

## Bemenet

Az `input.txt` fájl első sora a kezdőtábla  $n$  méretét (a számok darabszámát) tartalmazza.  $n$  páros és  $2 \leq n \leq 100$ . A második sor  $n$  számot tartalmaz, a játék kezdetén a táblán lévő számokat. A táblán 200-nál nagyobb szám nem szerepel.

## Kimenet

Ha a játék véget ért, akkor a programod írja ki a végeredményt az `OUTPUT.TXT` fájlba! A fájl első sorában két szám legyen! Az első szám az első játékos által választott számok összegével, a második szám a második játékos által választott számok összegével egyezzen meg! A programodnak a játékot le kell játszania és az output a lejátszott játék eredményét kell tartalmazza.

## Példa bemenet és kimenet

INPUT.TXT

6  
4 7 2 9 5 2

### Megoldás

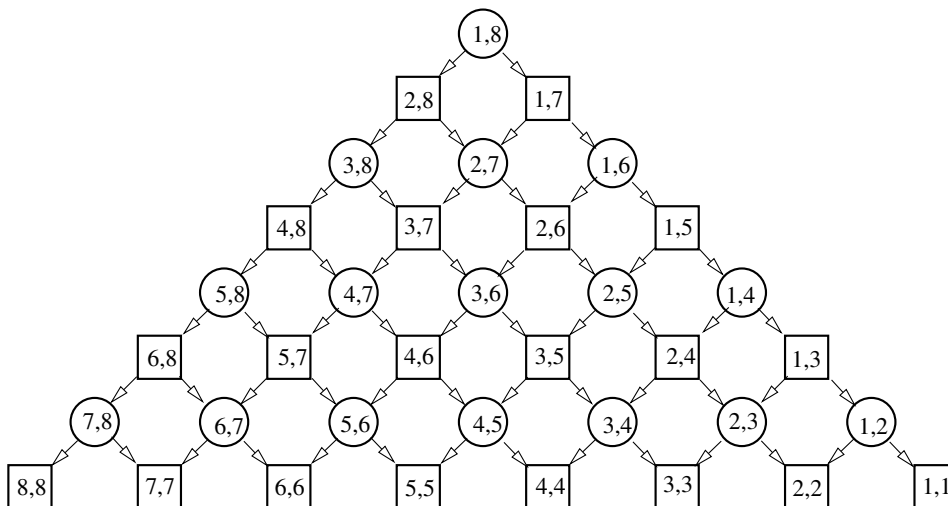
Jelölje  $\langle a_1, \dots, a_n \rangle$  a kezdeti játékállást. Minden lehetséges játékállást egyértelműen meghatározza az, hogy mely számok vannak még a táblán. Tehát minden játékállás azonosítható egy  $(i, j)$  számpárral, ami azt jelenti, hogy a táblán az  $\langle a_i, \dots, a_j \rangle$  számsorozat van. Mivel  $n$  páros szám, így minden esetben, amikor az első játékos lép, vagy  $i$  páros és  $j$  páratlan, vagy fordítva. Tehát az első játékos kényszerítheti a második játékos, hogy az mindig vagy csak páros, vagy csak páratlan indexű elemét válassza a számsorozatnak. Tehát ha a páros indexűek összege nagyobb, vagy egyenlő, mint a páratlanok összege, akkor az első játékos mindig páratlan indexűt választ, egyébként mindig párosat.

OUTPUT.TXT

18 11

Érdekesebb a játék, ha az a cél, hogy az első játékos a lehető legtöbbet szerezze meg, feltéve, hogy erre törekszik a második játékos is.

Ábrázoljuk a játékállásokat gráffal.



