Algoritmusok és adatszerkezetek gyakorlat – 04

Dinamikus programozás

Dinamikus programozás

A rekurzív algoritmusok sokszor azért lassúak, mert bizonyos részproblémákat többször is kiszámolnak. Ezt tudjuk elkerülni azzal, ha az egyszer már kiszámolt részmegoldásokat eltároljuk és újra felhasználjuk.

A több időt több tárra „cseréljük”: a rekurzióval egyébként exponenciális idő alatt megoldható problémákat polinomidőben megoldhatjuk. Ehhez viszont el kell tároljuk a részproblémák megoldását.

A megoldás lépései általában a következők:

- Adjuk meg a **rekurzív megoldás**t (alap- és rekurzív eset).
- A rekurzió alapján építsük fel **„lentről felfelé”**: az algoritmus kezdjen az alapesettel, majd a teljes megoldásig egy jó sorrendben számolja ki a részproblémákat.
  - **A részproblémák beazonosítása:** milyen különböző rekurzív hívások jöhetnek szóba? Milyen inputokat kaphat a program?
  - **Az adatszerkezet kiválasztása a részproblémák megoldásainak memorizálásához:** A legtöbb esetben minden részprobléma azonosítható néhány egész számmal, így pl használhatunk egy több dimenziós tömböt. Néha viszont ennél bonyolultabb adatszerkezetre van szükség.
  - **Idő- és tárigény:** A lehetséges részproblémák száma ad egy tárigényt. Az időigényt a részproblémák számának és a részproblémák kiszámolásához szükséges időnek a szorzataként kapjuk.
  - **Részproblémák közötti függőségek azonosítása:** Az alapesetek kivételével az összes rekurzív részprobléma függ a többitől. De melyektől? Jó taktika: rajzold le egy egyszerű példán!
  - **Jó kiértékelési sorrend megadása:** Olyan sorrendbe kell rendezni a részproblémákat, hogy minden probléma mire sorra kerül, addigra minden, amitől függ, már ki legyen számolva.

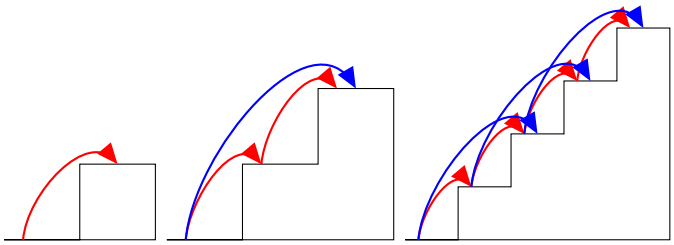
A DP nem csak táblázatkitöltés! Ha az adatszerkezet megválasztása, a rekurzív összefüggés vagy a hívási sorrend rossz, az egész algoritmus rossz!

1. Feladat Jelölje $P(n)$ azt a számot, ahányféleképpen mehetünk fel egy n lépcsőfokból álló lépcsőn, ha egyszerre csak 1 vagy 2 lépcsőfokot léphetünk. Adjunk egy dinamikus programozási eljárást a $P(n)$ érték kiszámítására!

Megoldás

Az előző órán a rekurzív algoritmushoz megadtuk az összefüggéseket:

- $P(1) = 1$ (ha egy lépcső van, egyféleképpen mehetünk)
- $P(2) = 2$ (ha két lépcső van, vagy kétszer egyet lépünk, vagy egyszer kettőt)
- $P(n) = P(n-1) + P(n-2)$, ha $n \geq 3$ (utolsó lépésként egyet vagy kettőt léphetünk)



Ezek alapján a következő dinamikus programozási eljárást adhatjuk meg, ahol egy T tömböt töltünk ki, $T[i]$ a $P(i)$ értékét tartalmazza:

Algoritmus:

```
int P(int n){
  int T[n+1];
  T[1] = 1;
  T[2] = 2;
  for(int i = 3; i<=n; i++)
    T[i] = T[i-2] + T[i-1];
  return T[n];
}
```

Nézzünk egy példát: egy 8 magas lépcsőre szeretnénk felmenni.

