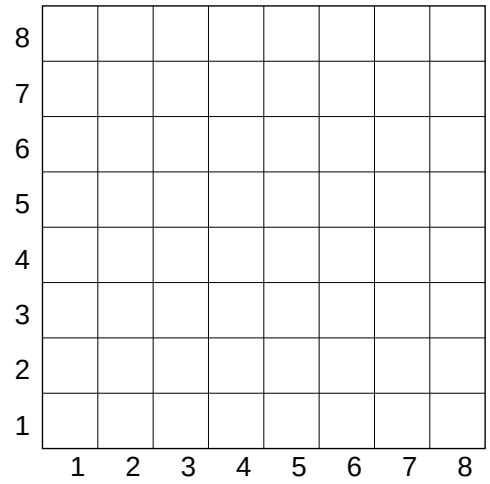
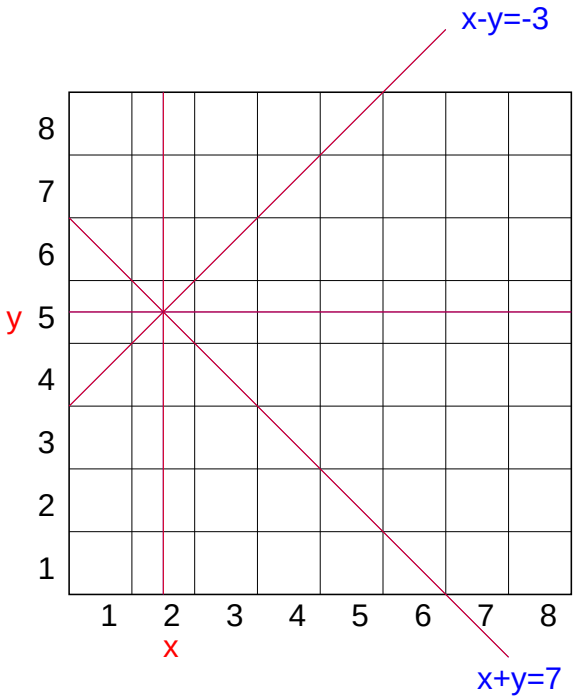
9. Megoldás keresése visszalépéssel (backtracking)

Probléma: n**-királynő**
Helyezzünk el az $n x n$-es sakktáblán n királynőt, hogy egyik se üsse a másikat!



1. ábra. Üres tábla

A megoldás megadható annak az n mezőnek a koordinátáival, amelyekre királynőket helyezzük: $M = \{(x_1,y_1),\cdots,(x_n,y_n)\}$. Tehát az (x_i,y_i) és (x_j,y_j) mezőkre helyezett két királynő akkor és csak akkor nem üti egymást, ha



2. ábra. Az (x,y) mezőn lévő királynő ütési mezői.

$$x_i \neq x_j \tag{1}$$
$$y_i \neq y_j \tag{2}$$
$$x_i - y_i \neq x_j - y_j \tag{3}$$
$$x_i + y_i \neq x_j + y_j \tag{4}$$

Tehát egy ilyen M halmaz akkor és csak akkor megoldása a feladatnak, ha $(\forall i, j)(1 \leq i < j \leq n)$teljesül a fenti 4 feltétel. Minden sorban, minden oszlopban pontosan egy királynőnek kell lenni, továbbá minden főátlóban és minden mellékátlóban legfeljebb egy királynő lehet. Tehát minden megoldás megadható egy $X = \langle x_1, \ldots, x_n \rangle$ vektorral, ami azt jelenti, a királynőket az $\{(x_1,1), \cdots, (x_n,n)\}$ mezökre helyezünk királynőket. Ekkor a megoldás feltétele:$(\forall i, j)(1 \leq i < j \leq n)$

$$x_i \neq x_j \tag{5}$$
$$x_i - i \neq x_j - j \tag{6}$$
$$x_i + i \neq x_j + j \tag{7}$$

9.1. Kimerítő keresés (nyers erő) módszere

Elvi algoritmus:
KIMERITOKERESES

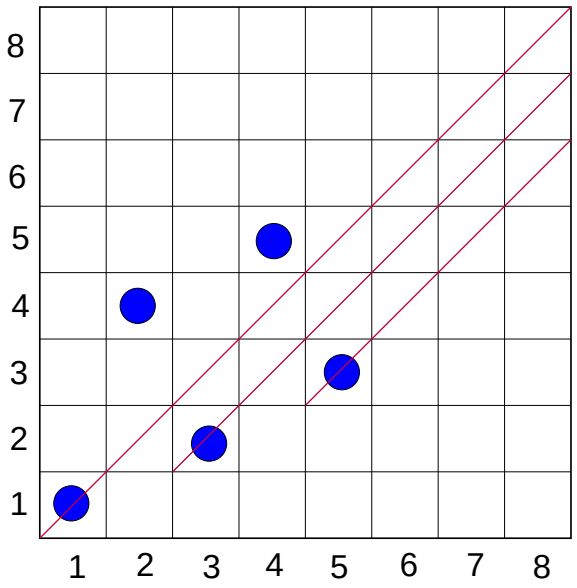
$\forall X = \langle x_1, \ldots, x_n \rangle \in [n] \times \ldots \times [n]$ do
    if Megoldás(X) then
        KiIr(X)

A KIMERITOKERESES algoritmus megvalósítása:

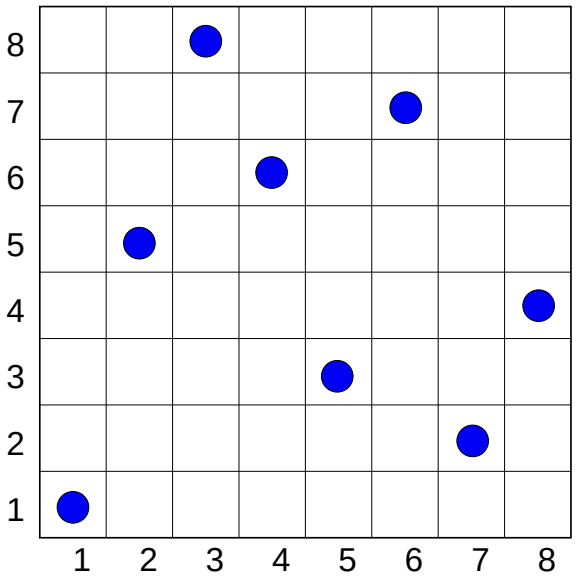
```
abstract class Kimerito{
   abstract boolean Megoldas(int[] X);
   abstract void KiIr(int[] X);

   public void Keres(int[] X, int k){
      int n=X.length;
      for(int i=1; i<=n; i++){
         X[k]=i;
         if (k==n-1 && Megoldas(X)){
            KiIr(X);
         }else
            Keres(X,k+1);
      }
   }
}
```

