Adatszerkezetek

2. hét – Az algoritmusok hatékonysága, lineáris és bináris keresés

Dr. Koczka Ferenc · Eszterházy Károly Egyetem · 2026

Ismétlés

- **Algoritmus:** véges, egyértelmű lépéssorozat
- **Adattípus:** értékhalmaz + műveletek + ábrázolás
- **Adatszerkezet:** elemek + kapcsolatok + műveletek
- Elérés: **közvetlen** (tömb) · **szekvenciális** (lista)

A mai óra kérdései

- Hogyan mondjuk meg, hogy egy algoritmus **gyors** vagy **lassú**?
- Hogyan találunk meg egy elemet egy tömbben – és lehet-e ezt **gyorsabban**?

Az algoritmusok hatékonysága

Miért nem elég stopperrel mérni?

A futási idő függ

- a géptől (processzor, memória),
- a programozási nyelvtől és a fordítótól,
- az egyéb futó programoktól,
- **a bemenet méretétől és tartalmától**.

> Ezért nem másodpercben mérünk, hanem azt vizsgáljuk, hogyan nő a **lépésszám** a bemenet **méretének (n)** függvényében.

Lépés (elemi művelet): értékadás, összehasonlítás, aritmetikai művelet, tömbelem elérése – egy lépésnek számít.

Lépésszám: egy példa

Pszeudokód – összegzés

```
Függvény Összeg(A, n)
  s ← 0                        // 1
  Ciklus i ← 1-től n-ig        // n-szer
    s ← s + A[i]               // n
  Ciklus vége
  Vissza: s                    // 1
Függvény vége
```

- Az értékadás és a visszatérés: **konstans** számú lépés
- A ciklusmag: **n**-szer fut le
- Összesen kb. $\mathbf{c_1 \cdot n + c_2}$ lépés

A konstansok (c_1, c_2) gépfüggők – a lényeg, hogy a lépésszám **lineárisan** nő n-nel.

Legjobb, legrosszabb és átlagos eset

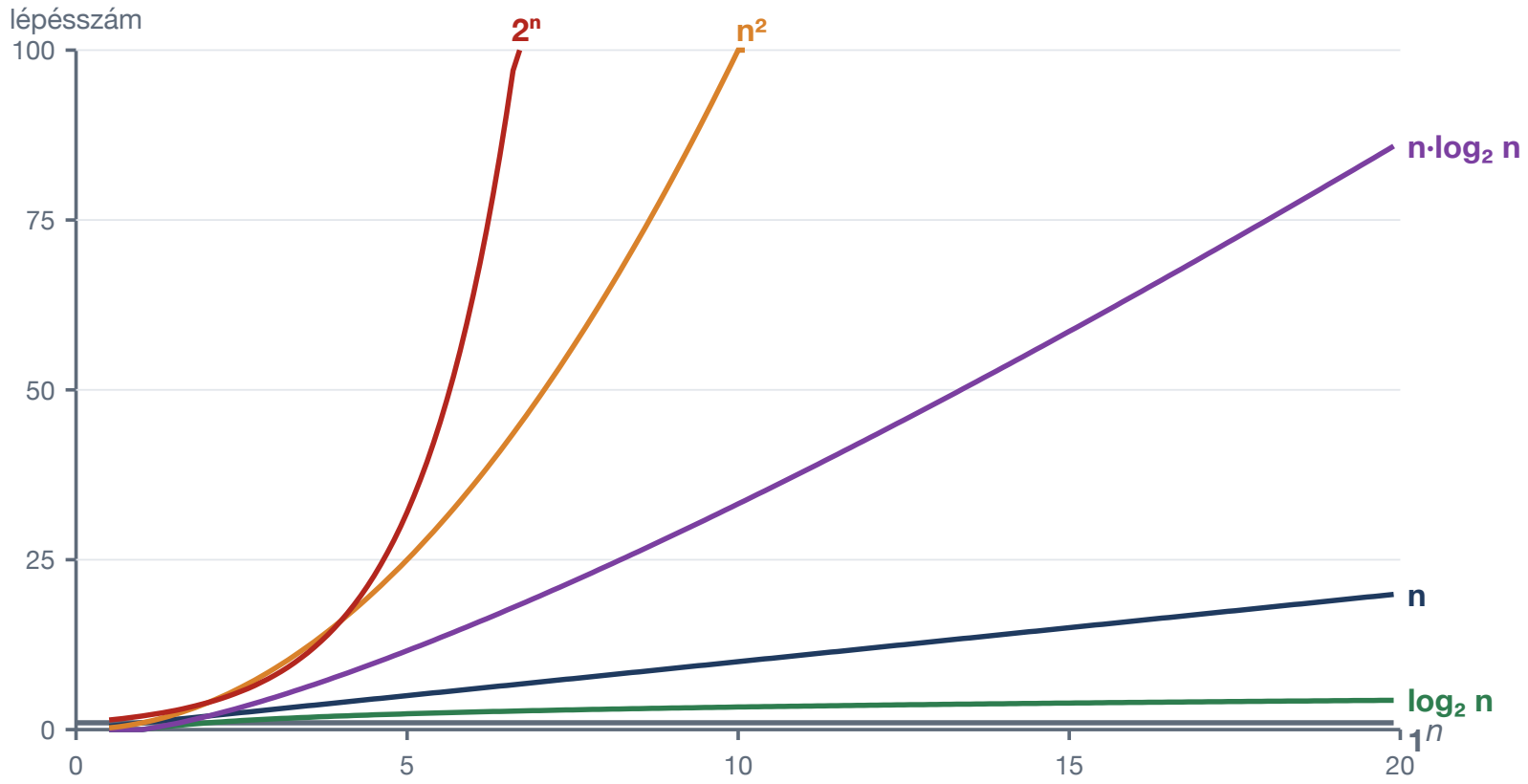
Ugyanolyan méretű bemenetre is eltérhet a lépésszám!