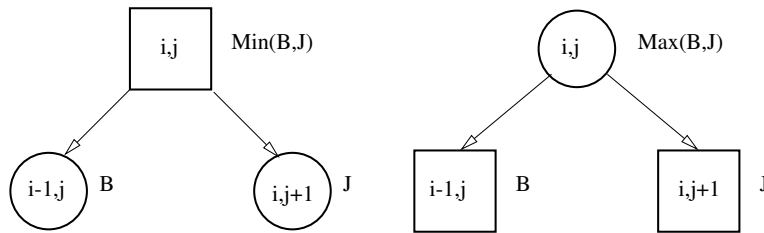
14. ábra. A játékállások gráfja  $n = 8$  esetén. Körrel jelölt állásból  $(i + j)$  páratlan) az első, négyzettel jelölt állásból  $(i + j)$  páros) a második játékos lép.

Definiáljuk minden  $(i, j)$  játékállásra azt a maximális pontszámot, amit az első játékos elérhet ebből a játékállásból indulva. Jelölje ezt az értéket  $Opt(i, j)$ .

$Opt(i, j)$  a következő rekurzív összefüggéssel számítható.

$$Opt(i, j) = \begin{cases} 0 & \text{ha } i = j \\ \max(a_i + Opt(i + 1, j), a_j + Opt(i, j - 1)) & \text{ha } i < j \text{ és } i + j \text{ páratlan} \\ \min(Opt(i + 1, j), Opt(i, j - 1)) & \text{ha } i < j \text{ és } i + j \text{ páros} \end{cases}$$

Tehát alkalmazható a dinamikus programozás módszere, vagyis az  $Opt(i, j)$  értékeket a játék megkezdése előtt kiszámítjuk. Tároljuk minden  $(i, j)$  játékállásra a  $Lep[i, j]$  tömbelemben az optimális lépést, tehát az 'L' karaktert, ha a képletben  $a_i + Opt(i + 1, j) > a_j + Opt(i, j - 1)$ , mert ekkor balról kell elvenni, egyébként pedig az 'R' karaktert, mert ekkor jobbról kell elvenni.



15. ábra. Mini-max szabály.

```

1 Program Jatek;
2 Uses Play; {a második játékost megvalósító modul}
3 const
4   MaxN=100;
5 var
6   InpF,OutF: Text;
7   A:array [1..MaxN] of word;{ a táblán lévő számok sorozata}
8   N: word;           { a tábla mérete}
9   Opt:array [1..MaxN,1..MaxN] of word;
10  Lt:array [1..MaxN,1..MaxN] of char; {az 1. játékos optimális lépései}
11
12 procedure Beolvas; var i:word; begin
13   Assign(InpF, 'input.txt '); Reset(InpF);
14   ReadLn(InpF,N);
15   for i:=1 to N do
16     Read(InpF,A[i]);
17   Close(InpF);
18 end;
19
20 procedure Elofeldolgoz;
21 var i,j:word;
22   Pont,PontBal,PontJobb:word;
23 begin
24   for j:=1 to N do begin
25     Opt[j,j]:=0;
26     for i:=j-1 downto 1 do begin
27       if Odd(j-i+1) then begin{2. játékos lép}
28         if Opt[i+1,j]<Opt[i,j-1] then
29           Opt[i,j]:=Opt[i+1,j]
30         else
31           Opt[i,j]:=Opt[i,j-1]
32       end else begin{1. játékos lép}
33         PontBal:=A[i]+Opt[i+1,j];
34         PontJobb:=A[j]+Opt[i,j-1];
35         if PontBal>PontJobb then begin
36           Opt[i,j]:=PontBal; Lt[i,j]:= 'L'
37         end else begin
38           Opt[i,j]:=PontJobb; Lt[i,j]:= 'R'
39         end
40       end;
41     end{for i};
42   end{for j};
43 end {Elofeldolgoz};
44
45 procedure Jatszaz;
46 var

```

```

45  Bal ,Jobb :word ;{az aktuális játékállás: A[Bal..Jobb]}
46  L1,L2: Char;  {a két játékos aktuális lépése}
47  begin
48  Bal:=1; Jobb:=N;           {a kezdő játékállás beállítása}
49  while Bal<=Jobb do begin  {amíg nem üres a tábla}
50  MyMove(Lt[Bal , Jobb]);   {az én lépésem}
51  if Lt[Bal , Jobb]='L' then {a játékállás aktualizálása}
52  Inc(Bal)
53  else
54  Dec(Jobb);
55  L2:=YourMove;           {az ellenfél lépése}
56  if L2='L' then         {a játékállás aktualizálása}
57  Inc(Bal)
58  else
59  Dec(Jobb);
60  end{while};
61  end{Jatszias};

62  begin
63  Beolvas;
64  Elofeldolgoz;
65  StartGame;
66  Jatszias;
67  end.

