CURS 2
COMPLEXITATEA ALGORITMILOR

Determinarea complexității computaționale a unui algoritm

Vom începe prin a prezenta câțiva algoritmi a căror complexitate poate fi estimată corect mai greu:

a)

```
printf("n = ");
scanf("%d", &n);

i = 0;
p = 1;

while(i <= n)
{
     j = 1;
     while(j <= p)
     {
          printf("%d ", j);
          j++;
     }
     printf("\n");
     i++;
     p = p*2;
}
```

După o analiză superficială, acest algoritmul pare să aibă complexitatea $\mathcal{O}(n \cdot p)$, datorită celor două instrucțiuni `while` imbricate. Analizând cu atenție algoritmul, vom observa faptul că, pentru fiecare valoare i cuprinsă între 0 și n, el afișează numerele de la 1 la 2^i, ceea ce înseamnă că, în total, va afișa $2^0 + 2^1 + \cdots + 2^n = 2^{n+1} - 1$ numere, deci complexitatea sa este $\mathcal{O}(2^n)$, adică o complexitate exponențială! Această complexitate exponențială este indusă de faptul că variabila p își dublează valoarea la fiecare pas, deci are o creștere exponențială. Dacă variabila p ar avea o creștere liniară (e.g., `p = p + 1`), atunci, pentru fiecare valoare i cuprinsă între 0 și n, se vor afișa numerele de la 1 la $i + 1$, ceea ce înseamnă că, în total, va afișa $1 + 2 + \cdots + (n + 1) = \frac{(n+1)(n+2)}{2}$ numere, deci complexitatea sa ar fi $\mathcal{O}(n^2)$, adică o complexitate pătratică!

b)

```
printf("a = ");
scanf("%d", &a);

printf("b = ");
scanf("%d", &b);
```

```
p = 1;
while(p < a)
     p = p * 2;

while(p <= b)
{
     printf("%d ", p);
     p = p * 2;
}
```

Inițial, acest algoritmul pare să aibă complexitatea $\mathcal{O}(b)$, datorită celor două instrucțiuni `while` secvențiale. Analizând cu atenție algoritmul, vom observa faptul că la fiecare pas valoarea variabilei p se dublează, deci, numărul total de iterații k pe care le va efectua algoritmul se obține rezolvând inecuația $2^k \leq b$, de unde obținem $k \leq \log_2 b$. În concluzie, acest algoritm, care afișează puterile lui 2 cuprinse între a și b, are complexitatea $\mathcal{O}(\log_2 b)$, deci o complexitate logaritmică!

c)

```
printf("n = ");
scanf("%d", &n);

for(i = 0; i < n; i++)
{
     printf("v[%d] = ", i);
     scanf("%d", &v[i]);
}

i = 0;
j = n - 1;
while (i < j)
{
     while(i < n && v[i] < 0)
          i++;
     while(j >= 0 && v[j] >= 0)
          j--;
     if (i < j)
     {
          aux = v[i];
          v[i] = v[j];
          v[j] = aux;
     }
}

printf("\nTabloul:\n");
for(i = 0; i < n; i++)
     printf("%d ", v[i]);
```

La prima vedere, acest algoritmul pare să aibă complexitatea $\mathcal{O}(n^2)$, unde n este numărul de elemente din tabloul unidimensional v, datorită celor două instrucțiuni `while` secvențiale imbricate în altă instrucțiune `while`. Analizând cu atenție algoritmul, vom observa faptul că, în realitate, cei 2 indici i și j nu se suprapun, ci parcurg, fiecare, o porțiune din tabloul v (i.e., indicele i parcurge tabloul de la stânga spre dreapta, iar indicele j de la dreapta spre stânga), iar în momentul în care cei 2 indici se "întâlnesc", algoritmul se termină. Astfel, per total, cei 2 indici vor parcurge o singură dată întregul tablou v, deci complexitatea algoritmului este $\mathcal{O}(n)$, respectiv o complexitate liniară. Algoritmul realizează un fel de sortare a elementelor tabloului, respectiv mută valorile negative înaintea celor pozitive, fără ca valorile negative sau cele pozitive să fie sortate conform unui criteriu. Metoda de parcurgere utilizată se numește *metoda arderii lumânării la două capete* (*two pointers*) și, într-o formă puțin modificată, este folosită și în metoda de sortare Quicksort.

