COMPLEXITATEA ALGORITMILOR

Profesor Radu Ana-Maria
Colegiul Național Pedagogic ”Ștefan cel Mare”, Bacău

Pentru rezolvarea unei probleme, chiar dacă aceeaşi metodă de elaborare a algoritmului este abordată de mai multe persoane, algoritmii prezentaţi pot să difere. Cu atât mai mult acest lucru este posibil atunci când metodele de rezolvare sunt diferite. Atunci când există mai mulţi algoritmi de rezolvare ai unei probleme, ar trebui să se stabilească, firesc, care dintre algoritmi este „mai performant”. Se impune astfel a găsi o măsura a gradului de performanţă sau de **eficienţa a algoritmilor** şi în funcţie de aceasta, o valoare optimă. Analiza unui algoritm presupune determinarea timpului necesar de rulare al algoritmului. Acesta nu se masoară în secunde, ci in numarul de operaţii pe care le efectuează. Numărul de operaţii este strâns legat de timpul efectiv de executie (în secunde) a algoritmului, dar nu acesta nu constituie o modalitate de măsurare a eficienţei algorimului deoarece un algoritm nu este "mai bun" decât altul dacă îl vom rula pe un calculator mai rapid sau este "mai incet" dacă il rulăm pe un calculator mai puţin performant. Analiza algoritmilor se face independent de calculatorul pe care va fi implementat sau de limbajul în care va fi scris. În schimb, presupune estimarea numărului de operaţii efectuate.

Analiza complexităţii unui algoritm are ca scop estimarea volumului de resurse de calcul necesare pentru execuţia algoritmului. Prin resurse se înţelege

- **Spaţiul de memorie** necesar pentru stocarea datelor pe care le prelucrează algoritmul.
- **Timpul** necesar pentru execuţia tuturor prelucrărilor specificate în algoritm

Dintre cele două resurse de calcul, spaţiu şi timp, cea critică este timpul de execuţie.

Un *exemplu* care scoate în evidenţa diferenţa de eficienţa legată de consumul de spaţiu de memorie, îl constituie doi algoritmi, care calculeaza suma primelor n numere naturale. Primul algoritm constă în a construi o funcţie care să calculeze succesiv sumele 0, 0 + 1, 0 + 1 + 2, funcţie care va întoarce valoarea sumei 1 + 2 + 3 + + n.

```
int suma (int n)
{ int i=1, s=0;
while (i <= n)
      {s = s + i;
       i = i + 1; }
return s; }
```

Funcţia ocupă memorie pentru parametru, variabilele locale, pentru adresa de revenire şi evident cu codul. Deci nu ar fi nevoie de spaţiu de memorie variabil. Al doilea algoritm construieşte o funcţe recursivă care calculeaza suma dupa relaţia de recurenţă.

s(n) = s(n-1) + n, cu s(0) = 0.

```
int suma (int n)
{ if ( p = 0 )
              return 0;
      else
      suma = suma(p-1) + p;}
```

Algoritmul care foloseşte funcţia recursivă foloseşte mai mult spaţiu de memorie decât în cazul primului algoritm. Timpul de execuţie al programului este direct proporţional cu numărul de operaţii simple efectuate de algoritm, acest număr oferind un criteriu de comparaţie, între algoritmi şi programele care îi implementează, fară a mai fi necesară implementarea efectivă si execuţia.. Analiza complexitatii algoritmului ca timp de executie presupune determinarea numărului de operaţii elementare efectuate de algoritm, nu şi a timpului total de execuţie a acestora, ţinând cont doar de ordinul de mărime a numărului de operaţii elementare.

<u>Notaţii asimptotice</u>

- **Ordinul de mărime al timpului** de execuţie al unui algoritm:
  - Reprezintă măsura eficienţiei algoritmului
  - Face compararea relativă a variantelor de algoritmi.
- **Studiul eficienţei asimptotice** a unui algoritm, este realizat utilizând mărimi de intrare cu dimensiuni suficient de mari pentru a face relevant numai **ordinul de mărime al timpului de execuţie al algoritmului**.

- Este interesantă **limita la care tinde timpul de execuţie** al algoritmului odată cu creşterea nelimitată a dimensiunii intrării.