i	0	1	2	3	4	5	6	7	8
$T[i]$	–	1	2	3	5	8	13	21	34

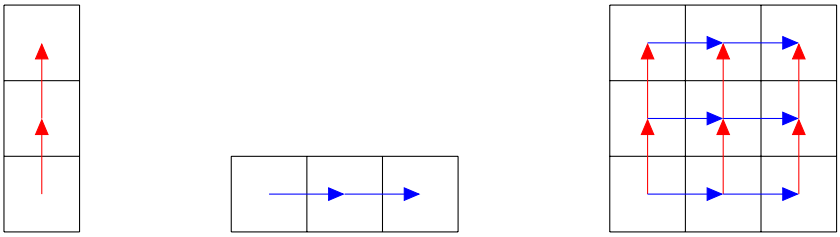
Itt futtatva az előző órán megírt rekurzív algoritmust, láthatjuk, hogy észrevehetően sok számítást spóroltunk meg már ezen a kicsi inputon is.

2. Feladat Jelölje $R(k, n)$ azt a számot, ahányféleképpen eljuthatunk egy $k \times n$ méretű sakktábla bal alsó sarkából a jobb felső sarkába, ha csak a jobbra vagy a felfelé szomszédos mezőre léphetünk. Adjunk meg két dinamikus programozási eljárást (egyet négyzetes, egyet lineáris táblázatkitöltéssel) az $R(k, n)$ érték kiszámítására!

Megoldás

Az előző órán megadtuk a rekurzív algoritmusokhoz az összefüggéseket. A következő összefüggések állnak fent:

- $R(k, 1) = 1$ (csak felfelé mehetünk)
- $R(k, n) = 1$ (csak jobbra mehetünk)
- $R(k, n) = R(k, n-1) + R(k-1, n)$, ha $n, k \geq 2$ (az első lépés jobbra vagy felfelé történhet)



Ezek alapján a következő dinamikus programozási eljárást adhatjuk meg, $T[i, j]$ az $R(i, j)$ értékét tartalmazza:

Algoritmus:

```
int R(int k, int n){
  int T[k+1, n+1];
  for(int j = 1; j <= n; j++)
    T[1, j] = 1;
  for(int i = 2; i <= k; i++)
    T[i, 1] = 1;
  for(int i = 2; i <= k; i++)
    for(int j = 2; j <= n; j++)
      T[i, j] = T[i-1, j] + T[i, j-1];
  return T[k, n];
}
```

Példa 5×4-es táblára:

5	1	5	15	35
4	1	4	10	20
3	1	3	6	10
2	1	2	3	4
1	1	1	1	1
j / i	1	2	3	4

Lineáris táblázatkitöltéssel: elég mindig megtartanunk az aktuális sort, és abban updateelni az elemeket annyiszor, ahány sor van.

```
int R(int k , int n){
  int T[n+1];
  for(int j = 1; j <= n; j++)
    T[j] = 1;
  for(int i = 2; i <= k; i++)
    for(int j = 2; j <= n; j++)
      T[j] = T[j] + T[j-1];
  return T[n];
}
```

j	0	1	2	3	4
$T[j]$	–	1	1	1	1

Alapeset

Megjegyzés: mivel itt arról van szó, hogy egy $n+k$ lépésből álló sorozatban ki kell választani, hogy melyik n legyen a ,,felfele" lépés (a többi k lesz a ,,jobbra"), így $R(n,k)$ értéke éppen $\binom{(n+k)}{n}$ lesz, amit konstans tárban $O(n)$ idő alatt ki lehet számolni, nem kell ez a $n \cdot k$ időigény, a feladat célja most csak az általános módszer bemutatása, a következő feladatra ilyen ,,trükk" már nincs.

3. Feladat Adott egy $k \times n$-es tábla. Minden mezőre meg van adva egy c_{ij} pozitív szám, ami a mezőről begyűjthető érték. Egy játékos a bal alsó sarokból szeretne eljutni a jobb felső sarokba úgy, hogy csak jobbra és felfelé léphet a szomszédos mezőre. Az útja során összegyűjtheti a mezőkről az értékeket. Mennyi értéket tudunk összeszedni maximálisan? Hogyan határoznánk meg ezt az utat, amin a maximális értéket szedhetjük össze?