Estimarea timpului de executare al unui algoritm

Complexitatea unui algoritm ne oferă o modalitate de comparare a performanțelor mai multor algoritmi care rezolvă o anumită problemă, dar, în practică, ne interesează și estimări ale timpului de executare al unui algoritm pe un anumit sistem de calcul. Pentru a realiza acest lucru, pe lângă complexitatea computațională, vom lua în calcul și frecvența procesorului sistemului respectiv, deoarece aceasta influențează în mod decisiv timpul de executare al unui algoritm.

Practic, pentru a estima timpul de executare al unui algoritm cu o anumită complexitate computațională vom utiliza următoarele două observații:

- durata unui ciclu de procesor (exprimată în secunde), este egală cu inversul frecvenței procesorului (exprimată în hertzi);
- orice operație elementară durează maxim 5 cicli de procesor.

De exemplu, un procesor cu frecvența f_P = 3GHz execută $3\text{GHz} = 3 \times 10^9$ cicli pe secundă, deci un ciclu de procesor durează $\frac{1}{3\text{GHz}} = \frac{1}{3 \cdot 10^9} = 0.3 \cdot 10^{-9}\text{secunde} = 3 \cdot 10^{-10}\text{secunde}$. În concluzie, rezultă că, pentru un astfel de procesor, o operație elementară durează aproximativ $5 \cdot 3 \cdot 10^{-10}\text{secunde} = 15 \cdot 10^{-10}\text{secunde}$.

Dacă vom rula un algoritm cu complexitatea $\mathcal{O}(n^2)$ pe un PC echipat cu un procesor având frecvența $f_P = 3\text{GHz}$ pentru $n = 20000 = 2 \cdot 10^4$, atunci timpul de executare t poate fi aproximat, în secunde, astfel:

$$t = \underbrace{(2 \cdot 10^4)^2}_{\substack{\text{numărul total} \\ \text{de operații elementare}}} \cdot \underbrace{15 \cdot 10^{-10}}_{\substack{\text{durata unei} \\ \text{operații elementare}}} \text{ s} = 6 \cdot 10^{-1} \text{ s} = 0.6 \text{ s}$$

În imaginea de mai jos, puteți observa faptul că estimarea realizată anterior este corectă (am utilizat un PC echipat cu un procesor având frecvența $f_P = 3\text{GHz}$):

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    unsigned int i, j, n;
    unsigned long long int s;

    s = 0;
    n = 20000;
    for(i = 0; i < n; i++)
        for(j = 0; j < n; j++)
            s = s + i + j;

    printf("Suma: %llu\n", s);