Nyilvánvaló, hogy ez a módszer sok vektort feleslegesen vizsgál, hiszen ha például $x_1 = x_2$ (vagy általában, ha $X = \langle x_1, \ldots, x_k \rangle$ esetén két királynő üti egymást), akkor X biztosan nem lehet megoldás. Azaz, elegendő lenne csak a permutációkat vizsgálni.
Ötlet: probáljuk a megoldást lépésenként előállítani úgy, hogy ha már elhelyeztünk a tábla első $k-1$ sorában királynőket úgy, hogy egyik sem üti a másikat, akkor a következő lépésben a k-adik sorba próbáljunk rakni egy királynőt.

Megoldáskezdemény:
$X = \langle x_1, \ldots, x_k \rangle$, $0 \leq k \leq n, 1 \leq x_i \leq n$
X jó megoldáskezdemény, ha kezdőszelete lehet egy megoldásnak, tehát nem üti egymást a táblára már elhelyezett k darab királynő, azaz:
$(\forall i, j)(1 \leq i < j \leq k)$
$(x_i \neq x_j) \wedge (x_i - i \neq x_j - j) \wedge (x_i + i \neq x_j + j)$ $V = \langle 1,4,2,5,3 \rangle$ jó megoldáskezdemény, de nem folytatható, mert a 6. sorba nem helyezhető királynő, hogy ne üsse a már táblán lévők egyikét sem.
Visszalépést kell végezni: az 5. sorban lévőt más helyre kell rakni.
Ez az a pont, ahol ez a módszer különbözik a mohó stratégiától. Ott a megoldáskezdemény mindig folytatható volt, és meg tudtuk mondani, hogy melyik lépéssel.
Itt azonban nem tudjuk megmondani, hogy egy megoldáskezdemény folytatható-e, és ha igen milyen lépéssel.



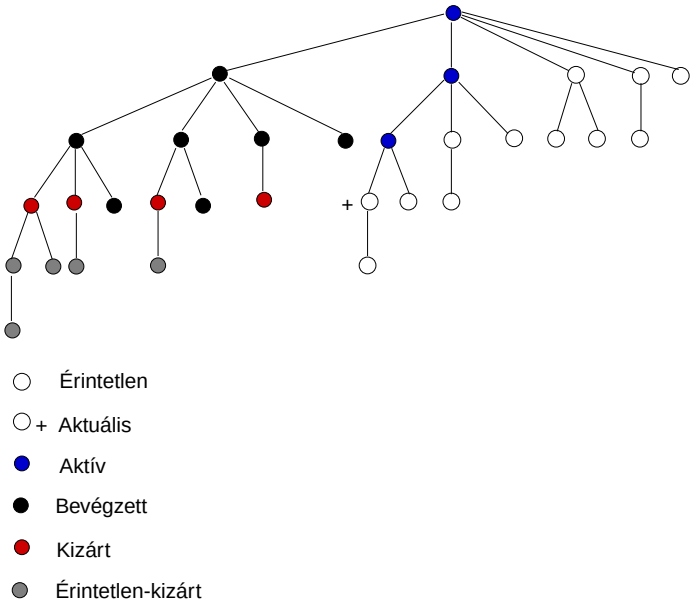
3. ábra. Nem folytatható állás (megoldáskezdemény)



4. ábra. Az első megtalált megoldás

Vegyük észre, hogy a megoldáskezdemények fát alkotnak. Egy $X = \langle x_1, \ldots, x_k \rangle$ megoldáskezdemény lehetséges közvetlen folytatásai, azaz X fiai az $Y = \langle x_1, \ldots, x_k, i \rangle$ $i = 1, \ldots, n$ lehetséges megoldáskezdemények.
A kimerítő keresést lényeges gyorsíthatjuk, ugyanis ha az $X = \langle x_1, \ldots, x_k \rangle$ megoldáskezdeményre nem teljesül az (1-4) feltételek valamelyike, akkor kihagyhatjuk a keresésből az összes olyan megoldáskezdeményt, amely folytatása X-nek. Ekkor azt mondjuk, hogy az X megoldáskezdemény kizárt lesz.

Tehát minden fabejáró algoritmus alkalmazható megoldás keresésére, azzal a módosítással, hogy ha az aktuális X pont (megoldáskezdemény) nem választható, azaz kizárt lesz, akkor X-et úgy kell tekinteni a bejárás során, mint ha levél pont lenne.
A megoldás keresését meg tudjuk fogalmazni olyan általános formában, hogy az algoritmus érdemi része, azaz a megoldástér bejárása csak néhány problémaspecifikus műveletet alkalmaz.
Ezt módszert "application framework" módszernek is nevezik. Adott problémára nem kell újraírni az érdemi részt, csak a problémaspecifikus műveletek megvalósítását kell megadni.
Érintetlen az olyan pontja a megoldástérnek, amelyet a keresés során még nem érintettünk.

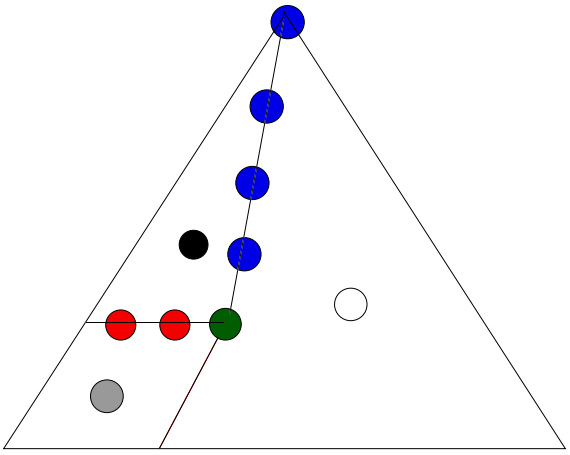


5. ábra. A megoldástér pontjainak osztályozása visszalépéses keresésnél

Aktuális az a pont, amelyet éppen vizsgálunk.
Aktív az olyan pont, amelyet már érintettünk a keresés során, de még nem bevégzett. Pontosan azok az aktív pontok, amelyek az aktuális pont ősei a fában. A keresés során az aktív pontokba még visszatérünk.
Kizárt a pont, ha olyan megoldáskezdemény, amelynek egyetlen folytatása (leszármazottja a fában) sem lehet megoldás.
Bevégzett a pont, ha minden fia vagy kizárt vagy bevégzett.
Érintetlen-kizárt a pont, ha leszármazottja valamely kizárt pontnak. Tehát a megoldástér ezen pontjait nem érinti a keresés.