Példa: keressük az `x` értéket egy **n** elemű tömbben, elölről haladva.

Eset	Mikor?	Lépésszám
Legjobb	`x` az első elem	1
Legrosszabb	`x` az utolsó, vagy nincs a tömbben	n
Átlagos	`x` egyenlő eséllyel bárhol	kb. n / 2

> Általában a **legrosszabb esetet** vizsgáljuk: ez garancia – ennél lassabb biztosan nem lesz.

Növekedési ütemek



Mennyi idő ez a gyakorlatban?

Tegyük fel, hogy a gép **10^9 lépést** végez másodpercenként.

n	$\log_2 n$	n	$n \cdot \log_2 n$	n^2	2^n
10	3 ns	10 ns	33 ns	100 ns	1 µs
1 000	10 ns	1 µs	10 µs	1 ms	∞ (≈ 10^{284} év)
10^6	20 ns	1 ms	20 ms	**17 perc**	∞
10^9	30 ns	1 s	30 s	**32 év**	∞

Gyorsabb géppel ezen alig lehet segíteni: egy **10× gyorsabb** gép a 32 évből is 3,2 évet csinál. A **jobb algoritmus** a megoldás.

Az O-jelölés (nagy ordó)

Szemléletesen: `f(n) = O(g(n))`, ha **f** legfeljebb olyan gyorsan nő, mint **g** (egy konstans szorzótól eltekintve).

Pontosan: `f(n) = O(g(n))`, ha léteznek **c** > 0 és $\mathbf{n_0}$ konstansok, hogy

> $f(n) \leq c \cdot g(n)$ minden $n \geq n_0$ esetén.

Szabályok:

- A konstans szorzók elhagyhatók: `5n = O(n)`
- Csak a legnagyobb rendű tag számít: `3n² + 10n + 7 = O(n²)`
- A logaritmus alapja nem számít: `log₂ n = O(log n)`

Rokon jelölések: Ω (alsó korlát), Θ (pontos nagyságrend).