    return 0;
}
```

```
C:\Users\Ours\Desktop\Test_C\bin\Debug\Test_C.exe
Suma: 7999600000000

Process returned 0 (0x0)   execution time : 0.556 s
Press any key to continue.
```

Pentru același sistem de calcul, putem estima timpul de executare t al unui algoritm cu complexitatea $\mathcal{O}(2^n)$ pentru $n = 100$ astfel (vom folosi aproximarea $2^{10} \approx 10^3$):

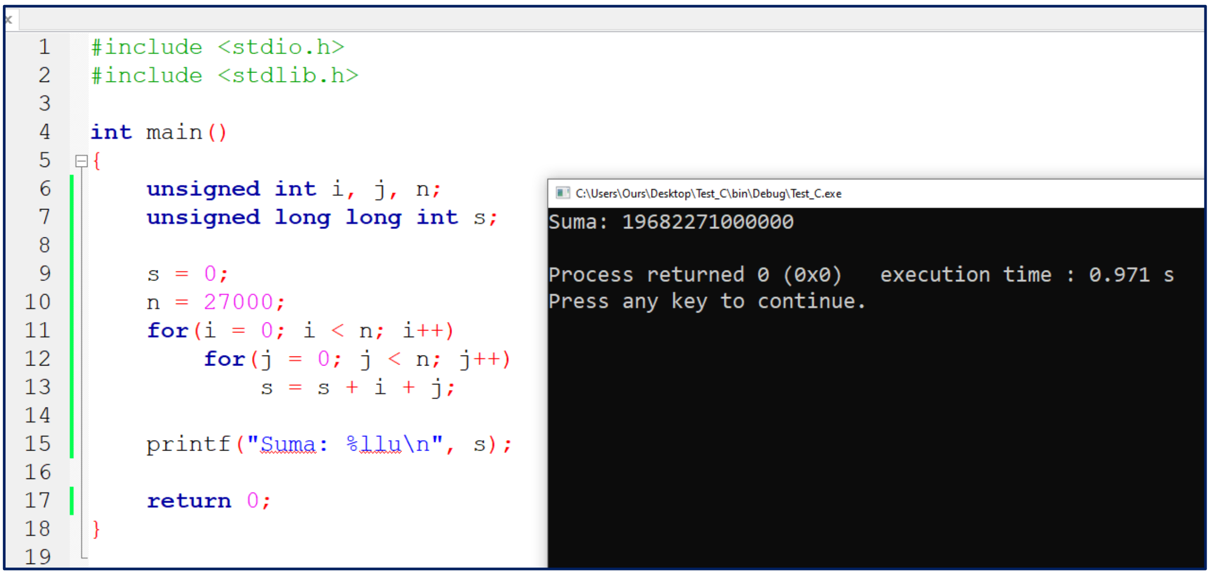
$$t \approx \underbrace{2^{100}}_{\substack{\text{numărul total}\\ \text{de operații elementare}}} \cdot \underbrace{15 \cdot 10^{-10}}_{\substack{\text{durata unei}\\ \text{operații elementare}}} \text{ s} \approx 10^{30} \cdot 15 \cdot 10^{-10} \text{ s} \approx 15 \cdot 10^{20} \text{ s} \approx$$

$$\approx \frac{15 \cdot 10^{20}}{3600} \text{ ore} \approx \frac{15 \cdot 10^{16} \cdot \cancel{10^4}}{\cancel{3600}} \text{ ore} \approx 15 \cdot 10^{16} \cdot 3 \text{ ore} \approx 45 \cdot 10^{16} \text{ ore} \approx \frac{45 \cdot 10^{14} \cdot \cancel{10^2}}{\cancel{24}} \text{ zile} \approx$$

$$\approx 45 \cdot 10^{14} \cdot 4 \text{ zile} \approx 180 \cdot 10^{14} \text{ zile} \approx \frac{180 \cdot 10^{14}}{365} \text{ ani} \approx \frac{180 \cdot 10^{11} \cdot \cancel{10^3}}{\cancel{365}} \text{ ani} \approx$$

$\approx 180 \cdot 10^{11} \cdot 3$ ani $\approx 54000 \cdot 10^9$ ani, adică un timp mai mare de aproximativ 3900 de ori decât vârsta estimată a Universului!

Folosind estimări de tipul celor prezentate anterior, putem să determinăm dimensiunea maximă a datelor de intrare pentru care un anumit algoritm se va încadra într-un timp maxim dat. De exemplu, pentru un algoritm cu complexitatea $\mathcal{O}(n^2)$ vrem să determinăm dimensiunea maximă n a datelor de intrare pentru care acesta va rula în mai puțin de o secundă. În acest caz, timpul de executare va fi $t = n^2 \cdot 15 \cdot 10^{-10}$ s, deci vom rezolva inecuația $t \leq 1 \Leftrightarrow n^2 \cdot 15 \cdot 10^{-10} \leq 1 \Leftrightarrow n^2 \leq \frac{1}{15 \cdot 10^{-10}} = 0.07 \cdot 10^{10} \Leftrightarrow n \leq \sqrt{7 \cdot 10^8} = 2.7 \cdot 10^4 = 27000$. Astfel, obținem că algoritmul respectiv se va executa într-un timp cel mult egal cu o secundă dacă dimensiunea datelor sale de intrare este cel mult 27.000, fapt care se poate observa și în imaginea de mai jos:



O altă problemă care poate să apară în practică este aceea a verificării faptului că o anumită prelucrare poate fi efectuată într-un timp maxim dat. De exemplu, folosind sistemul de calcul utilizat anterior, putem să sortăm crescător un milion de numere în cel mult o secundă? Pentru a răspunde la această întrebare, trebuie să ne amintim faptul că algoritmii de sortare studiați (uzuali) se împart în două categorii, din punct de vedere al complexității computaționale:

a) algoritmi de sortare cu complexitatea $\mathcal{O}(n^2)$ (e.g., sortarea prin selecție sau Bubblesort), pentru care timpul de executare va fi aproximativ egal cu $t_1 \approx (10^6)^2 \cdot 15 \cdot 10^{-10}$ secunde ≈ 1500 secunde ≈ 25 minute, ceea ce arată foarte clar faptul că acești algoritmi nu pot fi utilizați pentru a sorta un milion de numere în cel mult o secundă;

b) algoritmi de sortare cu complexitatea $\mathcal{O}(n \log_2 n)$ (e.g., Quicksort sau Mergesort), pentru care timpul de executare va fi aproximativ egal cu $t_2 \approx 10^6 \cdot \log_2 10^6 \cdot 15 \cdot 10^{-10}$ secunde $\approx 270 \times 10^{-4}$ secunde ≈ 0.027 secunde, ceea ce arată faptul că acești algoritmi ar putea să sorteze un milion de numere în cel mult o secundă.

Pentru a valida și practic acest rezultat, putem utiliza următorul program, care sortează crescător un milion de numere generate aleatoriu folosind funcția `qsort` din biblioteca `stdlib.h`, care implementează algoritmul de sortare Quicksort:

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>

int cmpCrescator(const void* a, const void* b)
{
    int va = *(int*)a;
    int vb = *(int*)b;

    if(va < vb) return -1;
```

```
    if(va > vb) return +1;
    return 0;
}

int main()
{
    int i, n = 1000000;
    int *v = (int*)malloc(n*sizeof(int));
    double t;

    //initializarea generatorului de numere pseudoaleatorii
    //time(NULL) = numarul de secunde trecute de la 01.01.1970
    srand(time(NULL));

    for(i = 0; i < n; i++)
        //functia rand() va genera, la fiecare apel,
        //un numar pseudoaleatoriu cuprins intre 0 si RAND_MAX = 32767
        v[i] = rand();

    //numarul de cicli de procesor utilizati de la
    //inceputul programului
    t = clock();

    //apelam functia qsort
    qsort(v, n, sizeof(int), cmpCrescator);

    //calculez numarul de cicli de procesor utilizati
    //pentru secventa cronometrata
    t = clock() - t;

    //calculez numarul de secunde in care a fost executata
    //secventa cronometrata
    //CLOCKS_PER_SEC = numarul de cicli per secunda
    t = t / CLOCKS_PER_SEC;

    printf("Timp de executare qsort: %.2f secunde\n", t);

    free(v);

    return 0;
}
```

Rulând programul de mai sus pe un sistem de calcul având un procesor cu frecvența $f_P = 3$ GHz, vom obține timpi de executare cuprinși între 0.12 și 0.20 de secunde, ceea ce validează și experimental estimarea anterioară.