

Rekurzió

Ionescu Klára

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 közben 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.

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*

különben

Ki: 'Fordított szó: '

vége(ha)

Ki: betű

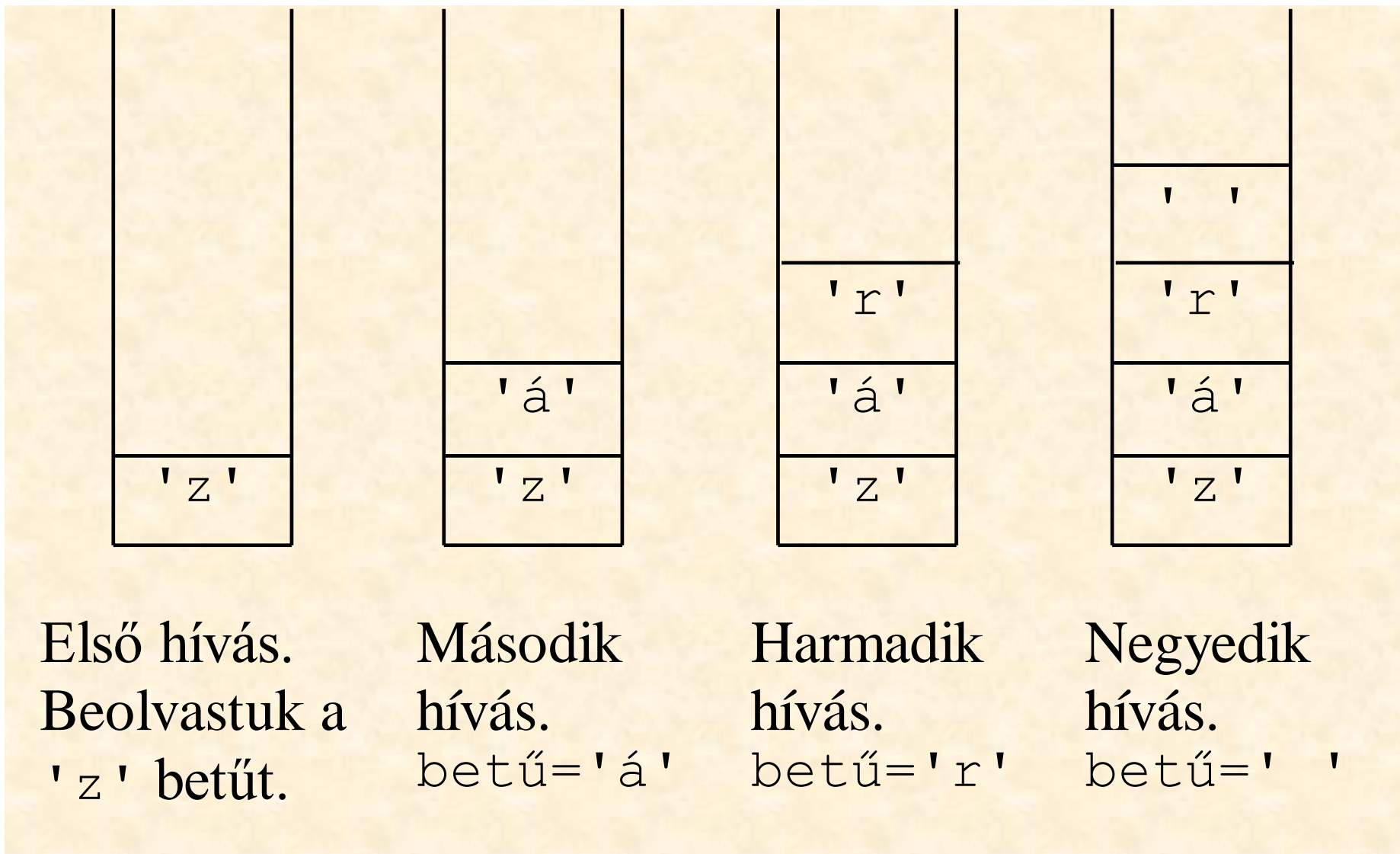
Vége(algoritmus)

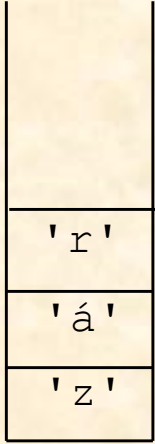
Az alprogram hívása: **Fordít**

Meg lehet valósítani a következőképpen is:

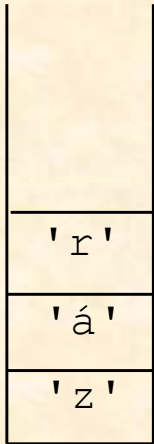
```
Algoritmus Fordít:                                     // nincs paraméter, mivel  
                                                         // az alprogramban olvasunk be és írunk ki  
  
    Be: betű  
    Ha betű ≠ '*' akkor  
        Fordít                                           // meghívja önmagát, hogy  
                                                         // megfordíthassa a szó fennmaradt részét  
  
        Ki: betű  
    különben  
        Ki: 'Fordított szó: '  
    vége(ha)  
Vége(algoritmus)
```

Példa: szó = zár. Megfordítva = ráz

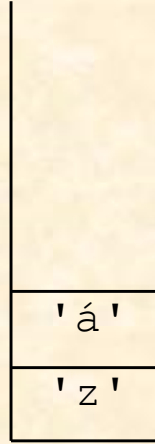




Kilépünk az aktuális hívásból.



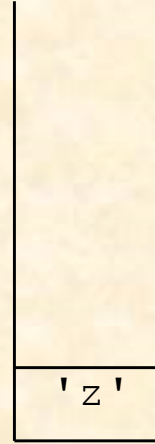
Kiírjuk a betűt, amely 'r'



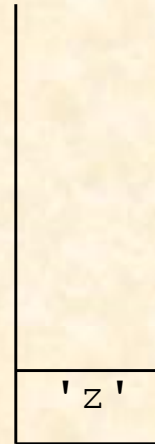
Kilépünk az aktuális hívásból.



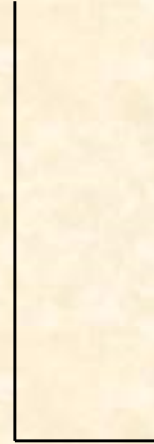
Kiírjuk a betűt, amely 'á'



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, n):

// az első hívás aktuális paramétere 1

Be: szó

Ha $i < n$ **akkor**

Szavakat_fordít_2($i + 1$, n)

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 **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 **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 Lngo(m, n):

mar $\leftarrow m \bmod n$

// mar = lokális változó,

// így nem fogjuk egy híváson belül kétszer számítani a maradékot

Ha mar = 0 akkor

térítsd n

különb

térítsd Lngo(n, mar)

vége(ha)

Vége(algoritmus)

Az első hívás történhet például egy kiíró utasításból (**Ki: Lngo(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 $n = 3$, $M^3 = M \times M \times M$, $M = \{1, 2\}$

	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, n, x):    {  $M = (1, 2)$  }  
     $x_i \leftarrow 1$   
    Ha  $i < n$  akkor  
        Descartes_szorzat_1(i + 1, n, x)  
    különben  
        Kiír  
    vége(ha)  
     $x_i \leftarrow 2$   
    Ha  $i < n$  akkor  
        Descartes_szorzat_1(i + 1, n, x)  
    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_j értéke előbb 1, majd 2. Megírjuk úgy, hogy egy j változó értékét 1-től 2-ig mozgatjuk:

```
Algoritmus Descartes_szorzat_2(i, n, x):           {  $M = (1, 2)$  }  
    Minden j = 1, 2 végezd el:  
         $x_i \leftarrow j$   
        Ha  $i < n$  akkor  
            Descartes_szorzat_2(i + 1, n, x)  
        különben  
            Kiír  
        vége(ha)  
    vége(minden)  
Vége(algoritmus)
```


Megjegyzések

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

Algoritmus Descartes_3(i, n, m, x): $\{ M = (1, 2, \dots, m) \}$
 Minden $j = 1, m$ végezd el:
 $x_i \leftarrow j$
 Ha $i < n$ akkor
 Descartes_3($i + 1, n, m, x$)
 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.
- Vigyázzunk, 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_2 és x_3 elemek kezdőértéket kaphassanak, x_1 csak 1 illetve 2 lehet; ugyanígy, ha $x_1 = 1$, x_2 csak 2 , illetve 3 lehet, és ha $x_1 = 2$, x_2 csak 3 lehet.
- Általánosítva, mivel valamennyi x_i szigorúan nagyobb mint x_{i-1} , az értékei $x_{i-1} + 1$ -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, n, x):

// $M = (1, 2, \dots, n)$, x 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($i + 1, n, x$)

különben

 Kiír

vége(ha)

vége(minden)

Vége(algoritmus)

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

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, n, x):

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

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

$x_i \leftarrow j$ *// amint kiválasztottunk az x_i számára egy új
// értéket, a részhalmazt azonnal kiírjuk*

Kiír(i, x) *// x_1, x_2, \dots, x_i*

Részhalmaz($i + 1, n, x$)

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 , p):

Minden $j = 1, n$ végezd el:

$p_i \leftarrow j$

Ha $j < n$ akkor

Partíció($i + 1$, $n - j$, p)

különben

Kiír(i)

vége(ha)

vége(minden)

Vége(algoritmus)