Complexitatea algoritmilor

In practica, eficienta algoritmilor este foarte importanta. Ca programatori, noi vom cauta intotdeauna sa implementam algoritmii intr-o maniera cat mai eficienta.

Vom vrea sa comparam algoritmii in functie de eficienta lor…dar cum definim aceasta eficienta?

Definim eficienta unui algoritm uitandu-ne la:

1. **timpul** de care are nevoie pentru a rezolva problema
2. **spatiul de memorie** folosit pentru stocarea datelor

Vrem ca **timpul** sa fie cat mai **scurt**, iar **spatiul de memorie** folosit, cat mai **mic**.

Complexitate de timp

Timpul de rulare (complexitatea de timp) al algoritmului depinde de numarul de operatii realizate.
Numarul de operatii poate sa difere in functie de datele de intrare. Daca ne gandim de exemplu la algoritmii de cautare studiati (cautare secventiala si cautare binara), pentru aceleasi date de intrare: 10, 20, 30, 40, 50, 60, 70 cautarea numarului 40 poate fi realizata de cautarea secventiala prin efectuarea a 4 comparatii, in schimb cautarea binara va efectua doar 1. Pe de alta parte, pentru cautarea numarului 10 cautarea binara va efectua mai multe comparatii, 3, in timp ce cautarea secventiala va avea nevoie doar de 1.
Daca dimensiunea datelor de intrare creste semnificativ (asa cum se intampla in aplicatiile din viata reala) aceste diferente devin din ce in ce mai mari. De multe ori vom face compromisuri de genul acesta: “Pentru ca algoritmul de cautare binara face, in general, mai putine operatii decat cel de cautare secventiala cand vine vorba de un volum mare de date, il vom alege pe el. Da, este adevarat ca pot exista cazuri in care volumul de date de intrare este mic, sau elementul cautat este chiar primul din secventa (si cautarea secventiala l-ar gasi facand doar o singura operatie), dar cautarea binara este mai eficienta pentru cazul general.”

Cum calculam timpul de rulare al unui algoritm?
Putem calcula timpul de rulare cronometrand numarul de secunde, dar aceasta metoda nu e una de incredere. Pentru ca acelasi algoritm poate dura mai mult sau mai putin in functie de echipamentul nostru hardware (cat de performant e laptop-ul de pe care il rulam) sau de limbajul de programare in care algoritmul este scris.

Cea mai buna solutie este **analiza asimptotica**, o analiza matematica care estimeaza eficienta unui algoritm in functie de posibilele date de intrare, si felul in care ea se modifica pe masura ce volumul de date creste. Astfel, in loc sa numaram secunde, vom studia timpul de rulare al unui algoritm in raport cu volumul de date.

Pentru un algoritm A si o secventa de date de intrare I, vom nota cu E(I) numarul de operatii efectuate asupra secventei I si cu P(I) probabilitatea de a avea secventa I drept secventa de intrare.

Pentru analiza asimptotica, distingem 3 cazuri:

1. **cel mai bun caz (best case sau BC)** - cazul in care algoritmului ii ia cel mai putin ca sa termine rularea pentru ca datele de intrare sunt prielnice (pentru cautarea binara, cel mai bun caz este cazul in care secventa de intrare are elementul cautat chiar pe pozitia din mijloc - ne amintim, atunci se excuta o singura comparatie)

   BC(A) = min(E(I))

2. **cel mai rau caz (worst case sau WC)** - cazul in care algoritmului ii ia cel mai mult ca sa termine rularea (pentru cautarea binara, cel mai rau caz este, in mod paradoxal, cazul in care elementul cautat se afla chiar la inceputul (sau la sfarsitul) secventei. De asemenea, cel mai rau caz pentru orice algoritm de cautare este acela in care elementul cautat nu se afla in secventa si trebuie sa efectuam toti pasii pentru a deduce asta.

