Adatszerkezetek

12. hét – Rekurzió: a probléma visszavezetése önmaga kisebb példányára

Dr. Koczka Ferenc · Eszterházy Károly Egyetem · 2026

Ismétlés

- **Hívási verem** (7. hét): minden függvényhívás egy keretet tesz a verembe, a visszatérés kiveszi
- **Láncolt lista:** egy elem + a lista maradéka (ami szintén lista!)
- **Bináris keresés:** a feladatot a felére csökkentjük, és ugyanúgy folytatjuk

A mai óra kérdései

- Mi a rekurzió, és hogyan működik a számítógépben?
- Hogyan írjunk helyes rekurzív függvényt?
- Mikor jó a rekurzió, és mikor veszélyes?

Mi a rekurzió?

Rekurzió: egy függvény **önmagát hívja**, a feladat egy **kisebb** példányára.

Minden rekurzív függvény két részből áll:

1. **Alapeset (báziseset):** a legegyszerűbb eset, amelyet közvetlenül megoldunk – itt **megáll** a rekurzió
2. **Rekurzív eset:** a feladatot visszavezetjük ugyanannak a feladatnak egy **kisebb** példányára

Faktoriális: $n! = 1 \cdot 2 \cdot \ldots \cdot n$

$0! = 1$ **(alapeset)**

$n! = n \cdot (n - 1)!$ **(rekurzív eset)**

Matematikában ez a **teljes indukció** párja: ha az alapeset igaz, és a kisebből következik a nagyobb, akkor minden esetre igaz.

Faktoriális

Pszeudokód

```
Függvény Fakt(n)
  Ha n ≤ 1 akkor
    Vissza: 1                  // alapeset
  különben
    Vissza: n · Fakt(n − 1)    // rekurzív eset
  Elágazás vége
Függvény vége
```

Java

```java
static long fact(int n) {
    if (n <= 1) return 1;
    return n * fact(n - 1);
}
```

Go

```go
func fact(n int) int64 {
    if n <= 1 {
        return 1
    }
    return int64(n) * fact(n-1)
}
```

A program kimenete (`Factorial`):

```
fact(4) hívás
  fact(3) hívás
    fact(2) hívás
      fact(1) hívás
      fact(1) = 1
    fact(2) = 2
  fact(3) = 6
fact(4) = 24
```

Hogyan működik? A hívási verem

fakt(4) kiértékelése: lefelé hívások (a verem nő), felfelé visszatérések (a verem csökken)



- Minden hívás saját **keretet** kap: saját **n** paraméter, saját visszatérési cím
- A szorzások csak **visszafelé** (a visszatérések során) hajtódnak végre
- Tárigény: **O(n)** – n darab keret a veremben

Helyes rekurzió – három szabály

1. **Van alapeset**, amely rekurzió nélkül ad eredményt
2. Minden rekurzív hívás **közelebb visz** az alapesethez (kisebb n, rövidebb lista, kisebb tartomány)
3. **Bízz a rekurzióban:** tételezd fel, hogy a kisebb feladatra a hívás helyes eredményt ad, és csak azt gondold végig, hogyan lesz belőle a nagyobb feladat megoldása

Ha nincs alapeset, vagy nem közelítünk felé: **végtelen rekurzió** → a verem megtelik
Java: `StackOverflowError` (kb. 10 000–30 000 hívás után) · Go: a verem nő, de 1 GB-nál összeomlik

A rekurzió fajtái

Fajta	Jelentés	Példa
Közvetlen	f önmagát hívja	faktoriális
Közvetett	f hívja g-t, g hívja f-et	páros(n) → páratlan(n – 1)
Lineáris	egy rekurzív hívás	faktoriális, összegzés, lista hossza
Fa (többszörös)	több rekurzív hívás	Fibonacci, Hanoi tornyai, fabejárás
Farokrekurzió	a rekurzív hívás az utolsó művelet	`gcd(b, a mod b)`

A farokrekurzió gépiesen ciklussá alakítható (egyes fordítók ezt automatikusan meg is teszik – a Java és a Go nem).

