Pénzváltás nagy számokkal

Adott n darab pénz; $p_1, \ldots, p_n$ és egy kifizetendő E összeg.

Feladat

Ijunk olyan programot, amely kiszámít egy kifizetést, azaz pénzek egy olyan halmazát, amelyek összege E!

Bemenet

A standard bemenet első sora két egész számot tartalmaz, a pénzek n ($1 \leq n \leq 25$) számát és a felváltandó E ($1 \leq E \leq 2\,000\,000\,000$) összeget. A második sor pontosan n pozitív egész számot tartalmat egy-egy szóközzel elválasztva, a felváltáshoz használható pénzek értékeit. Egy pénz csak egyszer szerepelhet a felváltásban.

Kimenet

A standard kimenet első sora a kifizetéshez kiválasztott pénzek m számot tartalmazza. A második sor tartalmazza a kifizetésben szereplő pénzek sorszámait egy-egy szóközzel elválasztva, tetszőleges sorrendben. Több megoldás esetén bárnelyik megadható.
Ha az E összeget nem lehet kifizetni, akkor az első és egyetlen sor a -1 számot tartalmazza.

Példa

Bemenet	Kimenet
`6 100000`	`2`
`1000 40000 50000 60000 70000 3000`	`2 4`

Korlátok

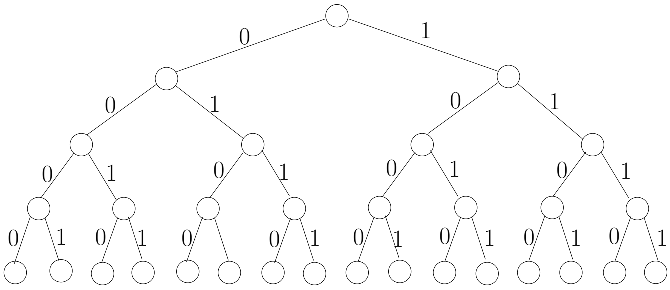
Időlimit: 1 mp.
Memórilimit: 32 MiB
Pontozás: a tesztesetek 40

Megoldás

Láttuk, hogy ha $n \times E$ nem túl nagy, azaz használhatunk $4 \times n \times E$ byte memóriát, akkor még optimális megoldást is elő tudunk állítani $n \times E$ -val arányos futási időben, a dinamikus programozás módszerével.
Esetünkben, mivel E nagyon nagy is lehet, ez a módszer nem alkalmazható.
Minden megoldása pénzek egy részhalmaza, ami megadható a kiválasztott pénzek indexeinek megadásával. Azaz egy megoldás az $\{1, \ldots, n\}$ halmaz egy olyan $M = \{i_1, \ldots i_k\} \subseteq \{1, \ldots, n\}$ részhalmaz, hogy

$$E = \sum_{u=1}^{k} p_{i_u}$$

Alkalmazhatjuk a nyers erőszak módszerét: vegyük sorra az összes lehetséges M részhalmazt, és ellenőrízzük, hogy az adott indexű pénzek összege E-e? Mivel összesen 2^n részhalmaz van, ezért az ilyen algoritmus futási ideje exponenciálisan függ n-től.
Gyorsíthatunk az algoritmuson, ha figyelembe vesszük, hogy ha egy M részhalmaz nem megoldás, akkor nem lehet megoldás semmilyen olyan $\overline{M}$ részhalmaz, amelynek M részhalmaza. Tehát a keresésből kizárjuk az ilyen $\overline{M}$ részhalmazokat.
Hogyan valósítható meg egy ilyen keresés?
A megoldás megadható egy sorozattal (vektorral). Tekintsük az összes olyan sorozatot, amely valamely megoldás vektor kezdőszelete. Ezek a vektorok alkotják a **megoldásteret**. Tehát úgy keresünk megoldást, hogy szisztematikusan sorravesszük a megoldástér elemeit és minden pontról eldöntjük, hogy megoldás-e. Ha megoldás, akkor befejezzük a keresést. Ha nem, akkor azt ellenőrizzük, hogy lehet-e olyan folytatása a megoldáskezdeménynek, amely megoldás. Ha nem lehet, akkor a keresésből kizárjuk az összes folytatását a megoldáskezdeménynek.



