Alapvető algoritmusok

12. előadás

Dr. Pătcaș Csaba

BABEȘ-BOLYAI TUDOMÁNYEGYETEM
Matematika és Informatika Kar

Tartalom

Mester tétel

Mester tétel

Ha $a \geq 1, b > 1$ állandókra egy rekurzív algoritmus futási ideje kifejezhető $T(n) = a \cdot T(\frac{n}{b}) + f(n)$ formában, akkor

1. Ha $f(n) = O(n^{log_b a - \epsilon})$ valamely $\epsilon > 0$ állandóra, akkor $T(n) = \Theta(n^{\log_b a})$
2. Ha $f(n) = \Theta(n^{log_b a})$, akkor $T(n) = \Theta(n^{log_b a} \log n)$
3. Ha $f(n) = \Omega(n^{log_b a + \epsilon})$ valamely $\epsilon > 0$ állandóra és teljesül a *regularitási feltétel*, akkor $T(n) = \Theta(f(n))$

A regularitási feltétel akkor teljesül, ha $a \cdot f(\frac{n}{b}) \leq c \cdot f(n)$, valamely $c < 1$ állandóra és elég nagy n-re.

Mester tétel

Hogyan jegyezzük meg?

- Olyan *Divide et impera* típusú algoritmusok bonyolultságának a megállapítására alkalmazhatjuk, amelyek egy n méretű feladatot a darab, egyenként $\frac{n}{b}$ méretű részfeladatra bontanak fel és az összerakáshoz $f(n)$ időt használnak.
- Az eredményt mindig aszimptotikus éles korlát formájában adja meg, vagyis a Θ jelölést használva.
- Első lépésben érdemes kiszámolni $n^{\log_b a}$ értékét.
- Az így kapott kifejezést hasonlítsuk össze $f(n)$-el és ha valamelyik lényegesen (polinomiálisan) nagyobb, akkor az lesz az eredmény.
- Ha a két kifejezés azonos nagyságrendű, az eredmény $\Theta(n^{log_b a} \log n) = \Theta(f(n) \log n)$ lesz.

Mester tétel

Mikor nem alkalmazható?

1. Az 1. és 2. eset között van egy „lyuk”, amikor $f(n)$ ugyan kisebb, mint $n^{\log_b a}$, de nem polinomiálisan kisebb, vagyis nem létezik egy n^ϵ szorzótényező, amivel kisebb.
   Példa: $T(n) = 2T(\frac{n}{2}) + \frac{n}{\log n}$
2. A 2. és 3. eset között is van egy hasonló lyuk, amikor $f(n)$ ugyan nagyobb , mint $n^{\log_b a}$, de nem polinomiálisan nagyobb.
3. A tétel nem alkalmazható, ha a 3. esetben nem teljesül a regularitási feltétel (ez a helyzet relatív ritkán fordul elő).
   Példa: $T(n) = T(\frac{n}{2}) + n(2 - \cos n)$
4. Ha a részfeladatok mérete nem egyenlő (pl. `Gyorsrendezés`). Ilyenkor segíthet a mester tétel általánosított változata az *Akra–Bazzi-módszer*, amely matematikailag valamivel intenzívebb számításokat igényel.
5. Ha $b \leq 1$ (pl. `Hanoi tornyok` feladata).
6. Egyéb „furcsa” esetek (pl. $a < 1$, $f(n)$ negatív stb.)

Mester tétel

Példa: Szorzat

- A sorozatot két egyforma hosszúságú részsorozatra bontjuk, melyeknek hossza az eredeti sorozat fele, tehát $a = 2, b = 2$.
- Az összerakás egy egyszerű szorzás, vagyis $f(n) = \Theta(1)$.
- $n^{\log_b a} = n$, ami polinomiálisan nagyobb, mint $\Theta(1)$, tehát a végső bonyolultság $\Theta(n)$.

Mester tétel

Példa: Bináris keresés

- A sorozatnak csak a felére történik a rekurzív hívás, tehát $a = 1, b = 2$
- Összerakás nincs, ezért tekinthetjük úgy, hogy $f(n) = \Theta(1)$.
- $n^{\log_b a} = n^0 = \Theta(1)$, vagyis azonos nagyságrendű, mint $f(n)$, ezért az eredmény $\Theta(f(n) \log n) = \Theta(\log n)$.