Legyen MTer a megoldástér absztrakt osztály, amely az alábbi absztrakt metódusokat definiálja.
A megoldástér bejárásához szükséges műveletek:
ElsoFiu(), Testver(), Apa()
Megoldas(), LehetMego(), VisszaAllit()
Megoldástér osztály visszalépéses kereséshez:

```
public abstract class MTer implements Cloneable {
// probléma specifikus műveletek:
   public abstract boolean ElsoFiu();
/** Ha van X-nek fia, akkor X az első fiúra változik és a
   függvényhívás értéke true, egyébként false és X nem változik.
*/
   public abstract boolean Testver();
/** Ha van X-nek még benemjárt testvére, akkor X a következő testvér lesz
```



6. ábra. A megoldástér sematikus képe visszalépéses keresésnél

```
 *   és a függvényhívás értéke true, egyébként false és X nem változik.
 */
   public abstract boolean Apa();
/** Ha van X-nek apja, akkor X az apjára változik és a
    függvényhívás értéke true, egyébként false és X nem változik.
 */
   public abstract boolean Megoldas();
/** Akkor és csak akkor ad igaz értéket, ha X megoldása a problémának.
 */
   public abstract boolean LehetMego();
/** X.LehetMego() false értéket ad, ha nincs megoldás az X gyökerű részfában.
    Ha X.LehetMego() true, abból nem következik, hogy van X-nek olyan
    folytatása, ami megoldás.
    Bejegyzéseket tehet, ami segíti a műveletek hatékony megvalósítását.
 */
   public abstract void VisszaAllit();
/** Törli a Lehetmego() által tett bejegyzéseket.
 */
   public abstract void Input(String filenev)throws IOException;
/** A bemeneti adatok beolvasása és üres megoldáskezdemény előállítása.
 */
   public abstract void Output(String filenev)throws IOException;
/** A megoldás kiíratása.
 */
   public MTer clone() {
      MTer result = this;
      try {
       result = (MTer)super.clone();
      } catch (CloneNotSupportedException e) {
         throw new InternalError();
      }
      return result;
   }
}

public class VisszalepR{
//Megoldás rekurzív keresése az X gyökerű megoldástér fában
   public MTer Megold(MTer X){
```

```
      if (X.Megoldas())       //ha az összes megoldást keressük:
         return X;            //X.Output();
      MTer XX=X.clone();      //másolat készítés
      if (!XX.ElsoFiu())      //áttérés az első fiúra,
         return null;         //ha nincs, akkor visszatérés

      do{                     //X fiainak bejárása
         if (!XX.LehetMego()) //ha XX nem folytatható megoldássá
            continue;         //továbblépés a következő testvérre
         MTer X0=Megold(XX);  //rekurzív hívás
         if (X0!=null)        //megoldást találtunk,
            return X0;        //készenvagyunk
         XX.VisszaAllit();    //a LehetMego által tett bejegyzések törlése
      }while(XX.Testver());   //van-e még benemjárt testvér?

   return null;               //nincs megoldás az X-gyökerű fában
   }
}

public class Visszalep{
   private static enum Paletta {Feher, Kek, Piros};
//Megoldás visszalépéses keresése az X-gyökerű megoldástérben
   public MTer Megold(MTer X){
      Paletta szin=Paletta.Feher;
      while (true){
         switch (szin) {
            case Feher:
                  if (X.LehetMego()){     //folytatható-e X megoldássá?
                     if (X.Megoldas())    //ha az összes megoldást keressük:
                        return X;         //X.Output();
                     if (!X.ElsoFiu())    //átlépés az első fiúra, ha van
                        szin=Paletta.Kek; //uj aktuális pont
                  }else                   //X kizárt pont lesz
                     szin=Paletta.Piros;
                  break;
            default:

                  if (szin==Paletta.Kek)  //ha aktív pontból lépünk tovább
                     X.VisszaAllit();     //akkor előbb visszaállítás kell
                  if (X.Testver())        //átlépés a testvérre, ha van
                     szin=Paletta.Feher;  //X új aktív pont lesz
                  else{                   //ha X-nek nincs benemjárt testvére
                     if (X.Apa())         //visszalépés az apára
                        szin=Paletta.Kek; //az apa mindig aktív pont
                     else                 //X a gyökér
                        return null;      //vége a keresésnek
                  }
                  break;
         }//switch
      }//while
   }//Megold
}
```

A probléma-specifikus műveletek megvalósítása az n-királynő problémához.

```
public class Kiralynok extends MTer{
```

```
private static int N;
private int k;
private static int Sor[];
private static boolean Oszlop[];
private static boolean Fatlo[];
private static boolean Matlo[];
Kiralynok(){
   k=0;
}
public void Input(String filenev){
   Scanner stdin=new Scanner(System.in);
   N=stdin.nextInt();
   stdin.close();
   Sor=new int[N+1];
   Oszlop=new boolean[N+1];
   Fatlo=new boolean[2*N+1];
   Matlo=new boolean[2*N+1];
}

public void Output(String filenev){
   for (int i=1; i<=N; i++)
      System.out.print(Sor[i]+" ");
    System.out.println();
}
public boolean ElsoFiu(){
   if (k<N){
      Sor[++k]=1;
      return true;
   }
   return false;
}
public boolean Testver(){
   if (Sor[k]<N){
      ++Sor[k];
      return true;
   }
   return false;
}
public boolean Apa(){
   if (k>1){
      --k; return true;
   }  return false;
}

public boolean Megoldas(){
   return k==N;
}
public boolean LehetMego(){
   if (k==0) return true;
   if(!Oszlop[Sor[k]] &&
      !Fatlo[N+Sor[k]-k]&&
      !Matlo[Sor[k]+k]) {
      Oszlop[Sor[k]]=true;
      Fatlo[N+Sor[k]-k]=true;
      Matlo[Sor[k]+k]=true;
      return true;
```

