# Dinamikus programozás

Horváth Gyula
**horvath@inf.elte.hu**

# 2. 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.

# 3. Feladat: A pénzváltás probléma.

**Probléma:** Pénzváltás
**Bemenet:** $P = \{p_1, ..., p_n\}$ pozitív egészek halmaza, és $E$ pozitív egész szám.
**Kimenet:** Olyan $S \subseteq P$, hogy $\sum_{p \in S} = E$.
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

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

egy megoldása a feladatnak. Ekkor

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

megoldása lesz annak a feladatnak, amelynek bemenete a felváltandó $E - p_{i_k}$ érték, és a felváltáshoz legfeljebb a első $i_k - 1$ ($p_1, \ldots, p_{i_k - 1}$) 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 $p_1, \ldots, p_i$ pénzzel. Jelölje $V(X,i)$ az $(X,i)$ részprobléma megoldását, ami logikai érték; $V(X,i) = Igaz$, ha az $X$ összeg előállítható legfeljebb az első $i$ pénzzel, egyébként $Hamis$.

**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 $i > 1$

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

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

**Rekurzív megoldás.**

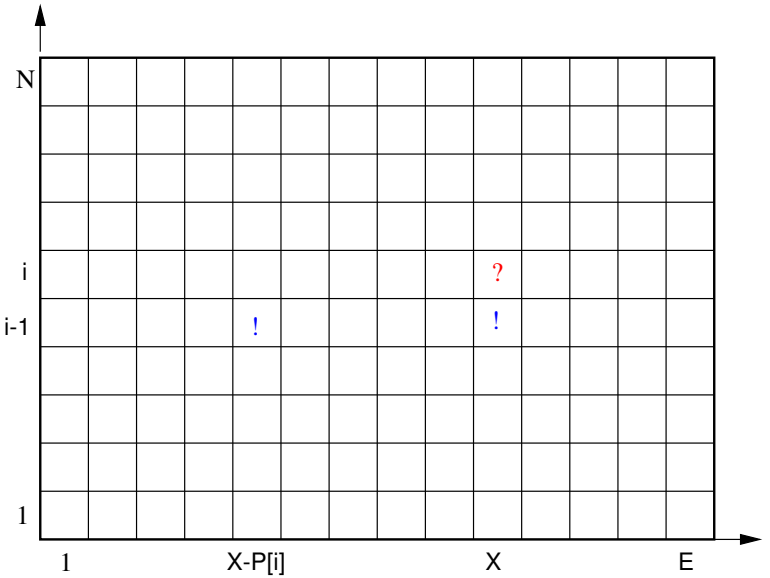
Mivel a megoldás kifejezhető egy V(X,i) logikai értékű függvénnyel, ezért a felírt összefüggések alapján azonnal tudunk adni egy rekurzív függvényeljárást, amely a pénzváltás probléma megoldását adja.

```
1 bool V(int P[], int x, int i ){
2    return   P[i]==x ||
3             i>1 && V(P, x,i-1) ||
4             i>1 && x>P[i] && V(P, x-P[i], i-1);
5 }
```

Elemezzük a rekurzív megoldás futási idejét. Számítsuk ki, hogy legrosszabb esetben hány eljáráshívás 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 $2^{30}$ eljáráshívást hajt végre, a futási idő legrosszabb esetben $17592$ milliárd év lenne! Másképpen fogalmazva, a rekurzív megoldás a pénzek halmazának összes lehetséges részhalmazát megvizsgálja, hogy az adott részhalmaz összege megegyezik-e $E$-vel. De $n$-elemű halmaz összes részhalmazainak 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.




1. ábra. A pénzváltás táblázata

**A részproblémák 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  bool V(int P[], int E, int n){
2     bool VT[E+1][n+1];
3     for (int x=1;x<=E;x++) VT[x][1]=false;//1. sor kitĂśltĂŠse
4     if (P[1]<=E) VT[P[1]][1]=true;
5     for (int i=2;i<=n;i++){//az i. sor, azaz V(x,i) szĂĄmĂtĂĄsa
6        for (int x=1;x<=E;x++)
7           VT[x][i]=x==P[i] ||
8                    VT[x][i-1] ||
9                    P[i]<=x && VT[x-P[i]][i-1];
10    }
11    return VT[E][n];
12 }
```

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 létezi-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 (hátulról 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  bool V(int P[], int E, int n ){
2  //Pénzváltás lineáris táblázatkitöltéssel
3     bool T[E+1];
4     for (int x=1;x<=E;x++) T[x]=false;//1. sor kitÄšltÄŠse
5     T[P[1]]=P[1]<=E;
6     for (int i=2;i<=n;i++){//az i. sor, azaz V(x,i) szÄĄmÄtÄĄsa
7        for (int x=1;x<=E;x++)
8           T[x]= T[x] || x==P[i] ||
9                 P[i]<=x && T[x-P[i]];
10    }
11    return T[E];
12 }
```

## 3.1. 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:** $P = \{p_1, ..., p_n\}$ pozitív egészek halmaza, és $E$ pozitív egész szám.
**Kimenet:** Olyan $S \subseteq P$, hogy $\sum_{p \in S} = E$
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  int V(int P[], int E, int n, int S[] ){
2     bool VT[E+1][n+1];
3     for (int x=1;x<=E;x++) VT[x][1]=false;//1. sor kitÄšltÄŠse
4     if (P[1]<=E) VT[P[1]][1]=true;
5     for (int i=2;i<=n;i++){//az i. sor, azaz V(x,i) szÄĄmÄtÄĄsa
6        for (int x=1;x<=E;x++)
7           VT[x][i]=x==P[i] ||
8                    VT[x][i-1] ||
9                    P[i]<x && VT[x-P[i]][i-1];
10    }
11    int m=0;
12    int x=E; int i=n;
```

```
13      do{
14          while(i>0 && VT[x][i]) i--;
15          S[++m]=i+1;
16          x-=P[i+1];
17      }while(x>0);
18      return m;
19  }
```

# 4. Feladat: Optimális pénzváltás

**Bemenet:** $P = \{p_1, ..., p_n\}$ pozitív egészek halmaza, és $E$ pozitív egész szám.
**Kimenet:** Olyan $S \subseteq P$, hogy $\sum_{p \in S} = E$ 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

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

egy optimális megoldása a feladatnak. Ekkor

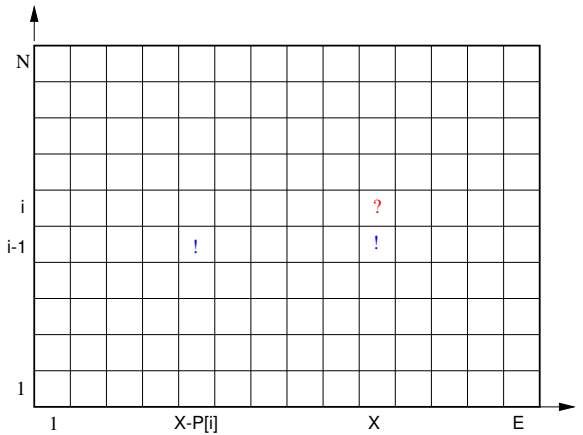
$$E - p_{i_k} = p_{i_1} + \ldots + p_{i_{k-1}}$$

optimális megoldása lesz annak a feladatnak, amelynek bemenete a felváltandó $E - p_{i_k}$ érték, és a felváltáshoz legfeljebb a első $i_k - 1$ ($p_1, \ldots, p_{i_k - 1}$) 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 $i$ $\{p_1,\ldots,p_i\}$ 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.**

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




2. ábra. A pénzváltás táblázata

```
1  int OptValto(int P[], int E, int n, int S[] ){
2     int Opt[E+1][n+1];
3     for (int x=1;x<=E;x++) Opt[x][1]=n+1;//1. sor kitöltése
4     if (P[1]<=E) Opt[P[1]][1]=1;
5     for (int i=2;i<=n;i++){//az i. sor, azaz Opt(x,i) számítása
6        for (int x=1;x<=E;x++){
7           if (P[i]==x) Opt[x][i]=1; else Opt[x][i]=Opt[x][i-1];
8           if (P[i]<x && Opt[x][i]>Opt[x-P[i]][i-1]+1)
9              Opt[x][i]=Opt[x-P[i]][i-1]+1;
10       }
11    }
12    int m=0;
13    int x=E; int i=n;
14    do{
15       while(i>1 && Opt[x][i]==Opt[x][i-1]) i--;
16       S[++m]=i;
17       x-=P[i--];
18    }while(x>0);
19    return m;
20 }
```

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.

# 5. 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 $\{e_1, \ldots, e_n\}$ 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 $\{e_1, \ldots, e_n\}$ egy részhalmazának összegeként.

Tehát a megoldás visszavezethető a pénzváltás probléma megoldására.

# 6. 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 közösben is maradhat kinemosztott ajándék, de ragaszkodnak ahhoz, hogy mindketten azonos összértéket kapjanak, és a közösben 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
```