Gyakori nagyságrendek

Jelölés	Név	Példa
$O(1)$	konstans	tömbelem elérése index alapján, verem `push`
$O(\log n)$	logaritmikus	bináris keresés
$O(n)$	lineáris	lineáris keresés, összegzés, maximum
$O(n \log n)$	„n log n”	összefésülő rendezés, gyorsrendezés (átlagosan)
$O(n^2)$	négyzetes	buborékrendezés, minden pár vizsgálata
$O(2^n)$	exponenciális	minden részhalmaz vizsgálata

Ökölszabály: $n = 10^6$ elemre az $O(n \log n)$ még bőven belefér, az $O(n^2)$ már nem.

A nagyságrend meghatározása kódból

```
// O(n)
for (int i = 0; i < n; i++)
    sum += a[i];

// O(n²)
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        count++;
```

```
// O(log n) – felezés
while (n > 1)
    n = n / 2;

// O(n) + O(n²) = O(n²)
for (int i = 0; i < n; i++) ...
for (int i = 0; i < n; i++)
    for (int j = i; j < n; j++) ...
```

- **Egymás után:** összeadódnak → a nagyobbik számít
- **Egymásba ágyazva:** összeszorzódnak
- **Felezés:** ha minden lépés a felére csökkenti a feladatot → log n

Tárigény

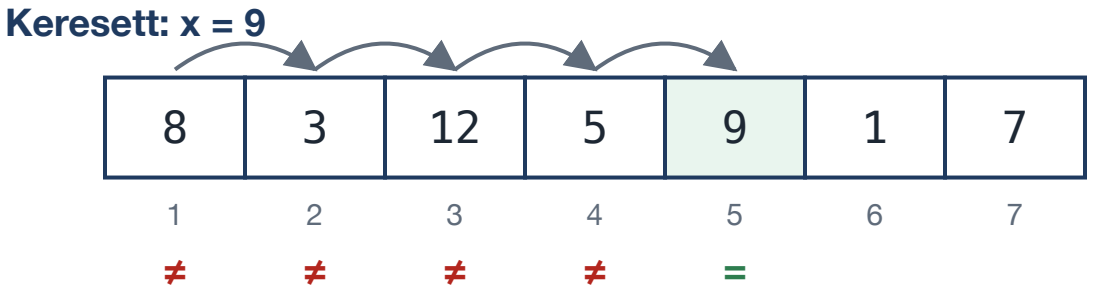
Nemcsak az idő, hanem a **memória** is erőforrás.

- **Tárigény:** a bemeneten felül szükséges memória nagysága, n függvényében
- Lineáris és bináris keresés: **O(1)** – csak néhány segédváltozó
- Egy tömb másolatának elkészítése: **O(n)**

> Gyakran **idő és tár között** kell választani: több memóriával (pl. hash tábla, 11. hét) gyorsabb lehet a keresés.

Keresés tömbben

Lineáris (szekvenciális) keresés



5 összehasonlítás

legrosszabb eset: n

- Elölről haladva minden elemet összehasonlítunk a keresett értékkel
- **Nem kell**, hogy a tömb rendezett legyen
- Működik láncolt listán is (szekvenciális elérés)

Lineáris keresés – algoritmus

Pszeudokód

```
Függvény LineárisKeresés(A, n, x)
  i ← 1
  Ciklus amíg i ≤ n és A[i] ≠ x
    i ← i + 1
  Ciklus vége
  Ha i ≤ n akkor
    Vissza: i
  különben
    Vissza: 0      // nincs benne
  Elágazás vége
Függvény vége
```

Lépésszám: legrosszabb esetben **O(n)**

Java

```java
static int linearSearch(int[] a, int x) {
    int i = 0;
    while (i < a.length && a[i] != x)
        i++;
    return i < a.length ? i : -1;
}
```

Go

```go
func linearSearch(a []int, x int) int {
    for i, v := range a {
        if v == x {
            return i
        }
    }
    return -1
}
```

Figyelem: a feltételek sorrendje!

```
Ciklus amíg i ≤ n és A[i] ≠ x
```

- Ha `i = n + 1`, akkor `A[i]` **nem létező** elem
- A **rövidzár-kiértékelés** miatt ez mégis helyes: ha `i ≤ n` hamis, `A[i] ≠ x`-et már nem értékeli ki

Fordított sorrendben (`A[i] ≠ x és i ≤ n`) Java-ban `ArrayIndexOutOfBoundsException`, Go-ban `index out of range` hibát kapunk.