Mester tétel

Példa: Összefésülő rendezés

- $a = 2$, $b = 2$, $n^{\log_b a} = n$
- Az összerakás ideje az `Összefésülés` programozási tétel futási idejével egyezik meg, ami $\Theta(n)$.
- A két függvény azonos nagyságrendű, tehát az eredmény $\Theta(n \log n)$.

Mester tétel

Példa: Úszómedence (kedvező eset)

- $a = 4$, $b = 2$, $n^{\log_b a} = n^2$
- $f(n) = \Theta(1)$.
- Az eredmény $\Theta(n^2)$.

Tartalom

Dinamikus programozás módszere

- A módszer nevében a „programozás” szó nem számítógép-programozásra utal, hanem táblázatos megoldási módszer használatára (mint a *lineáris programozás* esetében).
- Láttuk az *oszd meg és uralkodj* módszernél, hogy a részfeladatokat top-down sorrendben jártuk be.
- Ezzel ellentétben, a dinamikus programozás módszerénél bottom-up sorrendben fogjuk be is járni és meg is oldani a részfeladatokat.
- A módszer egy másik jellemzője, hogy akkor is hatékony, ha a részfeladatok nem függetlenek egymástól.
- Ahhoz, hogy ebben az esetben a futási idő ne váljon exponenciálissá, megőrizzük a részfeladatok megoldását egy „táblázatban”, így csak egyszer számoljuk ki ezeket.

Dinamikus programozás módszere

- Olyankor alkalmazható, amikor a táblázat mérete relatív kicsi (másképpen nem tudnánk tárolni a memóriában, vagy sok időbe telne a kitöltése).
- Látni fogjuk, hogy használhatjuk optimalizálási és számolási feladatok megoldására is.
- Az optimalizálási feladatok esetén a feladat optimális résszstruktúrájú kell legyen, ahhoz, hogy a módszert alkalmazhassuk. Ezt a későbbiekben definiáljuk majd.
- A *divide et impera*-val szembeni előnye akkor válik egyértelművé, amikor a részfeladatok nem függetlenek, vannak átfedések közöttük.

Feljegyzéses módszer (Memoizálás)

- A feljegyzéses módszer (memoizálás) a „határon” helyezkedik el az *oszd meg és uralkodj* és a *dinamikus programozás* módszerei között, ötvözi a kettőből ismert elveket.
- Rekurzívan implementáljuk, mint általában a *divide et impera* algoritmusokat, viszont egy táblázatban elmentjük a kiszámolt részfeladatok megoldásait, így a *dinamikus programozáshoz* hasonlóan csak egyszer oldunk meg egy részfeladatot, amelyek nem kell függetlenek legyenek egymástól.

Fibonacci
Divide et impera módszerrel

```
ALGORITMUS FiboDivImp(n)
  HA (n < 2) akkor
    VISSZATÉRÍT: n
  KÜLÖNBEN
    VISSZATÉRÍT: FiboDivImp(n - 1) + FiboDivImp(n - 2)
  VÉGE(Ha)
VÉGE(Algoritmus)
```

Fibonacci
Dinamikus programozás módszerével

```
ALGORITMUS FiboDP(n)
  fibo[0] = 0
  fibo[1] = 1
  MINDEN i = 2, n végezd el:
    fibo[i] = fibo[i - 1] + fibo[i - 2]
  VÉGE(Minden)
  VISSZATÉRÍT: fibo[n]
VÉGE(Algoritmus)
```

Fibonacci

Feljegyzéses módszerrel

```
ALGORITMUS InitMemo(n)
  MINDEN i = 0, n végezd el:
    fibo[i] = -1 \\még nem számoltuk ki, "ismeretlen"
  VÉGE(Minden)
  fibo[0] = 0
  fibo[1] = 1
VÉGE(Algoritmus)
```

Fibonacci

Feljegyzéses módszerrel

```
ALGORITMUS FiboMemo(n)
  HA (fibo[n] != -1) akkor \\már kiszámoltuk
    VISSZATÉRÍT: fibo[n]
  KÜLÖNBEN
    fibo[n] = FiboMemo(n - 1) + FiboMemo(n - 2)
    VISSZATÉRÍT: fibo[n]
  VÉGE(Ha)
VÉGE(Algoritmus)
```

Hívás: `FiboMemo(n)`

Összehasonlítás

1. Divide et impera
   Előnyök: rövid és egyszerű kód
   Hátrányok: exponenciálissá válhat a futási idő, terheli a vermet
2. Dinamikus programozás
   Előnyök: időhatékony amikor a teljes táblázatra szükség van, a vermet nem terheli
   Hátrányok: a táblázat memóriát foglal, nem mindig egyértelmű a kitöltési sorrend, a teljes táblázatot kitölti (néha csak részlegesen van rá szükség), gyakran a leghosszabb kódot eredményezi a három közül
3. Feljegyzéses módszer
   Előnyök: könnyen implementálható, a kitöltési sorrendet a rekurzív hívások által automatikusan kezeli, csak azt a részét tölti ki a táblázatnak amelyikre szükség van
   Hátrányok: a legtöbb memóriát igényli

Kombinációk

Feladat

Határozzuk meg C_n^k-t, vagyis n elem k-ad rendű kombinációinak számát.

Megoldás:

- Felhasználjuk a jól ismert rekurzív képletet: $C_n^k = C_{n-1}^k + C_{n-1}^{k-1}$
- A legegyszerűbb részfeladatok: $C_n^0 = 1$ és $C_n^n = 1$
- Tulajdonképpen a Pascal-háromszöget építjük fel a dinamikus programozást alkalmazva.

Kombinációk

```
ALGORITMUS KombinációkDP(n, k)
  c[1][0] = 1
  c[1][1] = 1
  MINDEN i = 2, n végezd el:
    c[i][0] = 1
    c[i][i] = 1
    MINDEN j = 1, i - 1 végezd el:
      c[i][j] = c[i - 1][j] + c[i - 1][j - 1]
    VÉGE(Minden)
  VÉGE(Minden)
  VISSZATÉRÍT: c[n][k]
VÉGE(Algoritmus)
```

Számháromszög

Feladat

Adott egy számháromszög, amely pozitív egész számokat tartalmaz. Az első sorában egy elem van, a másodikban kettő és így tovább. Határozzuk meg a legnagyobb összeget, amelyet úgy kaphatunk, hogy a háromszög tetejéről indulva egészen az utolsó sorig haladunk úgy, hogy egy mezőről csak a két alatta lévő mezőre léphetünk.

Példa:

```
     2
    9 4
   3 5 2
  1 2 3 4
 7 6 5 4 3
3 2 1 1 7 8
```

Optimális összeg: 30

Számháromszög

Megoldás

- Először vegyük észre, hogy a *mohó* megközelítés nem működik helyesen, a fenti példára csak 25-ös összeget ad.
- A *backtracking*-re alapuló megoldás minden sorban (az elsőt leszámítva) két lehetőség közül választhatna, így 2^{n-1} útvonalat kellene megvizsgáljon.
- Mivel a feladat csak a maximális összeg értékét kéri, a háromszögben lentről felfelé indulunk és a felső mezőben fogjuk megkapni a megoldást.
- Vegyük észre, hogy a háromszöget tárolhatjuk egy mátrix főátlóján és az az alá eső részében.
- Legyen `maxÖsszeg[i][j]` az a legnagyobb összeg, amit az (i, j) pozícióról lefelé indulva kaphatunk.
- A rekurzív összefüggést könnyedén levezethetjük: `maxÖsszeg[i][j]` $=$ max(`maxÖsszeg[i + 1][j]`, `maxÖsszeg[i + 1][j + 1]`) $+$ `a[i][j]`

```
ALGORITMUS Számháromszög(n, a)
  MINDEN j = 1, n végezd el:
    maxÖsszeg[n][j] = a[n][j]
  VÉGE(Minden)
  MINDEN i = n - 1, 1, -1 végezd el:
    MINDEN j = 1, i végezd el:
      maxÖsszeg[i][j] =
      max(maxÖsszeg[i + 1][j], maxÖsszeg[i + 1][j + 1]) + a[i][j]
    VÉGE(Minden)
  VÉGE(Minden)
  VISSZATÉRÍT: maxÖsszeg[1][1]
VÉGE(Algoritmus)
```

Leghosszabb növekvő részsorozat

Feladat

Adva van egy n elemű egész számokat tartalmazó a sorozat. Határozzuk meg a leghosszabb szigorúan növekvő részsorozatának a hosszát és egy ilyen részsorozatot!

Példa: $n = 8$, [0 8 4 12 2 10 6 14]

A leghosszabb növekvő részsorozat hossza: 4

Egy lehetséges megoldás: [0 8 12 14]

- A dinamikus programozás módszerének alkalmazását kezdhetjük hasonlóképpen, mint ahogy már láttuk a *backtracking* és *divide et impera* módszereknél: a paraméterek meghatározásával, amelyek egy részfeladatot jellemeznek.
- Miután megállapítottuk a paramétereket, bizonyítanunk kell, hogy a feladat optimális részstuktúrával rendelkezik.
- Ezt általában könnyedén megtehetjük a *reductio ad absurdum* módszert alkalmazva.
- Ezután meghatározzuk a legkisebb részfeladatok megoldását („megállási feltétel”), a rekurzív összefüggéseket és az eredeti feladat paramétereit („kezdeti hívás”).
- Megoldjuk lentről felfelé haladva a részfeladatokat és elmentjük egy táblázatba az így kapott eredményeket. Ha szükséges a megoldás visszakeresése, egyéb segédinformációkat is elmenthetünk, ezek segítségével fentről lefelé építhetjük majd vissza a megoldást.

Leghosszabb növekvő részsorozat

Elemzés

- Jelöljük `maxHossz[i]`-vel a leghosszabb növekvő részsorozat hosszát, amely az i. elemben végződik.
- Bizonyítanunk kell, hogy a részfeladatok ezen paraméterezése mellett, a feladat optimális részstruktúrával rendelkezik, de ehhez előbb definiálnunk kell, hogy ez mit jelent.

Definíció

Azt mondjuk, hogy egy feladat optimális résztruktúrájú, ha a probléma minden optimális megoldása önmagán belül a részfeladatok optimális megoldásait tartalmazza.

- Az optimális részstuktúra meglétét a „szétvágás és összeragasztás” elvét követve bizonyíthatjuk *reductio ad absurdum*mal.
- Jelen feladat esetén azt kell bizonyítani, hogy egy i. pozíción végződő S leghosszabb növekvő részsorozat bármely $j < i$ pozíción végződő S' darabja is optimális, vagyis leghosszabb azok közül a részsorozatok közül, melyek a j. pozíción végződnek.
- Feltételezzük, hogy ez nem igaz, tehát létezik egy S'' hosszabb növekvő részsorozat, amely a j. pozíción végződik.
- Ekkor ezt a hosszabb S'' részsorozatot behelyettesíthetnénk S' helyére, így S meghosszabbodna, ami ellentmond azzal, hogy a leghosszabb i. pozíción végződő részsorozat. Ezzel bizonyítottuk az optimális részstuktúra meglétét.
- Vegyük észre, hogy feltételeztük S'-ről, hogy nem optimális, ezt „kivágtuk” és a helyére „ragasztottuk” S''-et.
- Kis gyakorlattal ezeket a bizonyításokat fejben is elvégezhetjük, ezért a további feladatok esetén kihagyjuk őket.

Leghosszabb növekvő részsorozat

Megoldás

- A legkisebb részfeladatnak tekinthetjük `maxHossz[1] = 1`-et.
- Az eredmény a tömb legnagyobb eleme lesz, ott végződik a leghosszabb növekvő részsorozat.
- A rekurzív összefüggéshez arra gondolhatunk, hogy az i. elemet mely eddig felépített növekvő részsorozatok végére ragaszhatjuk és ezek közül melyik adja a legjobb megoldást:
  $\texttt{maxHossz[i]} = 1 + \max\limits_{j=\overline{1,i-1}} \{\texttt{maxHossz[j]}, \text{ ahol } a[j] < a[i]\}$
- Nyilván ha `a[i]` kisebb vagy egyenlő, mint az összes előtte lévő elem, akkor `maxHossz[i] = 1`. Ha nem akarjuk ezt az esetet külön kezelni, akkor tekinthetjük úgy, hogy `a[0]` = $-\infty$ és `maxHossz[0] = 0`, ekkor a fenti rekurzív összefüggésben j 0-tól kell induljon.
- Ahhoz, hogy egy lehetséges megoldást könnyedén visszakereshessünk, érdemes tárolni az `előző` tömbben minden i-re, azt a j-t, amelyre a maximumot kaptuk.