Megoldás

Az előző algoritmust annyiban kell módosítanunk, hogy nem a lépések számát, hanem az addig összeszedhető maximum értéket kell eltároljuk.

Algoritmus:

```
int R(int k, int n, int C[][]){
  int T[k+1, n+1];
  T[1,1] = C[1,1]
  for(int j = 2; j <= n; j++)
    T[1, j] = T[1,j-1] + C[1,j];
  for(int i = 2; i <= k; i++)
    T[i, 1] = T[i-1,1] + C[i,1];
  for(int i = 2; i <= k; i++)
    for(int j = 2; j <= n; j++)
      T[i, j] = max{T[i-1, j], T[i, j-1]}+C[i,j];
  return T[k, n];
}
```

4. Feladat Mindjárt eljön az idő, és megmutatod tökéletes tánctudásod! Holnap lesz az a táncverseny, amire egész életedben készültél - kivéve azt a nyarat, amikor a bácsikáddal Forksban vámpírra vadásztál. Szereztél egy előzetes példányt az n dal listájáról, amit a versenyen ebben a sorrendben fognak lejátszani. Nagyon jól ismered az összes dalt, az összes bírót és a saját képességeidet.

Minden k-ra tudod, hogy ha eltáncolod a k-adik dalt, pontosan $S[k]$ pontot fogsz kapni, de utána a tánctér szélén lihegve a kimerültségtől próbálsz életben maradni, így pihenned kell $W[k]$ számot (azaz nem táncolhatsz a $k+1$. számtól a $k+W[k]$. számig, azaz a $k+W[k]+1$. szám a következő, amin táncolhatsz). Az a táncos nyeri a versenyt, akinek az este végére a legtöbb pontja lesz, szóval el akarod érni a lehető legmagasabb pontot.

Adj meg egy hatékony algoritmust, amely megadja, mennyi a maximálisan elérhető pont. Az algoritmus inputja két tömb: $S[1 \ldots n]$ és $W[1 \ldots n]$ és azoknak mérete n.[1]

[1] A feladatot innen loptam.

Megoldás

Előlről hátrafelé most nem fogunk tudni számolni. Hiszen ahhoz, hogy megtudjuk, hogy ha az első számot eltáncoljuk, majd pihenünk, mennyi pontot szedhetünk még össze, ahhoz tudnunk kéne, hogy a maradék számot letáncolva (és pihenve) mennyi pontot tudunk összeszedni. Ezeknek az összege lesz a végeredmény, ha az első számot kiválasztjuk.

Az alapesetünk tehát az utolsó tánc. Arról tudjuk mennyi pontot ér és nem függ további táncoktól. A rekurzív eset pedig hátulról előre megy: az i. lépésben el kell döntenünk, hogy eltáncoljuk az i. számot vagy sem, azaz hogy megéri e az i. szám pontjait bezsebelni és várni, vagy várjunk most és táncoljunk a következőn. Megjegyzés: a ciklus lefelé megy, így a ,,következő" tánc pontjai már rendelkezésünkre állnak.

```
int Dance(int S[], int W[], int n){
  int D[n+1];
  D[n]=S[n];
  for(int i=n-1;i>0;i--){
    if(i +W[i]<n){
      D[i]=max{S[i]+D[i +W[i] + 1],D[i+1]};
    }
    else D[i]=max{S[i],D[i+1]};
  }
  return D[1];
}
```

S	–	3	1	5	4	3	6	5

W	–	1	2	2	4	1	3	0

i	0	1	2	3	4	5	6	7
$D[i]$	–	–	–	–	–	–	–	5

Alapeset

Érdekesség: ha meg akarod tudni, mi az a lineáris rekurzió, és hogyan számolhatóak ki a nagy Fibonacci számok gyorsan, kattints a nyuszira: