Alapvető algoritmusok

7. előadás

Dr. Pătcaș Csaba

BABEȘ-BOLYAI TUDOMÁNYEGYETEM
Matematika és Informatika Kar

Tartalom

Faktoriális

Feladat

Számítsuk ki rekurzívan a beolvasott n szám faktoriálisát!

```
ALGORITMUS Faktoriális(n)
  HA (n = 0) akkor
    VISSZATÉRÍT: 1
  KÜLÖNBEN
    VISSZATÉRÍT: Faktoriális(n - 1) * n
  VÉGE(Ha)
VÉGE(Algoritmus)
```

Gyakorlati tanács: Rajzoljuk le a rekurzív hívások fáját!

Faktoriális

Megjegyzések

- Mivel minden egyes alkalommal más n paraméterre van szükség, fontos, hogy ezt érték szerint adjuk át. Így kifejezéseket is írhatunk az aktuális paraméter helyére, mint fentebb az $n-1$-et.
- A faktoriális rekurzív kiszámítása előnytelen, mivel $n+1$-szer kerül meghívásra a `Faktoriális` alprogram.

Faktoriális

Végződés szerinti rekurzió

Definíció

Amikor egy alprogramban az utolsó végrehajtott utasítás a rekurzív hívás, akkor végződés szerinti rekurzióról (tail recursion) beszélünk.

Végződés szerinti rekurzív-e a bemutatott faktoriálisszámoló függvény? Miért?

Faktoriális
Végződés szerinti rekurzió

A bemutatott algoritmus nem végződés szerinti rekurzív, mivel az utolsó végrehajtott utasítás nem a rekurzív hívás, hanem a szorzás.
Egy végződés szerinti rekurzív változat:

```
ALGORITMUS FaktoriálisTailRec(n, x)
  HA (n = 0) akkor
    VISSZATÉRÍT: x
  KÜLÖNBEN
    VISSZATÉRÍT: FaktoriálisTailRec(n - 1, x * n)
  VÉGE(Ha)
VÉGE(Algoritmus)
```

Hívás: `FaktoriálisTailRec(n, 1)`

Fibonacci

Feladat

Generáljuk a Fibonacci-sorozat *n*. elemét!

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

Fibonacci

Avagy így ne!

- Ha lerajzoljuk a rekurzív hívások fáját, vagy implementáljuk és futtatjuk az algoritmust, megfigyelhetjük, hogy ugyanarra az n-re többször is meghívódik a `Fibo` alprogram, ugyanazt a részfeladatot többször is megoldja az algoritmus.
- Ez exponenciális futási időhöz vezet, vagyis már kis értékekre is elfogadhatatlan lesz a hatékonyság.
- Ennek egy lehetséges kiküszöbölése a rekurzív szerkezet megtartásával a memoizáció módszere, amiről a dinamikus programozás módszerénél lesz majd szó.

Legnagyobb közös osztó

Feladat

Számítsuk ki rekurzívan két természetes szám legnagyobb közös osztóját.

```
ALGORITMUS LnkoIteratív(x, y)
  AMÍG (y != 0) véged el:
    r = x % y
    x = y
    y = r
  VÉGE(Amíg)
  VISSZATÉRÍT: x
VÉGE(Algoritmus)
```

```
ALGORITMUS Lnko(x, y)
  HA (y = 0) akkor
    VISSZATÉRÍT: x
  KÜLÖNBEN
    VISSZATÉRÍT: Lnko(y, x % y)
  VÉGE(Ha)
VÉGE(Algoritmus)
```

Gyors hatványozás

```
ALGORITMUS Gyorshatvány(alap, kitevő)
  HA (kitevő = 0) akkor
    VISSZATÉRÍT: 1
  KÜLÖNBEN
    HA ((kitevő % 2) = 1) akkor
      VISSZATÉRÍT: alap * Gyorshatvány(alap * alap, kitevő / 2)
    KÜLÖNBEN
      VISSZATÉRÍT: Gyorshatvány(alap * alap, kitevő / 2)
    VÉGE(Ha)
  VÉGE(Ha)
VÉGE(Algoritmus)
```

Konverzió

Feladat

Alakítsuk át az n számot a p ($p \leq 10$) számrendszerből a 10-esbe!

```
ALGORITMUS Konverzió(szám, újszám, hatvány, p)
  HA (szám > 0)
    szj = szám % 10
    VISSZATÉRÍT: Konverzió(szám / 10, újszám + szj * hatvány,
                                hatvány * p, p)
  KÜLÖNBEN
    VISSZATÉRÍT: újszám
  VÉGE(Ha)
VÉGE(Algoritmus)
```