**kimenet**

```
15
6 3
4 2 1
```

**Megoldás**

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

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

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

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

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 tekintsük 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:

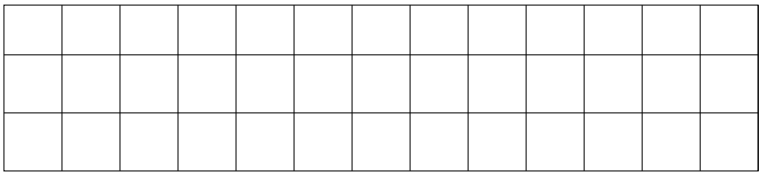
$$E(x,y,i) \Leftrightarrow \begin{cases} E(x,y,i-1) \vee \\ a_i \leq x \wedge E(x-a_i,y,i-1) \vee \\ a_i \leq x \wedge E(x,y-a_i,i-1) \end{cases} \tag{3}$$

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

# 7. Feladat: Járdakövezés

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

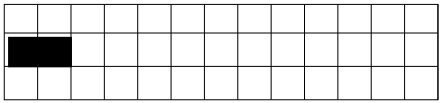
**Megoldás**



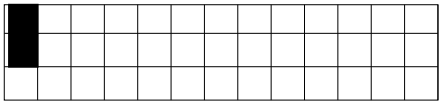
3. ábra.

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

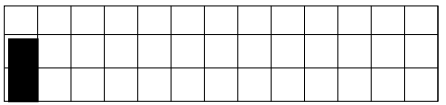
Az első oszlop középső négyzete háromféleképpen fedhető le.



4. ábra. 1. eset



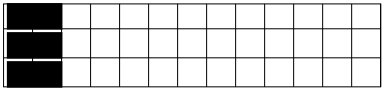
5. ábra. 2. eset



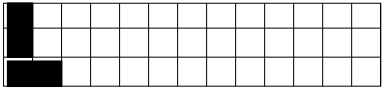
6. ábra. 3. eset

Az egyes esetek csak az alábbi módon folytathatók:

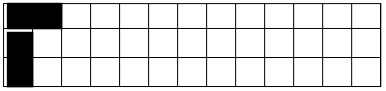
Jelölje $B(n)$ azt, hogy hányféleképpen fedhető le egy $3 \times n$ egység méretű járda, amelynek a bal alsó sarka már le van fedve. Szimmetria miatt a jobb felső sarok lefedettsége esetén is $B(n)$-féle lefedés van.



