Analiza complexitatii algoritmilor

Pentru rezolvarea unei probleme, chiar daca aceeasi metoda de elaborare a algoritmului este abordata de mai multe persoane, algoritmii prezentati pot sa difere. Cu atat mai mult acest lucru este posibil atunci cand metodele de rezolvare sunt diferite. Atunci cand exista mai multi algoritmi de rezolvare ai unei probleme, ar trebui sa se stabileasca, firesc, care dintre algoritmi este „mai performant”. Se impune astfel a gasi o masura a gradului de performanta sau de eficienta a algoritmilor si in functie de aceasta, o valoare optima.

Doua criterii stabilesc masura performantei unui algoritm: *timpul* in care se obtine solutia problemei si resursele (*spatiul*) de *memorie* utilizate pentru obtinerea ei. Analiza acestor parametri de eficienta ai algoritmilor este cunoscuta in literatura de specialitate sub numele de *analiza complexitatii algoritmilor*.

Spatiului de memorie utilizat de programul care implementeaza algoritmul intr-un limbaj de programare este format dintr-o parte constanta, independenta de datele de intrare, in care se afla memorat codul executabil, variabile si structuri de date de dimensiune constanta alocate static si o parte variabila ca lungime care depinde de volumul de date de prelucrat, spatiul necesar pentru structurile de date alocate dinamic, stive pentru apelul functiilor si procedurilor si a caror lungime depinde in mod cert de algoritmul de rezolvare.

Un exemplu care scoate in evidenta diferenta de eficienta legata de consumul de spatiu de memorie, il constituie doi algoritmi, care calculeaza suma primelor n numere naturale.

Primul algoritm consta in a construi o functie care sa calculeze succesiv sumele $0, 0 + 1, 0 + 1 + 2,$ functie care va intoarce valoarea sumei $1 + 2 + 3 + + n$.

```
function suma(n : byte):word;
var i : byte;
    s : word;
begin
    s := 0;
    i := 1;
    while (i <= n) do
        begin
            s := s + i;
            i := i + 1;
        end;
   suma := s;
end;
```

Functia va ocupa memorie pentru parametru, variabilele locale, pentru adresa de revenire si evident cu codul. Deci nu este necesar spatiu de memorie variabil.

Al doilea algoritm presupune construirea unei functii recursive care calculeaza suma dupa relatia de recurenta $s(n) = s(n-1) + n$, cu $s(0) = 0$.

```
function suma(p : byte):word;
begin
    if ( p = 0 ) then
      suma := 0 ;
    else
       suma := suma(p-1) + p;
end;
```

Pentru fiecare apel al functiei vor fi ocupati *5* octeti; unul pentru memorarea parametrului *p*, unul pentru valoarea functiei si 2 octeti pentru adresa de revenire. Se fac *n* apeluri recursive, deci spatiul de memorie variabil este de *5n* octeti. Algoritmul care foloseste functia recursiva foloseste mai mult spatiu efectiv de memorie decat in cazul primului algoritm.

Pentru a analiza teoretic algoritmul din punct de vedere a duratei de executie a programului care-l implementeaza, vom presupune ca o operatie elementara se executa intr-o unitate de timp sau daca timpul de executie a doua operatii elementare difera, acesta este bine stabilit (si constant) pentru fiecare operatie elementara si este independent de datele cu care opereaza. Nu vom restrange generalitatea daca presupunem ca timpul de executie este acelasi pentru toate operatiile elementare.

Astfel, timpul de executie al programului este direct proportional cu numarul de operatii simple efectuate de algoritm, numar care ofera un criteriu de comparatie, intre algoritmi si programele care-i implementeaza, fara a mai fi necesara implementarea efectiva si executia. Teoretic, aprecierea timpului de executie se poate face pentru orice volum de date de intrare, lucru care practic nu este realizabil, daca am incerca sa masuram efectiv timpul pe perioada executiei programelor, iar aceasta s-ar face pentru seturi particulare de date de intrare.

Analiza complexitatii algoritmului ca timp de executie presupune determinarea numarului de operatii elementare efectuate de algoritm, nu si a timpului total de executie a acestora, tinand cont doar de ordinul de marime a numarului de operatii elementare.

Analiza complexitatii din punct de vedere a timpului de executie presupune determinarea unor functii care sa limiteze comportarea in timp a algoritmului, functii care au ca parametri *caracteristicile relevante* ale datelor de intrare.