Hívás: `Konverzió(n, 0, 1, p)`

Palindrom

Feladat

Írjunk rekurzív algoritmust, amely eldönti egy karakterláncról, hogy palindrom-e!

```
ALGORITMUS Palindrom(s)
  HA (hossz < 2) akkor
    VISSZATÉRÍT: IGAZ
  KÜLÖNBEN
    HA (s[1] != s[hossz]) akkor
      VISSZATÉRÍT: HAMIS
    KÜLÖNBEN
      VISSZATÉRÍT: Palindrom(s[2..hossz - 1])
    VÉGE(Ha)
  VÉGE(Ha)
VÉGE(Algoritmus)
```

Számjegyek

Feladat

Számoljuk meg, hogy egy adott x számnak hány y számjegye van!

```
ALGORITMUS Számjegyek(x, y)
  HA (x < 10) akkor
    HA (x = y) akkor
      VISSZATÉRÍT: 1
    KÜLÖNBEN
      VISSZATÉRÍT: 0
    VÉGE(Ha)
```

Számjegyek

Feladat

Számoljuk meg, hogy egy adott x számnak hány y számjegye van!

```
  KÜLÖNBEN
    HA ((x % 10) = y) akkor
      VISSZATÉRÍT: 1 + Számjegyek(x / 10, y)
    KÜLÖNBEN
      VISSZATÉRÍT: Számjegyek(x / 10, y)
    VÉGE(Ha)
  VÉGE(Ha)
VÉGE(Algoritmus)
```

Orosz szorzás

- A módszer lényege, hogy ismételt kettővel való osztások és szorzások felhasználásával egy összegből alakítjuk ki a szorzatot.
- Ha x és y a két szorzandó szám, x-et mindig osztjuk és y-t mindig szorozzuk kettővel.
- Ha x páratlan, akkor a megoldáshoz hozzáadjuk y-t.

Orosz szorzás

Példa

x	y	p
85	18	0
42	36	18
21	72	18
10	144	90
5	288	90
2	576	378
1	1152	378
0	2304	1530

Tartalom

1. Rekurzió
   - Feladatok

2. Visszalépéses keresés (Backtracking)
   - Descartes-szorzat
   - Részhalmazok
   - Permutációk
   - Királynők

Feladat

Generáljuk az $M_1 \times M_2 \times \ldots \times M_n$ Descartes-szorzatot!

Példa:

$n = 3, M_1 = \{1, 2\}, M_2 = \{1\}, M_3 = \{1, 2, 3\}$

1 1 1
1 1 2
1 1 3
2 1 1
2 1 2
2 1 3

Descartes-szorzat

- Nézzük először a feladat egyszerűsített változatát, melyben $|M_i| = m$.
- Ekkor könnyen belátható, hogy m^n darab megoldásunk lesz.

Példa $n = 2, m = 3$

1 1

1 2

1 3

2 1

2 2

2 3

3 1

3 2

3 3

Descartes-szorzat

Ha más eszköz nem áll rendelkezésünkre, egymásba ágyazott `MINDEN` ciklusokkal próbálhatjuk először megoldani a feladatot:

```
HA (n = 1) akkor
  MINDEN i1 = 1, m végezd el:
    KI: i1
  VÉGE(Minden)
KÜLÖNBEN
  HA (n = 2) akkor
    MINDEN i1 = 1, m végezd el:
      MINDEN i2 = 1, m végezd el:
        KI: i1, i2
      VÉGE(Minden)
    VÉGE(Minden)
  KÜLÖNBEN ...
```

Descartes-szorzat

- Látjuk, hogy ez a megoldás nem elég általános, gyakorlatilag fel kellene sorolni a kódban minden lehetséges n értékre n darab egymásba ágyazott `MINDEN` ciklust.
- Legyen egy `i` tömb, amiben a ciklusváltozókat tároljuk és a `k` változó, amely azt mutatja, hogy mennyi az éppen aktuális ciklus sorszáma.

```
ALGORITMUS DescartesIteratív(n, m)
  MINDEN k = 1, n végezd el:
    i[k] = 0
  VÉGE(Minden)
  k = 1
  AMÍG (k > 0) végezd el:
```

Descartes-szorzat

```
      HA (k = n + 1) akkor
        KI: i[1..n]
        k = k - 1
      KÜLÖNBEN
        HA (i[k] < m) akkor
          i[k] = i[k] + 1
          k = k + 1
        KÜLÖNBEN
          i[k] = 0
          k = k - 1
        VÉGE(Ha)
      VÉGE(Ha)
    VÉGE(Amíg)
  VÉGE(Algoritmus)
```

Descartes-szorzat

Könnyen belátható, hogy egy rekurzív implementáció sokkal rövidebb, átláthatóbb és érthetőbb:

```
ALGORITMUS DescartesRekurzív(i, k)
  HA (k = n + 1) akkor
    KI: i[1..n]
  KÜLÖNBEN
    MINDEN i[k] = 1, m végezd el:
      DescartesRekurzív(i, k + 1)
    VÉGE(Minden)
  VÉGE(Ha)
VÉGE(Algoritmus)
```

Részhalmazok

Feladat

Határozzuk meg az $\{1, 2, \dots, n\}$ halmaz összes részhalmazát!