7. ábra. Az 1. eset csak így folytatható

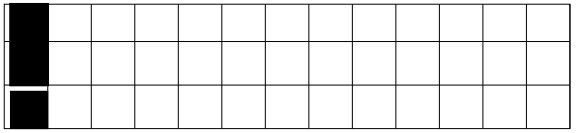


8. ábra. A 2. eset csak így folytatható

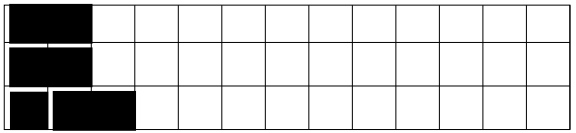


9. ábra. A 3. eset csak így folytatható

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



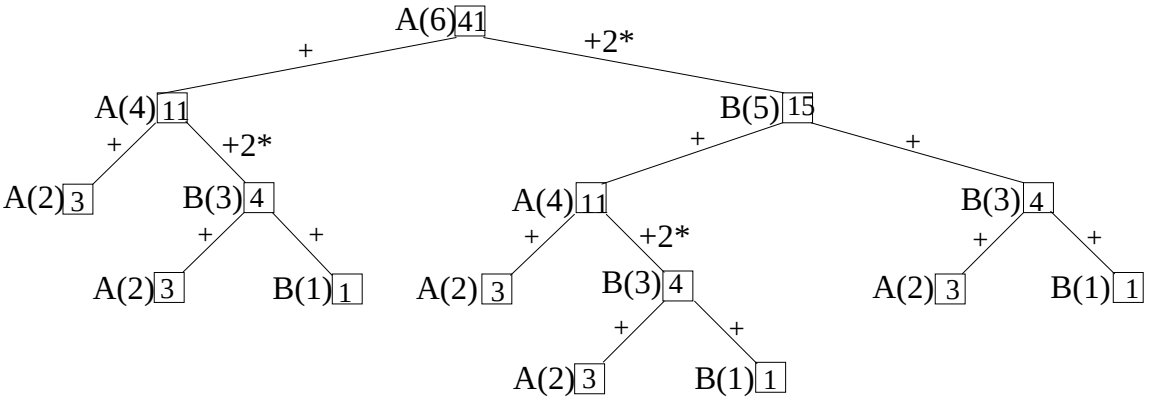
10. ábra. Az 1. eset csak így folytatható



11. ábra. Az 2. eset csak így folytatható

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

```
1  program jarda;
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
23   writeln(A(32));
24 end.
```




12. ábra. Rekurziós fa

```
1  long long JardaKovezo(int n){
2     long long A[n+1],B[n+1];
3     A[1]=0; B[1]=1;
4     A[2]=3; B[2]=0;
5     for (int i=3; i<=n;i++){
6        A[i]=A[i-2]+2*B[i-1];
7        B[i]=A[i-1]+B[i-2];
8     }
9     return A[n];
10 }
```

```
A(64) = 1582048049556775361
```

# 8. Feladat: Tükörszó (IOI'2000)

Egy karaktersorozatot 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: $'a' \leq c \leq' z'$ 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 törlendő 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.**

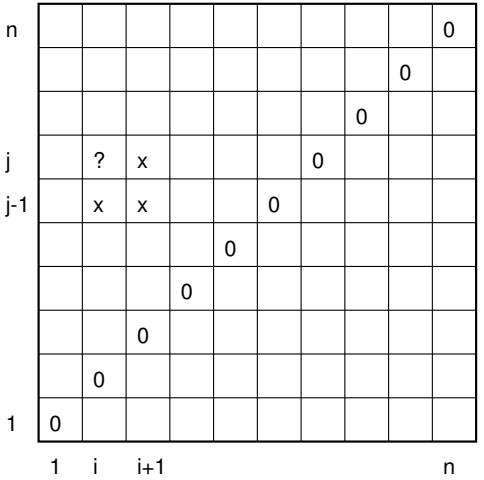
Ha $i > j$ esetre $M(i, j)$ értékét $0$-ként értelmezzük, akkor a részproblémák megoldásai között az alábbi rekurzív összefüggést lehet felírni.

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

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.




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

```
1  int Tukorszo(char* S, int n){
2     int T[n+1];
3     int ment, menti;
4     T[0]=0;
5     for (int j=1; j<n; j++){
6        T[j]=0; menti=0;
7        for (int i=j-1;i>=0; i--){
8           ment=T[i];
9           if (S[i]==S[j])
10             T[i]=menti;
11          else
12             T[i]=1+min(T[i],T[i+1]);
13          menti=ment;
14       }
15    }
16    return T[0];
17 }
```

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 $\Theta(n^2)$.

```
int Tukorszo(char* S, int n){
   int M[n+1][n+1];
   for (int j=0; j<n; j++){
      M[j][j]=0;
      for (int i=j-1;i>=0; i--){
         if (S[i]==S[j])
            M[i][j]=M[i+1][j-1];
         else
            M[i][j]=1+min(M[i+1][j],M[i][j-1]);
      }
   }
   cout<<M[0][n-1]<<endl;
   int i=0, j=n-1;
   while (i<j){
      if (S[i]==S[j]){
         i++; j--;
      }else{
         if (M[i+1][j]<M[i][j-1]){
            cout<<i<<" ";
            i++;
         }else{
            cout<<j<<" ";
            j--;
         }
      }
```

```
26      }
27      return M[0][n-1];
28  }
```

# 9. 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 játszmát 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 játékossal, 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 $2 <= n <= 100$. 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.