```

#### 4.9. Feladat: Vágás

Adott egy fémrúd, amelyet megadott számú darabra kell felvágni úgy, hogy a vágások pontos helyét is tudjuk. A vágások helyét a rúd egyik végétől mért, milliméterben kifejezett értékek adják meg. Olyan vágógéppel kell a feladatot megoldani, amely egyszerre csak egy vágást tud végezni. A vágások tetszőleges sorrendben elvégezhetőek. Egy vágás költsége megegyezik annak a darabnak a hosszával, amit éppen (két darabra) vágunk. A célunk optimalizálni a műveletsor teljes költségét.

Írjunk olyan programot, amely kiszámítja a vágási műveletsor optimális összköltségét, és megad egy olyan vágási sorrendet, amely optimális költséget eredményez.

#### Bemenet

A `vag.be` szöveges állomány első sora egyetlen egész számot tartalmaz, a vágandó rúd  $h$  hosszát ( $0 < h \leq 1000$ ). A második sorban az elvégzendő vágások  $n$  száma van ( $1 \leq n \leq 1000$ ). A harmadik sor  $n$  darab egész számot tartalmaz egy-egy szóközzel elválasztva, az elvégzendő vágások helyeit. A számok szigorúan monoton növekvő sorozatot alkotnak, és mindegyik nagyobb, mint 0 és kisebb, mint  $h$ .

#### Kimenet

A `vag.ki` szöveges állomány első sorába egyetlen számot, a vágási műveletsor optimális összköltségét kell írni! A második sor  $n$  darab egész számot tartalmazzon, ami a vágási helyek sorszámainak egy olyan felsorolása legyen, hogy ebben a sorrendben elvégezve a vágásokat, az összköltség optimális lesz.

#### Példa bemenet és kimenet

**Bemenet**

24

Megoldás.

**Kimenet**

70

2 1 4 3 7 5 6

**Az optimális megoldás szerkezetének vizsgálata.**

Ha az optimális vágás során először a  $v_k$  ( $1 \leq k \leq n$ ) helyen történik a vágás, akkor az első darabon a  $v_0, v_1, \dots, v_{k-1}, v_k$ , a másodikon pedig  $v_k, v_{k+1}, \dots, v_n, h$  vágásoknak is optimálisnak kell lenni.

**Az optimális megoldás értékének rekurzív kifejezése.**

Legyen minden  $i, j$  párra, ( $0 \leq i < j \leq n+1$ )  $Opt(i, j)$  a  $v_i$ , vágási helytől a  $v_j$ , vágási hely által meghatározott rúddarab optimális vágásának költsége.