Hatékonyság: a Fibonacci-példa

Fibonacci-számok

Definíció: $F(0) = 0$, $F(1) = 1$, $F(n) = F(n-1) + F(n-2)$

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …

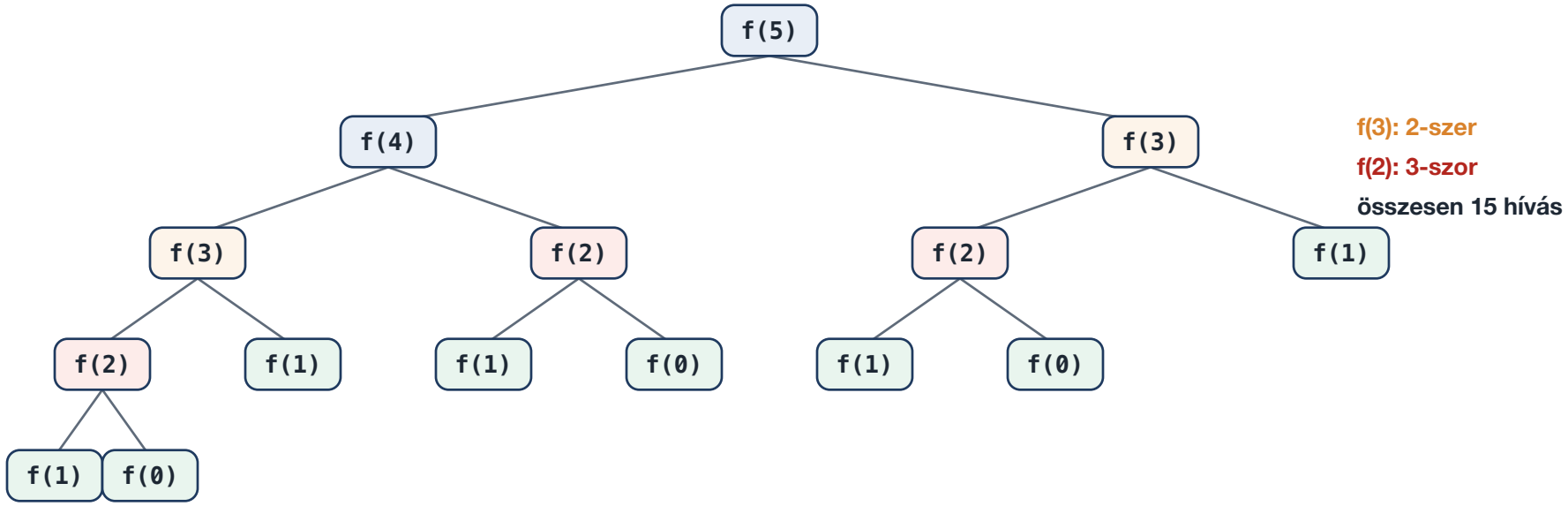
```
Függvény Fib(n)
  Ha n < 2 akkor
    Vissza: n
  különben
    Vissza: Fib(n - 1) + Fib(n - 2)
  Elágazás vége
Függvény vége
```

```
static long fib(int n) {
    if (n < 2) return n;
    return fib(n - 1) + fib(n - 2);
}
```

Egyszerű és pontosan követi a definíciót – de…

A naiv Fibonacci hívási fája

fib(5) hívási fája: ugyanazokat az értékeket újra és újra kiszámolja



f(3): 2-szer

f(2): 3-szor

összesen 15 hívás

Mérési eredmények

n	fib(n)	naiv hívások	memoizálva	idő (naiv)
10	55	177	19	0,0 ms
20	6 765	21 891	39	0,1 ms
30	832 040	2 692 537	59	2,6 ms
40	102 334 155	**331 160 281**	79	**320 ms**

- A naiv változat **exponenciális**: $O(\phi^n) \approx O(1{,}618^n)$ – n = 50-re percek, n = 100-ra évmilliók
- **Memoizáció:** a már kiszámolt értékeket egy **hash táblában** tároljuk → **O(n)**
- **Iteráció:** két változóval, O(n) idő, **O(1)** tár

Javított változatok

Memoizáció (Java)

```java
static HashMap<Integer, Long> memo = new HashMap<>();

static long fibMemo(int n) {
    if (n < 2) return n;
    Long cached = memo.get(n);
    if (cached != null) return cached;
    long r = fibMemo(n - 1) + fibMemo(n - 2);
    memo.put(n, r);
    return r;
}
```

Iteráció (Go)

```go
func fibIter(n int) int {
    a, b := 0, 1
    for i := 0; i < n; i++ {
        a, b = b, a+b
    }
    return a
}
```

A memoizáció a **dinamikus programozás** alapötlete: minden részfeladatot csak egyszer oldunk meg.