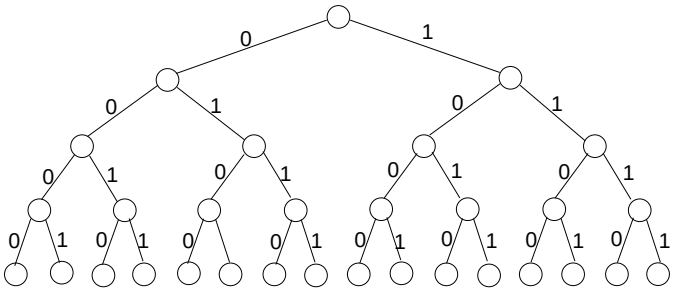
```
      }
      return false;
   }
   public void VisszaAllit(){
      if (k==0) return;
      Oszlop[Sor[k]]=false;
      Fatlo[N+Sor[k]-k]=false;
      Matlo[Sor[k]+k]=false;
   }
}
```

9.2. A visszalépéses keresés alkalmazása a pénzváltás problémára.

Probléma: Pénzváltás
Bemenet: $P = \{p_1, ..., p_n\}$ pozitív egészek halmaza, és E pozitív egész szám.
Kimenet: Olyan $S \subseteq P$, hogy $E = \sum_{p \in S}$
A megoldást kifejezhetjük és kereshetjük bitvektor formában, tehát olyan $X = \langle x_1, \ldots, x_n \rangle$ vektort keresünk, amelyre

$$E = \sum_{i=1}^{n} x_i \, p_i$$

Ekkor a megoldástér fája bináris fa lesz. A megoldást kifejezhetjük és kereshetjük mint a pénzek indexeinek olyan $S \subseteq \{1, \ldots, n\}$



7. ábra. Bináris megoldástér a pénzváltás probléma $n = 4$ esetében

halmazának $X = \langle i_k, \ldots, i_m \rangle$ növekvő felsorolásaként is, azaz $i_1 < i_2 < \ldots < i_m$, hogy .

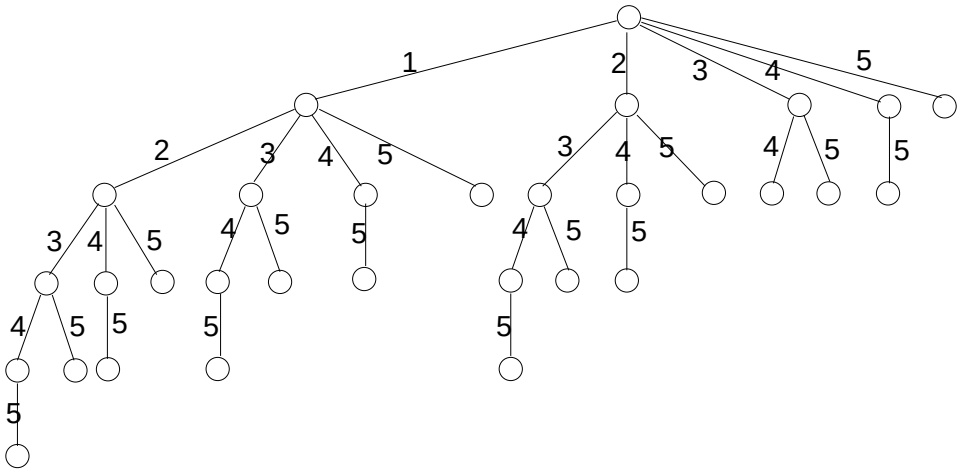
$$E = \sum_{k=1}^{m} p_{i_k}$$

Ekkor a megoldástér formája a 8. ábrán látható $n = 5$ esetére. A pénzváltás probléma megoldásához elegendő megadni a probléma-specifikus ELSOFIU , TESTVER, (APA,) VISSZAALLIT, LEHETMEGO, MEGOLDAS műveletek megvalósítását, és az RKERES (KERES) eljárás változtatás nélkül alkalmazható egy megoldás előállítására.
(Az APA művelet csak a nemrekurzív keresés esetén kell.)
Legyen $X = \langle i_k, \ldots, i_m \rangle$ tetszőleges megoldáskezdemény.
ELSOFIU(X) $= \langle i_1, \ldots, i_m, i_m + 1 \rangle$, ha $i_m < n$
TESTVER(X) $= \langle i_1, \ldots, i_{m-1}, i_m + 1 \rangle$, ha $i_m < n$
APA(X) $= \langle i_1, \ldots, i_{m-1} \rangle$, ha $m > 0$
LEHETMEGO(X) akkor és csak akkor adjon igaz értéket, ha

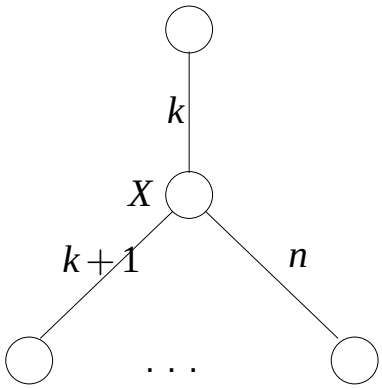
$$\sum_{k=1}^{m} p_{i_k} \leq E \wedge \sum_{k=1}^{m} p_{i_k} + \sum_{j=i_m+1}^{n} p_j \geq E$$

MEGOLDAS(X) akkor és csak akkor adjon igaz értéket, ha

$$E = \sum_{k=1}^{m} p_{i_k}$$



8. ábra. Nem bináris megoldástér a pénzváltás probléma $n = 5$ esetében



9. ábra. A fa pontjai

A VISSZAALLIT művelet megvalósítása előtt dönteni kell, hogy milyen segéd információt tárolunk egy megoldástérpontban. Célszerű tárolni a

