

# Rekurzió

Ionescu Klára

[clara@cs.ubbcluj.ro](mailto:clara@cs.ubbcluj.ro)

# Iteratív és rekurzív algoritmusok

- Bármely algoritmus megvalósítható *iteratívan és/vagy rekurzívan*.
- Mindkét technikának a lényege: *bizonyos utasítások ismételt végrehajtása*
- Az iteratív algoritmusokban az ismétlést *ciklusokkal* valósítjuk meg
- A rekurzív algoritmusokban az ismétlés azáltal valósul meg, hogy az illető alprogram *meghívja önmagát*, amikor még aktív.

# Mikor érdemes rekurzív algoritmust tervezni?

1. ha a feladat *eredménye rekurzív szerkezetű*;
2. ha a megoldás legjobb módszere a visszalépéses keresés módszer (*backtracking*);
3. ha a megoldás legjobb módszere az *oszd meg és uralkodj* módszer (*divide et impera*);
4. ha *a feldolgozandó adatok rekurzívan definiáltak* (pl. bináris fák).

**Pro:** egy rekurzív algoritmus *tömörebb, olvashatóbb* az iteratívnál

**Kontra:** néha túlságosan *igénybe veszi a végrehajtási vermet*, és előfordulhat, hogy a *futási ideje is nagyobb*, mint az iteratív változatnak

# Bevezetés

- A rekurzió egy különleges *programozási stílus*, inkább „technika” mint módszer.
- A rekurzív programozás, mint fogalom, a matematikai értelmezéshez közelálló módon került közhasználatba.

## Példák

1. Az aranyhal és a három kívánság: az első (valami...), a második (valami...), a harmadik: *„teljesítsd még három kívánságomat”*.
2. A matematikában, egy fogalmat rekurzív módon definiálunk, *ha a definíción belül felhasználjuk magát a definiálandó fogalmat*. Például, a faktoriális rekurzív definíciója adott *n* szám esetében:

$$n! = \begin{cases} 1, & \text{ha } n = 0 \\ n \cdot (n-1)!, & \text{ha } n \in \mathbb{N}^* \end{cases}$$

## Példák

3. **Knuth**: Egy **bináris fa** vagy üres, vagy tartalmaz egy csomópontot, amelynek van egy bal meg egy jobb utóda, amelyek szintén **bináris fák**.
- A programozásban: azokat az algoritmusokat, amelyeknek **szerkezete rekurzív**, vagy amelyeket **rekurzívan definiált adatszerkezetek**re alkalmazunk, általában rekurzívan kódolunk.
  - **Olyan alprogramokat nevezünk rekurzívoknak, amelyek meghívják önmagukat.**

# Közvetlen és közvetett rekurzió

- Ha az önmeghívás az illető alprogram összetett utasításában található, **közvetlen (direkt) rekurzióról** beszélünk.
- Ha egy rekurzív alprogramot egy másik alprogram hív meg, amelyet ugyanakkor az illető alprogram hív (közvetve, vagy közvetlenül) akkor **közvetett (indirekt) rekurzióról** beszélünk.
- **Közvetlen rekurzió:** *a rekurzív hívás aközben történik, miközben a számítógép az illető alprogram összetett utasítását hajtja végre.*

# Megjegyzések

1. A rekurzív eljárások esetében is, hasonlóan a nem rekurzívakhoz, az aktiválás **feltételezi a veremhasználatot**, ahol a paramétereket, a visszatérés helyének címét, valamint a lokális változókat tárolja (minden aktuális aktiválás idejére) a programozási környezet.
2. Új aktiválás csak az újrahívási feltétel teljesülésekor történik;
3. Az **újrahívások száma** meghatározza a **rekurzió mélységét**;  
Az **1.** megjegyzést figyelembe véve, egy rekurzív megoldás csak akkor hatékony, ha **ez a mélység nem túl nagy**.

# Megjegyzések

3. Amikor az újrahívási feltétel nem teljesül, az újraaktiválások sora leáll;
- ⇒ Az **F** feltétel tagadása *a rekurzióból való kilépés feltétele*;
  - ⇒ Az **F** feltétel a rekurzív eljárás *paramétereitől* függ és/vagy a helyi változóktól;
  - ⇒ A paraméterek olyan változók lesznek, amelyek *változnak egyik hívástól a másikig*
- A kilépést a paraméterek és a lokális változók módosulása (egyik hívástól a másikig) biztosítja.*
- (Ha ezeket a követelményeket nem tartjuk be, a program hibaüzenettel (**Stack overflow**) kilép.)



# Megjegyzések

4. **Újrahívás** (közvetlen rekurzió esetén), **többször** is előfordulhat egy rekurzív eljárásban; ebben az esetben, különbözni fognak a visszatérési címek.
5. A rekurzió **késlelteti** az eljárás azon utasításainak végrehajtását, amelyek a rekurzív hívás **utáni** részhez tartoznak.
6. Minden eddigi állítás igaz a rekurzív **függvények** esetében is, csak a hívás módja más.
7. *Egy rekurzív függvényt egy kifejezésből hívunk meg.*
8. Egy rekurzív függvény összetett utasítása, hasonlóan a nem rekurzív függvényekhez, **tartalmazni fog egy értékadó utasítást**, amely a függvény azonosítójának ad át értéket. Ebbe az utasításba kerül, többnyire, az újrahívás.