   WC(A) = max(E(I))

3. **cazul mediu (average case sau AC)** - facem media numarului de pasi executati pentru toate cazurile (atunci cand asa ceva este posibil)

   $$AC(A) = \sum_{I \in Domeniu} P(I)E(I)$$

   Ex.: Cazul mediu pentru cautarea secventiala pentru o lista cu n elemente.
   Pentru fiecare posibila pozitie a valorii cautate 10 in secventa de intrare:
   I1: 10, .., .., .., .., … E(I1) = 1 comparatie
   I2: .., 10, .., .., .., … E(I2) = 2 comparatii
   I3: .., .., 10, .., .., … E(I3) = 3 comparatii
   … …
   In: …, .., .., .., .., 10 E(In) = n comparatii
   Pentru fiecare secventa de intrare: I1, …, In exista aceeasi probabilitate ca ea sa fie data: P(I1) = P(I2) = … = P(In) = $\frac{1}{n}$
   Deci cazul mediu:

   $$AC(A) = \sum_{I \in Domeniu} P(I)E(I) = \frac{1}{n} + \frac{2}{n} + \frac{3}{n} + \ldots + \frac{n}{n} = \frac{1 + 2 + 3 + \ldots + n}{n} = \frac{n(n+1)}{2n}$$

   $$AC(A) = \frac{n+1}{2}$$

Trebuie sa fim atenti cand alegem modul in care implementam algoritmii si sa tinem cont de existenta acestor cazuri, deoarece cel mai rau caz, sau cazuri la fel de rele, se pot produce des. De asemenea, de cele mai multe ori cazul mediu este la fel ca cel mai rau caz. Deci faptul de a fi optimisti si a presupune ca cel mai rau caz nu se va intampla (sau ca se va intampla foarte rar) nu functioneaza.

Putem estima timpul de rulare al unui algoritm folosind o functie *T(N)* care asociaza unei dimensiuni *N* a datelor de intrare, un numar de pasi executati de algoritm.
Asadar, atunci cand calculam complexitatea de timp a unui algoritm ne concentram pe modul in care timpul de rulare creste in raport cu volumul datelor de intrare. De cele mai multe ori nu e posibil, si nici necesar sa determinam aceasta complexitate exact. Mai degraba, ea va fi aproximata folosind *ordine de magnitudine* prin care este descris estimativ numarul de pasi necesari rularii algoritmului.

Cele mai des intalnite ordine de magnitudine sunt $log_2 N$, N, $Nlog_2 N$, N^2, N^3, dar exista si ordine de magnituine mai putin practice precum 2^N, N^N sau $N!$.

In practica, volumul datelor de intrare este foarte mare, deci complexitatea de timp calculata pentru o secventa de *N* numere poate fi privita ca si cum $N \to \infty$.
De aceea, complexitatea in cazul mediu pentru cautarea secventiala atunci cand $N \to \infty$ este: $T(N) = \frac{N+1}{2} \approx N$, unde *T(N)* este functia care asociaza dimensiunii *N* a datelor de intrare, un numar de pasi.

Dar ce se intampla daca $T(N) = 10log_2 N + 40N + 33\ Nlog_2 N + 12\ N^2 + 49N^3$?

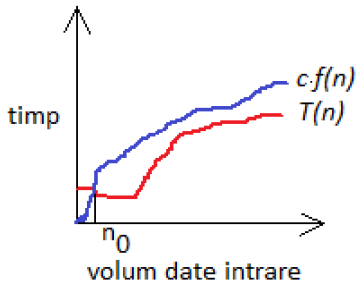
Cum estimam *T(N)* in acest caz?

Deoarece presupunem ca $N \to \infty$ ne intereseaza numai termenul dominant al ecuatiei: N^3. Observam ca am ignorat complet constanta 49.
Daca sunt relativ mici, astfel de constante pot fi ignorate deoarece pe masura ce valoarea lui *N* creste (aici $N \to \infty$), timpul de executie devine din ce in ce mai putin dependent de acea constanta si din ce in ce mai dependent de *N*.
Deci putem spune ca $T(N) = 10log_2 N + 40N + 33\ Nlog_2 N + 12\ N^2 + 49N^3 \approx N^3$.

Aceasta gandire poate fi formalizata prin introducerea unor definitii.

Definim functiile $f : N \to \Re$ si $T : N \to N$, unde functia T care da timpul de rulare al algoritmului.

1. **Notatia *O* ("Big-oh")**
   Notatia *O* ofera o limita superioara a timpului de rulare. Este folosita pentru a evalua cel mai rau caz.



   $T(n) \in O(f(n))$ daca $\exists$ doua constante pozitive c si n_0 independente de n astfel incat

$$0 \le T(n) \le c \cdot f(n), \qquad \forall n \ge n_0$$

sau

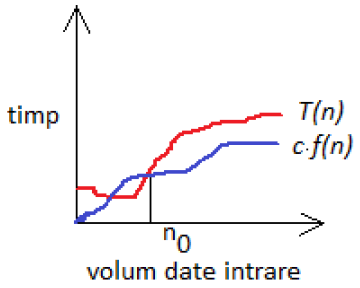
   $T(n) \in O(f(n))$ daca $\lim_{n \to \infty} \frac{T(n)}{f(n)}$ este 0 sau constanta, dar nu ∞.