Példa: $n = 3$

$\emptyset$

$\{1\}$

$\{2\}$

$\{3\}$

$\{1, 2\}$

$\{1, 3\}$

$\{2, 3\}$

$\{1, 2, 3\}$

Részhalmazok

Megoldás

- Alkalmazzuk az előbb látott Descartes-szorzat generáló algoritmust, úgy, hogy m értékének 2-t választunk!
- Így n tagú sorozatokat fogunk kapni, melyeknek minden eleme 1 vagy 2.
- Ha egy megoldásban a k. elem 1, akkor úgy tekintjük, hogy k benne van az aktuális részhalmazban, ha meg 2, akkor úgy tekintjük, hogy nincs benne.

Permutációk

Feladat

Generáljuk az összes n elemű permutációt!

Példa: $n = 3$

1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1

Permutációk

Első ötlet

- Egy első ötlet lehetne, hogy $m = n$-t választunk és az így generált Descartes-szorzatokra ellenőrizzük, hogy halmazok-e. Ha igen, akkor van egy jó megoldásunk.
- Ennek a megoldásnak a bonyolultsága $\Theta(n^n \cdot n^2)$ lenne, ami már kis értékekre is elfogadhatatlan futási időt eredményez.
- Ábrázoljuk gyökeres faként az állapotokat, amiket az algoritmus bejár.
- A gyökér a kezdeti állapotot jelképezi (ahol $k = 0$) és a levelek egy-egy lehetséges megoldást. Ebben a feladatban a fának pontosan $n + 1$ szintje lesz és minden levél az utolsó szinten található majd meg.
- Egy k. szinten lévő csúcs leszármazottai azok a csúcsok, amelyekbe úgy juthatunk az eredeti csúcs által jelképezett állapotból, hogy ahhoz hozzáfűzzük `i[k]` valamilyen értékét.

- Észrevehetjük, hogy az első ötlet sok felesleges konfigurációt generál, ezt tekinthetjük nyers erő (brute force) algoritmusnak is, mert tulajdonképpen a teljes megoldásteret bejárja.
- Ezzel szemben a visszalépéses keresés (backtracking) megpróbálja csökkenteni a generálandó próbálkozások számát (néha sikertelenül).
- Amikor lehetséges, csak azokat a konfigurációkat generálja, amelyek eleget tesznek bizonyos belső feltételeknek.
- Jelen esetben a belső feltételek azt kell kikössék, hogy a generált megoldásban ne legyenek egyforma elemek.
- Ezek alapján fogalmazzuk meg a folytatási feltételeket, amelyek azt írják le, hogy a k. szintről milyen esetben léphetünk (vagy *kell* lépnünk) a $k+1$.-re.

Permutációk

Második ötlet

```
ALGORITMUS Permutáció(i, k)
  HA (k = n + 1)
    KI: i[1..n]
  KÜLÖNBEN
    MINDEN i[k] = 1, n végezd el:
      HA (FolytatásiFeltétel(i, k)) akkor
        Permutáció(i, k + 1)
      VÉGE(Ha)
    VÉGE(Minden)
  VÉGE(Ha)
VÉGE(Algoritmus)
```

```
ALGORITMUS FolytatásiFeltétel(i, k)
  MINDEN j = 1, k - 1 végezd el:
    HA (i[k] = i[j])
      VISSZATÉRÍT: HAMIS
    VÉGE(Ha)
  VÉGE(Minden)
  VISSZATÉRÍT: IGAZ
VÉGE(Algoritmus)
```

Permutációk

Harmadik ötlet

- A módszer eredményessége nagyban függ a folytatási feltételek szerencsés kiválasztásától.
- Minél hamarabb állítjuk le az eredmény generálását, annál kisebb a rekurzió mélysége.
- Ellenben a feltételek nem lehetnek túl bonyolultak, mivel ezeket minden lépésben végrehajtja az algoritmus.
- A második változatban a folytatási feltételeket lineáris időben valósítottuk meg.
- Egy n elemű logikai tömbbel ezt $\Theta(1)$-re javíthatjuk.

```
ALGORITMUS Permutáció(i, k, volt)
  HA (k = n + 1)
    KI: i[1..n]
  KÜLÖNBEN
    MINDEN i[k] = 1, n végezd el:
      HA (NEM volt[i[k]]) akkor
        volt[i[k]] = IGAZ
        Permutáció(i, k + 1, volt)
        volt[i[k]] = HAMIS
      VÉGE(Ha)
    VÉGE(Minden)
  VÉGE(Ha)
VÉGE(Algoritmus)
```