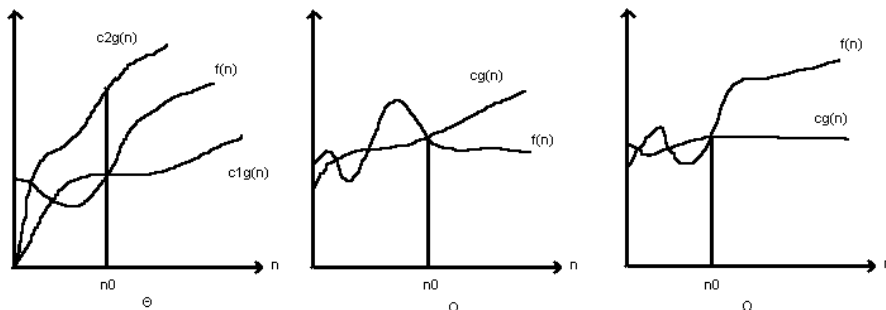
Notaţia O (O mare)

- Notaţia **O** desemnează **marginea asimptotică superioară** a unei funcţii. Pentru o funcţie dată g(n), se defineşte O(g(n)) ca şi mulţimea de funcţii:

$O(g(n)) = \{ f(n)$: există constantele pozitive c şi n_0 astfel încât $0 \le f(n) \le c * g(n)$, pentru $\forall\ n \ge n_0 \}$

- Notaţia O este utilizată pentru a desemna o margine superioară a unei funcţii în interiorul unui factor constant.
- Pentru toate valorile n superioare lui n_0, valoarea funcţiei f(n) este pe sau dedesuptul lui g(n).

În **figura** următoare se poate observa interpretarea grafică a notaţiilor de complexitate



$\Omega(g(n)) = \{ f(n)$: există constantele pozitive c şi n_0 astfel încât $0 \le c * g(n) \le f(n)$ pentru $\forall\ n \ge n_0 \}$

$\Theta\ (g(n)) = \{ f(n)$: există constantele pozitive c_1, c_2 şi n_0 astfel încât $0 \le c_1 * g(n) \le f(n) \le c_2 \cdot g(n)$ pentru $\forall\ n \ge n_0 \}$

f(n)∈O(g(n)) inseamnă că pentru valori mari ale dimensiunii intrarii, c*g(n) este o limită superioară pentru f(n); algoritmul se va purta mereu mai bine decât această limită.

f(n)∈Ω(g(n)) inseamnă că, pentru valori mari ale dimensiunii intrării, c*g(n) este o limită inferioară pentru f(n); algoritmul se va purta mereu mai prost decat această limită.

f(n)∈Θ(g(n)) inseamnă că, pentru valori mari ale dimensiunii intrării, c_1 *g(n) este o limită inferioară pentru f(n), iar c_2 *g(n) o limita superioara.

Notaţia O este de obicei cea mai utilizată în aprecierea timpului de execuţie al algoritmilor respectiv a performanţei acestora. Cele mai obişnuite ordine de mărime ale notaţiei O se află în relaţiile de ordine sunt:

$O(1) < O(\log n) < O(n) < O(n \cdot \log n) < O(n^2) < O(n^3) < < O(2^n) < O(10^n) < O(n!) < O(n^n)$

Sume întregi utile care sunt frecvent utilizate în **calculul complexităţii algoritmilor**:

$$\sum_{i=1}^{n} i = \frac{n*(n+1)}{2} = O(n^2)$$

$$\sum_{i=1}^{n} i^2 = \frac{n*(n+1)*(2n+1)}{2} = O(n^3)$$

$$\sum_{i=1}^{n} i^3 = \frac{n^2*(n^2+1)}{4} = O(n^4)$$

$$\sum_{i=1}^{n} i^k = \frac{n^{k+1}}{k+1} + \cdots = O\left(\frac{n^{k+1}}{k+1}\right) = O(n^{k+1})$$

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

Clase de complexitate

Clasa	Exemple	Număr de operaţii		
		n=3	n=10	n=100
O(1)	Operaţii aritmetice simple, căutare folosind tabele de dispersie	1	1	1
O(ln(n))	Căutare binară	≅1	≅2	≅5
O(n)	Căutare secvenţială, suma unui şir	3	10	100
O(n*ln(n))	Sortările rapide (interclasare, sortare rapidă, ...)	≅3	≅23	≅460
O(n^2)	Sortările "naive"(metoda bulelor, inserţie, ...), parcurgeri de matrice pătratice de ordin n	9	100	10000
O(n^3)	Înmulţirea clasică a matricelor	27	1000	1000000
O(2^n)	Generare aranjamente	8	1024	≅10^{30}

Aplicaţii pentru analiza complexităţi algoritmilor utilizaţi în problemele respective [9]:

Exerciţiul 1:

Fie două numere întregi x şi y. Să se realizeze interschimbarea valorilor lor.