Strázsás (sentinel) keresés

Ötlet: tegyük a keresett értéket a tömb végére (**n** + 1. hely) – így biztosan megtaláljuk, és nem kell az indexhatárt vizsgálni.

Pszeudokód

```
Függvény StrázsásKeresés(A, n, x)
  A[n + 1] ← x          // strázsa
  i ← 1
  Ciklus amíg A[i] ≠ x
    i ← i + 1
  Ciklus vége
  Ha i ≤ n akkor Vissza: i
  különben Vissza: 0
Függvény vége
```

- Lépésenként **egy** összehasonlítás kettő helyett
- Nagyságrendben ugyanúgy **O(n)**, de kb. kétszer kevesebb összehasonlítás
- Kell egy **plusz hely** a tömb végén

Változat	x = 4 nincs benne, n = 7
egyszerű	15 összehasonlítás
strázsás	8 összehasonlítás

A strázsa ötletét a 10. héten a strázsás listánál újra használjuk.

Keresés rendezett tömbben – lineárisan

Ha a tömb **rendezett**, a keresést abbahagyhatjuk, amint `x`-nél **nagyobb** elemhez érünk.

```
Függvény RendezettLineáris(A, n, x)
  i ← 1
  Ciklus amíg i ≤ n és A[i] < x
    i ← i + 1
  Ciklus vége
  Ha i ≤ n és A[i] = x akkor Vissza: i
  különben Vissza: 0
Függvény vége
```

- A sikertelen keresés átlagosan gyorsabb
- De a legrosszabb eset továbbra is **O(n)** – lehet-e jobb?

Bináris keresés – az ötlet

Feltétel: a tömb **rendezett**.

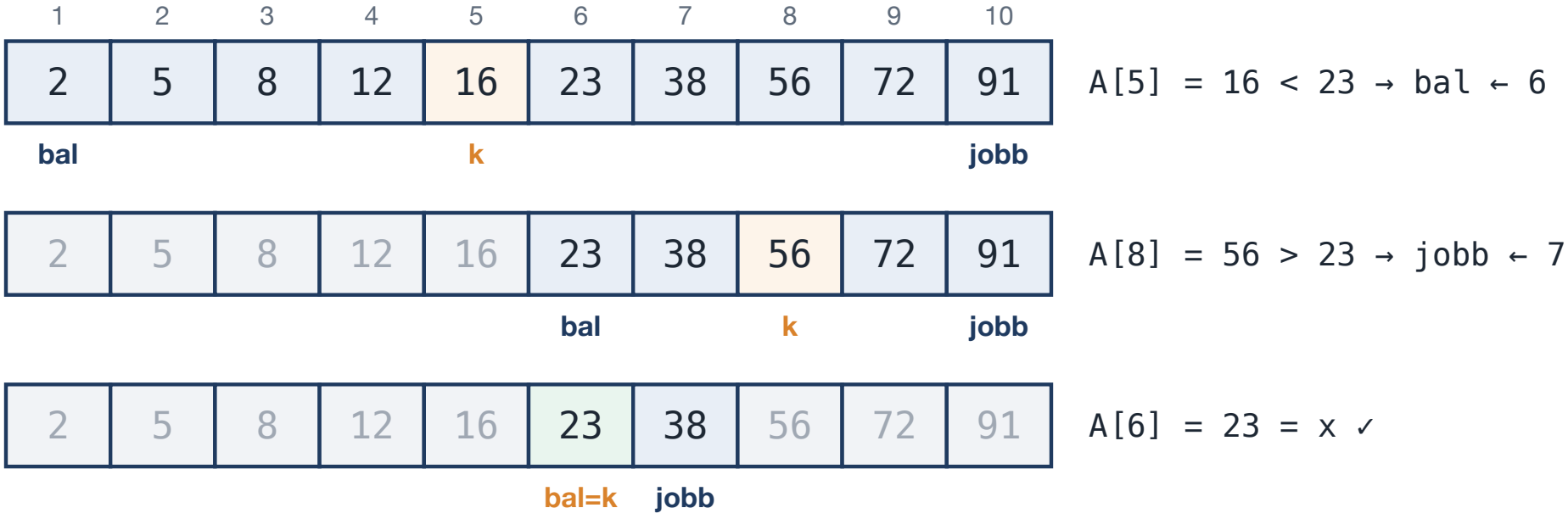
1. Nézzük meg a **középső** elemet
2. Ha ez a keresett: kész
3. Ha a keresett **kisebb**: a bal felében folytatjuk
4. Ha a keresett **nagyobb**: a jobb felében folytatjuk