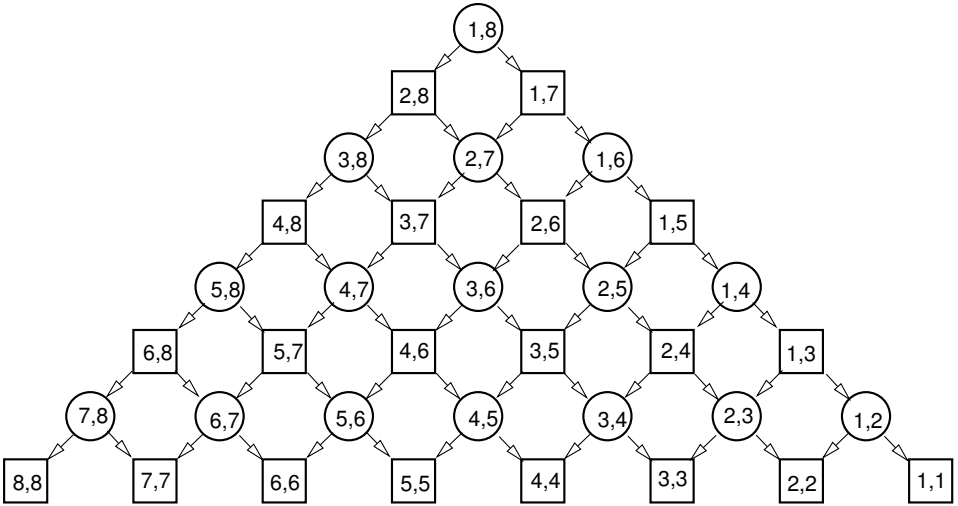
## Példa bemenet és kimenet

| INPUT.TXT | OUTPUT.TXT |
| --- | --- |
| 6<br>4 7 2 9 5 2 | 18 11 |

**Megoldás**

Jelölje $\langle a_1, \ldots, a_n \rangle$ 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 $\langle a_i, \ldots, a_j \rangle$ 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 játékost, 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.

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.

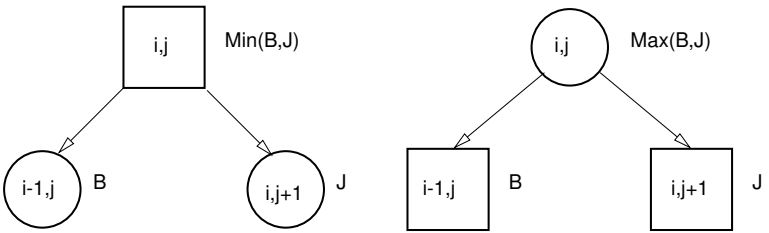



14. ábra. A játékállások gráfja $n = 8$ esetén. Körrel jelölt állásból ($i + j$ páratlan) az első, négyzettel jelölt állásból ($i + j$ páros) a második játékos lép.

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 meg-




15. ábra. Mini-max szabály.

kezdése 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.

```cpp
#include <iostream>
#include "stdlib.h"
#include "Play.h"
#define maxN 201
//Számjáték
using namespace std;
int A[maxN];
int n;
int Opt[maxN][maxN];
char Lt[maxN][maxN];

void Beolvas(){
   cin>>n;
   for (int i=1;i<=n; i++)
      cin>>A[i];
}
```

```
18 void Elofeldolgoz(){
19    int pbal,pjobb;
20    for (int j=1;j<=n;j++){
21       Opt[j][j]=0;
22       for (int i=j-1;i>0;i--)
23          if ((i+j)%2==1){//1. jÄĄtÄŠkos lÄŠp
24             if (Opt[i+1][j]<Opt[i][j-1]){
25                pbal=A[i]+Opt[i+1][j];
26                pjobb=A[j]+Opt[i][j-1];
27                if (pbal>pjobb){
28                   Opt[i][j]=pbal;
29                   Lt[i][j]='L';
30                }else{
31                   Opt[i][j]=pjobb;
32                   Lt[i][j]='R';
33                }
34             }
35          }else{//2. jÄĄtÄŠkos lÄŠp
36             if (Opt[i+1][j]<Opt[i][j-1])
37                Opt[i][j]=Opt[i+1][j];
38             else
39                Opt[i][j]=Opt[i][j-1];
40          }
41    }
42 }
```

```
43 void Jatszas(){
44    char lep;
45    int bal=1, jobb=n;
46    while (bal<jobb){
47 //       MyMove(Lt[bal][jobb]);
48       if (Lt[bal][jobb]=='L')
49          bal++;
50       else
51          jobb--;
52 //       lep=YourMove();
53       if (lep=='L')
54          bal++;
55       else
56          jobb--;
57    }
58 }
59 int main(){
60    Beolvas();
61    Elofeldolgoz();
62 //    StartGame();
63    Jatszas();
64 }
```