# Megoldott feladatok

## Egy szó betűinek megfordítása

*Írjuk ki egy szó betűit, fordított sorrendben, tömbhasználat nélkül. A szó betűi után '\*' karakter következik.*

### Elemzés

- A feladat követelményének megfelelően betűk szintjén dolgozunk.
- A megfordított kiírás azt jelenti, hogy *miután beolvastunk egy betűt, nem írjuk ki, csak azután, hogy megfordítottuk a szó fennmaradt részét.*
- A fennmaradt rész esetében ugyanígy járunk el; a módszer addig folytatódik, amíg eljutunk az utolsó betűhöz, amikor nincs mit megfordítani.

# Egy szó betűinek megfordítása

Rekurzív módon ezt a következőképpen lehet leírni:

*szó megfordítás =*

- az első (épp soron levő első) betű beolvasása, a fennmaradt rész megfordítása, majd az első betű kiírása, amennyiben ez egyben nem az utolsó;
- betű beolvasása és kiírása, amennyiben ez az utolsó betű.

**Algoritmus Fordít:** { *nincs paraméter, mivel* }  
{ *az alprogramban olvasunk be és írunk ki* }

**Be:** betű

**Ha** betű  $\neq$  ' \* ' **akkor**

Fordít { *meghívja önmagát, hogy* }  
{ *megfordíthassa a szó fennmaradt részét* }

**különb**

**Ki:** 'Fordított szó: '

**vége(ha)**

**Ki:** betű

**Vége(algoritmus)**

Az alprogram hívása: **Fordít**

Meg lehet valósítani a következőképpen is:

**Algoritmus** Fordít: { *nincs paraméter, mivel* }  
{ *az alprogramban olvasunk be és írunk ki* }

**Be:** betű

**Ha** betű  $\neq$  '\*' **akkor**

Fordít { *meghívja önmagát, hogy* }  
{ *megfordíthassa a szó fennmaradt részét* }

**Ki:** betű

**különben**

**Ki:** 'Fordított szó: '

**vége(ha)**

**Vége(algoritmus)**

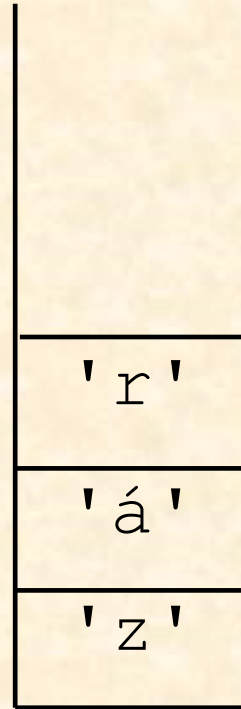
Példa: szó = zár. Megfordítva = ráz



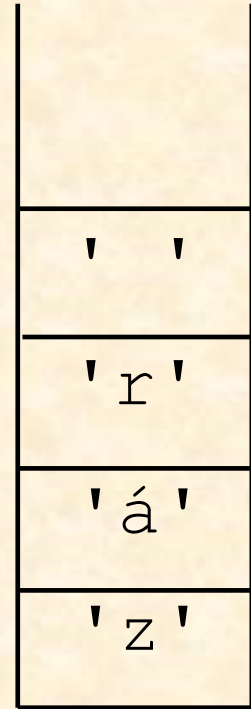
Első hívás.  
Beolvastuk a  
' z ' betűt.



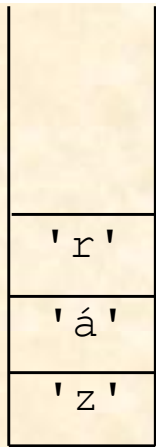
Második  
hívás.  
betű=' á '



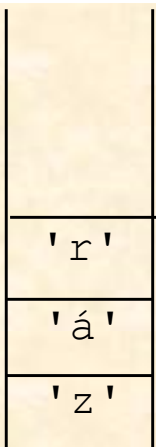
Harmadik  
hívás.  
betű=' r '



Negyedik  
hívás.  
betű=' '



Kilépünk az aktuális hívásból.



Kiírjuk a betűt, amely 'r'



Kilépünk az aktuális hívásból.



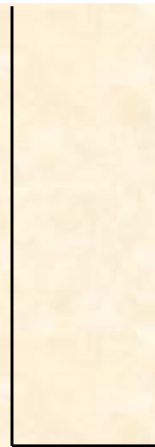
Kiírjuk a betűt, amely 'á'



Kilépünk az aktuális hívásból.



Kiírjuk a betűt, amely 'z'



Kilépünk az aktuális hívásból.

# Szavak sorrendjének megfordítása

*Beolvassunk egy  $n$  természetes számot és  $n$  szót a billentyűzetről, különböző sorokban. Írjuk ki a szavakat a beolvasás fordított sorrendjében! Ne használjunk tömböket!*

**Algoritmus Szavakat\_fordít\_1( $n$ ):**

**Be:** szó

**Ha  $n > 1$  akkor**

    Szavakat\_fordít\_1( $n-1$ )

**különben**

**Ki:** 'Fordított sorrendben:'

**vége(ha)**

**Ki:** szó

**Vége(algoritmus)**

Az első hívás aktuális paramétere a beolvasott  $n$  szám. Az eredeti feladat tehát  $n$  szó megfordítását valósítja meg, a részfeladatok pedig egyre kevesebb szó megfordítását.



