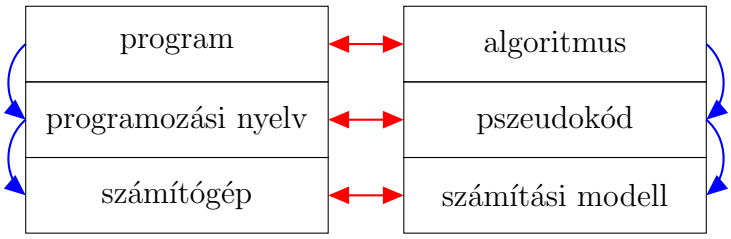
Algoritmusok és adatszerkezetek gyakorlat – 02

Algoritmusok futásidő elemzése

Algoritmus:

- Számítógépes program matematikai absztrakciója
- Egy adott problémát megoldó számítási eljárás



Egy számítási modellben specifikáljuk

- Az algoritmusban milyen műveletek megengedettek
- Minden elemi művelet költségét (idő,tár,...)
- Az algoritmus költségét, ami nálunk a műveletek összköltsége

Tehát az algoritmus olyan elemi műveletekből kompozíciós szabályok szerint felépített összetett művelet, amelyet megadott feltételt teljesítő bemeneti adatra végrehajtva, a megkívánt kimeneti adatot eredményezi.

Egy számítógépes programnak / algoritmusnak sok tulajdonságát vizsgálhatjuk gyakorlati szempontból, mint például megbízhatóság, karbantarthatóság, mennyire felhasználóbarát. Mi ezen a kurzuson - eggyel elméletibb szempontból - az algoritmusok helyességével, idő- és tárigényével foglalkozunk.

Hogyan vizsgálunk időigényt?

- Az időigény függ az inputtól: például egy már rendezett tömböt könnyebb lehet rendezni, mint egy nem rendezettet.
- Függ az input hosszától: egy nagy tömbben hosszabb ideig tart egy elem megkeresése, mint egy rövidebben.
- Általában jobban szeretünk egy garantált **felső korlát**ot mondani az időigényre.

Időigény (legrosszabb eset)

Egy algoritmus időigénye $T(n)$, ha az algoritmus tetszőleges n méretű inputon $T(n)$ időben megáll.

Az időigény analízis további fajtái: előfordul, hogy nem a fenti legrosszabbeset-korlátot akarjuk becsülni, mert az adott esetben indokolatlanul pesszimista (pl a gyakorlatban előforduló problémákra jellemzően gyorsabb). Használjuk még:

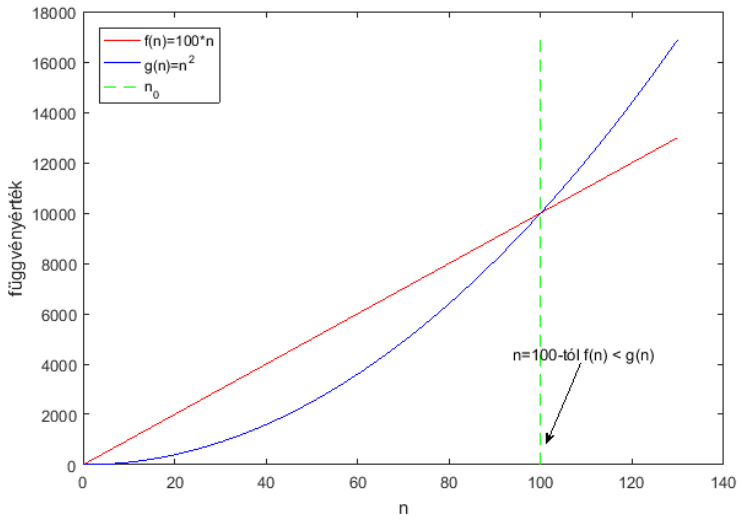
- **Átlagos eset:** $T(n)$ = az algoritmus várható időigénye az összes lehetséges n méretű bemeneten.
- **Legjobb eset:** olyan cheat, ahol az amúgy lassú algoritmus néhány speciális inputra mégis gyors

Hogyan számítjuk ki egy algoritmus legrosszabb eset időigényét? Nem akarjuk kiszámolni precízen a függvényt. (Pl így: $T(n) = 3n^4 + 7n^2 + 2n + 5024$, mert általában ennyire pontosan nem is lényeges.)

