

Fundamentele programării

Recursivitate

O funcție recursivă: este o funcție care se auto-apliquează.

```
def factorial(n):
    """
        compute the factorial
        n is a positive integer
        return n!
    """
    if n == 0:
        return 1
    return factorial(n-1)*n
```

- Recursivitate directă: P apelează P
- Recursivitate indirectă: P apelează Q, Q apelează P

Cum rezolvăm probleme folosind recursivitatea

- Definim cazul de bază: soluția cea mai simplă adică punctul unde se oprește apelul recursiv.
 - Pas inductiv: împărțim problema într-o variantă mai simplă al același probleme plus ceva puși simpli;
- ex: apel cu $m-1$ săm donec apeluri cu $m/2$

```
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:])
```

```
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)
```

Recurzivitate în Python

- la fiecare apel de metodă se crează un nou namespace.
Acesta conține valorile pentru parametri și variabilele locale.
- tabula de simboluri este salvată pe stack, iar când apelul se termină tabula se elimină din stivă

```
def isPalindrome(str):  
    """  
    Verify if a string is a palindrome  
    str - string  
    return True if the string is a palindrome False otherwise  
    """  
    dict = locals()  
    print (id(dict),dict)  
  
    if len(str)==0 or len(str)==1:  
        return True  
  
    return str[0]==str[-1] and isPalindrome(str[1:-1])
```

Recurzivitate :

- avantaje :
 - claritate
 - cod mai simplu
- dezavantaje :
 - consum de memorie mai mare (pt fiecare recursiune creză o nouă tabulă de simboluri)

Analiza complexității

Eficiența algoritmilor în raport cu:

- timpul de execuție
- spațiul de memorie

Timpul de execuție, depinde de:

- algoritmul folosit
- datele de intrare
- hardware-ul folosit
- sistemul de operare

Exemplu timp de execuție.

<pre>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)</pre>	<pre>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</pre>
<pre>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"</pre>	
<pre>measureFibo(32) fibonacci2(32) = 3524578 fibonacci2 take 0.0 seconds fibonacci(32) = 3524578 fibonacci take 1.7610001564 seconds</pre>	

Eficiența algoritmilor

Este definită ca fiind cantitatea de resurse utilizate de algoritm
(Timp, memorie)

Măsurarea eficienței:

- analiză matematică (asimptotică) a algoritmului:
descrie eficiența sub formă unei funcții matematice
estimează timpul de execuție pentru toate intervalele posibile
- analiză empirică a algoritmului:
determină timpul exact de execuție pentru date specifice
nu se poate prezice timpul pentru toate datele de intrare

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

- Estimăm timpul de execuție în funcție de dimensiunea datelor.
- Realizăm o analiză asimptotică și determinăm ordinul de
mărime pentru resurse utilizate (Timp, memorie)

Complexitate

- **Caz favorabil** - datele de intrare conduc la timp de execuție minim

$$BC(A) = \min_{I \in D} E(I)$$

- **Caz nefavorabil** - datele de intrare conduc la timp de execuție maxim

$$WC(A) = \max_{I \in D} E(I)$$

- **Caz mediu** - Timp de execuție

(average complexity) $AC(A) = \sum_{I \in D} P(I) E(I)$

A - algoritm, $E(I)$ - numărul de operații,

$P(I)$ - probabilitatea de a avea I ca și date de intrare

D - multimea tuturor datelor de intrare

Complexitate timp de execuție

- numărul pasării (nr de instrucțiuni, comparări, adunări, etc.)
- numărul de pași nu este un număr fixat, este o funcție $T(n)$, este în funcție de dimensiunea datelor n , nu rezultă timpul exact de execuție
- se surprinde deosebită esențială: cum crește timpul de execuție în funcție de dimensiunea datelor

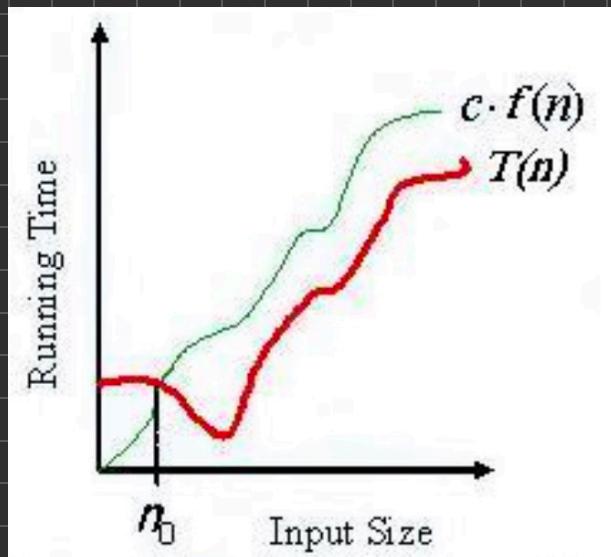
• putem ignora constantele mici

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

$0 < \log_2 n < n$, $\sqrt{n} < n$ și $n > 1 \Rightarrow n^3$ domină expresia când n este mare, ca urmare timpul de execuție crește cu ordinul lui n^3 , motivul: $T(n) \in O(n^3)$ și se cunoaște $T(n)$ este de ordinul n^3

În continuare, avem o funcție $f: N \rightarrow R$ și $T: N \rightarrow N$ care dă complexitatea timp de execuție a unui algoritm.