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

## Bemenet

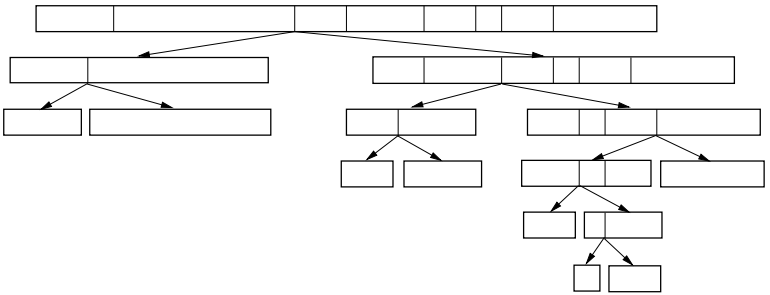
A `vag.be` szöveges állomány első sora egyetlen egész számot tartalmaz, a vágandó rúd $h$ hosszát $(0 < h \leq 1000)$. A második sorban az elvégzendő vágások $n$ száma van $(1 \leq n \leq 1000)$. A harmadik sor $n$ darab egész számot tartalmaz egy-egy szóközzel elválasztva, az elvégzendő vágások helyeit. 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 | Kimenet |
|---|---|
| 24<br>7<br>3 10 12 15 17 18 20 | 70<br>2 1 4 3 7 5 6 |

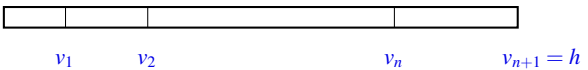


16. ábra.

**Megoldás.**

**Az optimális megoldás szerkezetének vizsgálata.**

Vegyünk fel egy $v_0 = 0$ és $v_{n+1} = h$ fiktív vágási helyet a rúd elejére, illetve végére.




17. ábra. A vágási helyek: $v_0 = 0$ , $v_{n+1} = h$

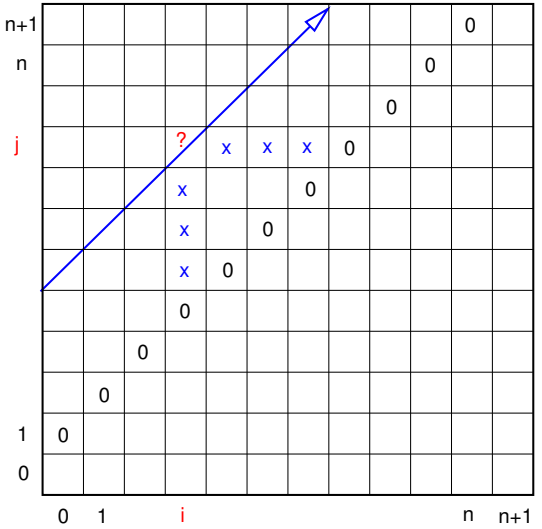
Ha az optimális vágás során először a $v_k$ ($1 \leq k \leq n$) helyen történik a vágás, akkor az első darabon a $v_0, v_1, \ldots, v_{k-1}, v_k$, a másodikon pedig és $v_k, v_{k+1}, \ldots, v_n, h$ vágásoknak is optimálisnak kell lenni.

**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.

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

Legyen $S(i,j)$ az a $k$, amelyre a minimum adódik.