$$Resz = \sum_{k=1}^{m} p_{i_k}$$

$$Maradt = \sum_{j=i_m+1}^{n} p_j$$

összegeket, hogy a LEHETMEGO(X) és MEGOLDAS(X) műveleteket konstans időben ki tudjuk számítani.

```
public class Penzvalto extends MTer{
   private static int N, E;
   private int k;
   private static int Penz[];
   private static int S[];
   private int resz;
   private int maradt;

   Penzvalto(){
      k=0;
   }
   public void Input(String filenev)throws IOException{
      Scanner bef;
      if (filenev.length()==0)
         bef=new Scanner(System.in);
      else
         bef=new Scanner(new File(filenev));

      N=bef.nextInt();
      E=bef.nextInt();
      bef.nextLine();
      Penz=new int[N+1];
      S=new int[N+1];

      S[0]=0;
      resz=0;
      maradt=0;
      for (int i=1; i<=N; i++){
         Penz[i]=bef.nextInt();
         maradt+=Penz[i];
      }
      bef.close();
   }
   public void Output(String filenev)throws IOException{
      PrintWriter kif;
      if (filenev.length()==0)
         kif=new PrintWriter(System.out);
      else
         kif=new PrintWriter(
               new BufferedWriter(
                  new FileWriter(filenev)
               )
             );

      for (int i=1; i<=k; i++)
         System.out.print(S[i]+" ");
       System.out.println();
```

```
   }

   public boolean ElsoFiu(){
      if (k<N && S[k]<N){
         k++;
         S[k]=S[k-1]+1;
         return true;
      }
      return false;
   }
   public boolean Testver(){
      if (k>0 && S[k]<N){
         ++S[k];
         return true;
      }
      return false;
   }
   public boolean Apa(){
      if (k>0){
         S[k]=0;
         --k;
         return true;
      }
      return false;
   }

   public boolean Megoldas(){
      return resz==E;
   }
   public boolean LehetMego(){
      if (resz<=E && resz+maradt>=E){
         resz+=Penz[S[k]];
         maradt-=Penz[S[k]];
         return true;
      }
      return false;
   }
   public void VisszaAllit(){
      for (int i=S[k-1]+1; i<=S[k]; i++)
         maradt+=Penz[i];
      resz-=Penz[S[k]];
   }
}
```

Visszalépéses keresési algoritmusok futási ideje

Legrosszabb esetben a keresés a megoldástér-fa minden pontját bejárja, ami exponenciális.

Visszalépéses keresési algoritmusok tárigénye

A rekurzív megvalósítás tárigénye függ az alkalmazott programozási nyelvtől. Minden esetben tárolni kell az aktív pontokat. Ha objektum orientált nyelven történik a megvalósítás, ahol automatikus memóriagazdálkodás van, akkor a tényleges tárigény ennél több. Ennek az az oka hogy a rekurzív eljáráspéldány terminálásakor nem szabadul fel automatikusan az eljárás során létesített objektumok által elfoglalt memória.

Nemrekurzív megvalósítás során elég csak csak az aktuális pontot tárolni.

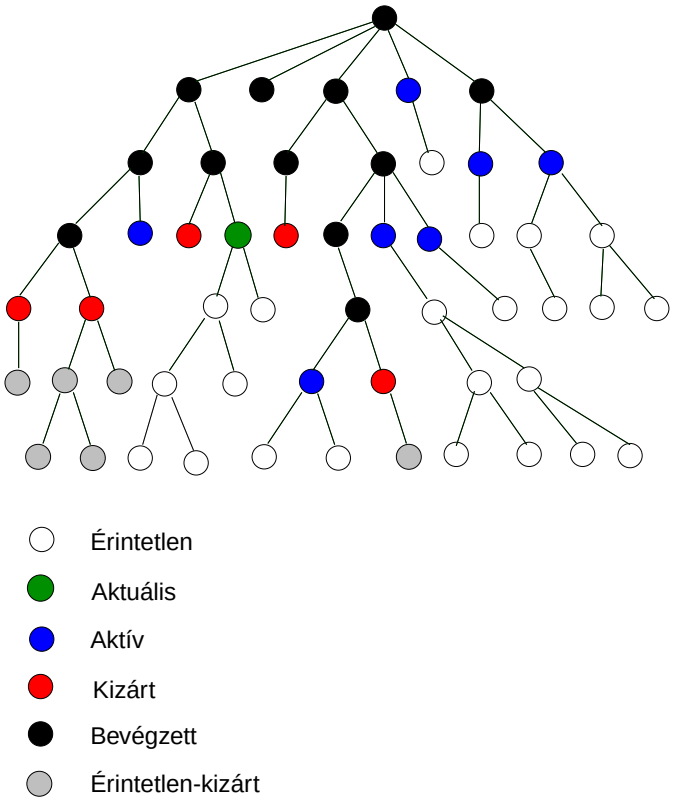
A visszalépéses keresés alkalmazható optimális megoldás előállítására is.

Legyen $C(X)$ a célfüggvény, tehát olyan X-et keresünk, amelyre: MEGOLDAS(X) és $C(X)$ minimális. Az összes megoldás keresése során ha jobb megoldást találunk, feljegyezzük.

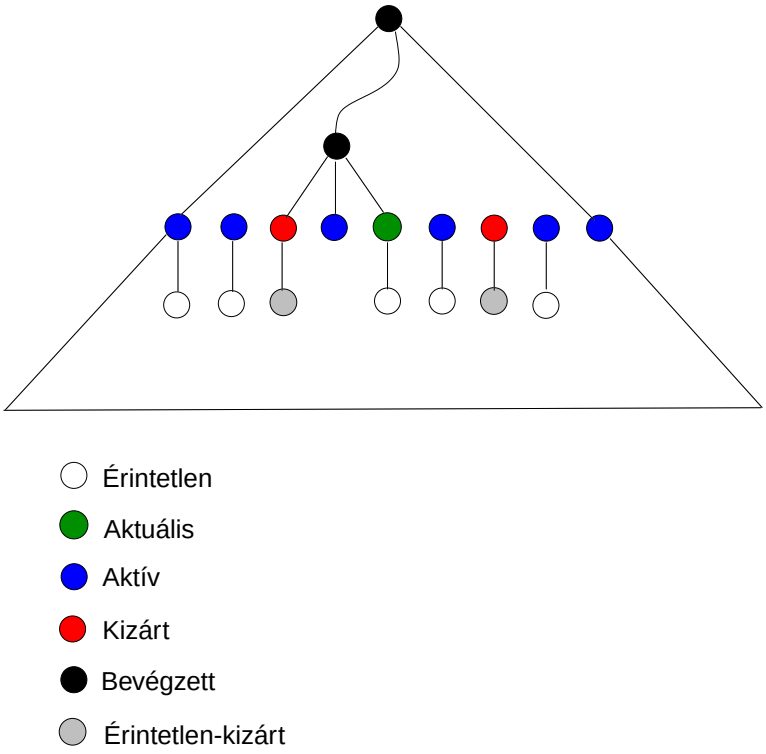
```
if (X.Megoldas() && X.C()<OptC){
  X0=X;
  OptC=C(X);
}
```

