CURS 1
COMPLEXITATEA ALGORITMILOR

Un *algoritm* este o succesiune de prelucrări care se aplică mecanic asupra unor date de intrare în scopul obținerii unor date de ieșire.

Principalele caracteristici ale unui algoritm sunt:

- *corectitudine* – proprietatea unui algoritm de a furniza date de ieșire corecte pentru orice date de intrare;
- *determinism* – pentru un anumit set de date de intrare un algoritm trebuie să furnizeze întotdeauna același date de ieșire;
- *generalitate* – un algoritm nu trebuie să rezolve doar o problemă particulară, ci o întreagă clasă de probleme;
- *claritate* – prelucrările efectuate de un algoritm trebuie să nu fie ambigue;
- *finitudine* – un algoritm trebuie să se termine într-un timp finit (i.e., după un număr finit de pași).

În practică, finitudinea unui algoritm nu este suficientă pentru a-i garanta utilitatea, deoarece s-ar putea ca timpul după care el se va termina să fie prea mare în raport cu cerințele noastre. Din acest motiv, un algoritm trebuie analizat și din punctul de vedere al *eficienței* sale, respectiv să verificăm dacă algoritmul se termină într-un anumit timp maxim, convenabil în practică din punctul nostru de vedere. Dacă, în plus, numărul de pași după care algoritmul se termină este minim, atunci algoritmul respectiv este *optim*.

De exemplu, dacă am calcula suma numerelor naturale mai mici sau egale decât un n dat adunând, pe rând, fiecare număr de la 1 la n am obține un algoritm care s-ar termina într-un timp finit și, în plus, ar fi eficient pentru valori "rezonabile" ale lui n. Totuși, algoritmul optim constă în calculul sumei $1 + 2 + \ldots + n$ folosind formula $\frac{n(n+1)}{2}$.

În practică, eficiența unui algoritm se studiază prin prisma complexității sale computaționale, respectiv a resurselor necesare pentru executarea sa. Cele mai importante resurse luate în considerare în evaluarea complexității computaționale a unui algoritm sunt *timpul de executare* și *spațiul de memorie* necesar. Deoarece exprimarea timpului de executare în unități de timp nu este relevantă, aceasta depinzând foarte mult de configurația hardware și software a sistemului de calcul, se preferă exprimarea acestuia printr-o expresie care estimează numărul maxim de operații elementare efectuate de algoritmul respectiv în raport de dimensiunile datelor de intrare, folosind notația asimptotică $\mathcal{O}(expresie)$. De exemplu, dacă un algoritm are complexitatea computațională $\mathcal{O}(n^2)$ înseamnă că dimensiunea datelor sale de intrare este egală cu n (variabila din expresie) și algoritmul efectuează aproximativ n^2 operații elementare (expresia) pentru a rezolva problema respectivă. Pentru a exprima complexitatea unui algoritm din punct de vedere al spațiului de memorie necesar se utilizează aceeași notație, însă precizând explicit acest lucru (în mod implicit, complexitatea unui algoritm se referă la timpul său de executare).

Utilizarea notației asimptotice permite o comparare simplă a performanțelor a doi sau mai mulți algoritmi care rezolvă o aceeași problema, fiind evident faptul că un algoritm este cu atât mai eficient cu cât numărul de operații elementare efectuate și, eventual, spațiul de memorie necesar este mai mic.

Operațiile elementare pe care le efectuează un algoritm sunt:
- operația de atribuire și operațiile aritmetice;
- operația de decizie și operația de salt;
- operațiile de citire/scriere.

În programarea actuală, de tip structurat, nu se mai permite utilizarea operației de salt (instrucțiunea `goto` din limbajul C). Totuși, instrucțiunile repetitive sunt descompuse în astfel de instrucțiuni înainte de a fi executate de procesor! De exemplu, o instrucțiuni repetitivă de tip `for/while` poate fi simulată în limbajul C astfel:

```
#include <stdio.h>

int main()
{
    int x = 1;

    Eticheta:   printf("%d\n", x);
                x++;
                if(x <= 10) goto Eticheta;

    return 0;
}
```

Astfel, pentru a estima complexitatea unei instrucțiuni repetitive care execută de n ori un anumit bloc de instrucțiuni, vom considera faptul că instrucțiunea repetitivă este atomică (i.e., nu vom lua în considerare numărul de operații elementare efectuate pentru executarea sa), deci complexitatea sa totală va fi $\mathcal{O}(n \cdot complexitate\_bloc\_instructiuni)$:

Instrucțiune repetitivă	Complexitatea totală
`for(i = 0; i < n; i++)` `  bloc de instrucțiuni cu complexitatea` $\mathcal{O}(1)$	$\mathcal{O}(n)$
`for(i = 0; i < n; i++)` `  bloc de instrucțiuni cu complexitatea` $\mathcal{O}(m)$	$\mathcal{O}(n \cdot m)$

`for(i = 0; i < n; i++)`
`  bloc de instrucțiuni cu complexitatea` $\mathcal{O}(1)$
`for(j = 0; j < m; j++)`
`  bloc de instrucțiuni cu complexitatea` $\mathcal{O}(1)$
	$\mathcal{O}(n + m)$

`for(i = 0; i < n; i++)`
`  for(j = 0; j < m; j++)`
`  bloc de instrucțiuni cu complexitatea` $\mathcal{O}(1)$
	$\mathcal{O}(n \cdot m)$

Instrucțiune repetitivă	Complexitatea totală

`for(i = 0; i < n; i++)`
`  for(j = 0; j < m; j++)`
`  bloc de instrucțiuni cu complexitatea` $\mathcal{O}(p)$
	$\mathcal{O}(n \cdot m \cdot p)$

`for(i = 0; i < n; i++)`
`{`
`  for(i = 0; i < m; i++)`
`  bloc de instrucțiuni cu complexitatea` $\mathcal{O}(1)$
`  for(j = 0; j < p; j++)`
`  bloc de instrucțiuni cu complexitatea` $\mathcal{O}(1)$
`}`
	$\mathcal{O}(n(m+p))$

Evident, observația anterioară este valabilă pentru orice alt tip de instrucțiune repetitivă (`while` sau `do...while`), dar în exemplele prezentate am preferat utilizarea instrucțiunii repetitive `for` pentru a evidenția într-un mod simplu faptul că instrucțiunea respectivă se execută de n ori.

Practic, complexitatea computațională a unui algoritm se determină estimând numărul maxim de operații elementare efectuate în raport de dimensiunile datelor de intrare. De exemplu, complexitatea maximă a algoritmului clasic pentru calculul valorii minime dintr-un tablou unidimensional format din n valori se poate determina astfel:

Instrucțiune	Operații elementare	
`printf("n = ");`	**1 afișare**	
`scanf("%d", &n);`	**1 citire**	
`for(i = 0; i < n; i++)`	**de n ori:**	**2n operații elementare**
`{` `  printf("v[%d] = ", i);`	**1 afișare**	
`  scanf("%d", &v[i]);` `}`	**1 citire**	
`minv = v[0];`	**1 atribuire**	
`for(i = 1; i < n; i++)`	**de n-1 ori:**	**maxim 2(n-1) operații elementare**
`  if(v[i] < minv)`	**1 operație de decizie**	
`  minv = v[i];`	**maxim 1 atribuire**	
`printf("Minimul: %d\n", minv);`	**1 afișare**	
TOTAL:	**maxim 4n+2 operații elementare**	

În concluzie, complexitatea algoritmului din punct de vedere al timpului de executare este $\mathcal{O}(4n+2) \approx \mathcal{O}(n)$, iar complexitatea din punct de vedere al memoriei utilizate este tot $\mathcal{O}(n)$, deoarece se memorează cele n valori (în tabloul unidimensional v).

Expresiile utilizate în notația asimptotică a complexității computaționale se simplifică folosind următoarele două reguli (se presupune faptul că $n \to \infty$):

- constantele (multiplicative sau aditive) sunt ignorate: $\mathcal{O}(4n + 2) \approx \mathcal{O}(4n) \approx \mathcal{O}(n)$
- dintr-o expresie se păstrează doar termenul dominant: $\mathcal{O}(3n^2 + 5n + 7) \approx \mathcal{O}(3n^2) \approx \mathcal{O}(n^2)$ sau $\mathcal{O}(2^n + 3n^2) \approx \mathcal{O}(2^n)$

Clase uzuale de complexitate computațională

Vom începe prin a prezenta clasele de complexitate computațională cele mai des întâlnite, în ordine crescătoare:

a) **clasa $\mathcal{O}(1)$ – *complexitate constantă***

Exemple: suma a două numere sau alte formule simple (e.g., rezolvarea ecuației de gradul I sau II, calculul minimului sau maximului dintre două numere etc.), adăugarea unui element la sfârșitul unei liste, determinarea numărului de elemente dintr-o colecție, accesarea unui element al unei liste prin indexul său

b) **clasa $\mathcal{O}(\log_b n)$ – *complexitate logaritmică***

Exemple: suma cifrelor unui număr natural - $\mathcal{O}(\log_{10} n)$, operația de căutare binară a unei valori într-o listă sortată cu n elemente - $\mathcal{O}(\log_2 n)$

