7. Dinamikus programozás

7.1. Rekurzió memorizálással.

Láttuk, hogy a partíció probléma rekurzív algoritmusa $\Omega(2^{\sqrt{n}})$ eljáráshívást végez, pedig a lehetséges részproblémák száma csak n^2 (vagy $n(n+1)/2$, ha csak az $n \leq k$ eseteket vesszük.) Ennek az az oka, hogy ugyanazon részprobléma megoldása több más részprobléma megoldásához kell, és az algoritmus ezeket mindig újra kiszámítja. Tehát egyerűen gyorsíthatjuk a számítást, ha minden részprobléma (azaz $P2(n,k)$) megoldását tároljuk egy tömbben. Ha hivatkozunk egy részprobléma megoldására, akkor először ellenőrizzük, hogy kiszámítottuk-e már, és ha igen, akkor kiolvassuk az értéket a táblázatból, egyebként rekurzívan számítunk, és utána tároljuk az értéket a táblázatban.
A táblázat inicializálásához válasszunk egy olyan értéket, amely nem lehet egyetlen részprobléma megoldása sem. Esetünkben ez lehet a 0.

```
public class ParticioRM{
   private static long[][] T2;

   public static long P(int n){
      T2=new long[n+1][n+1];
      return P2(n,n);
   }//P
   private static long P2(int n, int k){
      if (T2[n][k]!=0)              //P2(n,k)-t már kiszámítottuk
         return T2[n][k];           //értéke a T2 táblázatban
      long Pnk;
      if (k==1 || n==1)             //rekurzív számítás
         Pnk=1;
      else if (k>=n)
         Pnk=P2(n,n-1)+1;
      else
         Pnk=P2(n, k-1)+P2(n-k, k);
      T2[n][k]=Pnk;                 //memorizálás
      return Pnk;
   }//P2
}
```

Nyilvánvaló, hogy az algoritmus futási ideje $\Theta(n^2)$, és a tárigénye is $\Theta(n^2)$ lesz, ha csak n^2 méretű táblázatnak foglalunk memóriát dinamikusan az aktuális paraméter függvényében.

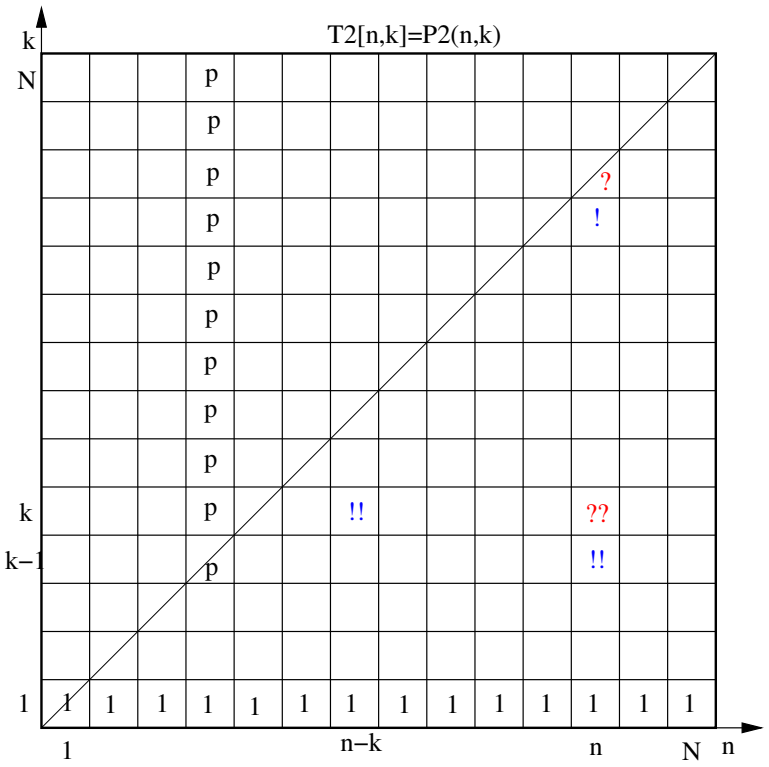
7.2. A partíció probléma megoldása táblázat-kitöltéssel.