Cuantificand caracteristicile relevante (de exemplu dimensiunea) ale datelor de intrare, pentru un algoritm stabilit, putem defini o functie $f: N \; R_+^*$ care da ordinul de marime al numarului de operatii elementare si implicit al timpului de executie, care se va nota cu *t(f(n))*.

Definitie. Un algoritm este de ordinul *f(n)*, notat *O(f(n))*, daca si numai daca exista o constanta pozitiva *c>0* si $n_0 N$, astfel incat $t(f(n)) \; c \cdot f(n), \; n n_0$.

Exemplu.

a) Daca t(n) an b, a>0 si f(n) n, atunci t(n)O(n) pentru ca exista c a 1, c> 0 si n_0 bN, astfel incat an b cf(n), n b.

b) Daca t(n) an^2 bn c, a>0, pentru f(n) n^2, atunci t(n)O(n^2), pentru ca an^2 bn c (a 1) f(n), n max(b,c) 1.

c) Daca t(n) $6\cdot 2^n+n^2$ si f(n) 2^n atunci t(n)O(2^n), pentru ca $T_A(n)$ $6\cdot 2^n$ n^2 7· f(n), n 4.

Propozitie. Daca t(n) $a_k n^k$ $a_{k-1}n^{k-1}$ $a_1 n$ a_0, atunci t(n)O(n^k).

Demonstratie. Pentru f(n) n^k,

t(n) t(n) $a_k n^k$ $a_{k-1}n^{k-1}$ $a_1 n$ a_0 $a_k n^k$ $a_{k-1}n^{k-1}$ $a_1 n$ a_0

(a_k a_{k-1} a_1 a_0)n^k, n 1.

Pentru c a_k a_{k-1} a_1 a_0 si n_0 1 T t(n)O(n^k).

Observatie. Evaluarea ordinului unui algoritm echivaleaza cu determinarea unei margini superioare a timpului de executie a algoritmului. Prin urmare:

- un algoritm cu *t(f(n)) O(1)* necesita un timp de executie *constant;*
- un algoritm cu *t(f(n)) O(log n)* se numeste *logaritmic*;
- un algoritm cu *t(f(n)) O(n)* se numeste *liniar*;
- un algoritm cu *t(f(n)) O(n^2)* se numeste *patratic*;
- un algoritm cu *t(f(n)) O(n^3)*, se numeste *cubic*;
- un algoritm cu *t(f(n)) O(n^k)* se numeste *polinomial*;
- un algoritm cu *t(f(n)) O(2^n)* se numeste *exponential*.

Aceasta notatie, numita asimptotica, determina o clasificare a algoritmilor impusa de valoarea ordinului de complexitate:

O(1) O(log n) O(n) O(n log n) O(n^2) O(n^k) O(2^n), k > 2.

Putem astfel clasifica algoritmii din punctul de vedere al performantei.

Reprezentarea grafica a functiilor care determina ordinul de complexitate, prezentata in figura de mai jos, ilustreaza comportarea lor. „*Daca t(f((n))O(2^n), pentru n = 40, unui calculator care face 1 bilion (10^9) de operatii pe secunda, ii sunt necesare aproximativ 18 minute. Pentru n = 50, acelasi program va rula 13 zile pe acest calculator, pentru n = 60, vor fi necesari peste 310 ani, iar pentru n = 100 aproximativ 4·10^{13} ani*”.

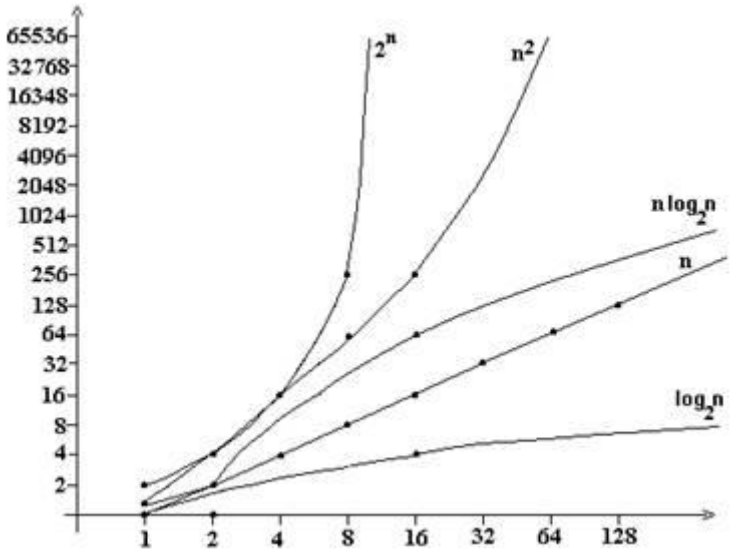

Fig. 18.