Klasszikus példák

Rekurzív gondolkodás: „első + maradék”

Tömb összege: az első elem + a maradék összege

```java
static int sum(int[] a, int from) {
    if (from == a.length) return 0;
    return a[from] + sum(a, from + 1);
}
```

String megfordítása: a maradék megfordítva + az első karakter

```java
static String reverse(String s) {
    if (s.isEmpty()) return s;
    return reverse(s.substring(1)) + s.charAt(0);
}
```

Palindrom: az első = az utolsó, és a belseje is palindrom

```go
func isPalindrome(r []rune) bool {
    if len(r) < 2 {
        return true
    }
    return r[0] == r[len(r)-1] &&
        isPalindrome(r[1:len(r)-1])
}
```

Láncolt lista hossza: 1 + a maradék hossza

```go
func length(p *Node) int {
    if p == nil {
        return 0
    }
    return 1 + length(p.Next)
}
```

Gyors hatványozás – oszd meg és uralkodj

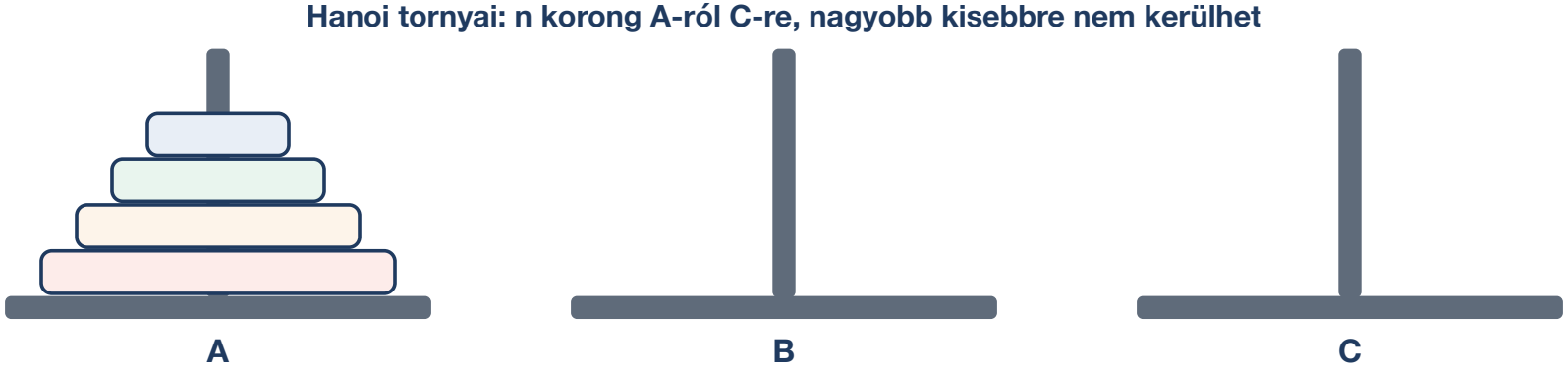
x^{13} kiszámítása 12 szorzás helyett:

- $x^{13} = (x^6)^2 \cdot x$ $x^6 = (x^3)^2$ $x^3 = (x^1)^2 \cdot x$ → összesen **5** szorzás

```
Függvény Hatvány(x, n)
  Ha n = 0 akkor Vissza: 1
  fél ← Hatvány(x, n div 2)
  Ha n mod 2 = 0 akkor Vissza: fél · fél
  különben Vissza: fél · fél · x
Függvény vége
```

- Minden hívás **felezi** n-t → **O(log n)** hívás (mint a bináris keresés)
- Ugyanez a minta: **bináris keresés rekurzívan**, összefésülő rendezés, gyorsrendezés (13. hét)

Hanoi tornyai




```
Eljárás Hanoi(n, honnan, hova, segéd)
  Ha n > 0 akkor
    Hanoi(n – 1, honnan, segéd, hova)
    Mozgat(honnan, hova)
    Hanoi(n – 1, segéd, hova, honnan)
  Elágazás vége
Eljárás vége
```

- Lépések száma: **$2^n - 1$** → $O(2^n)$
- n = 3: 7 lépés · n = 20: 1 048 575 lépés
- n = 64: másodpercenként egy lépéssel kb. **585 milliárd év**

Iteratívan sokkal nehezebb megírni!