   Pentru exemplul nostru cu $T(n) = 10log_2 n + 40n + 33\, nlog_2 n + 12\, n^2 + 49n^3$

   $\lim_{n \to \infty} \frac{T(n)}{n^3} = 49$, deci $T(n) \in O(n^3)$.

   Dar daca $T(n) \in O(n^3)$, inseamna ca $T(n) \in O(n^4)$, $T(n) \in O(n^5)$, deci nu e de ajuns sa definim limita superioara pentru timpul de rulare pentru a il estima corespunzator. Vom avea nevoie sa definim si o limita inferioara (Ω).

2. **Notatia Ω ("Big-omega")**
   Notatia Ω ofera o limita inferioara a timpului de rulare. Este folosita pentru a evalua cel mai bun caz.



   $T(n) \in \Omega(f(n))$ daca $\exists$ doua constante pozitive c si n_0 independente de n astfel incat

$$0 \le c \cdot f(n) \le T(n), \qquad \forall n \ge n_0$$

sau

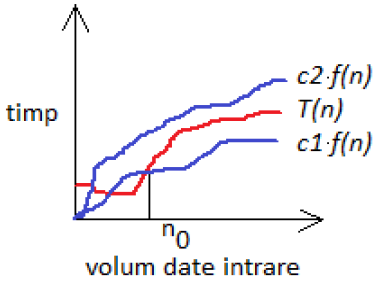
   $T(n) \in \Omega(f(n))$ daca $\lim_{n \to \infty} \frac{T(n)}{f(n)}$ este ∞ sau constanta, dar nu 0.

Pentru exemplul nostru cu $T(n) = 10log_2 n + 40n + 33\ nlog_2 n + 12\ n^2 + 49n^3$

$\lim_{n \to \infty} \frac{T(n)}{n^3} = 49$, deci $T(n) \in \Omega(n^3)$.

3. **Notatia Θ ("Big-theta")**

Notatia Θ ofera o limita cleste a timpului de rulare.



$T(n) \in \Theta(f(n))$ daca $T(n) \in O(f(n))$ si $T(n) \in \Omega(f(n))$, adica ∃ trei constante pozitive $c1$, c2 si n_0 independente de n astfel incat

$$c1 \cdot f(n) \leq T(n) \leq c2 \cdot f(n), \qquad \forall n \geq n_0$$

sau

$T(n) \in \Theta(f(n))$ daca $\lim_{n \to \infty} \frac{T(n)}{f(n)}$ este constanta, dar nu 0 sau ∞.

Pentru exemplul nostru cu $T(n) = 10log_2 n + 40n + 33\ nlog_2 n + 12\ n^2 + 49n^3$

$\lim_{n \to \infty} \frac{T(n)}{n^3} = 49$, deci $T(n) \in \Theta(n^3)$.

4. **Notatia *o* ("Little-oh")**

$T(n) \in o(f(n))$ daca ∀ c constanta pozitiva ∃ constanta pozitiva n_0 independenta de n astfel incat

$$0 \leq T(n) \leq c \cdot f(n), \qquad \forall n \geq n_0$$

Asta inseamna ca $f(n)$ creste mult mai repede decat $T(n)$. $log_2 n \in o(n)$

5. **Notatia ω ("Little-omega")**

$T(n) \in \omega(f(n))$ daca ∀ c constanta pozitiva ∃ constanta pozitiva n_0 independenta de n astfel incat

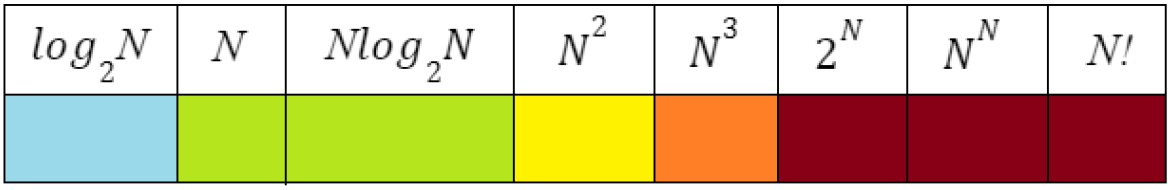
$$0 \leq c \cdot f(n) \leq T(n), \qquad \forall n \geq n_0$$