Ha fordítva indulunk, vagyis „megfordítjuk” egy szónak a sorrendjét, majd a többiét, akkor az algoritmus a következő:

**Algoritmus** Szavakat\_fordít\_2(i):

*{ az első hívás aktuális paramétere 1 }*

**Be:** szó

**Ha**  $i < n$  **akkor**

Szavakat\_fordít\_2( $i + 1$ )

**különben**

**Ki:** 'Fordított sorrendben:'

**vége(ha)**

**Ki:** szó

**Vége(algoritmus)**

# Faktoriális

Írjuk ki az  $n$  adott szám faktoriálisát.

## Elemzés

Felhasználjuk a faktoriális matematikai definícióját és ez a  $\text{Fakt}(n)$  alprogramban lesz felismerhető.

**Algoritmus Fakt( $n$ ):**

**Ha  $n = 0$  akkor**

**térítsd 1**

**különben**

**térítsd Fakt( $n - 1$ ) \*  $n$**

**vége(ha)**

**Vége(algoritmus)**

Az első hívás  $\text{Fakt}(n)$ -nel történik, ahol  $n$  a beolvasott szám.

# Megjegyzések

1. A faktoriális tulajdonképpen kiszámolása akkor történik, amikor kilépünk egy-egy hívásból. Mivel minden egyes alkalommal más-más  $n$  paraméterre van szükség, fontos, hogy ezt *értékként* adjuk át; így kifejezéseket is írhatunk az aktuális paraméter helyére.
2.  $\Rightarrow$  A faktoriális rekurzív kiszámítása előnytelen; sokkal időigényesebb mint az iteratív megoldás, hiszen szükséges volt  $n + 1$ -szer aktiválni a *Fakt* függvényt.

# Legnagyobb közös osztó

*Számítsuk ki rekurzívan két természetes szám legnagyobb közös osztóját!*

## *Elemzés*

Ha figyelmesen elemezzük Eukleidész algoritmusát, észrevesszük, hogy a legnagyobb közös osztó ( $\text{Lnko}(m, n)$ ) egyenlő  $n$ -nel (ha  $n$  osztója  $m$ -nek), különben egyenlő  $\text{Lnko}(n, m \bmod n)$ -nel.

**Algoritmus** Lnko(m, n):

mar  $\leftarrow$  m **mod** n { mar = *lokális változó*, }  
{ *így nem fogjuk kétszer számítani a maradékot* }

**Ha** mar = 0 **akkor**

**térítsd** n

**különb**en

**térítsd** Lnko(n, mar)

**vége**(ha)

**Vége**(algoritmus)

Az első hívás történhet például egy kiíró utasításból  
(**Ki**: Lnko(m, n)).

# Descartes szorzat

Egy rajzon  $n$  virágot fogunk kiszínezni. A festékeket az  $1, 2, \dots, m$  számokkal kódoljuk. Bármely virág, bármilyen színű lehet, de szeretnénk tudni, hogyan lehetne ezeket különböző módon kiszínezni.

- 1) Oldjuk meg a feladatot, ha  $m = 2$ ;
- 2) Oldjuk meg a feladatot, ha  $m \geq 2$ .

## Elemzés

A következő Descartes szorzatot generáljuk:  $x = (x_1, x_2, \dots, x_n)$ ,  
 $x_i \in M, i = 1, \dots, n$ . Ha  $m = 2$  és  $n = 3$ ,  $M_3 = M \times M \times M$

	$x_1$	$x_2$	$x_3$		$x_1$	$x_2$	$x_3$
1)	1	1	1	5)	2	1	1
2)	1	1	2	6)	2	1	2
3)	1	2	1	7)	2	2	1
4)	1	2	2	8)	2	2	2

# Descartes szorzat

- Ha  $x_1 = 1$  és eltekintünk tőle, az  $(x_2, x_3)$  elemek értékei az  $M \times M$  Descartes szorzatot jelentik.
  - Ugyanígy, ha  $x_1 = 2$  és eltekintünk tőle.
- $\Rightarrow$  az  $M^n$  Descartes szorzat elemeinek generálása feltételezi, hogy az  $1$  és  $2$  értékeket egymás után válasszuk ki és adjuk  $x_1$ -nek, majd generáljuk az  $M^{n-1}$  Descartes szorzatot.
- Ennek elemeit megőrizzük a sorozatunkban, a második elemtől kezdődően.
  - Ezen elemek generálása újból azt jelenti, hogy az  $1$  és  $2$  értékeket egymás után kiválasztjuk és átadjuk  $x_2$ -nek, majd generáljuk az  $M^{n-2}$  Descartes szorzatot.
  - Ennek elemeit megőrizzük a sorozatunkban a harmadik elemtől kezdődően és így tovább, amíg kiválasztottuk az  $1$  és  $2$  értékeket az  $x_n$  elem számára is.

**Algoritmus** Descartes\_szorzat\_1(i):  $\{ M = (1, 2) \}$

$x_i \leftarrow 1$

**Ha**  $i < n$  **akkor**

Descartes\_szorzat\_1(i+1)

**különben**

Kiír

**vége(ha)**

$x_i \leftarrow 2$

**Ha**  $i < n$  **akkor**

Descartes\_szorzat\_1(i+1)

**különben**

Kiír

**vége(ha)**

**Vége(algoritmus)**

Az első hívás

Descartes\_szorzat\_1(1) alakú.



Elemmezve az előbbi algoritmust észrevesszük, hogy két részlete majdnem azonos:  $x_i$  értéke előbb **1**, majd **2**. Megírjuk úgy, hogy egy  $j$  változó értékét **1**-től **2**-ig mozgatjuk:

**Algoritmus** Descartes\_szorzat\_2(i):  $\{ M = (1, 2) \}$

**Minden**  $j = 1, 2$  **végezd el:**

$x_i \leftarrow j$

**Ha**  $i < n$  **akkor**

Descartes\_szorzat\_2(i+1)

**különben**

Kiír

**vége(ha)**

**vége(minden)**

**Vége(algoritmus)**

# Megjegyzések

4. Ha  $m \geq 2$ ,  $x_i$  *nem két értéket vehet fel*, hanem  $m$  értéket.

**Algoritmus** Descartes\_3(i):     $\{ M = (1, 2, \dots, m) \}$

**Minden**  $j = 1, m$  **végezd el:**

$x_i \leftarrow j$

**Ha**  $i < n$  **akkor**

            Descartes\_3(i+1)

**különben**

            Kiír

**vége(ha)**

**vége(minden)**

**Vége(algoritmus)**

# $k$ elemű részhalmazok

Adva vannak az  $n$  és  $k$  ( $1 \leq k \leq n$ ) egész számok. Generáljuk rekurzívan az  $\{1, 2, \dots, n\}$  halmaz valamennyi,  $k$  elemet tartalmazó részhalmazát.

## Elemzés

- Az  $\{1, \dots, n\}$  halmaz  $k$  elemet tartalmazó részhalmaza egy tömb segítségével kódolható, amelynek  $k$  eleme van:  
 $x_1, x_2, \dots, x_k$ .
- A részhalmaz elemei különbözők és nem számít a sorrendjük.
- Vigyázunk, hogy az  $x$  sorozatba ne generáljuk kétszer, vagy többször ugyanazt a részhalmazt (esetleg, más sorrendű elemekkel).

# ***k* elemű részhalmazok**

- Ha az ***x*** sorozatba az elemek ***szigorúan növekvő sorrendben kerülnek:  $x_1 < x_2 < \dots < x_k$  egy részhalmazt csak egyszer állítunk elő.***

- Legyen ***k* = 3** és a halmaz:  **$\{1, 2, 3, 4\}$  (*n* = 4)**
- A részhalmazokat a Descartes szorzat elemeinek generálásához hasonlóan generáljuk:

1) **1 2 3**      2) **1 2 4**      3) **1 3 4**      4) **2 3 4**

- Ahhoz, hogy az ***x*<sub>2</sub>** és ***x*<sub>3</sub>** elemek kezdőértéket kaphassanak, ***x*<sub>1</sub>** csak **1** illetve **2** lehet; ugyanígy, ha ***x*<sub>1</sub> = 1**, ***x*<sub>2</sub>** csak **2**, illetve **3** lehet, és ha ***x*<sub>1</sub> = 2**, ***x*<sub>2</sub>** csak **3** lehet.
- Általánosítva, mivel valamennyi ***x<sub>i</sub>*** szigorúan nagyobb mint ***x<sub>i-1</sub>***, az értékei ***x<sub>i-1</sub> + 1***-től nőnek ***n - (k - i)***-ig.

# $k$ elemű részhalmazok

- Sorban kiválasztunk az  $\{1, \dots, n - k + i\}$  halmazból egy-egy értéket és a továbbiakban  $k - 1$  elemet tartalmazó részhalmazokat generálunk az  $\{x_1 + 1, \dots, n\}$  halmaz elemeiből.
- Ezeket az  $x$  sorozatban őrizzük meg a második helytől kezdődően.
- A  $k - 1$  elemet tartalmazó részhalmaz generálása céljából ugyanígy járunk el.
- Minden elem az előző utáni értéket veszi fel: ( $x$  elemeit  $0$ -tól indexeljük).
- $x$  globális változó és minden elemének kezdőértéke  $0$ .

# ***k* elemű részhalmazok**

**Algoritmus** Részhalmazok(*i*):

$\{ M = (1, 2, \dots, n), x \text{ globális} \Rightarrow x_i = 0, i = 0, 20 \}$

**Minden**  $j = x_{i-1} + 1, n - k + i$  **végezd el:**

$x_i \leftarrow j$

**Ha**  $i < k$  **akkor** Részhalmazok( $i+1$ )

**különb**en Kiír

**vége**(ha)

**vége**(minden)

**Vége**(algoritmus) [Programok\CPP\08kElemuRH.cpp](#)

A részhalmazokat generáló algoritmust az *i* paraméter **1** értékére hívjuk meg.

# Fibonacci sorozat

*Generáljuk a Fibonacci sorozat  $n$ -edik elemét!*

## *Elemzés*

Az  $n$ -edik elem kiszámításához szükségünk van az előtte található két elemre. De ezeket szintén az előttük levő elemekből számítjuk ki.

A sorozat: **0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89...**

$$F_1 \leftarrow 0, F_2 \leftarrow 1$$

$$F_i \leftarrow F_{i-2} + F_{i-1}, \text{ ha } i > 2$$

# Fibonacci sorozat

**Algoritmus Fibonacci(n):**

**Ha  $n = 1$  akkor**

**térítsd 0**

**különben**

**Ha  $n = 2$  akkor**

**térítsd 1**

**különben**

**térítsd  $\text{Fibonacci}(n-2) + \text{Fibonacci}(n-1)$**

**vége(ha)**

**vége(ha)**

**Vége(algoritmus)**



# Fibonacci sorozat

- Ha végrehajtjuk az algoritmust  $n = 6$ -ra, ez 15-ször hívja meg önmagát.
- Ezt a számot lecsökkenthetjük 9-re, ha a kiszámolt értékeket megőrizzük egy sorozatban. Legyen ez a sorozat  $F$ , amelyet globális változóként kezelünk.
- A két megoldás közötti különbséget annál jobban érzékeljük minél nagyobb  $n$ -re teszteljük ezeket.