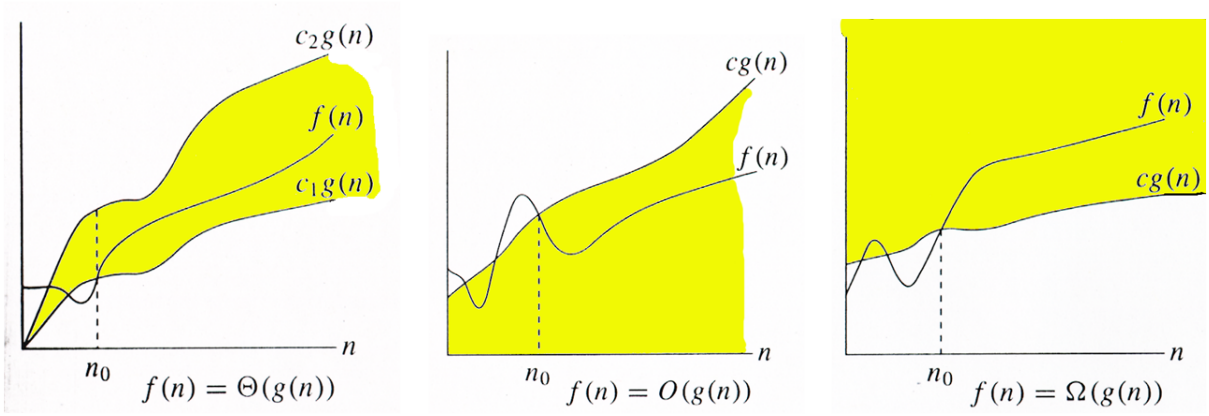
NAGY ÖTLET: Nézzük csak azt, hogy milyen gyorsan nő a $T(n)$, ha $n \to \infty$.

Futásidőt és tárigényt leíró f függvényekkel fogunk számolni, ezért hallgatólagosan feltesszük, hogy ezek a függvények monoton növőek lesznek („nagyobb inputra tovább fut és több tárat használ”)

A függvények közötti kapcsolatok leírására az O (e. „Ordó”), o („ordó”, „kisordó”), Θ („Théta”), Ω („Omega”) és ω („omega”, „kisomega”) jelöléseket fogjuk használni. Ezek kb. a $\leq$, $<$, $=$, $\geq$, $>$ kapcsolatok lesznek (de nem úgy, hogy „minden n-re”, hanem „egy idő után, egy küszöbszám után minden n-re”). Erre láthatunk példát az 1. Ábrán.



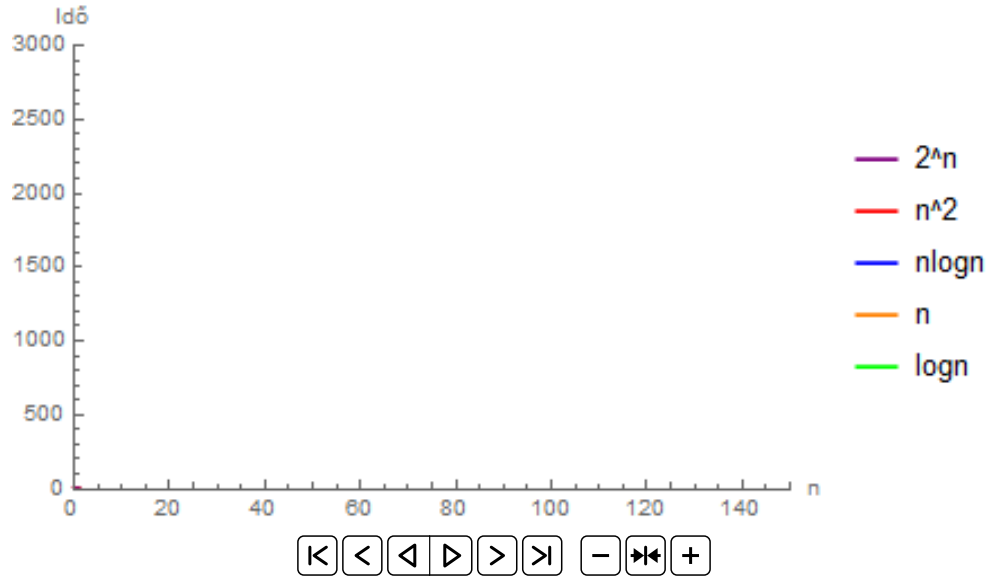
1. Ábra.: Példa $f(n) = O(g(n))$-re (már $c = 1$-re is)



2. Ábra.: Példák Θ-ra, O-ra és Ω-ra (forrás).

Milyen függvényekkel találkozhatunk, és melyik milyen gyorsan nő?

- logaritmikus: $T(n) = \log n$
- lineáris: $T(n) = n$
- polinom: pl $T(n) = n^2$, $T(n) = n^3$, de ide számítjuk a $T(n) = n \log n$-t is, hiszen ez nagyobb, mint a lineáris, de n^2-nél kisebb
  (Az eddig felsorolt időigényű algoritmusokat mondjuk a gyakorlatban megoldhatóknak)
- exponenciális: $T(n) = 2^n$
- faktoriális: $T(n) = n!$



1. Feladat Adjuk meg a következő algoritmus legrosszabbeset-analízis szerinti időigényét!

Szekvenciális keresés

- **Input**: Egy n elemet tartalmazó rendezett tömb és annak egy eleme
- **Output**: Benne van-e az elem a tömbben és ha igen, hanyadik indexen?
- **Algoritmus:**

```
public static int search(int key, int[] a) {
        for (int i = 0; i < a.length; i++)
                if ( a[i] == key ) return i;
        return -1;
}
```

Megoldás

Az algoritmus végigszkenneli a tömböt a 0. indextől egészen addig, amíg meg nem találja a keresett elemet vagy ha az nincs benne, végigmegy az egész tömbön, majd -1-gyel tér vissza.

Az elemi utasítások (itt például az összehasonlítás) költsége konstans, tehát $O(1)$. Így csak azt kell megszámoljuk, hogy a legrosszabb input esetén hányszor fut le a ciklus, mivel a ciklusmag költsége $O(1)$, így az algoritmus teljes költsége annyi, ahányszor ez a ciklusmag lefut.

A legrosszabb eset az, ha az elem nem szerepel a tömbben (vagy ha az utolsó helyen szerepel), mert ekkor a teljes tömböt be kell járni a 0. indextől az $n-1$. indexig, azaz összesen n-szer fut le a ciklus. Az algoritmus időigénye tehát $O(n)$.

Kérdés: Van-e ennél jobb?

Ötlet: Használjuk ki, hogy rendezett a tömb!

2. Feladat Adjunk egy olyan algoritmust, ami egy n elemű rendezett tömbön gyorsabb, mint a szekvenciális keresés! Mennyi ennek az eljárásnak az időigénye?

Megoldás

Ha egy tömb rendezett, akkor bármely elemét véve tudjuk, hogy előtte a nála $\leq$, mögötte a nála $\geq$ elemek szerepelnek. Emiatt tetszőleges elemmel összehasonlítva a keresett elemet tudjuk, hogy ,,merre" kell tovább keresni.

Hogyan tudjuk a legkevesebb ,,merre" összehasonlítást csinálni?

Ötlet: Mindig felezzük el a tömböt! Tehát nézzük meg a középső ($j = \lfloor n/2 \rfloor$) elemet. Ha az $a[j] \leq key$, akkor vizsgáljuk ugyanígy tovább az $a[0 \dots j-1]$ intervallumot, ha pedig nagyobb, vizsgáljuk meg az $a[j+1 \dots n-1]$ intervallumot.

Tehát az algoritmus a következő:

```
public static int search( int x, int arr[]){
  int l = 0, r = arr.length - 1;
  while (l <= r){
    int m = l + (r-l)/2;
    if (arr[m] == x) return m;
    if (arr[m] < x) l = m + 1;
    else
      r = m - 1;
  }
  return -1;
}
```

Időigény:

Hányszor tudunk összesen elfelezni egy n elemű tömböt? (Azaz hányszor tudjuk elosztani az n-et kettővel?)