9.3. Elágazás-korlátozás módszere (branch and bound)

Ha a megoldáskezdemények tere fa, akkor a probléma egy, vagy az összes megoldását kereshetjük a fák szint szerinti bejárásának megfelelő stratégiával. A stratégiát még általánosabban használhatjuk, ugyanis az aktuális pontot tetszőlegesen választhatjuk az aktív pontok közül. A lényeg, hogy a választott aktuális pont minden fiát kigeneráljuk, és ha lehetséges megoldás (azaz teljesül rá a Lehetmego feltétel), akkor betesszük az aktív pontok halmazába. Tehát az algoritmus egy adagolót használ az aktív pontok tárolására. A visszalépéses stratégia esetén elég volt egy pontot, az aktuális pontot tárolni, mert a következő aktív pont mindig ennek fia, testvére, vagy apja. Az adagolóval történő bejáráskor ez nem igaz, ezért a fabejáráshoz szükséges ElsőFiú és Testvér műveleteket célszerű úgy specifikálni, hogy mindig új objektumként hozzák létre a fiút vagy testvért, ha nem létezik, akkor pedig null értéket adjanak.



10. ábra. A megoldástér pontjainak osztályozása adagolóval történő keresés esetén.



11. ábra. A megoldástér pontjainak sematikus ábrázolása adagolós keresés esetén.

```
public static void AKeres(EKMegold X)throws Exception{
   Adagolo<EKMegold> A=new AdagoloL<EKMegold>();

   if (!X.LehetMego()) return;
   A.Betesz(X);
   while (A.nemUres()){
      X=A.Kivesz();
      X=X.ElsoFiu();
      while (X!=null){
         if (X.LehetMego()){
            if (X.Megoldas())
               X.Output("");
            A.Betesz(X);
         }
         X=X.Testver();
      }
   }//while
   return;
}
```

9.1. Állítás. *Ha X az üres megoldáskezdemény, akkor* AKERES(X) *a probléma összes megoldását adja.*

Bizonyítás. Az alábbi feltétel a `while (A.nemUres())` kezdetű ismétlés ciklusinvariánsa lesz.

Bármely Y megoldás vagy bevégzett (azaz egyszer már kivettük A-ból), vagy van olyan $Z \in A$, hogy Y leszármazottja Z-nek a megoldástér fában (azaz Y érintetlen).

A ciklus előtt az állítás teljesül, mert minden pont érintetlen, az adagolóban csak az üres megoldáskezdemény van és minden megoldáskezdemény leszármazottja az üresnek.

Tegyük fel, hogy a feltétel teljesül a ciklusmag végrehajtása előtt és az adagolóból X-et vesszük ki. Továbbá tegyük fel, hogy Y nem bevégzett. Ekkor két eset lehet.

a.) Y leszármazottja X-nek. Ekkor vagy $Y = X$ és így Y bevégzett lesz, vagy Y leszármazottja X valamely Z fiának. De a ciklusban

X minden fia kigenerálásra kerül, továbbá LEHETMEGO(Z) biztosan igazat ad, mert van megoldás a Z-gyökerű fában, nevezetesen Y. Ezért Z-t betesszük az adagolóba.
b.) Y nem leszármazottja X-nek. Ekkor Y olyan $X \neq Z \in A$ leszármazottja, amely Z továbbra is A-ban marad.
Tehát mindkét esetben teljesül a feltétel a ciklusmag után.
Ebből következik az algoritmus helyessége, hisz a ciklus terminálása után az A adagoló üres, így minden megoldás bevégzett. ■

Az algoritmus futási ideje sok esetben erősen függ az aktuális pont választásától. Ezen kívül további kizárásokat is tehetünk. Tegyük fel, hogy minimalizációs feladatot kell megoldani. Tehát adott a $C(X)$ valós értékű célfüggvény, és olyan X megoldást keresünk, amelyre a célfüggvény $C(X)$ értéke minimális.

Tegyük fel továbbá, hogy a megoldáskezdeményekre meg tudunk adni olyan $AK(X)$ *alsó korlát* függvényt, amelyekre teljesülnek az alábbi egyenlőtlenségek.
Bármely X megoldáskezdeményre és minden olyan Y megoldásra, amely leszármazottja X-nek:

$$AK(X) \leq C(Y)$$

Ekkor az adagoló lehet az AK szerinti minimumos prioritási sor, tehát az aktív pontok közül mindig a legkisebb alsó korlátú pontot választjuk aktuálisnak.
Az keresés során tároljuk az addig megtalált megoldások célfüggvény értékeinek minimumát.

$$G\_F\_K = \min\{C(X) : \text{MEGOLDAS(X) ÉS X BEVÉGZETT}\}$$

Ekkor a következő kizárásokkal (korlátozásokkal) gyorsíthatjuk a keresést:

1. Ha az X új aktuális pontra $G\_F\_K < AK(X)$, akkor X kizárt lehet.
2. Ha a sorból az X pontot vettük ki és $G\_F\_K < AK(X)$, akkor a keresést befejezhetjük, hiszen nem kaphatunk már jobb megoldást.

Ha a megoldáskezdeményekre meg tudunk adni felső korlát is, akkor az adagoló lehet a felső korlát szerinti minimumos prioritási sor. Felső korlát olyan $FK(X)$ függvényt értünk, amelyre teljesül, hogy minden X megoldáskezdeményre és minden olyan Y megoldásra, amely leszármazottja X-nek:

$$C(Y) \leq FK(X)$$

Ekkor azonban a 2. kizárást nem alkalmazhatjuk.
Az $FK(X)$ felső korlátot erős felső korlátnak nevezzük, ha bármely X megoldáskezdeményre:

$$FK(X) < \infty \Rightarrow (\exists Y)(Y \trianglelefteq X \wedge \text{MEGOLDAS}(Y) \wedge C(Y) \leq FK(X))$$