$$Opt(i, j) = \begin{cases} 0, & \text{ha } j = i + 1 \\ v_j - v_i + \min_{k=i+1}^{j-1} (Opt(i, k) + Opt(k, j)) & \text{ha } i < j + 1 \end{cases}$$



```

17 Assign(BeF, 'vag.be'); Reset(BeF);
18 ReadLn(BeF, H);
19 ReadLn(BeF, N);
20 for i:=1 to N do
21     Read(BeF, V[i]);
22 Close(BeF);
23 V[0]:=0; V[N+1]:=H;
24 end{Beolvas};

25 procedure Szamit; {Global: N, V, S}
26 var
27     Opt:array[0..MaxN+1,0..MaxN+1] of longint;
28     i,j,k,u,G:word;
29     Min,Uj:longint;
30 begin{Szamit}
31     for i:=0 to N do begin{inicializalas}
32         Opt[i,i+1]:=0; S[i,i+1]:=0;
33     end{for i};
34     for u:=2 to N+1 do begin{j-i=u}
35         for i:=0 to N-u+1 do begin
36             j:=i+u; Min:=Inf;
37             for k:=i+1 to j-1 do begin
38                 Uj:=Opt[i,k]+Opt[k,j];
39                 if Uj <Min then begin
40                     Min:=Uj;
41                     G:=k;
42                 end;
43             end{k};
44             Opt[i,j]:=Min+V[j]-V[i];
45             S[i,j]:=G;
46         end{i};
47     end{for u};
48     Kolts:=Opt[0,N+1];
49 end{Szamit};

50 procedure KiIr;
51 var
52     KiF:Text;
53     Ind,i:word;
54 procedure Bejar(i,j:word);
55 var k:word;
56 begin
57     if j<=i+1 then Exit;
58     k:=S[i,j];
59     write(KiF, k, ' ');
60     Bejar(i,k);
61     Bejar(k,j);
62 end{Bejar};
63
64 begin{KiIr}
65     Assign(KiF, 'vag.ki'); Rewrite(KiF);
66     WriteLn(KiF, Kolts);
67     Ind:=0;
68     Bejar(0,N+1);
69     WriteLn(KiF);
70     Close(KiF);
71 end{KiIr};

```



```

72 begin {program}
73   Beolvas;
74   Szamit;
75   KiIr;
76 end.

```

#### 4.10. Feladat: Torony építése kockákból.

Építőkövekből úgy lehet stabil tornyot építeni, hogy kisebb kockára nem lehet nagyobb, illetve könnyebb kockára nem lehet nehezebbet tenni.

Adjunk olyan algoritmust, amely adott  $N$  darab kocka alapján megadja a belőlük építhető legmagasabb tornyot!

#### Bemenet

A `torony.be` állomány első sorában a kockák  $n$  ( $1 \leq n \leq 1000$ ) száma van, a további  $n$  sorban, pedig az egyes kockák oldalhossza és súlya (mindkettő 20000-nél kisebb pozitív egész szám), egyetlen szóközzel elválasztva. Nincs két kocka, amelynek oldalhossza és a súlya is megegyezne.

#### Kimenet

A `torony.ki` állomány első sorába a legmagasabb torony  $k$  kockaszámát kell írni, a következő  $k$  sorba pedig az építés szerint alulról felfelé sorrendben a felhasznált kockák oldalhosszát és súlyát.

#### Példa bemenet és kimenet

Bemenet	Kimenet
5	3
10 3	20 5
20 5	10 3
15 6	10 2
15 1	
10 2	

#### Az optimális megoldás szerkezetének vizsgálata.

A kockák oldalhosszai:  $h_1, \dots, h_n$ , súlyai pedig  $s_1, \dots, s_n$ .

Elemezzük az optimális megoldás szerkezetét.

Tegyük fel, hogy a  $i_1, \dots, i_k$  sorszámú kockák ebben a sorrendben egymásra rakásával kapjuk a legmagasabb tornyot. Ekkor  $i_2, \dots, i_k$  torony a lehető legmagasabb olyan torony, amelynek legalsó kockája  $i_2$ . Mert ha lenne magasabb torony, amelynek legalsó kockája  $i_2$ , akkor ezt a  $i_1$  kockára rárakhatnánk, hisz a  $i_1$  kocka biztosan nem szerepelhet olyan toronyban, amelynek legalsó kockája  $i_2$ , és így magasabb tornyot kapnánk, mint a  $i_1, \dots, i_k$ . Ez azért igaz, mert az a gráf, amelynek pontjai a kockák (sorszámai), és élei azok a  $(i, j)$  párok, amelyekre igaz, hogy az  $i$ -edik kockára rárakható a  $j$ -edik,  $(h_i \geq h_j \wedge s_i \geq s_j)$  körmentes gráf.