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

A megoldást kifejezhetjük és kereshetjük bitvektor formában, tehát olyan $X = \langle x_1, \dots, x_n \rangle$ vektort keresünk, amelyre

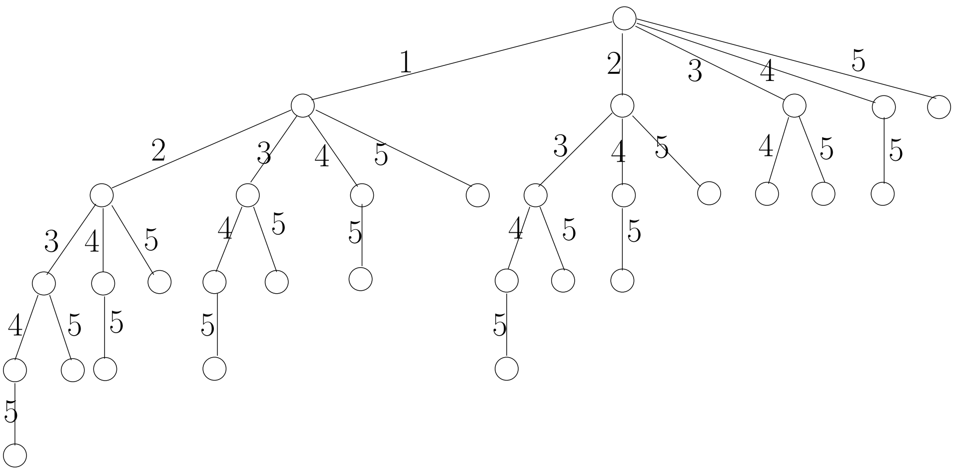
$$E = \sum_{i=1}^{n} x_i \, p_i$$

Ekkor a megoldástér fája bináris fa lesz.

A megoldást kifejezhetjük és kereshetjük mint a pénzek indexeinek olyan $M \subseteq \{1, \dots, n\}$ halmazának $X = \langle i_k, \dots, i_m \rangle$ növekvő felsorolásáként is, azaz $i_1 < i_2 < \dots < i_m$, hogy .

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

Ekkor a megoldástér formája a 8. ábrán látható $n = 5$ esetére. A megoldástér mindkét esetben fa szerkezetű, a



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

fa gyökere az üres megoldáskezdemény, és a q akkor és csak akkor fia a fa p pontjának, ha q küzvetlen folytatása a p megoldáskezdeménynek.

A megoldástér fájának bejárásához elegendő megadni az alábbi műveleteket:

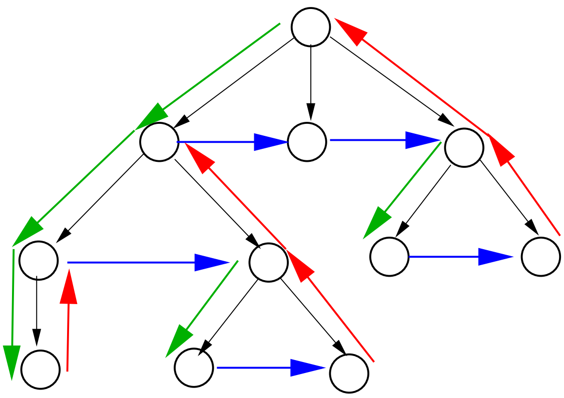
ElsoFiu(X) Ha van X-nek fia, akkor X az első fiúra változik és a függvényhívás értéke igaz, egyébként hamis és X nem változik.

Testver(X) Ha van X-nek még benemjárt testvére, akkor X a következő testvér lesz és a függvényhívás értéke igaz, egyébként hamis és X nem változik.

Apa(X) Ha van X-nek apja, akkor X az apjára változik és a függvényhívás értéke igaz, egyébként hamis és X nem változik.

Megoldas(X) Akkor és csak akkor ad igaz értéket, ha X megoldása a problémának.

Lehet(X) hamis értéket ad, ha nincs megoldás az X gyökerű részfában. Ha Lehet(X) igaz, abból nem következik, hogy van X-nek olyan folytatása, ami megoldás.



3. ábra. Fa teljes bejárása ElsoFiu, Testver, Apa műveletekkel

Ha a bejárás során olyan `p` pontba jutunk, amelyre a `Lehet(p)` hamis értéket ad, akkor a bejárásból kihagyható a `p`-gyökerű részfa minden pontja.

```
eljaras Keres(X)
van:=igaz; elsore:=igaz;
ciklus amíg van=igaz
    ha elsore=igaz akkor
        ha nem Lehet(X) akkor
            elsore=hamis
        egyébként
            ha Megoldas(X) akkor
                eljárás vége
            egyébként
                elsore=ElsoFiu(X)
            elágazás vége
        elágazás vége
    egyébként//nem elöször érintjük a pontot
        elsore=Testver(X)
        ha elsore=hamis akkor//visszalépés
            van=Apa(X)
        elágazás vége
    elágazás vége
ciklus vége
eljaras vege
```