- A módszer használatához meg kell állapítanunk az eredmény kódolását, vagyis az M_k halmazokat.
- Ezután meghatározzuk a belső feltételeket és a folytatási feltételeket.
- A megoldástömb elemei (eddig `i`, a továbbiakban `v`) egyenként kapnak értéket.
- `v[k]` számára csak akkor javaslunk értéket, ha `v[1]`, `v[2]`, ..., `v[k - 1]` már kaptak értéket az aktuálisan generált eredményben.
- A `v[k]`-ra vonatkozó javaslatot akkor fogadjuk el, amikor `v[1]`, `v[2]`, ..., `v[k - 1]` értékei a `v[k]` értékével együtt teljesítik a továbblépési feltételeket.
- Ha a k. lépésben ezek nem teljesülnek, akkor `v[k]` számára új értéket választunk az M_k halmazból.
- Ha az M_k halmaz valamennyi elemét kipróbáltuk, visszalépünk a $k-1$. elemhez, amely számára új értéket javasolunk az M_{k-1} halmazból.

- Végeredménynek nevezzük azokat az eredményeket, amelyek teljesítik a belső feltételeket és amelyek esetén a folytatási feltételek nem kérnek további elemeket.
- Az algoritmus fontos jellemzője a visszalépés:
  back = vissza
  to track = keresni, követni
  backtracking = visszalépéses keresés
- Előfordulhat, hogy a folytatási feltételek által nem ellenőríztünk minden belső feltételt, ekkor ezt meg kell tenni a kiíratás előtt.
- Ha egyetlen eredményt kell megkeressünk, a kiírás után leállítjuk az algoritmust.

Királynők

Feladat

Helyezzünk el minden lehetséges módon n királynőt egy $n \times n$ méretű sakktáblán, úgy, hogy ne támadják egymást! Két királynő támadja egymást, ha ugyanazon a soron, oszlopon, vagy átlón helyezkednek el.

Ha észrevesszük, hogy minden sorba csak egy királynő kerülhet, a feladat elkezd hasonlítani a permutációk feladatára, hiszen választhatunk olyan kódolást, amelyben a k. királynőt kötelező módon a k. sorba tesszük és azt jegyezzük meg rá, hogy melyik oszlopba kerül.

Királynők

- A királynőket egymás után helyezzük fel a sakktáblára, sorról-sorra.
- Az adott sorba az első oszloppal kezdődően próbáljuk elhelyezni az aktuális királynőt, addig, amíg meg nem találjuk azt az oszlopot, amelyben nem támad más, eddig elhelyezett királynőt.
- Ha egy királynőt nem lehet elhelyezni, visszatérünk az előzőhöz és számára tovább keresünk megfelelő, nagyobb sorszámú oszlopot.

A megoldás tánccal bemutatva: `https://youtu.be/R8bM6pxlrLY`

Királynők

A feladat megoldásának rekurzív megfogalmazása

- A k. királynő esetében meghatározunk minden helyet ahová azt el lehet helyezni a k. sorban úgy, hogy ne támadjon egyet sem azok közül, amelyeket az $1, 2, \ldots, k-1$ sorokba helyeztünk.
- Elhelyezzük a k. királynőt, majd megoldjuk ugyanezt a feladatot a $k+1$. királynő esetében.
- Ha valamennyi királynőt elhelyeztünk, van egy végeredményünk, amelyet kiíratunk.
- Ahhoz, hogy két királynő ne támadja egymást, a következő relációk kell igazak legyenek: $\texttt{v[i]} \neq \texttt{v[j]}, \texttt{i} - \texttt{j} \neq |\texttt{v[i]} - \texttt{v[j]}|, \forall j = \overline{1, i-1}$

Királynők

```
ALGORITMUS Királynő(v, k, n)
  MINDEN i = 1, n végezd el:
    v[k] = i
    HA (NemTámad(v, k)) akkor
    //a k. nem támadja egyik előzőt sem
      Ha (k < n)
        Királynő(v, k + 1, n)
      KÜLÖNBEN
        Kiír(v, n)
      VÉGE(Ha)
    VÉGE(Ha)
  VÉGE(Minden)
VÉGE(Algoritmus)
```

Királynők

```
ALGORITMUS NemTámad(v, k)
  nem = IGAZ
  j = 1
  AMÍG ((j <= (k - 1)) ÉS nem) végezd el:
    HA ((v[k] = v[j]) VAGY (k - j = |v[k] - v[j]|))
      nem = HAMIS
    KÜLÖNBEN
      j = j + 1
    VÉGE(Ha)
  VÉGE(Amíg)
  VISSZATÉRÍT: nem
VÉGE(Algoritmus)
```