**Algoritmus Fib(n):**

**Ha**  $(n > 2)$  **akkor**

$\text{Fib}(n-1)$

$F_n \leftarrow F_{n-1} + F_{n-2}$

**különben**

$F_1 \leftarrow 0$

**Ha**  $n = 2$  **akkor**

$F_2 \leftarrow 1$

**vége(ha)**

**vége(ha)**

**Vége(algoritmus)**

# Gyorshatvány

**Algoritmus** gyorsHativRek(n, x): {  $x^n$  al-Kvarizmi algoritmus }

**Ha**  $n = 0$  **akkor**

**térítsd** 1

**különben**

**Ha**  $n \bmod 2 = 1$  **akkor**

**térítsd**  $x * \text{gyorsHativRek}(n/2, x*x)$

**különben**

**térítsd**  $\text{gyorsHativRek}(n/2, x*x)$

**vége(ha)**

**vége(ha)**

**Vége(algoritmus)**

# Összes részhalmaz

Generáljuk az  $\{1, 2, \dots, n\}$  halmaz minden **részhalmazát**.

## Elemzés

Egy halmazt ebben az esetben is az  $x_1 < x_2 < \dots < x_i$  sorozattal ábrázolunk  $i = 1, \dots, n$ .

Ha  $n = 3$ , az üres halmaztól különböző részhalmazok a következők:

- |    | $x_1$ | $x_2$ | $x_3$ |
|----|-------|-------|-------|
| 1) | 1     |       |       |
| 2) | 1     | 2     |       |
| 3) | 1     | 2     | 3     |
| 4) | 1     | 3     |       |
| 5) | 2     |       |       |
| 6) | 2     | 3     |       |
| 7) | 3     |       |       |

# Megjegyzések

- A sorrend az úgynevezett *lexikográfikus sorrend*.
- A részhalmazok generálása feltételezi, hogy  $x_1$  számára rendre kiválasztunk egy bizonyos értéket és a továbbiakban az  $\{x_1 + 1, \dots, n\}$  halmaz elemeit generáljuk.
- Ezeket az  $x$  sorozatban tároljuk a második helytől kezdődően.
- Újabb részhalmazok generálása érdekében hasonlóan járunk el.
- Általános esetben, ha az  $\{x_{i-1} + 1, \dots, n\}$  halmazt szeretnénk generálni (az  $x$  sorozatban az  $i$ -edik helytől kezdve) rendre kiválasztjuk a halmaz elemeit az  $x_i$  számára és generáljuk az  $\{x_{i+1}, \dots, n\}$  részhalmazait.
- Ezeket az  $i + 1$ -edik helytől kezdődően őrizzük meg.
- Az  $\{x_{i+1}, \dots, n\}$  halmaznak van legalább egy eleme, ha  $x_i < n$ .

# Összes részhalmaz

**Algoritmus** Részhalmaz(i):

$\{ M = (1, 2, \dots, n), x \text{ globális} \Rightarrow x_i=0, i=0, n \}$

**Minden**  $j = x_{i-1} + 1, n$  **végezd el:**

$x_i \leftarrow j$        $\{ \text{amint kiválasztottunk az } x_i \text{ számára egy új} \}$   
 $\{ \text{értéket, a részhalmazt azonnal kiírjuk} \}$

Kiír(i)  $\{ x_1, x_2, \dots, x_i \}$

Részhalmaz(i+1)

**vége(minden)**

**Vége(algoritmus)**

A kilépési feltétel ( $x_i = n$ ) el van rejtve a **Minden** utasításba (ha  $x_i = n$ , a ciklusváltozó kezdőértéke  $x_{i+1}$  nagyobb mint a végső érték, így a **Minden** ciklusmagja nem lesz többet végrehajtva és a program kilép az aktuális hívásból).

# Partíciók

Generáljuk az  $n$  természetes szám partícióit!

- Partíció alatt azt a felbontást értjük, amely során az  $n$  számot  $0$ -nál nagyobb **pozitív számok összegeként írjuk fel**.
  - Két partíció **különbözik** egymástól, ha **vagy az előforduló értékek vagy az előfordulásuk sorrendje különbözik**.
- $(n = p_1 + p_2 + \dots + p_k, p_i \in \mathbb{N}^*, i = 1, \dots, k, k = 1, \dots, n).$

## Példa

Ha  $n = 4$ :

$$4 = 1 + 1 + 1 + 1$$

$$4 = 1 + 1 + 2$$

$$4 = 1 + 2 + 1$$

$$4 = 1 + 3$$

$$4 = 2 + 1 + 1$$

$$4 = 2 + 2$$

$$4 = 3 + 1$$

$$4 = 4$$

# Elemzés

- A generálás során, rendre *kiválasztunk egy lehetséges értéket* a partíció első  $p_1$  eleme számára és
- *Generáljuk a fennmaradt  $n - p_1$  szám partícióit.*
- Ez a maradék (tulajdonképpen különbség) egy új  $n$ , amellyel ugyanúgy járunk el.
- Egy partíciót legeneráltunk és kiírhatjuk ha  $n$  aktuális értéke  $0$ .
- Az algoritmust a *Partíció(1,n)* utasítással hívjuk meg először.