18. ábra. Táblázat-kitöltési sorrend: átlósan, vagy alulról-felfelé,jobbról-balra haladva.

```cpp
#include <iostream>
#include "stdlib.h"
#define maxN 201
#define maxM 1001
#define Inf 200000000L
//optimális vágás
using namespace std;
int V[maxN+1];
int h,n;
int S[maxN][maxN];

void Beolvas(){
//Global: h,n,V
   cin>>h>>n;
   for (int i=1;i<=n; i++)
      cin>>V[i];
   V[0]=0; V[n+1]=h;
}
```

```
19 int Szamit(){
20 //Global: V,n, S
21    int Opt[n+2][n+2];
22    int g, Min;
23    for (int i=0;i<=n;i++){
24       Opt[i][i+1]=0; S[i][i+1]=0;
25    }
26    for (int u=2;u<=n+1; u++){
27       for (int i=0;i<=n-u+1;i++){
28          int j=i+u; Min=Inf;
29          for (int k=i+1; k<=j-1;k++){
30             int uj=Opt[i][k]+Opt[k][j];
31             if (uj<Min){
32                Min=uj; g=k;
33             }
34          }//for k
35          Opt[i][j]=Min+V[j]-V[i];
36          S[i][j]=g;
37       }//for i
38    }//for u
39    return(Opt[0][n+1]);
40 }
```

```
41 void Kilr(int i, int j){
42     if (j<=i+1) exit;
43     int k=S[i][j];
44     cout<<k<<"␣";
45     Kilr(i,k);
46     Kilr(k,j);
47 }
48 int main(){
49     Beolvas();
50     int Koltseg=Szamit();
51     cout<<Koltseg<<endl;
52     Kilr(0,n+1);
53     return 0;
54 }
```

# 11. Feladat: Torony építése kockákból.

Építőkockákból úgy lehet stabil tornyot építeni, hogy kisebb kockára nem lehet nagyobbat, 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: $h_1, \ldots, h_n$, súlyai pedig $s_1, \ldots, s_n$.

Elemezzük az optimális megoldás szerkezetét.

Tegyük fel, hogy a $i_1, \ldots, i_k$ sorszámú kockák ebben a sorrendben egymásra rakásával kapjuk a legmagasabb tornyot. Ekkor $i_2, \ldots, i_k$ 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 $i_1, \ldots, i_k$. 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 tárigénye $n$-el arányos, futási ideje pedig legrosszabb esetben $n^2$-el.

```cpp
#include <iostream>
#include "stdlib.h"
#define maxN 1001
//toronyépítés kockákból
//Rekurzió memorizálással
using namespace std;
int n;
int S[maxN], H[maxN], M[maxN];
int Ra[maxN];

void Beolvas(){
//Global: n,H,S
   cin>>n;
   for (int i=1;i<=n; i++){
      cin>>H[i]>>S[i];
   }
}
```

```
18 int Magas(int i){
19 //Global: H,D,M, Ra
20    int Mi,Mj;
21    if (M[i]>0) return M[i];
22    Mi=0;
23    for (int j=1;j<=n;j++){
24       if (i!=j && H[i]>H[j] && S[i]>S[j]){
25          Mj=Magas(j);
26          if (Mj>Mi) {Mi=Mj; Ra[i]=j;}
27       }
28    }
29    M[i]=Mi+H[i]; //memorizálás
30    return M[i];
31 }
```

```
32 void Kilr(){
33 //Global: n, M,Ra
34    int maxi=0, max=0;
35    for (int i=1;i<=n; i++)
36       if (Magas(i)>max) {maxi=i; max=M[i]; }
37    cout<<M[maxi]<<endl;
38    int i=maxi;
39    while (i>0){
40       cout<<i<<"␣";
41       i=Ra[i];
42    }
43    cout<<endl;
44 }
45 int main(){
46    Beolvas();
47    Kilr();
48    return 0;
49 }
```

# 12. 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!

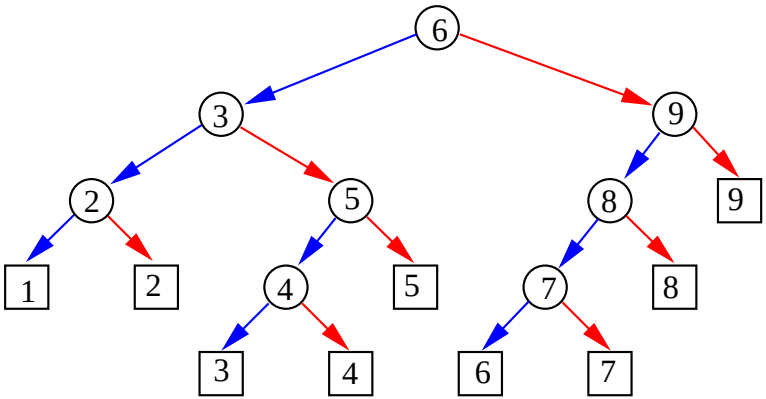
**Példa bemenet és kimenet**

| | |
|---|---|
| 9 2 | 4 |

**Megoldás**

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

- A fának $n$ levele van és ezek balról jobbra sorrendben az $1, \ldots, n$ 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.

