Curs 12

- **DTO**
- Complexitatea algoritmilor

Analiza complexității

Analiza complexități – studiul eficienței algoritmilor.

Eficiența algoritmilor în raport cu:

- timpul de execuție – necesar pentru rularea programului
- spațiu necesar de memorie

Timp de execuție, depinde de:

- algoritmul folosit
- datele de intrare
- hardware-ul folosit
- sistemul de operare (apar diferențe de la o rulare la alta).

Exemplu timp de execuție

```python
def fibonacci(n):
    """
     compute the fibonacci number
     n - a positive integer
     return the fibonacci number for a given n
    """
    #base case
    if n==0 or n==1:
        return 1
    #inductive step
    return fibonacci(n-1)+fibonacci(n-2)
```

```python
def fibonacci2(n):
    """
     compute the fibonacci number
     n - a positive integer
     return the fibonacci number for a given n
    """
    sum1 = 1
    sum2 = 1
    rez = 0
    for i in range(2, n+1):
        rez = sum1+sum2
        sum1 = sum2
        sum2 = rez
    return rez
```

```python
def measureFibo(nr):
    sw = StopWatch()
    print "fibonacci2(", nr, ") =", fibonacci2(nr)
    print "fibonacci2 take " +str(sw.stop())+" seconds"

    sw = StopWatch()
    print "fibonacci(", nr, ") =", fibonacci(nr)
    print "fibonacci take " +str(sw.stop())+" seconds"


measureFibo(32)
```

```
fibonacci2( 32 ) = 3524578
fibonacci2 take 0.0 seconds
fibonacci( 32 ) = 3524578
fibonacci take 1.7610001564 seconds
```

Eficiența algoritmilor

- Eficiența algoritmilor poate fi definită ca fiind cantitatea de resurse utilizate de algoritm (timp, memorie).

Măsurarea eficienței:

- analiză matematică a algoritmului - **analiză asimptotică**

Descrie eficiența sub forma unei funcții matematice.
Estimează timpul de execuție pentru toate intrările posibile.

- o analiză **empirică** a algoritmului

determinarea timpului exact de execuție pentru date specifice

nu putem prezice timpul pentru toate datele de intrare.

Timpul de execuție pentru un algoritm este studiat în relație cu dimensiunea datelor de intrare.

- Estimăm timpul de execuție în funcție de dimensiunea datelor.
- Realizăm o **analiză *asimptotică*.** Determinăm ordinul de mărime pentru resursa utilizată (timp, memorie), ne interesează în special pentru cazurile în care datele de intrare sunt mari

Complexitate

- **caz favorabil** - datele de intrare care conduc la timp de execuție minim
  - *best-case complexity* (BC): $BC(A) = \min_{I \in D} E(I)$
- **caz defavorabil –** date de intrare unde avem cel mai mare timp de execuție.
  - *worst-case complexity* (WC): $WC(A) = \max_{I \in D} E(I)$
- **caz mediu** - timp de execuție.
  - *average complexity* (AC): $AC(A) = \sum_{I \in D} P(I)E(I)$

A - algoritm; *E(I)* număr de operații; P*(I)* probabilitatea de a avea *I* ca și date de intrare
D – mulțimea tuturor datelor de intrare posibile pentru un n fixat

Obs. Dimensiunea datelor (n) este fixat (**un număr mare**) caz favorabil/caz defavorabil se referă la un **anumit aranjament al datelor** de intrare care produc timp minim/maxim

Complexitate timp de execuție

- **numărăm pași** (operații elementare) efectuați (de exemplu numărul de instrucțiuni, număr de comparații, număr de adunări).
- numărul de pași nu este un număr fixat, **este o funcție**, notat *T*(*n*), este in funcție de dimensiunea datelor (n), nu rezultă timpul exact de execuție
- Se surprinde doar esențialul: cum crește timpul de execuție în funcție de dimensiunea datelor. Ne oferă **ordinea de mărime** pentru timpul de execuție (dacă $n \to \infty$, atunci $3 \cdot n^2 \approx n^2$).
- putem **ignora constante mici** – dacă $n \to \infty$ aceste constante nu afectează ordinea de mărime.