A rekurziót teljesen kiküszöbölhetjük táblázat-kitöltéssel. Az 1. árán szemléltetett táblázatot használjunk a részproblémák megoldásainak tárolására. Tehát a $T2[n,k]$ táblázatelem tartalmazza a $P2(n,k)$ részprobléma megoldását. A táblázat első sora azonnal kitölthető, mert $P2(n,1)=1$. Olyan kitöltési sorrendet keresünk, hogy minden $(n,k), k>1$ részprobléma kiszámítása esetén azok a részproblémák, amelyek szükségesek $P2(n,k)$ kiszámításához, már korábban kiszámítottak legyenek.

Általánosan, rekurzív összefüggéssel definiált problémamegoldás esetén egy r (rész)probléma *összetevői* azok a részproblémák, amelyek megoldásától r megoldása függ. Tehát a táblázat-kitöltéssel alkalmazásához meg kell állapítani a részproblémáknak egy olyan sorrendjét, hogy minden r részprobléma minden összetevője előbb álljon a sorrendben, mint r. A

1. $P2(1,k)=1, P2(n,1)=1$,
2. $P2(n,n)=1+P2(n,n-1)$,
3. $P2(n,k)=P2(n,n)$ ha $n<k$,
4. $P2(n,k)=P2(n,k-1)+P2(n-k,k)$ ha $k<n$.

rekurzív összefüggések megadják az összetevőket:



1. ábra. Táblázat a Partíció probléma megoldásához.

1. $P2(1,k)$-nak és $P2(n,1)$-nek nincs összetevője,
2. $P2(n,n)$ összetevője $P2(n,n-1)$,
3. $P2(n,k)$ összetevője $P2(n,n)$, ha ($n<k$),
4. $P2(n,k)$ összetevői: $P2(n,k-1)$ és $P2(n-k,k)$, ha ($k<n$).

Tehát a táblázat kitöltése (k-szerint) soronként balról jobbra haladó lehet.
Az algoritmus futási ideje és tárigénye is $\Theta(n^2)$.

```
public static long P(int n){
   long[][] T2=new long[n+1][n+1];

   for (int i=1; i<=n; i++)
      T2[i][1]=1;                          //az első sor kitöltése
   for (int ki=2; ki<=n; ki++){            //az ki. sor kitöltése
      T2[ki][ki]=T2[ki][ki-1]+1;           //P2(n,n)=P2(n,n-1)+1
      for (int ni=ki+1; ni<=n; ni++){      //P2(ni,ki)=T2[ni,ki] számítása
         int n1=(ni-ki<ki) ? ni-ki : ki;   //P2(n,k)=P2(n,n), ha k>n
         T2[ni][ki]=T2[ni][ki-1]+T2[ni-ki][n1];//P2(n,k)=P2(n,k-1)+P2(n-k,k)
      }
   }
   return T2[n][n];
}
```

7.3. A partíció probléma megoldása lineáris táblázat-kitöltéssel.

Látható, hogy elegendő lenne a táblázatnak csak két sorát tárolni, mert minden (n,k) részprobléma összetevői vagy a k-adik, vagy a $k-1$-edik sorban vannak. Sőt, elég egy sort tárolni balról-jobbra (növekvő n-szerint) haladó kitöltésnél, mert amelyik részproblémát felülírjuk ($(n-k,k)$), annak később éppen az új értéke kell összetevőként.

```
public static long P(int n){
   long[] T=new long[n+1];

   for (int i=1; i<=n; i++)            //az első sor kitöltése
      T[i]=1;
   for (int ki=2; ki<=n; ki++){        //az ki. sor kitöltése
      ++T[ki];                         //P2(n,n)=P2(n,n-1)+1
      for (int ni=ki+1; ni<=n; ni++){  //
            T[ni]=T[ni]+T[ni-ki];      //P2(n,k)=P2(n,k-1)+P2(n-k,k)
      }
   }
   return T[n];
}
```

P(405)= 9147679068859117602

7.4. 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.
Először azt határozzuk meg, hogy van-e megoldás.
A megoldás szerkezetének elemzése.
Tegyük fel, hogy