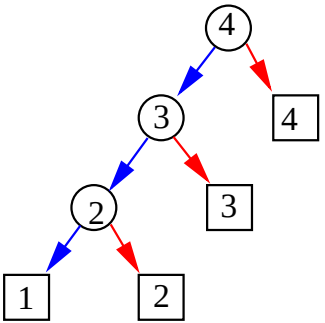



19. ábra. Egy 2-hibázó kérdezőfa a példa bemenetre.

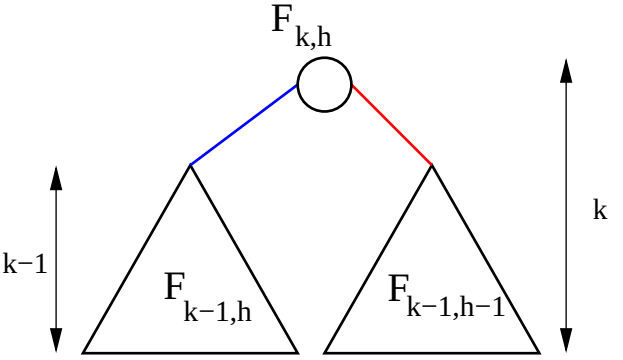
Kérdezőfa a következőképpen használható. Kezdetben a $p$ aktuális pont legyen a fa gyökere. Mindaddig, amíg $p$ nem levél, kérdezzünk rá a $p$-ben lévő értékre. Ha a válasz igen, akkor $p$ legyen a bal fia, egyébként a jobb fia. Az ismétlés befejeződése után a gondolt szám $p$-ben van.

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 kitaláló) $h$-hibázó kérdezőfa, a leveleinek száma pedig $L(k,h)$.



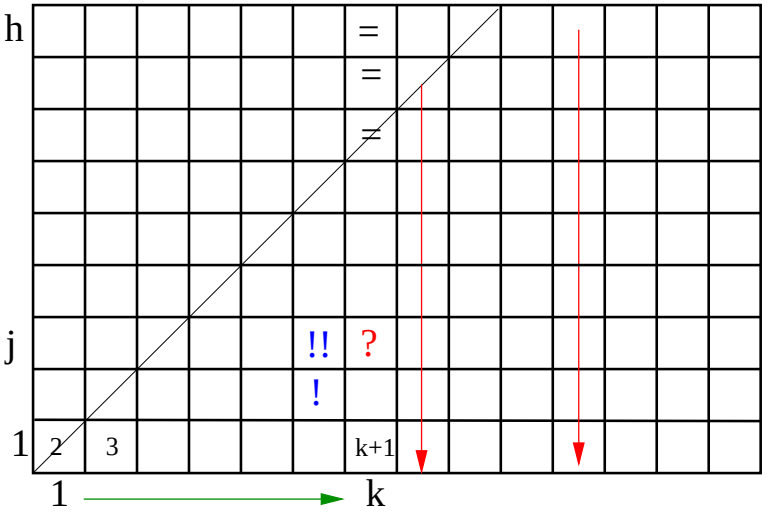

20. ábra. $F_{3,1}$: 1-hibázó 3 magas kérdezőfa.




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

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

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




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

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

```cpp
long long Lf(int n, int h){
    long long L[maxH];
    L[1]=2;    long long k=1,hh;
    do{
        k++;
        if (k<=h){
            hh=k;
            L[k]=L[k-1]+L[k-1];
        }else{
            hh=h;
            L[h]=L[h]+L[h-1];
        }
        for (int j=hh-1; j>1; j--)
            L[j]=L[j]+L[j-1];
        L[1]=k+1;
    }while(L[hh]<n);
    return k;
}
int main(){
    int n, h;
     cin>>n>>h;
    long long k=Lf(n,h);
    cout<<k<<endl;
    return 0;
}
```