Az erős felső korlát létezése azt jelenti, hogy bármely X megoldáskezdeményre, ha $FK(X) < \infty$, akkor biztosan létezik megoldás az X gyökerű megoldástér fában, és ennek célfüggvény értéke $\leq FK(X)$. Ekkor a keresést végezhetjük a felső korlát szerinti minimumos prioritási sorral úgy, hogy feljegyezzük az addig érintett pontok felső korlátjainak minimumát. Legyen ez a G_F_K. Tehát a keresés során azt állíthatjuk, hogy biztosan létezik olyan megoldás, amelynek célfüggvény értéke $\leq G\_F\_K$, de nem biztos, hogy már találtunk is ilyen megoldást. A 2. számú kizárást ennek ellenére továbbra is megtehetjük.

```
public abstract class EKMegold implements Cloneable, Comparable<EKMegold>{
 /** Ha vanfia, akkor a visszadott érték olyan uj objektum, amelynek értéke
 az első fiú, egyébként null.
*/
   public abstract EKMegold ElsoFiu();
/** Ha van még be nem járt testvére, akkor a visszadott érték olyan uj
    objektum, amelynek értéke a következő testvér, egyébként null.
*/
   public abstract EKMegold Testver();
/** Akkor és csak akkor ad igaz értéket, ha a megoldáskezdemény megoldása
 a problémának.
 */
   public abstract boolean Megoldas();
/** Hamis értéket ad, ha nincs megoldás az adott gyökerű részfában.
    Ha értéke igaz, abból nem következik, hogy van olyan folytatása,
    ami megoldás.
```

```
 */
   public abstract boolean LehetMego();
/** A célfüggvény. */
   public abstract float C();
/** Az alsókorlát függvény. */
   public abstract float AK();
/** A felsőkorlát függvény. */
   public abstract float FK();
/** Rendezés az alsókorlát szerint */
   public int compareTo(EKMegold X){
      return this.AK() < X.AK() ? -1: this.AK() > X.AK() ? 1: 0;
   }
/** A bemeneti adatok beolvasása és üres megoldáskezdemény előállítása.
 */
   public abstract void Input(String filenev)throws IOException;
/** A megoldás kiíratása. */
   public abstract void Output(String filenev)throws IOException;
   public EKMegold clone() throws CloneNotSupportedException{
      EKMegold result = this;
      try {
       result = (EKMegold)super.clone();
      } catch (CloneNotSupportedException e) {
         System.err.println("Az objektum nem klónozható");
      }
      return result;
   }

/* Megoldás keresése felsókorlát szerinti min. Prioritási sorral
  az X gyökerű megoldástér fában. */

public static EKMegold Keres1(EKMegold X)throws Exception{
   float G_F_K=Float.POSITIVE_INFINITY;
   PriSor<EKMegold> S=new PriSor<EKMegold>(1000);
   EKMegold X0=null;

   if (!X.LehetMego()) return null;
   S.SorBa(X);
   while ( S.Elemszam()>0 ){
      X=S.SorBol();
      if (G_F_K<=X.AK()) return X0;
      X = X.ElsoFiu();
      while ( X!=null ){
         if ( X.LehetMego() && X.AK()<G_F_K ){
            if (X.Megoldas() && X.C()<G_F_K){
               G_F_K=X.C();
               X0=X;
            }
            S.SorBa(X);
         }//else X kizárt
         X=X.Testver();
      }
   }//while
   return X0;
}

/* Megoldás keresése felsőkorlát szerinti min. Prioritási sorral,
 * erős felső korláttal. */
```

```
public static EKMegold Keres2(EKMegold X){
   float G_F_K=X.FK();
   PriSor<EKMegold> S=new PriSor<EKMegold>(1000);
   EKMegold X0=null;
   if (!X.LehetMego()) return null;
   S.SorBa(X);
   while ( S.Elemszam()>0 ){
      X=S.SorBol();
      if (G_F_K<X.AK()) continue;
      X = X.ElsoFiu();
      while ( X!=null ){
         if ( X.LehetMego() && X.AK()<=G_F_K ){
            if (X.Megoldas() && X.C()<=G_F_K){
               G_F_K=X.C();   X0=X;
            }else if (X.FK()<G_F_K)
               G_F_K=X.FK();
            S.SorBa(X);
         }//else X kizárt
         X=X.Testver();
      }
   }//while
   return X0;
}
}
```

9.3.1. Optimális pénzváltás

Az elágazás-korlátozás módszer alkalmazása az optimális pénzváltás probléma megoldására.
Probléma: 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.
Tegyük fel, hogy a pénzek nagyság szerint nemcsökkenő sorrendbe rendezettek: $p_1 \geq \ldots \geq p_n$. A megoldást keressük $X = \langle i_1, \ldots, i_m \rangle$, $i_1 < i_2 < \ldots < i_m$ alakú vektor formában. jelölje $Resz = \sum_{k=1}^{m} p_{i_k}$ és $Maradt = \sum_{j=i_m+1}^{n} p_j$ összegeket.

$$AK(X) = m + \lceil (E - Resz)/p_{i_m+1} \rceil$$
$$FK(X) = m + \lceil (E - Resz)/p_n \rceil$$

Ez a felső korlát nem erős, és nem is tudunk olcsón kiszámítható erős felső korlátot adni.

```
public class OPenzvalto extends EKMegold{
   private static int N, E;
   private int k;
   private static int Penz[];
   private int S[];
   private int resz;
   private int maradt;

   OPenzvalto(){
      k=0;
   }
   public OPenzvalto clone() {
      OPenzvalto co = this;
      try {
       co = (OPenzvalto)super.clone();
      } catch (CloneNotSupportedException e) {
         System.err.println("MyObject can't clone");
```

```
      }
      co.S=S.clone();
      return co;
   }

   public float C(){
      return k;
   }
   public float AK(){
      int i=S[k]<N ? S[k]+1 : N;
      return k+(float)Math.ceil((E-resz)/Penz[i]);
   }
   public float FK(){
      return k+(E-resz)/Penz[N];
   }

   public OPenzvalto ElsoFiu(){
      if (k<N && S[k]<N){
         OPenzvalto fiu=this.clone();
         ++fiu.k;
         fiu.S[k+1]=S[k]+1;
         fiu.resz+=Penz[S[k]+1];
         fiu.maradt-=Penz[S[k]+1];
         return fiu;
      }
      return null;
   }
   public OPenzvalto Testver(){
      if (k>0 && S[k]<N){
         OPenzvalto tv=this.clone();
         tv.resz+=Penz[S[k]+1]-Penz[S[k]];
         ++tv.S[k];
         tv.maradt-=Penz[S[k]];
         return tv;
      }
      return null;
   }

   public boolean Megoldas(){
      return resz==E;
   }
   public boolean LehetMego(){
      if (resz<=E && resz+maradt>=E){
         return true;
      }
      return false;
   }
}

   public void Input(String filenev)throws IOException{
      Scanner bef;
      if (filenev.length()==0)
         bef=new Scanner(System.in);
      else
         bef=new Scanner(new File(filenev));
```

```
      N=bef.nextInt();
      E=bef.nextInt();
      bef.nextLine();
      Penz=new int[N+1];
      S=new int[N+1];
      S[0]=0;
      resz=0;
      maradt=0;
      for (int i=1; i<=N; i++){
         Penz[i]=bef.nextInt();
         maradt+=Penz[i];
      }
      bef.close();
   }

   public void Output(String filenev)throws IOException{
      PrintWriter kif;
      if (filenev.length()==0)
         kif=new PrintWriter(System.out);
      else
         kif=new PrintWriter(
               new BufferedWriter(
                  new FileWriter(filenev)
               )
             );
      System.out.print(E+"=");
      for (int i=1; i<=k; i++)
         System.out.print(S[i]+"+ ");
       System.out.println();
   }
```