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

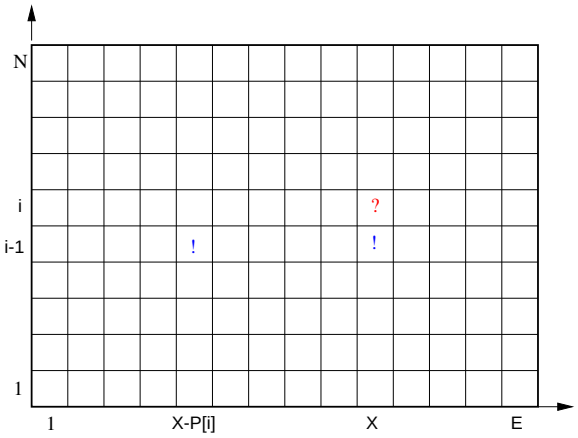
egy megoldása a feladatnak. Ekkor

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

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 $i > 1$
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.

```
public boolean V(int X, int i){
 // Globális:P
 // Módszer: Rekurzív megoldás
  return (X==P[i]) ||
         (i>1) && V(X,i-1) ||
         (i>1) && (X>P[i]) && V(X-P[i],i-1);
}
```

Ez a megoldás azonban igen lassú, legrosszabb esetben a futási idő $\Omega(2^n)$.
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**



2. ábra. A pénzváltás táblázata

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. **Egy megoldás előállítása a megoldás visszafejtésével.** Akkor és csak akkor van megoldása a problémának, ha a VT táblázat kitöltése után $VT[E,N]$ értéke igaz. Ekkor az (1-2.) képletek szerint a legnagyobb i indexű p_i pénz, amely szerepelhet E előállításában, az a legnagyobb index, amelyre

$$VT[E,i] = True \wedge (VT[E,i-1] = False)$$

De ekkor $VT[E-P[i],i-1]$ igaz, tehát $E-p_i$ előállítható az első $i-1$ pénz felhasználásával. Tehát a fenti eljárást folytatni kell $E := E-p_i, i := i-1$-re mindaddig, amíg E 0 lesz.

```
public class PenzValt1{

 public static int[] Valto(int E, int[] P){
   int n=P.length;
   int MaxE=1000;
   boolean VT[][]=new boolean[E+1][n+1];
   int i,x;

   for (x=1; x<=E; x++)       //inicializálás
      VT[x][0]=false;
   VT[0][0]=true;
   VT[P[0]][0]=true;
   for (i=1; i<n; i++)
      for (x=1; x<=E; x++)
         VT[x][i]=P[i]==x ||
                  VT[x][i-1] ||
                  x>=P[i] && VT[x-P[i]][i-1];
   int db=0; x=E; i=n-1;
   if (!VT[E][n-1]) return null;
   int C[]=new int[n];
   do{                        //megoldás visszafejtés
      while (i>=0 && VT[x][i])
         i--;
      C[db++]=++i;
      x-=P[i];
```

```
   }while(x>0);
   for (i=db; i<n; i++)
      C[i]=0;
   return C;
 }
}
```

Ha csak arra kell válaszolni, hogy létezi-e megoldása a problémának, akkor elég a táblázat egy sorát tárolni, mert soronként visszafelé (x-szerint csökkenő sorrendben) haladó kitöltést alkalmazhatunk.

```
public class PenzValt1L{
 public static boolean Valto(int E, int[] P){
   int n=P.length;
   boolean VT[]=new boolean[E+1];
   int i,x;

   for (x=1; x<=E; x++)
      VT[x]=false;
   VT[0]=true;
   if (P[0]<=E) VT[P[0]]=true;

   for (i=1; i<n; i++)
      for (x=E; x>0; x--)
         VT[x]=P[i]==x ||
               VT[x] ||
               x>=P[i] && VT[x-P[i]];

   return VT[E];
 }
}
```

7.5. Az optimális pénzváltás probléma.

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
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 optimális megoldásaival.**