Visszalépéses keresés: permutációk

Az `A, B, C` összes sorrendje: ABC, ACB, BAC, BCA, CAB, CBA

```
static void permute(List<Character> prefix, boolean[] used, char[] items) {
    if (prefix.size() == items.length) { print(prefix); return; }  // alapeset
    for (int i = 0; i < items.length; i++) {
        if (used[i]) continue;
        used[i] = true;  prefix.add(items[i]);          // döntés
        permute(prefix, used, items);                   // a maradék rekurzívan
        prefix.remove(prefix.size() - 1);  used[i] = false;   // visszalépés
    }
}
```

- n elemnek **n!** permutációja van – O(n · n!)
- Ugyanez a minta: sudoku, útvesztő, 8 királynő, részhalmazok

Rekurzió vagy iteráció?

	Rekurzió	Iteráció
Kód	rövid, a definíciót követi	gyakran hosszabb
Tárigény	O(mélység) a veremben	általában O(1)
Sebesség	függvényhívás költsége	gyorsabb
Veszély	verem túlcsordulása	–
Természetes alkalmazás	**fák**, oszd meg és uralkodj, visszalépés	egyszerű ciklusok, listák

Minden rekurzió átírható iterációvá (szükség esetén egy saját **veremmel**), és fordítva.

Ökölszabály: ha a feladat **fa szerkezetű** (több rekurzív hívás), a rekurzió sokkal egyszerűbb; ha lineáris, inkább ciklust írjunk.

Összefoglalás

- **Rekurzió:** a függvény önmagát hívja egy **kisebb** feladatra
- Kell: **alapeset** + a hívások **közelítsenek** az alapesethez
- Működés: **hívási verem** – minden hívás saját kerettel, tárigény O(mélység)
- **Fa rekurzió** könnyen exponenciális lehet (naiv Fibonacci) → **memoizáció** vagy iteráció
- Klasszikus példák: faktoriális, hatványozás (O(log n)), Hanoi tornyai (2^n – 1), permutációk
- A **fák** (14–15. hét) és a gyors rendezések (13. hét) természetes eszköze

Következő hét: rendezések II. – összefésülő rendezés és gyorsrendezés.

Ellenőrző kérdések

1. Mi lesz a `Fakt(5)` hívás során a hívási verem legnagyobb mélysége?
2. Mi a hiba? `Függvény F(n): Vissza: n + F(n – 1)`
3. Hányszor hívódik meg a naiv `Fib` függvény a `Fib(5)` kiszámítása során?
4. Mit ír ki? `Eljárás P(n): Ha n > 0 akkor P(n – 1); Kiír(n)` – `P(3)`
5. És ha a `Kiír(n)` a `P(n – 1)` elé kerül?
6. Hány lépésben rakható át 5 korong a Hanoi tornyai játékban?

Gyakorlat

✎ Bemutató programok

Program	Java	Go	Mit mutat?
Faktoriális	`Factorial.java`	`h12/factorial`	hívások behúzásos kiírása, túlcsordulás
Fibonacci	`Fibonacci.java`	`h12/fibonacci`	hívások száma, memoizáció, idő
Példák	`RecursionExamples.java`	`h12/examples`	összeg, hatvány, megfordítás, palindrom, bináris keresés
Hanoi	`Hanoi.java`	`h12/hanoi`	lépések kiírása, $2^n - 1$
Veremmélység	`StackDepth.java`	`h12/stackdepth`	`StackOverflowError` Java-ban, növekvő verem Go-ban
Permutációk	`Permutations.java`	`h12/permutations`	visszalépéses keresés

✎ Feladatok

Oldd meg **rekurzióval** (ciklus nélkül)!

1. **Számjegyösszeg:** `2026 → 10`
2. **Kettes számrendszer:** `50 → "110010"` (vesd össze a 7. heti veremmel!)
3. **Maximum:** egy tömb legnagyobb eleme
4. **Megszámlálás:** hányszor fordul elő **x** egy tömbben?
5. **Lista visszafelé:** írd ki egy láncolt lista elemeit fordított sorrendben!
6. **Lista hossza** rekurzívan
7. **Binomiális együttható:** C(n, k) = C(n – 1, k – 1) + C(n – 1, k) – miért lassú nagy n-re?

Megoldások: `Exercises12.java`, `h12/exercises`

✎ Házi feladat

1. Írd meg a binomiális együtthatót **memoizációval**, és mérd meg a gyorsulást C(30, 15) esetén!
2. Írj rekurzív függvényt, amely kiírja egy halmaz **összes részhalmazát**! Hány van?
3. **Útvesztő:** egy karakteres térképen (`#` fal, `.` út) keress utat a bal felső saroktól a jobb alsóig visszalépéses kereséssel!
4. Írd át a Hanoi tornyai megoldást **iteratívra** egy saját veremmel! (Tipp: a verembe a „még elvégzendő feladatokat” tedd.)