#### Részproblémákra és összetevőkre bontás.

Minden  $i$ -re ( $1 \leq i \leq n$ ) vegyük azt a részproblémát, hogy mekkora a magassága a legmagasabb olyan toronynak, amelynek legalsó kockája az  $i$ . Jelölje  $M(i)$  ezt a legmagasabb toronymagasságot, tehát a részprobléma optimális megoldásának az értékét.

**A részproblémák megoldásának kifejezése az összetevők megoldásaival.**

$$M(i) = h_i + \max(M(j) : i \neq j \wedge h_i \geq h_j \wedge s_i \geq s_j)$$

#### A részproblémák kiszámítási rekurzióval, memorizálva.

Az optimális megoldás értékét rekurzióval számítjuk a fenti kifejezés alapján, de ha egy részproblémára kiszámítottuk, akkor azt eltároljuk egy táblázatban, és ha később ismét szükség lesz rá, akkor a táblázatból olvassuk ki az értéket. Ehhez először a táblázatot olyan értékkel kell feltölteni, amely azt jelzi, hogy a megfelelő részprobléma értékét még nem számítottuk ki. Ez az érték lehet 0, mivel minden megoldás tartalmazza azt a kockát, tehát az optimális megoldás értéke  $> 0$ .

Ezzel a módszerrel elkerülhető a részproblémák megfelelő sorrendjének kiszámítása. Ez csökkentheti a kivitelezési időt és néha a futási időt is. Az algoritmus tárigenye  $n$ -el arányos, futási ideje pedig legrosszabb esetben  $n^2$ -el.

```

1 procedure Kocka(const H,S:Tomb; N:word; var Also:word; var Ra:Tomb);
2   const MaxN=1000;
3   var
4     MagasMem:array[1..MaxN] of longint;
5     MaxM:longint;      {az építhető max. toronymagasság}
6     i:integer;
7   function Magas(i:word):longint; {Global:N, S, H, MagasMem }
8   var
9     Mi,Mj,j:longint;
10  begin{Magas}
11    if MagasMem[i]>0 then begin {p-re már kiszámítottuk}
12      Magas:=MagasMem[i]; Exit {táblázatból vesszük }
13    end;
14    Mi:=0;
15    for j:=1 to N do { minden q-ra, amely rakható p-re }
16      if (i<>j)and(H[i]>=H[j])and(S[i]>=S[j]) then begin
17        Mj:=Magas(j); { Magas(q) rekurzív számítása }
18        if Mj>Mi then begin { Maximum számítás }
19          Mi:=Mj;
20          Ra[i]:=j; { bejegyzés: i-re j-t kell akni }
21        end;
22      end;
23      MagasMem[i]:=Mi+H[i]; { Memorizálás }
24      Magas:=MagasMem[i];
25    end{Magas};

26 var
27   KiF:Text; k:integer;
28 begin{kocka}
29   for i:=1 to N do {inicializálás, még semmit nem}
30     MagasMem[i]:=0 {számítottunk ki}
31   MaxM:=0;
32   for i:=1 to N do begin
33     Magas(i);
34     if MaxM<MagasMem[i] then begin{ hol a maximum? }
35       MaxM:=MagasMem[i];
36       Also:=i;
37     end;
38   end{for i};
39   assign(KiF,'kocka.ki'); rewrite(KiF); {kiíratás}
40   k:=0; i:=Also;
41   while i>0 do begin
42     inc(k); i:=Ra[i];
43   end;
44   writeln(KiF,k);
45   while Also>0 do begin
46     write(KiF, Also, ' ');
47     Also:=Ra[Also];
48   end;
49   writeln(KiF); close(KiF);
50 end;{kocka}

```

## 4.11. Feladat: Kitalálós játék

Ádám és Éva kitalálós játékot játszik. Éva gondol egy 1 és  $n$  közötti egész számot, amelyet Ádámnak ki kell találnia. Ádám olyan kérdést tehet fel, hogy "A gondolt szám kisebb-e, mint  $x$ ". Éva válasza "igen", vagy "nem" lehet. Hogy a játék érdekesebb legyen, megállapodtak abban, hogy Ádám legfeljebb  $h$ -szor tehet fel olyan kérdést, amelyre a válasz "nem", tehát ha már  $h$  kérdésére "nem" választ kapott, akkor tovább nem kérdezhet.