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

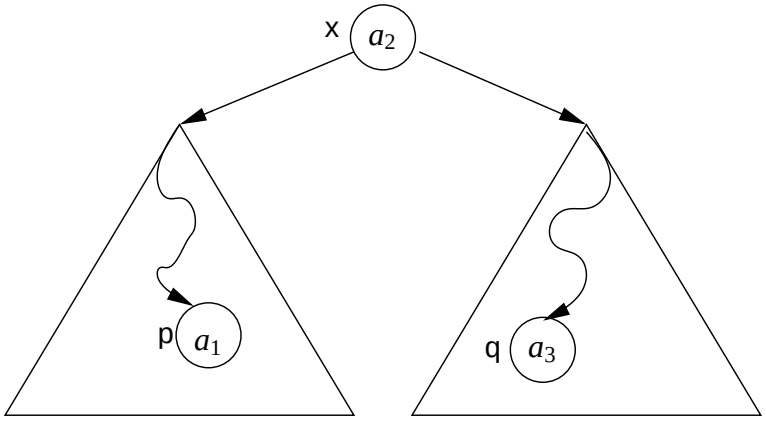
```
public class OptPenzValt {
 public static int[] Valto(int E, int[] P){
   int n=P.length;
   int [] Opt=new int[E+1];
   int V[][]=new int[E+1][n+1];
   int i,x,ropt;

   for (x=1; x<=E; x++){
      Opt[x]=n+1;
      V[x][0]=n+1;
   }
   Opt[0]=0;
   if (P[0]<=E){
      Opt[P[0]]=1;
      V[P[0]][0]=0;
   }

   for (i=1; i<n; i++)
      for (x=E; x>0; x--){
         if (x>=P[i])
            ropt=Opt[x-P[i]]+1;
         else
            ropt=n+1;
         if (ropt<Opt[x]) {
            Opt[x]=ropt;
            V[x][i]=i;
         }else
            V[x][i]=V[x][i-1];
      }

   int db=0; x=E; i=n-1;
   if (Opt[E]>n) return null;
      do{
      i=V[x][i];
      db++;
      x-=P[i];
      --i;
   }while(x>0);
   int C[]=new int[db];
   db=0; x=E; i=n-1;
   do{
      i=V[x][i];
      C[db++]=i;
      x-=P[i];
      --i;
   }while(x>0);

   return C;
 }
}
```

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.

7.6. Optimális bináris keresőfa előállítása

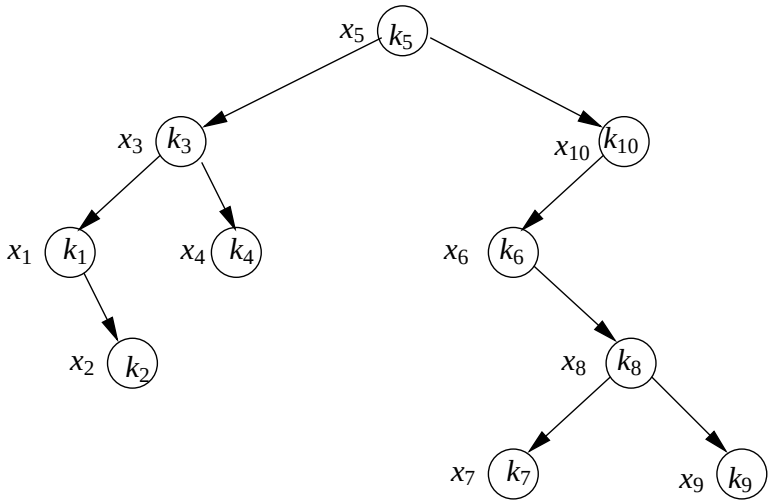
A $F = (M, R, Adat)$ absztrakt adatszerkezetet *bináris keresőfának* nevezzük, ha
1. F bináris fa,
2. $Adat : M \to Elemtip$ és $Elemtip$-on értelmezett egy $\leq$ lineáris rendezési reláció,
3. $(\forall x \in M)(\forall p \in F_{bal(x)})(\forall q \in F_{jobb(x)})(Adat(p) \leq Adat(x) \leq Adat(q))$
A BINKERFAKERES függvényeljárás egy nyilvánvaló megoldása a fában keresés feladatnak.