Minden lépésben **feleződik** a keresési tartomány.

> Mint a szótárban (vagy telefonkönyvben) való keresés: nem lapozunk végig minden oldalt, hanem középen kinyitjuk, és eldöntjük, merre menjünk.

Bináris keresés – példa

Keresett: x = 23



Bináris keresés – algoritmus

Pszeudokód

```
Függvény BinárisKeresés(A, n, x)
  bal ← 1
  jobb ← n
  Ciklus amíg bal ≤ jobb
    k ← (bal + jobb) div 2
    Ha A[k] = x akkor
      Vissza: k
    különben ha A[k] < x akkor
      bal ← k + 1
    különben
      jobb ← k − 1
    Elágazás vége
  Ciklus vége
  Vissza: 0
Függvény vége
```

Java

```java
static int binarySearch(int[] a, int x) {
    int lo = 0, hi = a.length - 1;
    while (lo <= hi) {
        int mid = lo + (hi - lo) / 2;
        if (a[mid] == x) return mid;
        if (a[mid] < x) lo = mid + 1;
        else hi = mid - 1;
    }
    return -1;
}
```

Go

```go
for lo <= hi {
    mid := lo + (hi-lo)/2
    switch {
    case a[mid] == x: return mid
    case a[mid] < x:  lo = mid + 1
    default:          hi = mid - 1
    }
}
```

Bináris keresés – hatékonyság

Minden lépés után a tartomány mérete legfeljebb a fele: n → n/2 → n/4 → … → 1

Legfeljebb $\lfloor \log_2 n \rfloor + 1$ lépés → **O(log n)**

n	lineáris (legrosszabb)	bináris (legrosszabb)
10	10	4
1 000	1 000	10
1 000 000	1 000 000	20
1 000 000 000	1 000 000 000	30

> Kína teljes lakossága (1,4 milliárd) között egy név megtalálása rendezett listában: **legfeljebb 31 lépés**.

Bináris keresés – tipikus hibák

- **Nem rendezett** a tömb → hibás eredmény (de nem hibaüzenet!)
- `bal < jobb` a `bal ≤ jobb` helyett → az egyelemű tartományt kihagyja
- `bal ← k` a `bal ← k + 1` helyett → **végtelen ciklus** lehet
- `(bal + jobb) / 2` → nagy indexeknél **túlcsordulás**
  helyette: `bal + (jobb − bal) / 2`

> Ezt a hibát évekig tartalmazta a Java könyvtár `Arrays.binarySearch` függvénye is (javítva 2006-ban).

Beépített változatok: Java `Arrays.binarySearch(a, x)` · Go `sort.SearchInts(a, x)`

Lineáris vagy bináris?