Leghosszabb növekvő részsorozat
Pszeudokód

```
ALGORITMUS LeghosszabbNövekvő(a, n)
  maxHossz[1] = 1
  m = 1 \\melyik pozíción végződik a megoldás
  MINDEN i = 2, n végezd el:
    maxHossz[i] = 0
    MINDEN j = 1, i - 1 végezd el:
      HA (maxHossz[j] > maxHossz[i] ÉS a[i] > a[j]) akkor
        maxHossz[i] = maxHossz[j]
        előző[i] = j
      VÉGE(Ha)
    VÉGE(Minden)
    maxHossz[i] = maxHossz[i] + 1
```

Leghosszabb növekvő részsorozat
Pszeudokód

```
    HA (maxHossz[i] > maxHossz[m]) akkor
      m = i
    VÉGE(Ha)
  VÉGE(Minden)
  KiírMegoldás(m)
VÉGE(Algoritmus)
```

Leghosszabb növekvő részsorozat
Pszeudokód

```
ALGORITMUS KiírMegoldás(i)
  HA (maxHossz[i] > 1) akkor
    KiírMegoldás(előző[i])
  VÉGE(Ha)
  KI: a[i]
VÉGE(Algoritmus)
```

Leghosszabb növekvő részsorozat

Bonyolultság

- Mennyi a bemutatott megoldás bonyolultsága a legrosszabb esetben?

Leghosszabb növekvő részsorozat

Bonyolultság

- Mennyi a bemutatott megoldás bonyolultsága a legrosszabb esetben? $\Theta(n^2 + n) = \Theta(n^2)$.
- A feladat megoldható $O(n \log n)$ időben is fejlett adatszerkezetekkel, amelyek lehetővé teszik a `maxHossz` tömb egy elemének kiszámolását $O(\log n)$ időben.
- Létezik a bináris keresésre alapuló megoldás is, melynek szintén $O(n \log n)$ a futási ideje.

Autó bérbeadás 2

Feladat

Egy szállítási vállalat autókat kölcsönöz. Egy bizonyos jármű iránt igen nagy az érdeklődés, ezért az igényeket egy évre előre jegyzik. Az igényt két számmal jelöljük, amelyek az év azon napjainak sorszámait jelölik, amellyel kezdődően, illetve végződően igénylik az illető autót. Állapítsuk meg a bérbeadást úgy, hogy a lehető legtöbb napra adjuk ki a járművet!

- Figyeljük meg a különbséget az első változathoz képest, amit a *greedy* módszernél vettünk!
- Ott a személyek számát kellett maximalizálni, itt a kiadott napok számát.
- Erre a változatra nyilvánvalóan nem működik a *mohó* módszer, pl. $n = 3$, (1, 2) (3, 4) (1, 10)

Autó bérbeadás 2

Megoldás

- Viszont az első változatnál megismert ötlet itt is segítségünkre lesz, tulajdonképpen kombinálni fogjuk a *greedy* és a *dinamikus programozás* módszereket!
- Rendezzük ismét az intervallumokat végpontjaik szerint növekvő sorrendbe.
- Ekkor ha a rendezett sorozatot tekintve `maxNap[i]`-vel jelöljük a napok maximális számát amelyre kiadható a jármű az i. kéréssel bezárólag, optimális részstruktúrát kapunk.
- Innen a feladat nagyon hasonlóvá vált a leghosszabb növekvő részsorozatéhoz.

Autó bérbeadás 2

Pszeudokód

```
ALGORITMUS Autó2(n, a)
  Rendez(n, a)
  eredmény = 0
```

Autó bérbeadás 2

Pszeudokód

```
  MINDEN i = 1, n végezd el:
    maxNap[i] = 0
    MINDEN j = 1, i - 1 végezd el:
      HA (a[i].kezd >= a[j].vég) akkor
        maxNap[i] = max(maxNap[i], maxNap[j])
      VÉGE(Ha)
    VÉGE(Minden)
    maxNap[i] = maxNap[i] + a[i].vég - a[i].kezd + 1
    eredmény = max(eredmény, maxNap[i])
  VÉGE(Minden)
  VISSZATÉRÍT: eredmény
VÉGE(Algoritmus)
```

Dominók

Feladat

Adott n darab dominó és a rajtuk szereplő értékek. Határozzuk meg a leghosszabb olyan sort, melyben a dominók betartják az eredeti sorrendjüket és az egymás melletti dominók megfelelő oldalain lévő számok egyenlőek. A dominókat el lehet forgatni $180°$-kal.