# Partíciók

**Algoritmus** Partíció( $i, n$ ):

**Minden**  $j = 1, n$  **végezd el:**

$p_i \leftarrow j$

**Ha**  $j < n$  **akkor**

    Partíció( $i+1, n-j$ )

**különben**

    Kiír( $i$ )

**vége**(ha)

**vége**(minden)

**Vége**(algoritmus)



# Mat–Info Verseny és Felvételi feladatok

## 1. Ackermann

- Legyenek az  $m$  és  $n$  természetes számok ( $0 \leq m \leq 10$ ,  $0 \leq n \leq 10$ ), valamint az **Ack( $m$ ,  $n$ )** algoritmus, amely kiszámítja az *Ackermann*-függvény értékét  $m$  és  $n$  esetében.
- Állapítsátok meg, hányszor hívja meg önmagát az **Ack( $m$ ,  $n$ )** algoritmus a következő utasítások végrehajtásának következtében.

$m \leftarrow 1$ $n \leftarrow 2$ <b>Ack(<math>m</math>, <math>n</math>)</b>
--

**Algoritmus** Ack(m, n):

**Ha**  $m = 0$  **akkor**

**térítsd**  $n + 1$

**különben**

**Ha**  $m > 0$  **és**  $n = 0$  **akkor**

**térítsd** Ack(m - 1, 1)

**különben**

**térítsd** Ack(m - 1, Ack(m, n - 1))

**vége(ha)**

**vége(ha)**

**Vége(algoritmus)**

A. 7-szer

B. 10-szer

C. 5-ször

D. ugyanannyiszor, mint a következő

utasítások végrehajtásának következtében:

$m \leftarrow 1$

$n \leftarrow 3$

Ack(m, n)

# Megoldás

- Adott egy rekurzív alprogram, és meg kell állapítanunk, hogy a paraméterek adott értékére hányszor hívja meg önmagát.
- A megoldáshoz követnünk kell a paraméterek értékeit a hívássorozaton belül, – esetleg lerajzoljuk a végrehajtási verem tartalmát és megszámloljuk a rekurzív hívásokat:

$\text{Ack}(1, 2) \Rightarrow$  (1. hívás):

$\text{Ack}(0, \text{Ack}(1, 1)) \Rightarrow$  (2. hívás):

$\text{Ack}(1, \text{Ack}(1, 0)) \Rightarrow$  (3. hívás):

$\text{Ack}(0, 1) \Rightarrow$  (4. hívás):

$\text{Ack}(0, 2) \Rightarrow$  (5. hívás)

$\text{Ack}(0, 3)$ : vége.

# Megoldás

- Tehát, a rekurzív hívások száma 5, vagyis a **C.** válasz jó
- Így  $\Rightarrow$  **A.** és **B.** válaszok nem jók.
- Az A esetben leírt módon megvizsgáljuk a **D.** esetet is
- $\Rightarrow$  a rekurzív hívások száma 7  $\Rightarrow$  a **D.** válasz sem jó

## 2. Mely értékek szükségesek?

Legyen a különbség( $a$ ,  $n$ ) algoritmus, ahol  $a$  egy  $n$  elemű ( $0 < n < 100$ ) sorozat, amely egész számokat tárol:

**Algoritmus** különbség( $a$ ,  $n$ ) 6, 4, 5, 5

**Ha**  $n = 0$  **akkor**

**térítsd** 0

**vége**(ha)

**Ha**  $|a[n]| \bmod 2 = 0$  **akkor**

**térítsd** különbség( $a$ ,  $n - 1$ ) +  $a[n]$

**különb**en

**térítsd** különbség( $a$ ,  $n - 1$ ) -  $a[n]$

**vége**(ha)

**Vége**(algoritmus)

Az  $n$  és  $a$  mely értékeire térít a fenti algoritmus 0-át?

**A.**  $n = 4$  és  $a = (6, 4, 5, 5)$

**B.**  $n = 4$  és  $a = (-6, 5, 4, -7)$

**C.**  $n = 8$  és  $a = (-6, 5, -1, -4, 1, 4, -7, 6)$

**D.**  $n = 8$  és  $a = (-6, -3, 0, 1, 2, 3, -1, 4)$

## Megoldás

- az aktuális összeget nem tároljunk egy változóban
- az utolsó híváskor 0 kezdőértéket térítünk
- minden rekurzív hívásból való visszatéréskor az aktuális összeghez hozzáadjuk a soron következő számot, ha az páros, illetve kivonjuk, ha páratlan.
- Azokat a bemeneti adatokat választjuk ki, amelyeknek esetében a páros számok összege egyenlő a páratlanok összegével. Helyes válaszok: **A.**, **B.** és **D.**

### 3. Kiegészítés

Legyen a **kizárPáratlan( $n$ )** algoritmus, ahol  $n$  ( $1 \leq n \leq 100\,000$ ) természetes szám.

Állapítsátok meg, melyik utasítást kellene a „...” helyére írni, ahhoz, hogy az algoritmus zárja ki az  $n$  számból a páratlan értékű számjegyeket.