Asta inseamna ca $T(n)$ creste mult mai repede decat $f(n)$. $n \in \omega(log_2 n)$

Cu alte cuvinte:

1. $T(n) \in O(f(n)) \simeq T \leq f$
2. $T(n) \in \Omega(f(n)) \simeq T \geq f$
3. $T(n) \in \Theta(f(n)) \simeq T = f$
4. $T(n) \in o(f(n)) \simeq T < f$
5. $T(n) \in \omega(f(n)) \simeq T > f$

Ordine de magnitudine pentru complexitati:



Exemple ordine de magnitudine:

1. **for** (sau **while**) **DUPA for** (sau **while**) (unde **for/while** fac N pasi):

```
for i in range(0, N):
        …

for j in range(0, N):
        …
```

```
for i in range(0, N):
        …

i = 0
while i < N:
        …
        i = i + 1
```

In ambele cazuri: T(N) = N + N ≈ N

2. **for** (sau **while**) **IN for** (sau **while**) (unde **for/while** fac N pasi):

```
for i in range(0, N):
        for j in range(0, N):
                …
        …
```

```
i = 0
while i < N:
        for j in range(0, N):
                …
        …
        i = i + 1
```

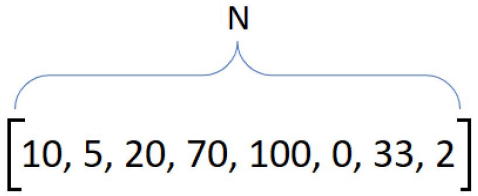
In ambele cazuri: T(N) = N · N ≈ N^2

Complexitate de spatiu

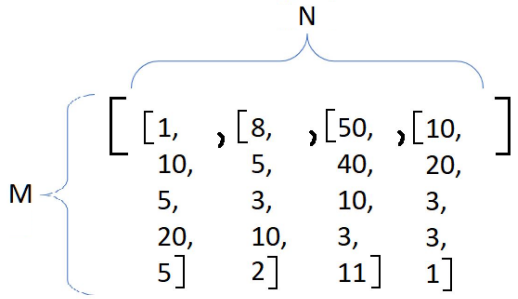
Complexitatea de spatiu se refera la spatiul de memorie necesar stocarii datelor aditionale. Dorim ca resursele alocate stocarii datelor aditionale sa fie cat mai putine.
Aproximarile folosind O, Ω, Θ, o, ω definite in cadrul complexitatii de timp se aplica si pentru complexitatea de spatiu.

In algoritmii nostri vom intalni cel mai adesea complexitati de spatiu avand doua ordine de magnitudine: N si N^2(sau $N \cdot M$)

Daca pentru algoritmul pe care vrem sa il evaluam este necesar sa folosim in plus fata de datele de intrare o lista de N elemente, complexitatea de spatiu va fi de ordin N.



Daca pentru algoritmul pe care vrem sa il evaluam este necesar sa folosim in plus fata de datele de intrare o lista de N elemente in care fiecare element este tot o lista, continand M elemente, complexitatea de spatiu va fi de ordin N^2(sau $N \cdot M$).



Complexitatea algoritmilor studiati de noi

Complexitate de timp:

	Best Case	Average Case	Worst Case
cautare secventiala	$\Omega(1)$	$\Theta(N)$	$O(N)$
cautare binara	$\Omega(1)$	$\Theta(log_2 N)$	$O(log_2 N)$
interclasare	$\Omega(N)$	$\Theta(N)$	$O(N)$
bubble sort	$\Omega(N)$	$\Theta(N^2)$	$O(N^2)$
insertion sort	$\Omega(N)$	$\Theta(N^2)$	$O(N^2)$
selection sort	$\Omega(N^2)$	$\Theta(N^2)$	$O(N^2)$
merge sort	$\Omega(Nlog_2 N)$	$\Theta(Nlog_2 N)$	$O(Nlog_2 N)$
quick sort	$\Omega(Nlog_2 N)$	$\Theta(Nlog_2 N)$	$O(N^2)$

Complexitate de spatiu:

$O(N)$, $\Omega(N)$, $\Theta(N)$ - interclasare si sortare prin interclasare (merge_sort)
$O(N)$ - backtracking
$O(log_2 N)$ - quick_sort

$O(1)$, $\Omega(1)$, $\Theta(1)$ - cautare secventiala, cautare binara
- bubble_sort, insertion_sort, selection_sort

Dupa: https://www.cs.ubbcluj.ro/~gabis/fp/Documentatii/Complexitati.pdf