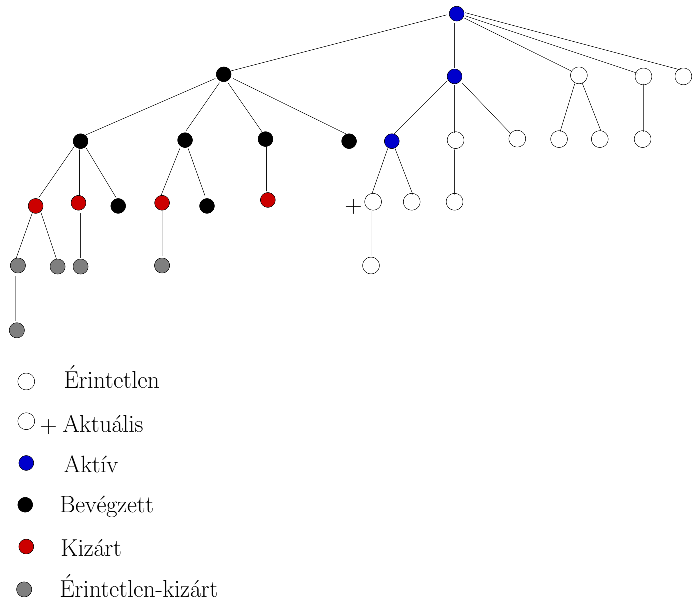
Látható, hogy a visszalépéses keresés algoritmusának fenti megfogalmazása nem függ attól hogy konkrétan mi a probléma, minden olyan probléma megoldására alkalmazható, ahol meg tudjuk adni az `ElsoFiu`, `Testver`, `Apa`, `Megoldas` és `Lehet` műveleteket.
Ha a pénzváltás problémánál a megoldáskezdeményt a beválasztott pénzek indexeinek (növekvő) felsorolásával adjuk meg, akkor a szükséges műveletek hatása az $X =< i_1, \ldots, i_k >$ esetén:

- ElsoFiu(X): $X =< i_1, \ldots, i_k, i_k + 1 >$ ha $i_k < n$
- Testver(X): $X =< i_1, \ldots, i_k + 1 >$ ha $i_k < n$
- Apa(X): $X =< i_1, \ldots, i_{k-1} >$ ha $k > 0$

Ahhoz, hogy a `Megoldás` és a `Lehet` műveleteket hatékonyan meg tudjuk valósítani, célszerű az $X =< i_1, \ldots, i_k >$ megoldáskezdemény esetén a megoldástér pontjában tárolni az

$$osszeg = \sum_{u=1}^{k} P[i_u] \text{ és } maradt = \sum_{u=i_k+1}^{n} P[u]$$

értékeket. Ekkor `Lehet(X)` akkor és csak akkor igaz, ha $X.osszeg \leq E$ és $X.osszeg + X.maradt \geq E$, továbbá `Megoldas(X)`= $X.osszeg = E$.



4. ábra. A megoldástér pontjainak osztályozása visszalépéses keresésnél

Érintetlen az olyan pontja a megoldástérnek, amelyet a keresés során még nem érintettünk.
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.
Látható, hogy a megoldás gyorsaságát alapvetően meghatározza az, hogy milyen kizárást tudunk megvalósítani a `Lehet` függvénnyel. A pénzváltás probléma esetén a futási idő legrosszabb esetben exponenciálisan függ a pénzek n számától, tehát 2^n-el arányos!