Algoritmii polinomiali de grad mare nu pot fi utilizati in practica, chiar daca viteza de executie a calculatoarelor moderne intrece adesea cele mai optimiste previziuni. Astfel, pentru $O(n^{10})$, *„pe un calculator care executa 1 bilion de operatii pe secunda sunt necesare 10 secunde pentru* $n = 10$*, aproximativ 3 ani pentru* $n = 100$ *si circa* $3 \cdot 10^{13}$ *ani pentru* $n = 1000$*"*. Tabelul de mai jos ne poate edifica asupra adevarului acestei afirmatii:

O(n) (liniar)	O(log(n)) (logaritmic)	O(n.log(n)) (log-liniar)	$O(n^2)$ (patratic)	$O(2^n)$ (exponential)	O(n!) (factorial)
1	0	0	1	2	1
2	1	2	4	4	2
4	2	8	16	16	24
8	3	24	64	256	40326
16	4	64	256	65536	20922789888000
32	5	160	1024	4294967296	26313.10^{33}

„Tragismul situatiei" reliefate de acest tabel este totusi diminuat de realitatea practica. Chiar daca timpul de executie al unui algoritm este direct proportional cu numarul de operatii elementare, totusi, acest numar poate varia considerabil in functie de caracteristicile relevante ale datelor de intrare (cum ar fi ordinul de marime al setului de date, cunoscut si sub denumirea de *volumul datelor de intrare*).

Vom exprima astfel numarul de operatii elementare in functie de volumul datelor de intrare (ordinul de marime), care determina altfel o masura exacta a notiunii de operatie elementara in contextul algoritmului pe care il analizam si in functie de care vom exprima timpul de executie.

Evaluarea complexitatii-timp a unui algoritm ca o functie de caracteristicile datelor de intrare este o problema dificila si ea se rezuma de cele mai multe ori la analiza cazurilor extreme (*cel mai favorabil si cel mai defavorabil*), sau *in medie*. Cazul cel mai defavorabil este acel caz in care numarul de operatii elementare efectuate de algoritm este maxim.

Chiar daca, in cazul cel mai defavorabil, numerosi algoritmi nu pot fi practic utilizati, acestia au o comportare acceptabila in practica curenta. Un cunoscut exemplu este cel al algoritmului de sortare rapida (quicksort) care are complexitatea in cazul cel mai defavorabil de $O(n^2)$, dar pentru datele intalnite in practica functioneaza in $O(n \log n)$.

O alta posibilitate este evaluarea complexitatii medii a unui algoritm, dar care presupune cunoasterea repartitiei probabilistice a datelor de intrare si din acest motiv analiza complexitatii in medie este mai dificil de realizat. In cazuri simple, cand putem caracteriza exact datele de intrare, daca am nota cu D – spatiul datelor de intrare, cu $p(d)$ probabilitatea aparitiei datei dD la intrarea algoritmului si cu $t(d)$ numarul de operatii elementare efectuate de algoritm pentru o intrare d din D, atunci *complexitatea medie* este $p(d)t(d)$. Vom ilustra afirmatiile anterioare prin doua exemple sugestive.

Un prim exemplu va prezenta un algoritm (implementat in limbajul **Pascal**), a carui complexitate nu depinde decat de volumul datelor de intrare si nu de alte caracteristici atipice.

Sortarea prin selectie (cu alegerea minimului). Sa se ordoneze crescator elementele vectorului a cu n componente, folosind metoda alegerii elementului minim neselectat din sirul initial.

```
for i := 1 to n-1 do
   begin
      min := a[i] ;
      poz := i;
      for j := i + 1 to n do
         if a[j] < min then
            begin
               min := a[j] ;
               poz := j;
            end;
      a[poz] := a[i] ;
      a[i] := min;
   end;
```

Vom face o evaluare a complexitatii algoritmului dupa numarul de componente ale vectorului. La o iteratie a ciclului **for** dupa variabila i se determina minimul din subsirul $a_{i+1}, , a_n$ si elementul minim este plasat pe pozitia i, elementele de la 1 la $i-1$ fiind deja plasate pe pozitiile lor definitive. Pentru a calcula minimul dintr-un sir de k elemente sunt necesare $k-1$ operatii elementare (se presupune primul element din sir ca fiind cel minim, apoi se fac $k-1$ comparatii si eventual atribuiri pana la epuizarea elementelor sirului); in total:

$$(n-1) + (n-2) + + 2 + 1 = n \cdot (n-1)/2.$$

Deci ordinul de complexitate al algoritmului este $O(n^2)$. Sa subliniem faptul ca timpul de executie al algoritmului nu depinde de ordinea initiala a elementelor vectorului.