9.3.2. Ütemezési probléma

Bemenet:
$M = \{m_1, \ldots, m_n\}$ munkák halmaza
$m[i].idotartam \geq 0$ egész
$m[i].hatarido \geq 0$ egész
$m[i].haszon \geq 0$ valós
Kimenet:
$H \subseteq 1..n$
1. A H-beli munkák beoszthatók határidőt nem sértő módon.
2.

$$\overline{C}(H) = \sum_{i \in H} m[i].haszon \rightarrow maxi \tag{8}$$

H elemeinek egy $\langle i_1, \ldots, i_k \rangle$ felsorolása határidőt nem sértő, ha $\forall 1 \leq j \leq k$

$$\sum_{u=1}^{j} m[i_u].idotartam \leq m[i_j].hatarido \tag{9}$$

Állítás: H-nak akkor és csak akkor van határidőt nem sértő beosztása, ha elemeinek határidő szerinti felsorolása határidőt nem sértő.
$\Leftarrow$ triviális.
$\Rightarrow$ Tfh. H-nak van határidőt nem sértő beosztása, de ebben van olyan egymást követő u és $u+1$, hogy $m[i_u].hatarido > m[i_{u+1}].hatarido$. Ekkor u és $u+1$ felcserélhető a sorban.
Visszavezetés minimalizációs feladatra.

$$\begin{aligned} C(H) &= \sum_{i \notin H} m[i].haszon \\ &= \sum_{i=1}^{n} m[i].haszon - \overline{C}(H) \rightarrow mini \end{aligned}$$

$\overline{C}(H) \rightarrow maxi \Leftrightarrow C(H) \rightarrow mini$

Tegyük fel, hogy a munkák határidő szerint nemcsökkenő sorrendben vannak felsorolva. Ekkor a megoldás kifejezhető $X = \langle i_1, \ldots, i_k \rangle$ vektorral, ahol $i_1 < i_2 < \cdots < i_k$

Minden X megoldáskezdeményre definiáljuk. a problémaspecifikus műveleteket.

$LehetMego(X) = igaz \Leftrightarrow$ ha a felsorolás határidőt nem sértő.

$Megoldas(X) = LehetMego(X)$

$$AK(X) = \sum_{j<i_k, j \notin X} m[j].haszon \tag{10}$$

$$FK(X) = \sum_{j \notin X} m[j].haszon \tag{11}$$

Tehát, ha $LehetMego(X)$, akkor $Megoldas(X)$ és $C(X) = FK(X)$, ezért FK erős felső korlát.

```
public class Utemez extends EKMegold{
   private static int N;
   private int k;
   private int[] S;
   private static int[] Ido;
   private static int[] Hat;
   private static int[] Hasz;
   private int oido;       //a bevalasztott munkak összideje
   private int ehaszon;    //az elmaradt haszon
   private int maradt;     //a még választható munkák haszna

   Utemez(){
      k=0;
   }
   public Utemez clone() {
      Utemez co = this;
      try {
       co = (Utemez)super.clone();
      } catch (CloneNotSupportedException e) {
         System.err.println("MyObject can't clone");
      }
      co.S=S.clone();
      return co;
   }

   public float C(){
      return ehaszon+maradt;
   }
   public float AK(){
      return ehaszon;
   }
   public float FK(){
      return ehaszon+maradt;
   }
```

```
/** Rendezés a felső korlát szerint
 */
   public int compareTo(EKMegold X){
      return this.FK() < X.FK() ? -1: this.FK() > X.FK() ? 1: 0;
   }

   public Utemez ElsoFiu(){
      if (k<N && S[k]<N){
         Utemez fiu=this.clone();
         ++fiu.k;
         fiu.S[k+1]=S[k]+1;
         fiu.oido+=Ido[S[k]+1];
         fiu.maradt-=Hasz[S[k]+1];
         return fiu;
      }
      return null;
   }
   public Utemez Testver(){
      if (k>0 && S[k]<N){
         Utemez tv=this.clone();
         tv.oido+=Ido[S[k]+1]-Ido[S[k]];
         ++tv.S[k];
         tv.maradt-=Hasz[S[k]+1];
         tv.ehaszon+=Hasz[S[k]];
         return tv;
      }
      return null;
   }

   public boolean Megoldas(){
      return oido<=Hat[S[k]];
   }
   public boolean LehetMego(){
      return (oido<=Hat[S[k]]);
   }
}

   public void Input(String filenev)throws IOException{
      Scanner bef;
      if (filenev.length()==0)
         bef=new Scanner(System.in);
      else
         bef=new Scanner(new File(filenev));
      N=bef.nextInt();  bef.nextLine();
      S =new int[N+1];
      Ido =new int[N+1];
      Hat =new int[N+1];
      Hasz =new int[N+1];
      S[0]=0;
      for (int i=1; i<=N; i++){
         Ido[i]=bef.nextInt();
      } bef.nextLine();
      for (int i=1; i<=N; i++){
         Hat[i]=bef.nextInt();
      }bef.nextLine();
      for (int i=1; i<=N; i++){
```

```
      Hasz[i]=bef.nextInt();
      maradt+=Hasz[i];
   }
   bef.close();
}

public void Output(String filenev)throws IOException{
   PrintWriter kif;
   if (filenev.length()==0)
      kif=new PrintWriter(System.out);
   else
      kif=new PrintWriter(
            new BufferedWriter(
               new FileWriter(filenev)
            )
         );
   System.out.print("= ");
   for (int i=1; i<=k; i++)
      System.out.print(S[i]+"+ ");
    System.out.println();
}
```