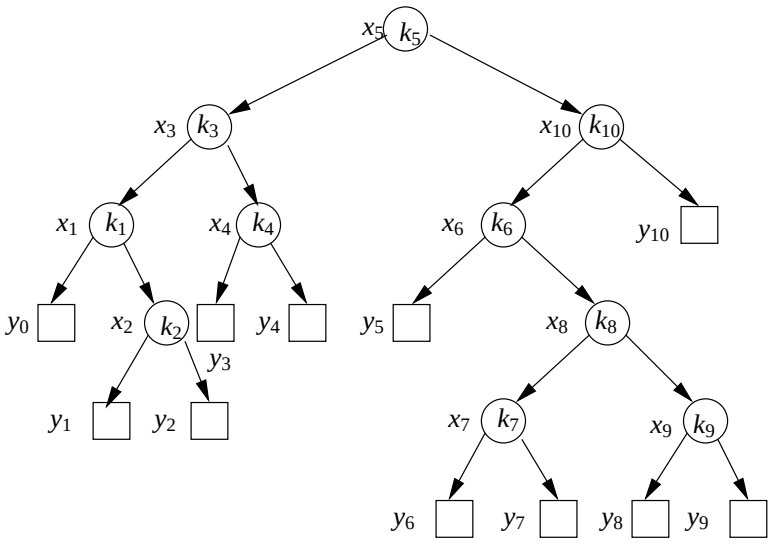
3. ábra. Bináris keresőfa

```
public BinFa<E> BinKerFaKeres(E a, BinFA F){
   while (F!=null) && (a!=F.elem)
      if (a<F.elem)
         F=F.bal
      else
         F=F.jobb;
  return F;
}
```

Tegyük fel, hogy ismerjük minden k_i kulcs keresési gyakoriságát, ami p_i ($i = 1, \ldots, n$) Továbbá ismert azon k kulcsok (sikertelen) keresési gyakorisága, amelyre $k_i < k < k_{i+1}$, ami q_i (i=1,. . . ,n), és q_0 a $k < k_1$ kulcsok keresési gyakorisága.



4. ábra. 10 adatot (kulcsot) tartalmazó bináris keresőfa

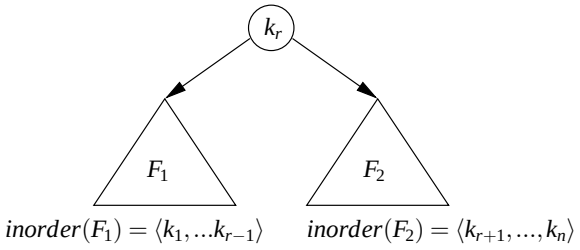


5. ábra. Bináris keresőfa kiegészítve sikertelen keresési pontokkal

Átlagos keresési idő (költség):

$$V(F) = \sum_{i=1}^{n} p_i \, d_F(x_i) + \sum_{i=0}^{n} q_i \, d_F(y_i)$$

, ahol $d_F(p)$ a p pont mélysége az F fában. **Probléma:** Optimális bináris keresőfa előállítása.
Bemenet: $P = \langle p_1, \ldots, p_n \rangle$ sikeres és $Q = \langle q_0, \ldots, q_n \rangle$ sikertelen keresési gyakoriságok.
Kimenet: Olyan F bináris keresőfa, amelynek a $V(F)$ költsége minimális.
Az optimális megoldás szerkezete.
Tegyük fel, hogy a $\langle k_1, \ldots, k_n \rangle$ kulcsokat tartalmazó F bináris keresőfa optimális, azaz $V(F)$ minimális. Jelölje x_r a fa gyökerét. Ekkor az $F_1 = F_{bal(x_r)}$ fa a $\langle k_1, \ldots, k_{r-1} \rangle$ kulcsokat, az $F_2 = F_{jobb(x_r)}$ fa pedig a $\langle k_{r+1}, \ldots, k_n \rangle$ kulcsokat tartalmazza. Mivel



6. ábra. Ha az optimális fa gyökerében a k_r kulcs van.