In urmatorul exemplu vom analiza complexitatea unui algoritm atat in cazul cel mai defavorabil cat si in medie.

Sortarea prin insertie directa. Sa se ordoneze crescator elementele unui vector considerand in fiecare moment ca se ordoneaza un subsir obtinut din cel anterior (deja ordonat), prin adaugarea unui nou element.

Algoritmul porneste de la subsirul cu un singur element (care este deja ordonat) si, odata cu adaugarea unui nou element pe urmatoarea pozitie din sir, acesta este promovat pana cand noul subsir devine din nou ordonat.

```
for i := 2 to n do
    begin
        j :=  i;
        while (a[j-1]  > a[j] ) and ( j > 1 ) do
            begin
                k := a[j-1] ;
                aj-1 :=  aj;
                a[j] := k;
                j := j-1;
            end;
    end;
```

Analizam algoritmul in functie de n, dimensiunea vectorului a ce urmeaza a fi sortat. La fiecare iteratie a ciclului **for** elementele $a_1,a_2,,a_{i-1}$ sunt deja ordonate si trebuie sa interschimbam elementul *aj]* cu *a[j-1]* (initial $j = i$) pana cand noul sir va deveni ordonat. In cazul cel mai defavorabil, cand fiecare element adaugat la sir este mai mic decat cele adaugate anterior, elementul a_i adaugat va fi deplasat pana pe prima pozitie, deci ciclul **while** se executa de $i-1$ ori.

Considerand drept operatie elementara compararea elementului *a[j-1]* cu *a[j]* si interschimbarea acestor elemente cat timp *aj-1 >a[j]*, vom avea in cazul cel mai defavorabil:

$1 + 2 + + (n-1) = n\cdot(n-1)/2$ operatii elementare, deci complexitatea algoritmului este $O(n^2)$.

Sa analizam comportarea algoritmului in medie. Pentru aceasta, vom considera ca orice permutare a elementelor sirului are aceeasi probabilitate de aparitie (orice ordine initiala este egal probabila).

Atunci:

- probabilitatea ca valoarea a_i, nou adaugata la sirul $a_1, a_2, , a_{i-1}$ sa fie plasata in final pe o pozitie oarecare, k, din $a_1, a_2, , a_i$ $(1 \le k \le i)$, este aceeasi : $1/i$;

- numarul mediu de operatii elementare (interschimbari de elemente), pentru ca elementul a_i sa ajunga pe pozitia k va fi $(i-k)\frac{1}{i}$, adica numarul de schimbari ce se efectueaza, inmultit cu probabilitatea ca aceste schimbari sa aiba loc;

- numarul mediu total de operatii elementare pentru un i fixat, va fi

$$\sum_{k=1}^{i}\frac{i-k}{i}=\frac{1}{i}\sum_{k=1}^{i}(i-k)=\frac{1}{i}(i^2-\frac{i(i+1)}{2})=\frac{i-1}{2}$$

- pentru a sorta cele n elemente sunt necesare

$$\sum_{i=1}^{n}\frac{i-1}{2}=\frac{1}{2}(\frac{n(n+1)}{2}-n)=\frac{n(n-1)}{4}$$

operatii elementare. Deci complexitatea algoritmului in medie este tot de $O(n^2)$.

Inafara tratarii complexitatii algoritmilor, la fel de important in practica este studiul coerent al terminarii si corectitudinii (nu verificarii!) acestuia. Sa subliniem doar ideea ca trebuie sa ne asiguram ca un program se termina pentru orice instanta admisibila si face ceea ce vrem, ***inainte*** ca el sa fie executat.

O alta tema de importanta deosebita este legata de introducerea si stapinirea (intuitiv si chiar formal) a logicii clasice (aristotelice si matematice). Trebuie sa se stie ce inseamna valoare de adevar, teorema directa, contrara, reciproca, rationament, sfera, diferenta specifica, etc.

O teorie generala a structurilor de date este de asemenea indispensabila.