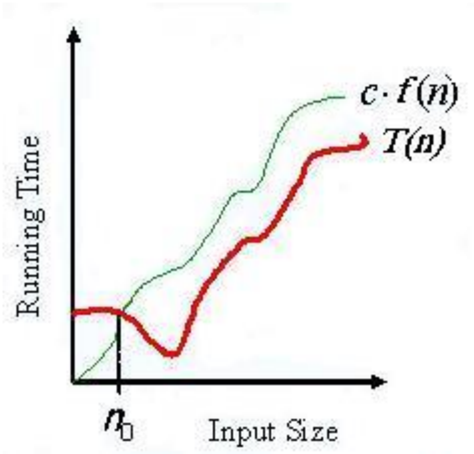
Ex : $T(n) = 13 \cdot n^3 + 42 \cdot n^2 + 2 \cdot n \cdot \log_2 n + 3 \cdot \sqrt{n}$

Fiindcă $0 < \log_2 n < n, \ \forall n > 1$ și $\sqrt{n} < n, \ \forall n > 1$, putem conclude că termenul n^3 domină această expresie când n este mare

Ca urmare, timpul de execuție a algoritmului crește cu ordinul lui n^3, ceea se scrie sub forma $T(n) \in O(n^3)$ și se citește “T(*n*) este de ordinul n^3

În continuare, vom nota prin *f* o funcție $f : N \to \Re$ și prin *T* funcția care dă complexitatea timp de execuție a unui algoritm, $T : N \to N$.

<u>Definiţia 1</u> (Notaţia O, "Big-oh"). Spunem că $T(n) \in O(f(n))$ dacă există **c** şi $\mathbf{n_0}$ constante pozitive (care nu depind de *n*) astfel încât $0 \le T(n) \le c \cdot f(n), \quad \forall n \ge n_0$.



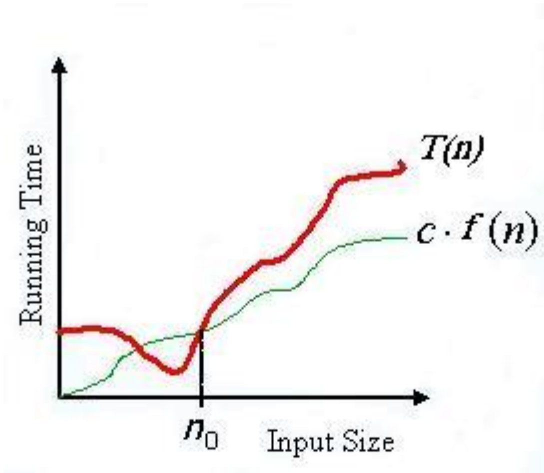
Cu alte cuvinte, notaţia *O* dă marginea superioară

Definiția alternativă: Spunem că $T(n) \in O(f(n))$ dacă $\lim\limits_{n\to\infty} \frac{T(n)}{f(n)}$ este 0 sau o constantă, dar **nu** ∞.

Observații.

1. Dacă $T(n) = 13 \cdot n^3 + 42 \cdot n^2 + 2 \cdot n \cdot \log_2 n + 3 \cdot \sqrt{n}$, atunci $\lim\limits_{n\to\infty} \frac{T(n)}{n^3} = 13$. Deci, putem spune că $T(n) \in O(n^3)$.
2. Notația O este bună pentru a da o limită superioară unei funcții. Observăm, totuși, că dacă $T(n) \in O(n^3)$, atunci este și $O(n^4)$, $O(n^5)$, etc atâta timp cât limita este 0. Din această cauză avem nevoie de o notație pentru limita inferioară a complexității. Această notație este Ω.

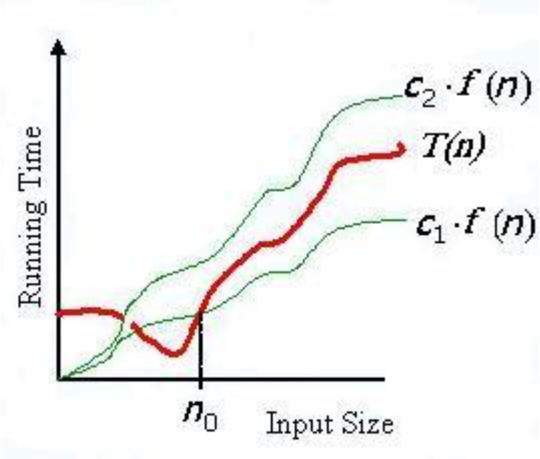
<u>Definiţia 2</u> (Notaţia Ω, "Big-omega"). Spunem că $T(n) \in \Omega(f(n))$ dacă există **c** şi **n_0** constante pozitive (care nu depind de n) astfel încât $0 \le c \cdot f(n) \le T(n), \quad \forall n \ge n_0$.



notația Ω dă marginea inferioară

Definiţia alternativă: Spunem că $T(n) \in \Omega(f(n))$ dacă $\lim_{n\to\infty} \frac{T(n)}{f(n)}$ este o constantă sau ∞, dar **<u>nu</u>** 0.