Ex.: Dacă iniţial x=5 şi y=7, după interschimbare valorile lor vor fi x=7 şi y=5.

Rezolvare: Secvenţa care utilizează interschimbarea, scrisă in limbajul de programare C++ este:

```
aux=x;
x=y;
y=aux;
```

În acest algoritm sunt utilizate trei atribuiri. Numărul de operaţii necesar interschimbării este mereu constant, indiferent ce valori vor fi memorate de x şi y. De aici deducem că timpul de execuţie este în acest caz O (1), adică timp constant. Pentru că este folosit un număr fix de variabile, x, z, aux, complextatea spaţiu este tot **O(1).**

Exerciţiul 2:

Fie un şir de numere întregi a_1, a_2, ..., a_n de numere întregi. Să se determine valoarea minimă din acest şir.

Rezolvare:

Pentru început se consideră că iniţial valoarea minimă este a_1. După care fiecare element a_i este comparat cu valoarea minimă, actualizându-se dacă este cazul. Secvenţa care calculează minimul, scrisă in limbajul de programare C++ este:

```
min=a[1];
for (i=2; i<=n; i++)
        if (a[i]<min)
                        min = a[i];
cout<<min;
```

Cel mai favorabil caz este atunci când minimul este chiar pe prima poziţie, iar cel mai nefavorabil este atunci când valorile din şir sunt în ordine descrescătoare, caz în care minimul se actualizează de n-1 ori. Deobicei nu putem stabili de câte ori se execută atribuirea min = a[i]. Prin urmare vom lua în seamă doar operaţia de bază.

Pentru această problemă este comparaţia (a[i]<min) care se execută de n+1 ori indferent pe ce poziţie se află minimul. Se mai pot considera ca operaţii şi comparaţia lui i cu n, precum şi incrementarea lui i, dar şi iniţialzarile lui i

şi ale lui min. Prin urmare numărul de operaţii ar fi 2+3(n-1) = 3n-1, care este un polinom de gradul întâi. Se va lua în considerare numai monomul de grad maxim şi se ignoră coeficientul acestuia. Complexitatea timp pentru acest algoritm de calculare a minimului dintr-un şir este O (n). Complexitatea spaţiu este determinată de folosirea unu vector de lungime n care memorează şirul, şi a variabilelor smple i, n, max. Aceasta complexitate spaţu este **O (n).**

Exerciţiul 3:

Fie citit un număr natural n<=20. Să se calculeze 2^n.

Rezolvare:

Ştiind că calculul lu 2^n înseamnă înmulţirea cu 2 de n ori, secvenţa care utilizează interschimbarea, scrisă in limbajul de programare C++ este:

```
p=1;
for (i=1; i<=n;i++)
        p=p*2;
```

Complexitatea timp pentru această soluţie este **O(n)**.

NP – completitudine

Mai există o clasă de probleme a celor NP – complete (*Non – determinist Polinomiale*) , pentru care se folosesc algoritmi de rezolvare exponenţială, dar nu se ştie dacă admit rezolvări de complexitate polinomială. Astfel avem ca exemplu problema comis-voiajorului, de la teoria grafurilor problema clicii, problema submulţimii de sumă dată etc. Dar nu trebuie confundată cu problema generării permutărilor care nu este NP – completă deoarec numărul permutărilor este n! Şi atunci orice algoritm de generare nu poate fi decât exponenţial.

Bibliografie

1) Răzvan Andonie, Ilie Gârbacea „*Algoritm fundamentali – o perspectivă C++*” , Editura Libris, Cluj Napoca 1995
2) Dan Pracsiu, *Culegere de probleme semnificative de informatică*, Editura Media Sind, Vaslui, 2015
3) Albeanu Gr., „Algoritm şi limbaje de programare”, Universtatea „Spiru Haret”, Editura România de Mâine, Bucureşti, 2000