Megvalósítás C++ nyelven

```
#include <iostream>
#define maxN 25
using namespace std;
typedef struct{
   int E;      //a kifizetendő összeg
   int n;      //a pénzek száma
   int P[maxN];//a pénzek
   int k;      //a megoldáskezdemény elemszáma
   int V[maxN];//a megoldáskezdemény vektora
   int osszeg, //a beválasztott pénzek osszege
   maradt;     //még ennyi pénz maradt=szum(P[k+1..n])
} MTerPont;

bool Lehet(MTerPont &X);
bool Megoldas(MTerPont &X);
bool ElsoFiu(MTerPont &X);
bool Testver(MTerPont &X);
bool Apa(MTerPont &X);
//nem rekurzív visszalépéses keresés
bool Keres(MTerPont &X){
   bool van=true, elsore=true;
   while(van){
      if(elsore){
         if(!Lehet(X))
            elsore=false;
         else{
            if(Megoldas(X))
               return true;
            else
               elsore=ElsoFiu(X);
         }
      }else{//nem elsore érintjük a pontot
         elsore=Testver(X);
         if(!elsore)
            van=Apa(X);
      }
   }
   return van;
}

void KiIr(MTerPont X){
   for(int i=1;i<=X.k;i++) cout<<X.V[i]<<","; cout<<endl;
}

int main(){
   MTerPont X;
   bool van;
   cin>>X.n>>X.E;
   for(int i=1;i<=X.n;i++){
      cin>>X.P[i];
      X.maradt+=X.P[i];
   }
   X.osszeg=0; X.k=0; X.V[0]=0;
```

```
   van=Keres(X);
   if(van) KiIr(X);
return 0;
}
bool Lehet(MTerPont &X){
   return X.osszeg<=X.E && X.osszeg+X.maradt>=X.E;
}
bool Megoldas(MTerPont &X){
   return X.osszeg==X.E;
}
bool ElsoFiu(MTerPont &X){
   if(X.V[X.k]<X.n){
      X.V[X.k+1]=X.V[X.k]+1;
      (X.k)++;
      X.osszeg+=X.P[X.V[X.k]];
      X.maradt-=X.P[X.V[X.k]];
      return true;
   }else
      return false;
}
bool Testver(MTerPont &X){
   if(X.V[X.k]<X.n){
      X.osszeg-=X.P[X.V[X.k]];
      X.V[X.k]++;
      X.osszeg+=X.P[X.V[X.k]];
      X.maradt-=X.P[X.V[X.k]];
      return true;
   }else
      return false;
}
bool Apa(MTerPont &X){
   if(X.k>1){
      X.osszeg-=X.P[X.V[X.k]];
      X.k--;
      return true;
   }else
      return false;
}
```

A visszalépéses keresés megfogalmazható rekurzív függvénnyel is. Ekkor azonban a függvény paramétere csak a legszükségesebb adatokat tartalmazza, minden más legyen globális. Ha a nemrekurzív változatban használt `MTerPont` típust használnánk, akkor nagyon memória pazarló lenne a megoldás. Elegendő a (k, u) párt megadni, ahol k a megoldáskezdemény elemszáma, u pedig a megoldáskezdemény utolsó eleme. Célszerű továbbá az *osszeg* és *maradt* értékeket is tárolni a pontban.

Rekurzív megvalósítás

```
#include <iostream>
#define maxN 25
using namespace std;
typedef struct{
   int k;      //a megoldáskezdemény elemszáma
   int u;      //a megoldáskezdemény k. eleme u
   int osszeg, //a beválasztott pénzek osszege
   maradt;     //még ennyi pénz maradt=szum(P[k+1..n])
} MTerPont;
```

```

typedef struct{
   int E;      //a kifizetendő összeg
   int n;      //a pénzek száma
   int P[maxN];//a pénzek
   int X[maxN];//a megoldásvektor
} Global;
Global GData;

bool Lehet(MTerPont &X);
bool Megoldas(MTerPont &X);
bool EFiu(MTerPont &X);
bool Testver(MTerPont &X);

bool RKeres(MTerPont X){
   if(Megoldas(X)) return true;
   if(!EFiu(X)) return false;
   do{
      if (Lehet(X))
         if (RKeres(X)){
            GData.X[X.k]=X.u; //a megoldás komponens bejegyzése
            return true;
         }
   }while(Testver(X));
}

void KiIr(){
   for(int i=1;i<=GData.n;i++) cout<<GData.X[i]<<",";
}

int main(){
   MTerPont X;
   bool van;
   cin>>GData.n>>GData.E;
   X.maradt=0;
   for(int i=1;i<=GData.n;i++){
      cin>>GData.P[i];
      X.maradt+=GData.P[i];
   }
   X.osszeg=0; X.k=0; X.u=0;
   RKeres(X);
   KiIr();

return 0;
}
bool Lehet(MTerPont &X){
   return true;
   return X.osszeg<=GData.E && X.osszeg+X.maradt>=GData.E;
}
bool Megoldas(MTerPont &X){
   return X.osszeg==GData.E;
}
bool EFiu(MTerPont &X){
   if(X.u<GData.n){
      X.k++;
```

```
      X.u++;
      X.osszeg+=GData.P[X.u];
      X.maradt-=GData.P[X.u];
      return true;
   }else
      return false;
}
bool Testver(MTerPont &X){
   if(X.u<GData.n){
      X.osszeg-=GData.P[X.u];
      X.u++;
      X.osszeg+=GData.P[X.u];
      X.maradt-=GData.P[X.u];
      return true;
   }else
      return false;
}
```