	Lineáris	Bináris
Feltétel	nincs	**rendezett** tömb
Elérés	szekvenciális is elég	**közvetlen** elérés kell
Lépésszám	O(n)	O(log n)
Láncolt listán	✓	✗ (nincs közvetlen elérés)

> Ha sokszor keresünk ugyanabban az adathalmazban, megéri **egyszer rendezni** (O(n log n), 13. hét), utána minden keresés O(log n).

Összefoglalás

- A hatékonyságot a **lépésszám növekedésével** mérjük a bemenet méretének függvényében
- Általában a **legrosszabb esetet** vizsgáljuk
- **O-jelölés:** konstansok és kisebb rendű tagok elhagyhatók
- Nagyságrendek: $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(2^n)$
- **Lineáris keresés:** $O(n)$, bármilyen tömbön · strázsával kevesebb összehasonlítás
- **Bináris keresés:** $O(\log n)$, csak **rendezett**, közvetlen elérésű tömbön

Következő hét: tömbök – egy- és többdimenziós tömbök, tárolás, címszámítás, dinamikus tömb.

Ellenőrző kérdések

1. Mi az O-jelöléssel kifejezett nagyságrendje: $3n^2 + 100n + 5$? Hát $n + \log n$-nek?
2. Igaz vagy hamis: a bináris keresés hatékonysága O(n).
3. Igaz vagy hamis: a bináris keresés láncolt listán is O(log n) lépésben működik.
4. Hány lépésben talál meg legfeljebb egy elemet a bináris keresés egy 500 elemű tömbben?
5. Mi történik, ha bináris keresést futtatunk egy rendezetlen tömbön?
6. Mi a következő kód nagyságrendje?

```
for (i = 1; i < n; i = i * 2) for (j = 0; j < n; j++) …
```

Gyakorlat

✎ Bemutató programok

Program	Java	Go	Mit mutat?
Lineáris keresés	`LinearSearch.java`	`h02/linearsearch`	egyszerű, strázsás, rendezett változat; összehasonlítások száma
Bináris keresés	`BinarySearch.java`	`h02/binarysearch`	lépések kiírása, legrosszabb eset táblázata
Időmérés	`Timing.java`	`h02/timing`	O(n) vs. O(log n) vs. O(n^2) a valóságban

A `Timing` programot futtasd le, és figyeld meg: ha n tízszeresére nő, hogyan változik a lineáris és a bináris keresés ideje? Ha n duplázódik, hogyan változik az O(n^2) algoritmus ideje?

✎ Feladatok

1. **Maximum:** add vissza a tömb legnagyobb elemének indexét! Mi a lépésszám?
2. **Megszámlálás:** hány eleme egyenlő a tömbnek `x` -szel?
3. **Első előfordulás:** rendezett tömbben, ahol egy érték **többször** is előfordulhat, keresd meg `x` **első** előfordulását O(log n) lépésben! (`[1,2,2,2,5] , x=2 → 1`)
4. **Beszúrási hely:** add meg azt az indexet, ahová `x` -et beszúrva a tömb rendezett marad!
5. **Egész négyzetgyök:** bináris kereséssel add meg a legnagyobb **r**-et, amelyre $r^2 \leq n$!
6. **Számkitalálás:** a gép 1 és 1000 között gondol egy számot, mi „kisebb/nagyobb” válaszokat kapunk. Legfeljebb hány kérdés kell?

Megoldások: `Exercises02.java` , `h02/exercises`

✎ Házi feladat

1. Add meg a nagyságrendet (O-jelöléssel)!
   a) `for i ← 1-től n-ig: for j ← 1-től 10-ig: …`
   b) `for i ← 1-től n-ig: for j ← 1-től i-ig: …`
   c) `i ← n; Ciklus amíg i > 0: i ← i div 3`
2. Játszd le papíron a bináris keresést a `[3, 8, 15, 21, 30, 42, 57, 66, 81]` tömbön `x = 57` és `x = 10` esetén! Írd fel minden lépésben `bal`, `jobb` és `k` értékét!
3. Írj függvényt, amely egy **rendezett** tömbben megszámolja, hányszor fordul elő `x`, O(log n) lépésben!