$d_F(x) = d_{F_1}(x) + 1$ és $d_F(x) = d_{F_2}(x) + 1$.

$$\begin{aligned}
V(F) &= \sum_{i=1}^{n} p_i \, d_F(x_i) + \sum_{i=0}^{n} q_i \, d_F(y_i) \\
&= \sum_{i=1}^{r-1} p_i \, d_F(x_i) + \sum_{i=0}^{r-1} q_i \, d_F(y_i) + p_r + \sum_{i=r+1}^{n} p_i \, d_F(x_i) + \sum_{i=r}^{n} q_i \, d_F(y_i) \\
&= \sum_{i=1}^{r-1} p_i \, (d_{F_1}(x_i) + 1) + \sum_{i=0}^{r-1} q_i \, (d_{F_1}(y_i) + 1) + p_r \\
&+ \sum_{i=r+1}^{n} p_i \, (d_{F_2}(x_i) + 1) + \sum_{i=r+1}^{n} q_i \, (d_{F_2}(y_i) + 1) \\
&= \sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i + \sum_{i=1}^{r-1} p_i \, d_{F_1}(x_i) + \sum_{i=0}^{r-1} q_i \, d_{F_1}(y_i) \\
&+ \sum_{i=r+1}^{n} p_i \, d_{F_2}(x_i) + \sum_{i=r}^{n} q_i \, d_{F_2}(y_i) \\
&= \sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i + V(F_1) + V(F_2)
\end{aligned}$$

Tehát

$$V(F) = \sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i + V(F_1) + V(F_2) \tag{2}$$

Az F_1 fa a $\langle k_1, \ldots, k_{r-1} \rangle$ kulcsokat tartalmazó optimális bináris keresőfa a $\langle p_1, \ldots, p_{r-1} \rangle$ sikeres és $\langle q_0, \ldots, q_{r-1} \rangle$ sikertelen keresési gyakoriságokra, az F_2 fa pedig $\langle k_{r+1}, \ldots, k_n \rangle$ kulcsokat tartalmazó optimális bináris keresőfa a $\langle p_{r+1}, \ldots, p_n \rangle$ sikeres és $\langle q_r, \ldots, q_n \rangle$ sikertelen keresési gyakoriságokra. A bizonyítás a kivágás-és-beillesztés módszerrel végezhető. Ha lenne olyan $\overline{F}_1$ bináris keresőfa a $\langle p_1, \ldots, p_{r-1} \rangle$ sikeres és $\langle q_0, \ldots, q_{r-1} \rangle$ sikertelen keresési gyakoriságokra, hogy $V(\overline{F}_1) < V(F_1)$, akkor az F fában F_1 helyett az $\overline{F}_1$ részfát véve olyan fát kapnánk a $\langle p_1, \ldots, p_n \rangle$ sikeres és $\langle q_0, \ldots, q_n \rangle$ sikertelen keresési gyakoriságokra, amelynek költsége $\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i + V(\overline{F}_1) + V(F_2) < V(F)$. Ugyanígy bizonyítható, hogy F_2 is optimális fa a $\langle k_{r+1}, \ldots, k_n \rangle$ kulcsokra a $\langle p_{r+1}, \ldots, p_n \rangle$ sikeres és $\langle q_r, \ldots, q_n \rangle$ sikertelen keresési gyakoriságokra.
Részproblémákra bontás.

Minden (i,j) indexpárra $0 \le i \le j \le n$ tekintsük azt a részproblémát hogy mi az optimális bináris keresőfa az $\langle p_{i+1}, \ldots, p_j \rangle$ sikeres és $\langle q_i, \ldots, q_j \rangle$ sikertelen keresési gyakoriságokra. Jelölje $Opt(i,j)$ az optimális fa költségét az (i,j) részproblémára.

Az optimális megoldás értékének rekurzív kiszámítása.

Vezessük be a

$$W(i,j) = \sum_{u=i+1}^{j} p_u + \sum_{u=i}^{j} q_u$$

jelölést.