$$n \to n/2 \to n/4 \to n/8 \to \dots \to 1$$

$$1 \to 1$$
$$2 \to 1$$
$$4 \to 2 \to 1$$
$$8 \to 4 \to 2 \to 1$$
$$16 \to 8 \to 4 \to 2 \to 1$$
$$32 \to 16 \to 8 \to 4 \to 2 \to 1$$
$$64 \to 32 \to 16 \to 8 \to 4 \to 2 \to 1$$
$$128 \to 64 \to 32 \to 16 \to 8 \to 4 \to 2 \to 1$$
$$256 \to 128 \to 64 \to 32 \to 16 \to 8 \to 4 \to 2 \to 1$$
$$512 \to 256 \to 128 \to 64 \to 32 \to 16 \to 8 \to 4 \to 2 \to 1$$

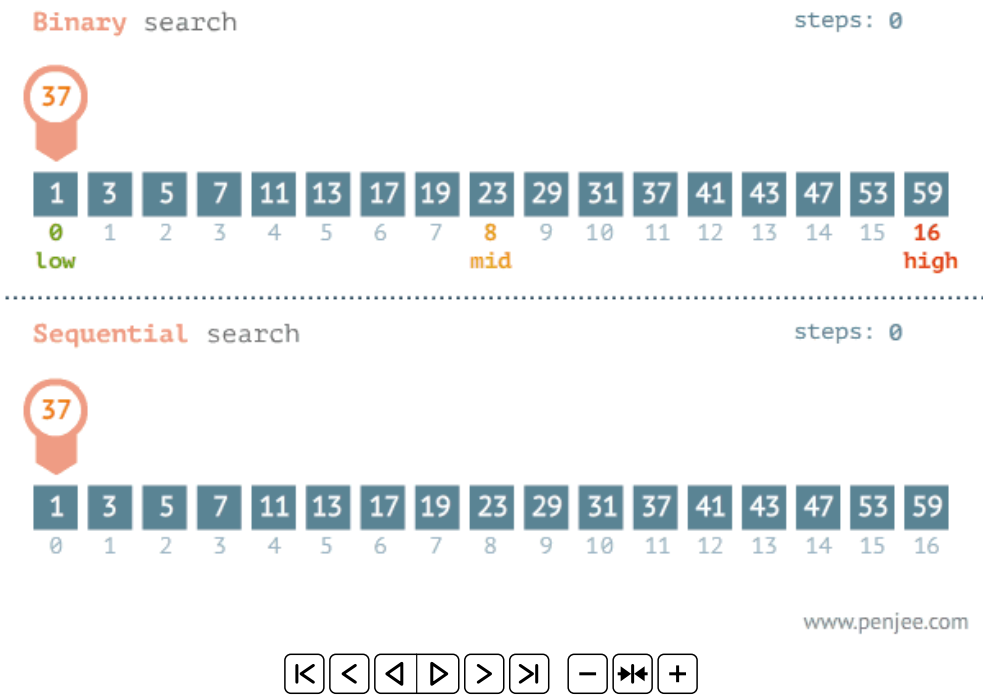
Mi az a függvény, ami a fenti helyettesítési értékekre visszaadja azt, hányszor osztható az adott szám kettővel (vagy annak az alsó egész részét)?

n	1	2	4	8	16	32	64	128	256	512
$\log_2 n$	0	1	2	3	4	5	6	7	8	9

A válasz tehát $O(\log_2 n)$.

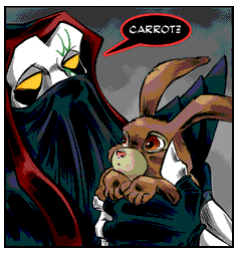
A továbbiakban ebből a tárgyból a logaritmust mindig kettes alapúnak tekintjük.

Nézzünk egy példát a két algoritmus futására: Az alábbi 17 elemű tömbben keressük a 37-es számot. Figyeljük meg, melyik hány lépésben jut el a keresett elemig! Ehhez játsszuk le a 3. Ábra animációját.



3. Ábra.: Bináris vs szekvenciális keresés. (forrás)

Érdekesség: Mi a helyzet akkor, ha k db n elemű rendezett tömbben szeretnék ugyanazt az elemet megkeresni? A triviális megoldás $O(k \cdot \log n)$ időigényű. Van-e ennél jobb?



Van: Fractional cascading módszerrel elérhető az $O(k + \log n)$ időigény. Ha bővebben érdekel (nem lesz zh-n), kattints a nyuszira.