Példa: $n = 6$, (4 2) (2 3) (3 4) (3 5) (6 9) (5 7)
A leghosszabb dominósor hossza: 4
Egy lehetséges megoldás: (2 4) (4 3) (3 5) (5 7)

Dominók

Elemzés

- Mivel a feladat hasonlít a leghosszabb növekvő részsorozat feladatára, az első ötletünk az lehetne, hogy egy hasonló `maxHossz` tömbbe számoljuk ki a részmegoldásokat.
- Ezzel a megközelítéssel viszont több problémába is ütközünk.
- Az egyik, hogy nem tudjuk egyszerűen ellenőrizni, hogy az i. dominót hozzácsatolhatjuk-e a j. dominóhoz, mert nem tudjuk, hogy a j. dominó elvan-e forgatva vagy sem abban a megoldásban, amelyet bővíteni próbálunk.
- A másik gond, hogy ezzel a paraméterezéssel nincs optimális résztstuktúránk!
- Figyeljük meg a fenti példát: a megadott optimális megoldásban a (4 3)-as dominó egy 2 hosszúságú részsorozat végén van, míg ugyanaz a dominó optimálisan egy 3 hosszúságú részsorozatot zárna: (4 2) (2 3) (3 4)

- A fenti problémákat kiküszöbölhetjük, ha módosítjuk a paraméterezést.
- Legyen `leghosszabb[i][j]` a leghosszabb dominósor hossza, amely az i. dominóval végződik úgy, hogy ennek a dominónak az esetében a j-nek megfelelő döntést hoztuk.
- Ha $j = 0$ a dominót nem forgattuk meg, ha $j = 1$, akkor igen.
- Tehát a `leghosszabb` mátrixnak n sora és két oszlopa lesz.
- Kezdetben `leghosszabb[1][0] = leghosszabb[1][1] = 1`
- $\texttt{leghosszabb[i][j]} = 1 + \max\limits_{k=\overline{1,i-1}} \{\texttt{leghosszabb[k][p]}\}$, ahol a k. dominóhoz p szerint forgatva hozzáilletszthető az i. dominó j szerint forgatva

Leghosszabb közös részsorozat

Feladat

Adott egy n elemű a sorozat és egy m elemű b sorozat. Határozzuk meg a két sorozat leghosszabb közös részsorozatát!

Példa:

$n = 9, a = [1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9]$

$m = 9, b = [1\ 9\ 3\ 5\ 2\ 2\ 2\ 9\ 3]$

Leghosszabb közös részsorozat hossza: 4

Megoldás: 1 3 5 9

Leghosszabb közös részsorozat

Megoldás

- Legyen `hossz[i][j]` a leghosszabb közös részsorozat hossza, amelyet az *a* sorozat első *i* eleméből és a *b* sorozat első *j* eleméből kaphatunk.
- Ha $i = 0$ vagy $j = 0$, akkor értelemszerűen `hossz[i][j]` is egyenlő nullával.
- Ha `a[i] = b[j]`, akkor a közös elemet hozzávehetjük a nélküle talált optimális megoldáshoz, tehát ekkor `hossz[i][j] = hossz[i - 1][j - 1] + 1`
- Egyébként vagy `a[i]` vagy `b[j]` nem lesz a részfeladat optimális megoldásának része, vagyis ekkor
  `hossz[i][j] = max(hossz[i - 1][j], hossz[i][j - 1])`
- Az eredmény visszakereséséhez tárolhatjuk minden részfeladat esetén, hogy melyik döntést hoztuk meg a három lehetséges közül, de ennél a feladatnál ez nem szükséges, mert `hossz[n][m]`-től indulva minden lépésben konstans időben eldönthetjük a mátrixban található értékek alapján, hogy merre menjünk a három irányból.

Leghosszabb közös részsorozat

A felhasznált memória csökkentése

Észrevétel

Figyeljük meg, hogy a `hossz` mátrix i. sorának a felépítéséhez csak az $i-1$. sorára van szükségünk. Így, ha csak a leghosszabb közös részsorozat hossza érdekel minket, elég mindig csak a mátrix utolsó két sorát tárolni, így a memóriabonyolultságot $\Theta(n \cdot m)$-ről $\Theta(m)$-re csökkenthetjük.
Ugyanezt a technikát alkalmazhatjuk a `Fibonacci`, `Kombinációk`, `Számháromszög`, `Szerkesztési távolság`, `Palindrom` és `Hátizsák` feladatoknál is.