**Algoritmus** **kizárPáratlan( $n$ )**

**Ha  $n = 0$  akkor**

**térítsd 0**

**vége(ha)**

**Ha  $n \bmod 2 = 1$  akkor**

**térítsd **kizárPáratlan( $n \div 10$ )****

**vége(ha)**

**térítsd ...**

**Vége(algoritmus)**

- A.**  $\text{kizárPáratlan}(n \bmod 10) * 10 + n \text{ DIV } 10$
- B.**  $\text{kizárPáratlan}(n) * 10 + n \bmod 10$
- C.**  $\text{kizárPáratlan}(n \text{ DIV } 10) * 10 + n \bmod 10$
- D.**  $\text{kizárPáratlan}((n \text{ DIV } 10) \bmod 10) * 10$

## Megoldás

- Egy új számot kell felépítenünk az adott  $n$  szám páros számjegyeiből.
- Amíg  $n$  páratlan szám, megtörténik a rekurzív hívás, anélkül, hogy módosítanánk az új számnak megfelelő térítendő értéket.
- A páros számjegyek a **C.** válaszban leírt utasítás eredményeként épülnek majd a térítendő értékbe.
- Az eddig kiszámolt értéket szorozzuk 10-zel és hozzáadjuk a páros számjegy értékét. Az  $n$  paraméter új értéke  $n / 10$ .
- Tehát a helyes válasz: **C.**



## 4. Számjegyszorzat

- A `sZámJegyek(n, d)` algoritmus ( $n$  és  $d$  természetes számok,  $10 \leq n \leq 100\,000$ ,  $1 \leq d \leq 9$ ), meghatározza és visszatéríti azt a legkisebb természetes számot, amelynek  $d$ -nél kisebb vagy  $d$ -vel egyenlő, nem nulla számjegyei vannak és amely számjegyeknek a szorzata egyenlő  $n$ -nel.
- Például, ha  $n = 108$  és  $d = 9$ , az algoritmus 269-et térít vissza. Ha ilyen szám nem létezik, az algoritmus -1-et térít.
- Állapítsátok meg, hányszor hívja meg önmagát a `sZámJegyek(n, d)` algoritmus az alábbi programrészlet végrehajtásának következtében:

**Algoritmus számJegyek(n, d)**

**Ha  $d = 1$  akkor**

**Ha  $n = 1$  akkor térítsd 0**

**különben térítsd -1**

**vége(ha)**

**különben**

**Ha  $n \bmod d = 0$  akkor**

**érték  $\leftarrow$  számJegyek( $n \text{ DIV } d$ ,  $d$ )**

**Ha érték  $< 0$  akkor térítsd -1**

**különben térítsd érték \* 10 + d**

**vége(ha)**

**különben térítsd számJegyek( $n$ ,  $d - 1$ )**

**vége(ha)**

**vége(ha)**

**Vége(algoritmus)**

## 5. Számjegyszorzat

beOlvas  $n$

érték  $\leftarrow$  számJegyek( $n$ , 9)

- A. Ha  $n = 108$ , az algoritmus 11-szer hívja meg önmagát.
- B. Ha  $n = 109$ , az algoritmus 8-szor hívja meg önmagát.
- C. Ha  $n = 13$ , az algoritmus egyszer sem hívja meg önmagát.
- D. Ha  $n = 100$ , az algoritmus 10-szer hívja meg önmagát.

### Megoldás

A rekurzív hívások számát megszámlolhatjuk, ha hívássorozatot készítünk vagy, ha követjük a paraméterek értékeit a végrehajtási veremben:

számjegyek(108, 9)  $\Rightarrow$  (0. hívás):

számjegyek(12, 9)  $\Rightarrow$  (1. hívás):

számjegyek(12, 8)  $\Rightarrow$  (2. hívás):

számjegyek(12, 7)  $\Rightarrow$  (3. hívás):

számjegyek(12, 6)  $\Rightarrow$  (4. hívás):

számjegyek(2, 6)  $\Rightarrow$  (5. hívás):

számjegyek(2, 5)  $\Rightarrow$  (6. hívás):

számjegyek(2, 4)  $\Rightarrow$  (7. hívás):

számjegyek(2, 3)  $\Rightarrow$  (8. hívás):

számjegyek(2, 2)  $\Rightarrow$  (9. hívás):

számjegyek(1, 2)  $\Rightarrow$  (10. hívás):

számjegyek(1, 1)  $\Rightarrow$  (11. hívás)

- A hívások száma 11, tehát **A.** helyes.
- A **B.** esetben, 8 rekurzív hívás után áll le a hívások sorozata.  
Így a **B.** válasz is helyes.
- A **C.** válasz nem helyes, mivel, ha  $n = 13$ , az algoritmusnak „nincs oka”, hogy egyszer se hívja meg önmagát.
- A **D.** esetben a rekurzív hívások száma 8, tehát **D.** sem helyes.

## 6. Varázslat

- Egy számjegymágus olyan varázslatot végez, amelynek eredményeképpen egy  $x$  természetes szám ( $100 < x < 1\,000\,000$ , amelynek a 10-es számrendszerben van legkevesebb két 0-tól különböző számjegye) szétválik két pozitív természetes számra: a **bal** és **jobb** számokra, amelyek egymás után ragasztva megadják az  $x$  számot. Ugyanakkor a **bal** és **jobb** számok szorzata a lehető legnagyobb. Például, ha  $x = 1\,092$ , a varázslat szétválasztja a **bal** = 10 és **jobb** = 92 számokra.
- Az adott algoritmusok közül melyik alkalmazza a varázslatot az  $x$  természetes számra, amelynek 10-es számrendszerben van legkevesebb két 0-tól különböző számjegye ( $100 \leq x \leq 1\,000\,000$ )? Az algoritmus meghatározza a  $z$  természetes számban ( $0 \leq z \leq 1\,000\,000$ ) az  $x$  szám **jobb** részét. Az alábbi algoritmusok léteznek:

# Varázslat

- `hatvány(b, p)` – meghatározza a  $b^p$  értéket ( $b$  a  $p$ . hatványon),  $b, p$  – természetes számok ( $1 \leq b \leq 20, 1 \leq p \leq 20$ );
- `szjSzáma(sz)` – meghatározza az  $sz$  szám ( $0 \leq sz \leq 1\,000\,000$ ) számjegyeinek darabszámát;

**A.**

**Algoritmus** varázslat(x, z)

maxSzorzat  $\leftarrow$  -1

eredmény  $\leftarrow$  0

**Amíg**  $x > 0$  **végezd el**

$z \leftarrow (x \text{ MOD } 10) * \text{hatvány}(10, \text{szjSzáma}(z)) + z$

$x \leftarrow x \text{ DIV } 10$

**Ha**  $x * z > \text{maxSzorzat}$  **akkor**

$\text{maxSzorzat} \leftarrow x * z$

$\text{eredmény} \leftarrow z$

**vége(ha)**

**vége(amíg)**

**térítsd** maxSzorzat

**Vége(algoritmus)**

**B.**

**Algoritmus** varázslat( $x$ ,  $z$ )

$t \leftarrow 0$

**Ha**  $x > 0$  **akkor**

$y \leftarrow (x \text{ MOD } 10) * \text{hatvány}(10, \text{szjSzáma}(z)) + z$

$t \leftarrow x \text{ DIV } 10$

**Ha**  $x * z < y * t$  **akkor**

**térítsd** varázslat( $y$ ,  $t$ )

**különben**

**térítsd**  $t$

**vége**( $ha$ )

**különben**

**térítsd**  $t$

**vége**( $ha$ )

**Vége**(algoritmus)



**C.**

**Algoritmus** varázslat(x, z)

maxSzorzat  $\leftarrow$  -1

eredmény  $\leftarrow$  0

**Amíg** x > 0 **végezd el**

z  $\leftarrow$  (x **MOD** 10) \* hatvány(10, szjSzáma(z)) + z

x  $\leftarrow$  x **DIV** 10

**Ha** x \* z > maxSzorzat **akkor**

maxSzorzat  $\leftarrow$  x \* z

eredmény  $\leftarrow$  z

**vége(ha)**

**vége(amíg)**

**térítsd** eredmény

**Vége(algoritmus)**

**D.**

**Algoritmus** varázslat(x, z)

**Ha**  $x > 0$  **akkor**

$y \leftarrow (x \text{ MOD } 10) * \text{hatvány}(10, \text{szjSzáma}(z)) + z$

$t \leftarrow x \text{ DIV } 10$

**Ha**  $x * z < y * t$  **akkor**

**térítsd** varázslat(y, t)

**különben**

**térítsd** z

**vége**(ha)

**különben**

**térítsd** z

**vége**(ha)

**Vége**(algoritmus)

# Megoldás

- Az **A.** algoritmus inicializálja az **eredmény** változó értékét, de sehol nem használja fel.
- Ez az észrevétel arra ösztönöz, hogy keressünk a változatok között egy másikat, amely szintén iteratív és nagyvonalakban hasonlít az **A.** algoritmushoz, de nem tartalmazza az előbb említett hibát, vagy ehhez hasonlót.
- Ez a **C.** algoritmus, amely nem a **maxSzorzat** változó értékét téríti, hanem az **eredmény** változóét. A két algoritmus közül egyértelmű, hogy az **A.** nem helyes, mivel a feladat nem a maximális szorzatot kéri, hanem az **x** szám **jobb** részét.

# Megoldás

- A **C.** algoritmus a **z** változóban építi a szám **jobb** részét **x** számjegyeiből jobbról balra haladva, és minden lépésben aktualizálja, ha szükséges, a **maxSzorzat** értékét is.
- Ha ez megtörtént, megjegyzi az **eredmény** változóban azt a **z** értéket (a szám **jobb** részét), amely nagyobb **maxSzorzat** értéket eredményez. Végül a **maxSzorzat** legnagyobb értékéhez „tartozó” **eredmény** változó értékét téríti. Tehát a **C.** algoritmus helyes.

# Megoldás

- A **B.** algoritmus két paramétere, hasonlóan a **C.** algoritmus-hoz: **x** és **z**, ahol **x**-ben az **x** szám **bal** részét, **z**-ben a **jobb** részét szeretné felépíteni az algoritmus.
- Ez az algoritmus a **t** változó értékét téríti, vagy meghívja önmagát az **y** és **t** aktuális paraméterekkel. De **t** az **x** változónak a **bal** része, hiszen 10-zel való osztások eredménye. Ebből következik, hogy a `varázslat(y, t)` rekurzív hívás helyett `varázslat(t, y)` lett volna a helyes, ezáltal **x** bal része **t**-től kapna értéket, jobb része **y**-től. A **térítsd t** utasítás is hibás, hiszen a feladat a szám **jobb** részét kéri, de **t** a **bal** része.
- Összehasonlítjuk a **B.** rekurzív megoldást a **D.**, szintén rekurzív megoldással és azonnal látjuk, hogy ugyanaz a gond a rekurzív hívás aktuális paramétereinek sorrendjével: `varázslat(y, t)` helyett `varázslat(t, y)` lett volna a helyes sorrend.
- Tehát csak egy helyes megoldást találtunk, a **C.**-t.