<u>Definiţia 3</u> (Notaţia θ, "Big-theta"). Spunem că $T(n) \in \theta(f(n))$ dacă $T(n) \in O(f(n))$ şi dacă $T(n) \in \Omega(f(n))$, altfel spus dacă există **c1, c2** şi **n_0** constante pozitive (care nu depind de *n*) astfel încât $c1 \cdot f(n) \le T(n) \le c2 \cdot f(n), \quad \forall n \ge n_0.$



notația θ mărginește o funcție până la factori constanți

Definiţia alternativă Spunem că $T(n) \in \theta(f(n))$ dacă $\lim_{n\to\infty} \frac{T(n)}{f(n)}$ este o constantă nenulă (dar **nu** 0 sau ∞).

Observaţii.

1. Timpul de execuție al unui algoritm este $\theta(f(n))$ dacă și numai dacă timpul său de execuție în cazul cel mai defavorabil este $O(f(n))$ și timpul său de execuție în cazul cel mai favorabil este $\Omega(f(n))$.
2. Notația $O(f(n))$ este de cele mai multe ori folosită în locul notației $\theta(f(n))$.
3. Dacă $T(n) = 13 \cdot n^3 + 42 \cdot n^2 + 2 \cdot n \cdot \log_2 n + 3 \cdot \sqrt{n}$, atunci $\lim_{n\to\infty} \frac{T(n)}{n^3} = 13$. Deci, $T(n) \in \theta(n^3)$. Acest lucru poate fi dedus şi din faptul că $T(n) \in O(n^3)$ şi $T(n) \in \Omega(n^3)$.

Sume

```
for i in range(0, n):
    #some instructions
```

presupunând că ceea ce este în corpul structurii repetitive (*) se execută în *f(i)* pași □ timpul de execuție al întregii structuri repetitive poate fi estimat astfel

$$T(n) = \sum_{i=1}^{n} f(i)$$

Se poate observa că, în cazul în care se folosesc bucle imbricate, vor rezulta sume imbricate. În continuare, vom prezenta câteva dintre sumele uzuale:

Calculul se efectuează astfel:

- se simplifică sumele – eliminăm constantele, separăm termenii in sume individuale
- facem calculul pentru sumele simplificate.

Exemple cu sume

Analizați complexitatea ca timp de execuție pentru următoarele funcții

<table>
<tr>
<td>

```
def f1(n):
    s = 0
    for i in range(1,n+1):
        s=s+i
    return s
```

</td>
<td>

$T(n) = \sum_{(i=1)}^{n} 1 = n \rightarrow T(n) \in \Theta(n)$

Complexitate (Overall complexity) $\Theta(n)$

Cazurile Favorabil/Mediu/Defavorabil sunt identice

</td>
</tr>
<tr>
<td>

```
def f2(n):
    i = 0
    while i<=n:
        #atomic operation
        i = i + 1
```

</td>
<td>

$T(n) = \sum_{(i=1)}^{n} 1 = n \rightarrow T(n) \in \Theta(n)$

Overall complexity $\Theta(n)$

Cazurile Favorabil/Mediu/Defavorabil sunt identice

</td>
</tr>
<tr>
<td>

```
def f3(l):
   """
   l - list of numbers
   return True if the list contains
an even nr
   """
   poz = 0
   while poz<len(l) and l[poz]%2 !=0:
       poz = poz+1
   return poz<len(l)
```

</td>
<td>

Caz favorabil:
primul element e număr par: $T(n) = 1 \in \Theta(1)$
Caz defavorabil:
Nu avem numere pare în listă: $T(n) = n \in \Theta(n)$
Caz mediu:
While poate fi executat 1,2,..n ori (același probabilitate).
Numărul de pași = numărul mediu de iterații

$T(n) = (1 + 2 + \ldots + n)/n = (n + 1)/2 \rightarrow T(n) \in \Theta(n)$

Complexitate $O(n)$

</td>
</tr>
</table>

Exemple cu sume

<pre>def f4(n):
    for i in range(1,2*n-2):
        for j in range(i+2,2*n):
            #some computation
            pass</pre>
	$T(n) = \sum_{(i=1)}^{(2n-2)} \sum_{(j=i+2)}^{2n} 1 = \sum_{(i=1)}^{(2n-2)} (2n - i - 1)$
	