Def: (Notatia O , "Big-oh"). Să spunem că $T(n) \in O(f(n))$ dacă există c și n_0 constante positive a.s. $0 \leq T(n) \leq c \cdot f(n)$, $\forall n \geq n_0$.



Notatia O dă marginea superioară

Def alternativă: Spunem că: $T(m) \in O(f(m))$ dacă \exists

$$\lim_{m \rightarrow \infty} \frac{T(m)}{f(m)} < +\infty$$

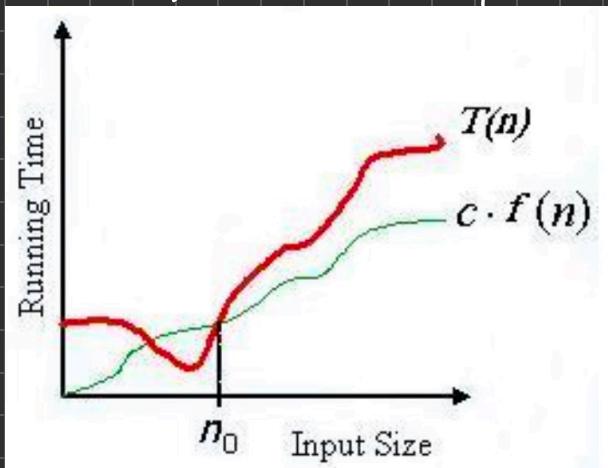
Opozitie:

- Dacă $T(m) = 13 \cdot m^3 + 42 \cdot m^2 + 2 \cdot m \cdot \log_2 m + 3 \cdot \sqrt{m}$, atunci $\lim_{m \rightarrow \infty} \frac{T(m)}{m^3} = 13$

Deci putem spune că $T(m) \in O(m^3)$

- Notatia O este bună pentru că dă o limită superioară a funcției. Totuși, dacă $T(m) \in O(m^3)$, atunci este și $O(m^4)$, $O(m^5)$, etc. pînă la limitele de O . Din această cauză avem nevoie de o notăție pentru limită inferioară a complexității, pe care o vom cîndea cu Ω .

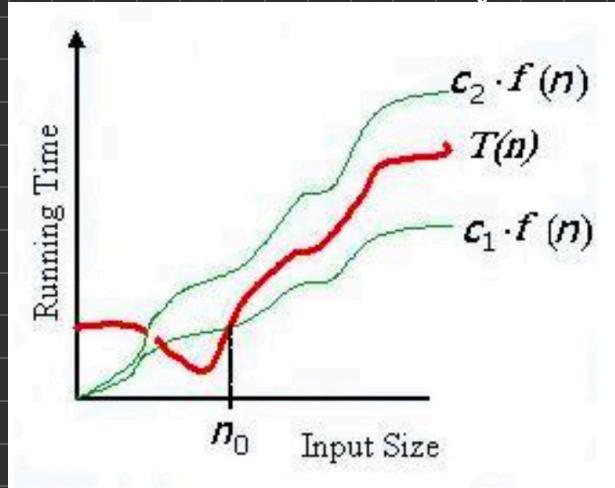
Def (Notatia Ω , "Big-omega"): Spunem că $T(m) \in \Omega(f(m))$ dacă există $c > 0$ și m_0 constante pozitive c.î. $0 \leq c \cdot f(m) \leq T(m)$, $\forall m \geq m_0$



notatia Ω de marginime inferioare

Dif alternativă: Spunem că: $T(m) \in \Omega(f(m))$ dacă \exists
 $\lim_{m \rightarrow \infty} \frac{T(m)}{f(m)} > 0$

Dif (Notatia Θ , "Big-theta"). Spunem că $T(m) \in \Theta(f(m))$ dacă
 $T(m) \in O(f(m))$ și $T(m) \in \Omega(f(m))$, astfel spus, există c_1, c_2 și m_0
constante positive c.i. $c_1 \cdot f(n) \leq T(n) \leq c_2 \cdot f(n)$, $\forall n \geq m_0$.



notatia Θ mărgineste o funcție până la factori constanți

Dif alternativă: Spunem că: $T(m) \in \Theta(f(m))$ dacă \exists

$$\lim_{m \rightarrow \infty} 0 < \frac{T(m)}{f(m)} < +\infty$$

Obs:

1. Timpul de execuție este $\Theta(f(m))$ dacă timpul algoritmului de execuție este $O(f(m))$ în cazul cel mai dezfavorabil și $\Omega(f(m))$ în cazul cel mai favorabil.