c) **clasa $\mathcal{O}(n)$ – *complexitate liniară***

Exemple: citirea/scrierea/o singură parcurgere a unui liste cu n elemente, testarea apartenenței unei valori la o listă cu n elemente sau un șir format din n caractere, numărarea aparițiilor unei valori într-o listă cu n elemente sau într-un șir format din n caractere

d) **clasa $\mathcal{O}(n \log_2 n)$**

Exemple: metodele de sortare *Quicksort* (sortarea rapidă), *Mergesort* (sortarea prin interclasare), *Timsort* (implementată în funcția `sort` din Python) și *Heapsort* (sortarea cu ansamble)

e) **clasa $\mathcal{O}(n^2)$ – *complexitate pătratică***

Exemple: metoda de sortare prin interschimbare, metoda de sortare Bubblesort, compararea fiecărui element al unui liste cu n elemente cu toate celelalte elemente din listă, citirea/scrierea/o singură parcurgere a unui matrice pătratice de dimensiune n

f) **clasa $\mathcal{O}(n^k), k \geq 3$ – *complexitate polinomială***

Exemple: sortarea fiecărei linii dintr-o matrice pătratică de dimensiune n folosind sortarea prin interschimbare sau Bubblesort - $\mathcal{O}(n^3)$, algoritmul Roy-Floyd-Warshall pentru determinarea drumurilor minime într-un graf orientat ponderat - $\mathcal{O}(n^3)$

g) **clasa $\mathcal{O}(a^n), a \geq 2$ – *complexitate exponențială***

Exemple: generarea tuturor submulțimilor unei mulțimi cu n elemente - $\mathcal{O}(2^n)$, partiționarea unei mulțimi cu n elemente în două submulțimi cu aceeași sumă a elementelor - $\mathcal{O}(2^n)$

Observații:

1. Complexitatea unui algoritm NU poate fi mai mică decât complexitatea citirii datelor de intrare și scrierii datelor de ieșire! De exemplu, complexitatea minimă a oricărui algoritm de generare a tuturor submulțimilor unei mulțimi cu n elemente este $\mathcal{O}(2^n)$, deoarece, indiferent de metoda de generare a submulțimilor, trebuie afișate cele 2^n submulțimi!

2. Complexitatea unei operații poate fi mai mică decât complexitatea unui algoritm care o implementează, deoarece un algoritm presupune citirea unor date de intrare și scrierea unor date de ieșire! De exemplu, operația de căutare binară a unei valori într-o listă sortată cu n elemente are complexitatea $\mathcal{O}(\log_2 n)$, dar un algoritm care utilizează această operație utilizează presupune și citirea listei sortate crescător, deci complexitatea algoritmului va fi $\mathcal{O}(n + \log_2 n) \approx \mathcal{O}(n)$.

3. Evident, există și alte clase de complexitate relativ des întâlnite, în afara celor menționate anterior. De exemplu, există clasa $\mathcal{O}(m + n)$ – interclasarea a două liste sortate crescător având m, respectiv n elemente; clasa $\mathcal{O}(m \cdot n)$ – parcurgerea unui tablou bidimensional cu m linii și n coloane; clasa $\mathcal{O}(m \cdot n \cdot p)$ – înmulțirea unei matrice cu m linii și n coloane cu o matrice cu n linii și p coloane etc.

4. Există complexități mai mari decât cea exponențială (e.g., generarea tuturor permutărilor unei mulțimi cu n elemente are complexitatea $\mathcal{O}(n!)$), dar algoritmii cu o astfel de complexitate sunt, în realitate, inutilizabili sau, mai precis, sunt utilizabili doar pentru valori foarte mici ale lui n.

În continuare, vom analiza din punct de vedere al complexității computaționale mai mulți algoritmi:

a) algoritmul de sortare prin selecția minimului

Instrucțiune	Operații elementare
`printf("n = ");`	1 afișare
`scanf("%d", &n);`	1 citire

`for(i = 0; i < n; i++)`	**de n ori:**	**2n operații elementare**
`{` `  printf("v[%d] = ", i);`	**1 afișare**	
`  scanf("%d", &v[i]);` `}`	**1 citire**	
`for(i = 0; i < n; i++)` `  for(j = i+1; j < n; j++)`	**de** $(n-1)+(n-2)+\ldots+1 = \frac{n(n-1)}{2}$ **ori:**	
`  if(v[i] > v[j])`	**1 operație de decizie**	**maxim 2n(n-1) operații elementare**

`  {`
`  aux = v[i];`
`  v[i] = v[j];`
`  v[j] = aux;`
`  }`
	maxim 3 atribuiri	

`printf("Tabloul sortat:\n");`	**n+1 afișări**	
`for(i = 0; i < n; i++)` `  printf("%d ", v[i]);`		
TOTAL:	**maxim $2n^2+n+3$ operații elementare**	