	$T(n) = \sum_{(i=1)}^{(2n-2)} 2n - \sum_{(i=1)}^{(2n-2)} i - \sum_{(i=1)}^{(2n-2)} 1$
	
	$T(n) = 2n \sum_{(i=1)}^{(2n-2)} 1 - (2n-2)(2n-1)/2 - (2n-2)$
	…
	$T(n) = 2n^2 - 3n + 1 \in \Theta(n^2)$ Overall complexity $\Theta(n^2)$

<pre>def f5():
    for i in range(1,2*n-2):
        j = i+1
        cond = True
        while j<2*n and cond:
         #elementary operation
         if someCond:
                cond = False</pre>
	Caz favorabil: While se execută odată $T(n) = \sum_{(i=1)}^{(2n-2)} 1 = 2n - 2 \in \Theta(n)$
	Caz defavorabil: While executat $2n - (i+1)$ori
	$T(n) = \sum_{(i=1)}^{(2n-2)} (2n - i - 1) = \ldots = 2n^2 - 3n + 1 \in \Theta(n^2)$
	Caz mediu: Pentru un i fixat While poate fi executat 1,2..2n-i-1 ori număr mediu de pași:$C_i = (1 + 2 + \ldots + 2n - i - 1)/2n - i - 1 = \ldots = (2n - i)/2$
	$T(n) = \sum_{(i=1)}^{(2n-2)} C_i = \sum_{(i=1)}^{(2n-2)} (2n - i)/2 = \ldots \in \Theta(n^2)$
	
	Overall complexity $O(n^2)$

Formule cu sume:

$$\sum_{i=1}^{n} 1 = n$$

suma constantă.

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

suma liniară (progresia aritmetică)

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

suma pătratică

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

suma armonică

$$\sum_{i=1}^{n} c^i = \frac{c^{n+1}-1}{c-1}, \quad c \neq 1$$

progresia geometrică (crește exponențial)

Complexităţi uzuale

$T(n) \in O(1)$ **- timp *constant*.** Algoritmul se executa in timp constant indiferent de dimensiunea datelor.

$T(n) \in O(\log_2 \log_2 n)$ - timp foarte rapid (aproape la fel de rapid ca un timp constant)

$T(n) \in O(\log_2 n)$ **-** complexitate *logaritmică*:
timp foarte bun (este ceea ce căutăm, în general, pentru orice algoritm);

$\log_2 1000 \approx 10, \quad \log_2 1.000.000 \approx 20$;

complexitate căutare binară, înălțimea unei arbore binar echilibrat

$T(n) \in O((\log_2 n)^k)$ **-** unde k este factor constant; se numește complexitate *polilogaritmică* (este destul de bună);

Complexităţi uzuale

$T(n) \in O(n)$	- complexitate *liniară;*
$T(n) \in O(n \cdot \log_2 n)$	- este o complexitate faimoasă, întâlnită mai ales la sortări (MergeSort, QuickSort);
$T(n) \in O(n^2)$	- *este complexitate pătratică (cuadratică); dacă n este de ordinul milioanelor, nu este prea bună;*
$T(n) \in O(n^k)$	- *unde k este constant; este complexitatea polinomială (este practică doar daca k nu este prea mare);*
$T(n) \in O(2^n), O(n^3), O(n!)$	- complexitate *exponențială* (algoritmii cu astfel de complexități sunt practici doar pentru valori mici ale lui *n*: $n \le 10, n \le 20$).

Recurențe

O **recurență** este o formulă matematică definită recursiv.

Ex. numărul de noduri (notat *N(h)*) dintr-un arbore ternar complet de înălțime *h* ar putea fi descris sub forma următoarei formule de recurență:

$$\begin{cases} N(0) = 1 \\ N(h) = 3 \cdot N(h-1) + 1, \qquad h \geq 1 \end{cases}$$

Explicația ar fi următoarea:

- Numărul de noduri dintr-un arbore ternar complet de înălțime 0 este 1.
- Numărul de noduri dintr-un arbore ternar complet de înălțime *h* se obține ca fiind de 3 ori numărul de noduri din subarborele de înălțime *h-1*, la care se mai adaugă un nod (rădăcina arborelui).