Írjon programot, amely  $n$  és  $h$  ismeretében kiszámítja azt a legkisebb  $k$  értéket, amelyre teljesül, hogy Ádám bármely 1 és  $n$  közötti gondolt számot ki tud találni legfeljebb  $k$  kérdéssel úgy, hogy legfeljebb  $h$ -szor kap "nem" választ!

### Bemenet

A `be.txt` szöveges állomány első sorában két egész szám van, az  $n$  értéke ( $1 \leq n \leq 2000000000$ ) és a  $h$  ( $2 \leq h \leq 100$ ) értéke.

### Kimenet

A `ki.txt` szöveges állomány első és egyetlen sorába egy számot kell írni, azt a minimális  $k$  értéket, amelyre teljesül, hogy Ádám bármely 1 és  $n$  közötti gondolt számot ki tud találni legfeljebb  $k$  kérdéssel úgy, hogy legfeljebb  $h$ -szor kap "nem" választ!

#### Példa bemenet és kimenet

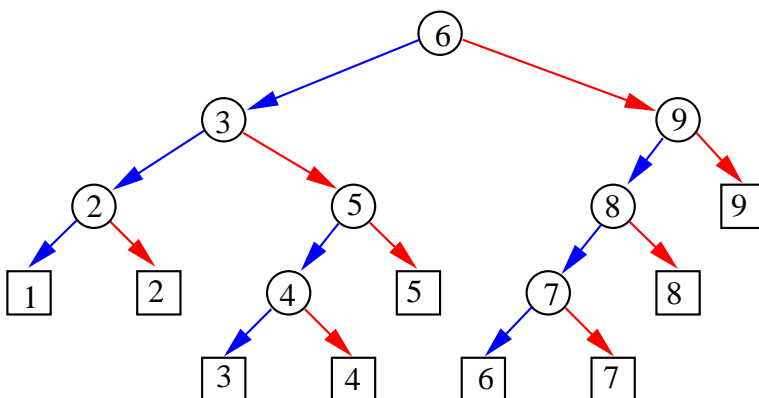
9 2

4

#### Megoldás

Minden olyan bináris fa, amely teljesíti az alábbi három feltételt, kifejez egy olyan kérdezősi stratégiát, amely során legfeljebb  $h$  kérdésre kaphatunk nem választ. Fordítva is igaz, tehát minden olyan kérdezősi stratégia, amely során legfeljebb  $h$  kérdésre kaphatunk nem választ, kifejezhető ilyen fával.

- A fának  $n$  levele van és ezek balról jobbra sorrendben az  $1, \dots, n$  számokat tartalmazzák.
- A fának  $n - 1$  belső pontja van. Minden  $p$  belső pont a  $p$  jobb-részfájában lévő levél értékek minimumát tartalmazza.
- Bármely gyökértől levélig vezető úton legfeljebb  $h$ -szor megyünk jobbra.



19. ábra. Egy 2-hibázó kérdezőfa a példa bemenetre.

Kérdezőfa a következőképpen használható. Kezdetben a  $p$  aktuális pont legyen a fa gyökere. Mindaddig, amíg  $p$  nem levél, kérdezzünk rá a  $p$ -ben lévő értékre. Ha a válasz igen, akkor  $p$  legyen a bal fia, egyébként a jobb fia. Az ismétlés befejeződése után a gondolt szám  $p$ -ben van.

Adott kérdezőfát használva a legrosszabb esetben annyi kérdést kell feltenni, amennyi a fa magassága. Tehát az a kérdés, hogy adott  $n$  és  $h$  esetén mekkora a legkisebb magasságú olyan kérdezőfa magassága, amelynek legalább  $n$  levele van és bármely gyökértől levélig vezető úton legfeljebb  $h$ -szor megyünk jobbra.