În concluzie, complexitatea acestui algoritm din punct de vedere al timpului de executare este $\mathcal{O}(2n^2+n+3) \approx \mathcal{O}(n^2)$, iar complexitatea din punct de vedere al memoriei utilizate este tot $\mathcal{O}(n)$, deoarece se memorează cele n valori în tabloul unidimensional v.

b) determinarea valorilor distincte (fără duplicate) dintr-un tablou unidimensional v format din n numere întregi

O variantă de rezolvare a acestei probleme constă în utilizarea unui tablou unidimensional auxiliar *dist* care să memoreze valorile distincte găsite până la un moment dat. Practic, parcurgem tabloul v element cu element şi dacă elementul curent $v[i]$ nu se găsește în tabloul *dist*, atunci îl adăugăm la sfârşitul său:

```
#include <stdio.h>

int main()
{
    int i, j, n, d, v[1000], dist[1000], gasit;

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

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

```
    d = 0;
    for(i = 0; i < n; i++)
    {
        gasit = 0;
        for(j = 0; j < d; j++)
            if(v[i] == dist[j])
            {
                gasit = 1;
                break;
            }

        if(gasit == 0)
        {
            dist[d] = v[i];
            d++;
        }
    }

    printf("Valorile distincte:\n");
    for(i = 0; i < d; i++)
        printf("%d ", dist[i]);

    return 0;
}
```

Se observă faptul că instrucţiunile marcate cu roşu induc complexitatea algoritmului, respectiv $\mathcal{O}(nd)$, unde d reprezintă numărul valorilor distincte din tabloul iniţial. Deoarece complexitatea unui algoritm trebuie exprimată doar în funcţie de dimensiunile datelor de intrare (iar valoarea d nu reprezintă o dimensiune a datelor de intrare!), vom aproxima complexitatea sa maximă prin $\mathcal{O}(n^2)$, care se obţine când numărul valorilor distincte d este aproximativ egal cu numărul n al valorilor din tabloul v. De asemenea, putem observa faptul că algoritmul are complexitatea $\mathcal{O}(n)$ când numărul valorilor distincte d este mult mai mic decât n (de exemplu, d va fi egal cu 10 dacă vom considera elementele tabloului v ca fiind doar cifre). Evident, complexitatea din punct de vedere al memoriei utilizate este $\mathcal{O}(n)$.

c) suma cifrelor unui număr natural n

Se observă faptul că numărul operaţiilor elementare efectuate este proporţional cu numărul c al cifrelor numărului n, deci este $\mathcal{O}(c)$. Complexitatea unui algoritm trebuie exprimată doar în funcţie de dimensiunile datelor de intrare, trebuie să calculăm numărul c în funcţie de numărul n. Presupunând faptul că numărul n are c cifre, rezultă că $10^{c-1} \leq n < 10^c$. Logaritmăm inegalitatea în baza 10 şi obţinem că $\log_{10} 10^{c-1} \leq \log_{10} c < \log_{10} 10^c$, deci $c - 1 \leq \log_{10} n < c$. Din ultima inegalitate rezultă că $[\log_{10} n] = c - 1$, deci $c = [\log_{10} n] + 1$. În concluzie, complexitatea acestui algoritm este $\mathcal{O}([\log_{10} n])$.

d) testarea primalității unui număr natural $n \geq 2$

Un algoritm simplu pentru testarea primalității unui număr natural n constă în verificarea faptului că el nu are niciun divizor propriu d cuprins între 2 și $\sqrt{n}$:

```
int prim(unsigned long long int n)
{
    unsigned long long int d;

    for(d = 2; d*d <= n; d++)
        if(n % d == 0)
            return 0;

    return 1;
}
```

Complexitatea acestui algoritm este $\mathcal{O}(\sqrt{n})$. Pentru a exprima complexitatea acestui algoritm în funcție de dimensiunile datelor de intrare, vom utiliza faptul că numărul de biți necesari pentru a reprezenta în memorie numărul n este $[\log_2 n] + 1$, deci $n \approx 2^{[\log_2 n]}$. În concluzie, complexitatea acestui algoritm este $\mathcal{O}(2^{[\log_2 n]/2})$, ceea ce reprezintă o complexitate exponențială în raport de lungimea reprezentării binare a numărului n. Pentru a va clarifica acest aspect, comparați timpul de executare al acestui algoritm pentru $n = 4294967291$ (un număr prim care se reprezintă pe 32 de biți) cu timpul de executare al algoritmului pentru $n = 18446744073709551557$ (un număr prim care se reprezintă pe 64 de biți)!