Dacă ar fi să rezolvăm recurența, am obține că numărul de noduri din arborele ternar complet de înălţime *h* este $N(h) = 3^h \cdot N(0) + (1 + 3^1 + 3^2 + ... + 3^{h-1}) = \sum_{i=0}^{h} 3^i$

Exemple

```python
def recursiveSum(l):
    """Compute the sum of numbers
    l - list of number
    return the sum of numbers """
    #base case
    if l==[]:
        return 0
    #inductive step
    return l[0]+recursiveSum(l[1:])
```

Recurrence: $T(n) = \begin{cases} 1 for n = 0 \\ T(n-1) + 1 otherwise \end{cases}$

$$\begin{aligned} T(n) &= T(n-1) + 1 \\ T(n-1) &= T(n-2) + 1 \\ T(n-2) &= T(n-3) + 1 \Rightarrow T(n) = n + 1 \in \Theta(n) \\ \ldots &= \ldots \\ T(1) &= T(0) + 1 \end{aligned}$$

```python
def hanoi(n, x, y, z):
    """
       n -number of disk on the x
stick
       x - source stick
       y - destination stick
       z - intermediate stick
    """
    if n==1:
      print ("disk 1 from",x,"to",y)
      return
    hanoi(n-1, x, z, y)
    print ("disk ",n,"from",x,"to",y)
    hanoi(n-1, z, y, x)
```

Recurrence: $T(n) = \begin{cases} 1 for n = 1 \\ 2T(n-1) + 1 otherwise \end{cases}$

$$\begin{aligned} T(n) &= 2T(n-1) + 1 \\ T(n-1) &= 2T(n-2) + 1 = \\ T(n-2) &= 2T(n-3) + 1 \\ \ldots &= \ldots \\ T(1) &= T(0) + 1 \end{aligned}$$

$\Rightarrow$

$$\begin{aligned} T(n) &= 2T(n-1) + 1 \\ 2T(n-1) &= 2^2T(n-2) + 2 \\ 2^2T(n-2) &= 2^3T(n-3) + 2^2 \\ \ldots &= \ldots \\ 2^{(n-2)}T(2) &= 2^{(n-1)}T(1) + 2^{(n-2)} \end{aligned}$$

$$T(n) = 2^{(n-1)} + 1 + 2 + 2^2 + 2^3 + \ldots + 2^{(n-2)}$$

$$T(n) = 2^n - 1 \in \Theta(2^n)$$

Complexitatea spațiu de memorare

Complexitatea unui algoritm din punct de vedere al *spațiului de memorare* estimează cantitatea de memorie necesară algoritmului pentru stocarea datelor de intrare, a rezultatelor finale şi a rezultatelor intermediare. Se estimează, ca și *timpul de execuție* al unui algoritm, în notațiile O, Θ, Ω.

Toate observațiile referitoare la notația asimptotică a complexității ca timp de execuție sunt valabile și pentru complexitatea ca spațiu de memorare.

Exemplu

Analizați complexitatea ca spațiu de memorare pentru următoarele funcții

<table>
<tr>
<td>

```
def iterativeSum(l):
    """
    Compute the sum of numbers
    l - list of number
    return int, the sum of numbers
    """
    rez = 0
    for nr in l:
        rez = rez+nr
    return rez
```

</td>
<td>

Avem nevoie de spațiu pentru numerele din listă

$T(n) = n \in \Theta(n)$

</td>
</tr>
<tr>
<td>

```
def recursiveSum(l):
    """
    Compute the sum of numbers
    l - list of number
    return int, the sum of numbers
    """
    #base case
    if l==[]:
        return 0
    #inductive step
    return l[0]+recursiveSum(l[1:])
```

</td>
<td>

Recurență: $T(n) = \begin{cases} 0 & for\ n = 1 \\ T(n-1) + 1 & otherwise \end{cases}$

</td>
</tr>
</table>

Analiza complexității (timp/spațiu) pentru o funcție

1 Dacă există caz favorabil/defavorabil:

- descrie **Caz favorabil**
- calculează complexitatea pentru **Best Case**
- descrie **Worst Case**
- calculează complexitatea pentru **Worst case**
- calculează complexitatea **medie**
- calculează complexitatea **generală**

2 Dacă Favorabil = Defavorabil = Mediu – (nu avem cazuri favorabile/defavorabile)

- calculează complexitatea

Calculează complexitatea:

- dacă avem recurență
  - calculează folosind egalități
- altfel
  - calculează folosind sume