2. Notația $O(f(m))$ este de cele mai multe ori folosită în analiza mărităi $\Theta(f(m))$.

3. Dacă $T(m) = 13 \cdot m^3 + 42 \cdot m^2 + 2 \cdot m \cdot \log_2 m + 3\sqrt{m}$ atunci
 $\lim_{m \rightarrow \infty} \frac{T(m)}{m^3} = 13$, deci $T(m) \in \Theta(m^3)$. Acest lucru este deoarece din $T(m) \in O(m^3)$ și $T(m) \in \Omega(m^3)$.

Sume

for i in range ($0, n$):

instructions.

Presupunând că ceea ce este în corpul structurii repetitive se execută în $f(i)$ pași, timpul de execuție poate fi estimat astfel $T(m) = \sum_{i=1}^n f(i)$

Exemplu cu sume:

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

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

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

Cazurile Favorabil/Mediu/Defavorabil sunt identice

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

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

Overall complexity $\Theta(n)$

Cazurile Favorabil/Mediu/Defavorabil sunt identice

```
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)
```

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 iterării

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

Complexitate $O(n)$

```
def f4(n):
    for i in range(1, 2*n-2):
        for j in range(i+2, 2*n):
            #some computation
            pass
```

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

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

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

...

$$T(n) = 2n^2 - 3n + 1 \in \Theta(n^2) \quad \text{Overall complexity } \Theta(n^2)$$

```
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
```

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

$$T(n) = \sum_{(i=1)}^{(2n-2)} (2n - i - 1) = \dots = 2n^2 - 3n + 1 \in \Theta(n^2)$$

Caz mediu: Pentru un i fixat While poate fi executat 1,2.. $2n-i-1$ ori număr mediu de pași: $C_i = (1 + 2 + \dots + 2n - i - 1)/2n - i - 1 = \dots = (2n - i)/2$

$$T(n) = \sum_{(i=1)}^{(2n-2)} C_i = \sum_{(i=1)}^{(2n-2)} (2n - i)/2 = \dots \in \Theta(n^2)$$

Overall complexity $O(n^2)$

Formule cu sume.

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

Suma constantă

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

Suma liniară

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

Suma pătratică

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

Suma armonică

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

progressia geometrică

Complexitatea uzuală

$T(n) \in O(1)$ - timp constant

$T(n) \in O(\log_2 \log_2(n))$ - timp foarte rapid

$T(n) \in O(\log_2 n)$ - complexitate logarithmică

$T(n) \in O((\log_2 n)^k)$ - complexitate polilogaritmică

$T(n) \in O(n)$ - complexitate liniară

$T(n) \in O(n \log_2 n)$ - (MergeSort, QuickSort)

$T(n) \in O(n^2)$ - complexitate patratice

$T(n) \in O(n^k)$ - complexitate polinomială

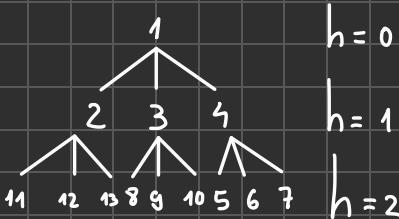
$T(n) \in O(2^n)$, $O(n^3)$, $O(n!)$ - complexitate exponențială

Recurențe

O recurență este o formulă matematică definită recursiv.

Ex: numărul de moduri în care se poate construi un arbore cu 3 noduri

complet de înalțime h poate să fie descris folosind o formulă de recurență astfel:



$$\left\{ \begin{array}{l} N(0) = 1 \\ \end{array} \right.$$

$$\left\{ \begin{array}{l} N(h) = 3 \cdot N(h-1) + 1, \quad h \geq 1 \\ \end{array} \right.$$

$$\Rightarrow N(h) = 3^0 + 3^1 + 3^2 + \dots + 3^h = \sum_{i=0}^h 3^i$$

Exemplu:

```
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:])
```

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

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

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

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

Complexitate spațiu de memorie

Estimarea cantității de memorie necesară algoritmului pentru stocarea datelor de intrare, a rezultatelor finale și ale intermedioare.

Să estimăm ca și timpul de execuție, în notăția O, Θ, Ω .

Exemplu:

```
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
```

```
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:])
```

Avem nevoie de spațiu pentru numerele din listă

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

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

Analiza complexității (temp / spațiu) pentru o funcție

1. Dacă există caz favorabil / unfavorabil.

• descrie BC

• calculează complexitatea pentru BC

• descrie WC

• calculează complexitatea pentru WC

• calculează complexitatea medie

• calculează complexitatea generală

2. Dacă $BC = WC = AC$

• calculează complexitatea

Calculul complexității:

• dacă avem recurență calculăm folosind egalitatea

• altfel calculăm folosind sume

Exemplu:

for i in range(1, n):

 for j in range(i, n):

 count += 1

$$\sum_{i=1}^{m-1} \sum_{j=i}^{m-1} 1 = \sum_{i=1}^{m-1} m-i = \sum_{i=1}^{m-1} m - \sum_{i=1}^{m-1} i = m(m-1) - \frac{m(m-1)}{2} = \frac{m(m-1)}{2}$$

$$= \frac{m^2-m}{2} \in \Theta(m^2)$$