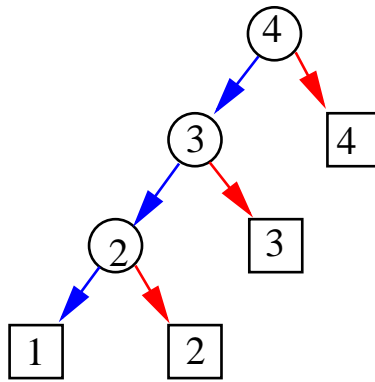
Jelölje  $F_{k,h}$  a legtöbb levelet tartalmazó  $k$  magasságú (legfeljebb  $k$  kérdéssel kitalálható)  $h$ -hibázó kérdezőfa, a leveleinek száma pedig  $L(k, h)$ .

$$L(k, 1) = k + 1$$

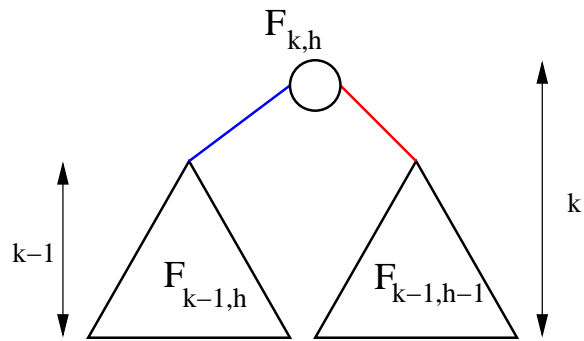
$$L(k, h) = L(k, k) \text{ ha } k < h$$

$$L(k, h) = L(k - 1, h) + L(k - 1, h - 1) \text{ ha } k > 1 \wedge k \geq h$$

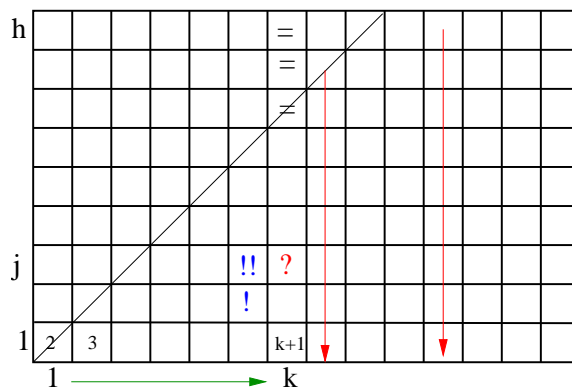
Tehát a probléma megoldása az a legkisebb  $k$ , amelyre  $L(k, h) \geq n$



20. ábra.  $F_{3,1}$ : 1-hibázó 3 magas kérdezőfa.



21. ábra.  $F_{k,h}$ :  $k$ -magas  $h$ -hibázó legtöbb levelet tartalmazó kérdezőfa.



22. ábra. Részproblémák számítási sorrendje: oszloponként felülről lefelé haladva.

$$\begin{aligned}
L(k,1) &= k+1 \\
L(k,j) &= L(k,k) \text{ ha } k < j \\
L(k,k) &= L(k-1,k) + L(k-1,k-1) = 2L(k-1,k-1) \\
L(k,j) &= L(k-1,j) + L(k-1,j-1) \text{ ha } k > 1 \wedge j < k
\end{aligned}$$

```

1 function Lf(n:longint; h:integer):longint;
2 const
3   maxH=100;
4 var
5   L:array[1..maxH] of int64;
6   k,j,hh:integer;
7 begin
8   L[1]:=2; {az első oszlop}
9   k:=1;
10  repeat {a k. oszlop kiszámítása}
11    inc(k);
12    if k<=h then begin
13      hh:=k;
14      L[k]:=L[k-1]+L[k-1];
15    end else begin
16      hh:=h;
17      L[h]:=L[h]+L[h-1];
18    end;
19    for j:=hh-1 downto 2 do
20      L[j]:=L[j]+L[j-1];
21    L[1]:=k+1;
22  until L[hh]>=n;
23  Lf:=k;
24 end{Lf};

```