Complexitatea algoritmilor

Ce inseamna si de ce depinde complexitatea

- un algoritm performant consuma cat mai putine resurse
- resursele cele mai importante: timpul si spatiul (memoria) => vom vorbi despre complexitate **temporala** si **spatiala**
- cantitatea de resurse folosita depinde de:
  - datele de intrare (ex: in general consumi mai putin sa sortezi un vector gata sortat decat unul “bine amestecat”)
  - dimensiunea datelor de intrare (ex: consumi mai mult sortand un vector de 1000 de elemente fata de unul de 3 elemente) => complexitatea va fi functie de dimensiunea datelor de intrare (notata tipic **T(n)**)
- masurarea experimentala a timpului consumat de algoritm introduce dependenta de hardware/software (conteaza masina pe care rulezi, limbajul in care ai programat – lucruri care nu tin de calitatea intrinseca a algoritmului) => vom prefera o **estimare matematica a numarului de pasi** parcursi de algoritm
- numarul exact de pasi este dificil de calculat, se impun o serie de simplificari
- in calculul de complexitate tinem cont numai de **operatiile critice** din algoritm (acele operatii care prin natura lor sunt f consumatoare, sau prin faptul ca sunt efectuate de un numar semnificativ de ori) (ex: intr-un algoritm de sortare o operatie critica este comparatia de elemente, intrucat se produce de f multe ori)
- in continuare este dificil de calculat un numar de pasi, ceea ce va interesa este “cam de ce ordin” este functia de complexitate (in ce **clasa de complexitate**), sau - altfel spus - **cum creste functia de complexitate** (cat de repede? creste liniar, creste logaritmic, creste exponential? etc); astfel deducem cum se va comporta algoritmul pe dimensiuni mari ale datelor de intrare, acolo unde diferentele se simt cel mai acut
- aceasta idee (spectaculoasa) sta la baza **analizei asimptotice** de complexitate, bazata pe notatiile asimptotice de complexitate pe care le vom introduce intr-un paragraf urmator
- in “viata reala” problemele au anumite particularitati care ne pot conduce sa alegem/modificam un algoritm mai putin performant dpdv matematic, insa mai performant pt necesitatile problemei in cauza (ex: daca un algoritm incepe sa fie mai rapid de la n=1000000 incolo iar problema noastra reala nu ajunge niciodata la un n atat de mare, alegem un algoritm care (doar) in teorie e mai putin performant); orice analiza teoretica trebuie dublata de un bun simt practic

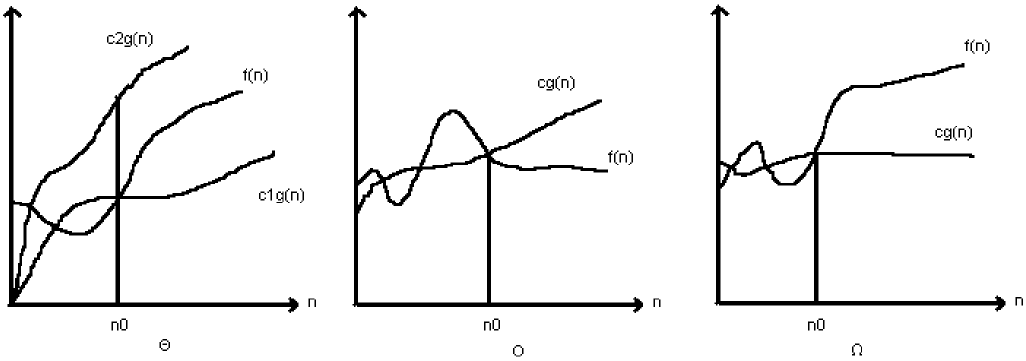
Tipuri de analiza de complexitate

- **cazul cel mai defavorabil** (= ce complexitate rezulta pt cel mai neprietenos input) (analiza cea mai frecventa, e usor de realizat si ofera utilizatorului o garantie: algoritmul nu se va purta niciodata mai rau decat atat)
- **cazul mediu** (= la ce complexitate ne putem astepta in cazul inputurilor aleatoare) (o analiza utila insa dificil de realizat, necesita cunoasterea unei distributii statistice a posibilelor inputuri, pt a pondera pe fiecare cu probabilitatea sa de aparitie si a calcula apoi o astfel de medie ponderata a complexitatilor)

- **cazul cel mai favorabil** (= ce complexitate rezulta pt cel mai prietenos input) (analiza cea mai putin frecventa, utila cel mult pt probleme care tind sa aiba inputuri favorabile; in plus, este usor de trisat, prin plasarea unui test pt un input anume la inceputul algoritmului, caz in care se da direct rezultatul, cu minim de efort)

Notatiile asimptotice de complexitate (O, Ω, Θ, o, ω)

- functiile de complexitate sunt functii asimptotic crescatoare de tip **N->R**$_+$ (in figura, f(n) reprezinta nr de pasi efectuat de algoritm pt o intrare de dimensiune n)



O(g(n)) = { $f: N \to R_+ \mid \exists$ constantele $c \in R_+$, $c>0$, si $n_0 \in N$ a.i. pt $\forall\ n \geq n_0$, $0 \leq f(n) \leq cg(n)$ }
Ω(g(n)) = { $f: N \to R_+ \mid \exists$ constantele $c \in R_+$, $c>0$, si $n_0 \in N$ a.i. pt $\forall\ n \geq n_0$, $0 \leq cg(n) \leq f(n)$ }
Θ(g(n)) = { $f: N \to R_+ \mid \exists$ constantele $c_1, c_2 \in R_+$, $c_1>0$, $c_2>0$, si $n_0 \in N$ a.i. pt $\forall\ n \geq n_0$,
$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ }

$f(n) \in O(g(n))$ inseamna ca, pentru valori mari ale dimensiunii intrarii, cg(n) este o **limita superioara** pentru f(n); algoritmul se va purta mereu mai bine decat aceasta limita.
$f(n) \in \Omega(g(n))$ inseamna ca, pentru valori mari ale dimensiunii intrarii, cg(n) este o **limita inferioara** pentru f(n); algoritmul se va purta mereu mai rau decat aceasta limita.
$f(n) \in \Theta(g(n))$ inseamna ca, pentru valori mari ale dimensiunii intrarii, $c_1 g(n)$ este o limita inferioara pentru f(n), iar $c_2 g(n)$ o limita superioara; algoritmul tinde sa se comporte “cam ca” g(n). Pt analize de complexitate cat mai precise, preferam notatia Θ.

o(g(n)) = { $f: N \to R_+ \mid \forall c \in R_+$, $c>0$, $\exists n(c) \in N$ a.i. pt $\forall\ n \geq n(c)$, $0 \leq f(n) < cg(n)$ }
ω(g(n)) = { $f: N \to R_+ \mid \forall c \in R_+$, $c>0$, $\exists\ n(c) \in N$ a.i. pt $\forall\ n \geq n(c)$, $0 \leq cg(n) < f(n)$ }

$f(n) \in o(g(n))$ inseamna ca g(n) creste **strict** mai repede decat f(n).
$f(n) \in \omega(g(n))$ inseamna ca f(n) creste **strict** mai repede decat g(n).
In alta exprimare, $f(n) \in o(g(n)) \Leftrightarrow f(n) \in O(g(n)) \wedge \text{not}(f(n) \in \Theta(g(n)))$.
Similar, $f(n) \in \omega(g(n)) \Leftrightarrow f(n) \in \Omega(g(n)) \wedge \text{not}(f(n) \in \Theta(g(n)))$.

Exercitiu: Sa se redefineasca notatiile asimptotice de complexitate folosind limite de functii.

Proprietati ale notatiilor asimptotice de complexitate
Tranzitivitatea
$f(n)\in O(h(n)) \wedge h(n)\in O(g(n)) \Rightarrow f(n)\in O(g(n))$
$f(n)\in \Omega(h(n)) \wedge h(n)\in \Omega(g(n)) \Rightarrow f(n)\in \Omega(g(n))$
$f(n)\in \Theta(h(n)) \wedge h(n)\in \Theta(g(n)) \Rightarrow f(n)\in \Theta(g(n))$
$f(n)\in o(h(n)) \wedge h(n)\in o(g(n)) \Rightarrow f(n)\in o(g(n))$
$f(n)\in \omega(h(n)) \wedge h(n)\in \omega(g(n)) \Rightarrow f(n)\in \omega(g(n))$
Reflexivitatea
$f(n)\in O(f(n))$
$f(n)\in \Omega(f(n))$
$f(n)\in \Theta(f(n))$
Simetria
$f(n)\in \Theta(g(n)) \Leftrightarrow g(n)\in \Theta(f(n))$
Antisimetria
$f(n)\in O(g(n)) \Leftrightarrow g(n)\in \Omega(f(n))$
Altele
$f(n)\in \Theta(g(n)) \Leftrightarrow f(n)\in O(g(n)) \wedge f(n)\in \Omega(g(n))$

Exercitii:
1) Demonstrati proprietatile notatiilor de complexitate.
2) $\Theta(n^2) = \Theta(n^2+n)$?
3) Este notatia o reflexiva?
4) $f(n) + g(n)\in O(\max(f(n),g(n)))$?
5) $2^{2n}\in \Theta(2^n)$?
6) $f(n)\in O(f^2(n))$?
7) $f(n)+O(f(n))\in \Theta(f(n))$?

Exemplu de analiza de complexitate: insertion-sort
- elementul curent se insereaza in bucata de vector aflata in stanga lui, care este mereu sortata
- se porneste cu al 2lea element

insertion-sort(v)	nr executii / instructiune
for j=2 to n	n
elem<-v[j]	n-1
i<-j-1	n-1
while i>0 and elem<v[i]	S1
v[i+1]<-v[i]	S2
i<-i-1	S2
v[i+1]<-elem	n-1

Cazul cel mai favorabil: v deja sortat, nu se intra in while
S1 = n-1 (prin fiecare instructiune while se trece fix o data, de n-1 ori)
S2 = 0 (nu se ajunge la instructiunile din interiorul while-ului)
$\Rightarrow T(n) = 5n-4 \Rightarrow T(n)\in \Theta(n)$

Obs: Ca regula generala cand stabilim clasa de complexitate, in functia de complexitate **ignoram termenii cu crestere mai inceata** si **ignoram constanta din fata termenului dominant**.

Cazul cel mai defavorabil: v sortat invers, fiecare while merge pana la i=0
S1 = $Suma_{j=2..n}(j) = n(n+1)/2 - 1$
S2 = $Suma_{j=2..n}(j-1) = n(n-1)/2$
=> $T(n) \in \Theta(n^2)$

Recomandare: pentru continuarea capitolului de complexitate, revizuiti proprietatile logaritmilor, exponentialelor, seriilor aritmetice si geometrice.