Minden (i,j)-re a (2) képlet miatt biztosan létezik olyan $i < r \le j$, hogy
$Opt(i,j) = W(i,j) + Opt(i,r-1) + Opt(r,j)$, csak azt nem tudjuk, hogy melyik r-re. Tehát azt az r-et keressük, amelyre a fenti összeg minimális lesz. Tehát $Opt(i,j)$ a következő rekurzív összefüggéssel számítható.

$$Opt(i,j) = \begin{cases} q_i & \text{ha} \quad i = j \\ W(i,j) + \min\limits_{i<r\le j} (Opt(i,r-1) + Opt(r,j)) & \text{ha} \quad i < j \end{cases} \tag{3}$$

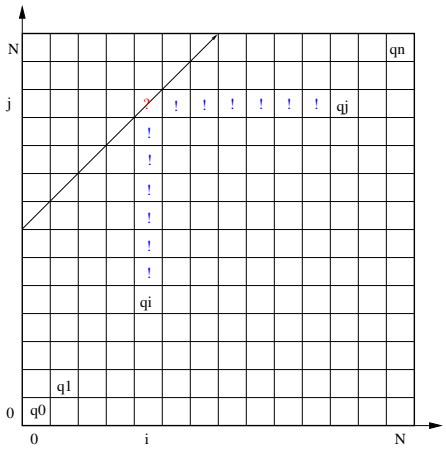
Az összetevők és a kiszámítási sorrend meghatározása.

Az (i,i) részproblémáknak nincs összetevőjük, mert $Opt(i,i) = q_i$.

Az (i,j), $i < j$ részprobléma összetevői az $(i,r-1)$ és (r,j), $r = i+1, \ldots, j$ részproblémák.

Tehát a táblázatot ki tudjuk tölteni átlósan haladva, a m-edik átlóban $m = 1, \ldots, n$ azon
(i,j)részproblémákat számítjuk, amelyekre $j - i = m$.

Kiszámítás alulról-felfelé haladva (táblázat-kitöltés).



7. ábra. Táblázat-kitöltési sorrend

Ahhoz, hogy egy optimális megoldást elő tudjunk állítani, minden (i,j) részproblémára tároljuk egy G táblázat $G[i,j]$-elemében azt az r értéket, amelyre a (3) képletben az minimum előáll. Ez az r lesz a $\langle k_{i+1}, \ldots, k_j \rangle$ kulcsokat tartalmazó optimális bináris keresőfa gyökere. A $G[i,j]$ értékeket felhasználva a FASIT rekurzív eljárás állítja elő ténylegesen az algoritmus kimenetét jelentő keresőfát.

```
public class OptBinKerFa {
   public class BinFapont{
      int bal, jobb;
   }
   private void Fasit(int Apa, int i, int j){
  // Előallítja az i+1..j elemek OB keresőfáját a G értékekből}
```

```
  // Globális: G, F
      if (Apa!=0){
         F[Apa].bal = G[i][Apa-1];
         F[Apa].jobb = G[Apa][j];
         Fasit(G[i][Apa-1], i, Apa-1);
         Fasit(G[Apa][j], Apa, j);
      }
   }
   private int[][] G;
   private BinFapont[] F;

public BinFapont[] Epit(float[] P, float[] Q){
   int n=P.length-1;
   F=new BinFapont[n+1];
   float[][] Opt=new float[n+1][n+1];
   float[][] W=new float[n+1][n+1];
   G=new int[n+1][n+1];

   int optr;
   float V, optV;

   for (int i=0; i<=n; i++){  //inicializálás
      W[i][i]=Q[i];
      G[i][i]=0;
      Opt[i][i]=Q[i];
      F[i]=new BinFapont();
   }

   for (int m=1; m<=n; m++)
      for (int i=0; i<=n-m; i++){
         int j=i+m;
         W[i][j]=W[i][j-1]+P[j]+Q[j];
         optr=j;
         optV=Opt[i][j-1]+Q[j]; //=Opt[j,j]
         for (int r=i+1; r<=j-1; r++){
            V = Opt[i][r-1]+Opt[r][j];
            if (V < optV){
               optV=V; optr=r;
            }
         };
      Opt[i][j]=W[i][j]+optV;
      G[i][j]=optr;
      }

   Fasit(G[0][n], 0, n);
   F[0].bal=G[0][n];
   return F;
 }
}
```

A OPTBINKERFA algoritmus futási ideje $\Theta(n^3)$.
Bizonyítás nélkül megjegyezzük, hogy a

```
      for (int r=i+1; r<=j-1; r++)
```

ciklusban elegendő lenne az `r` ciklusváltozót `G[i,j-1]+1` -től `G[i+1,j]`-ig futtatni, és ezzel az algoritmus futási ideje $\Theta(n^2)$ lenne.