

M. Caramihai, © 2020

**STRUCTURI DE DATE
& ALGORITMI**

CURS 3

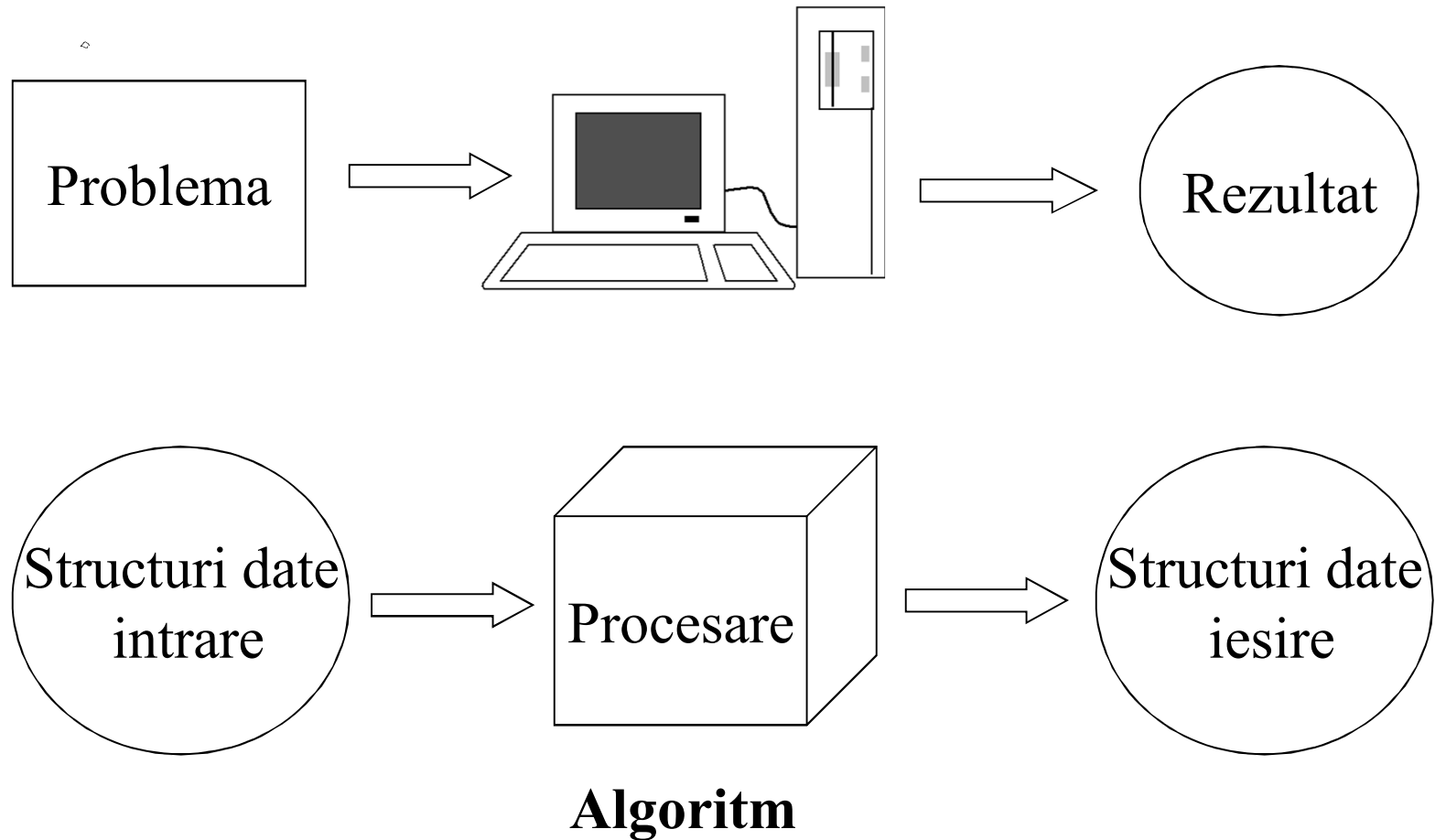
Analiza (temporală a) algoritmilor

Obiectivele programarii calculatoarelor

Aspecte antagonice:

- Proiectarea de algoritmi usor de inteles, codificat, depanat (**software engineering**)
- Proiectarea de algoritmi ce eficientizeaza resursele de calcul (**analiza algoritmilor**)
 - **Intrebare:** cum poate fi evaluat "costul unui algoritm"?
- Solutii:
- **Comparare empirica:** prin rularea pgm → dezav: consum mare de timp
- **Analiza asimptotica a algoritmilor:** determ. in fct de numarul operatiilor de baza si a marimii datelor de intrare
- **Observatie:** pentru a compara 2 algoritmi, acestia trebuie sa ruleze pe o aceeaasi platforma hw.

Algoritmi: pozitionare, *design*



Principalele proprietati ale algoritmilor

□ Corectitudine

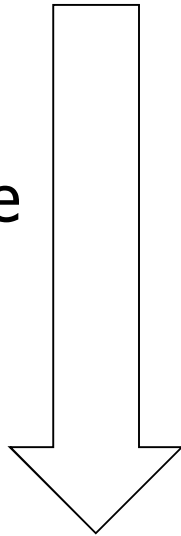
- Intoarce *intotdeauna* rezultatul (dorit) la iesire in cazul in care au fost respectate cerintele de structura.

□ Eficienta

- Se masoara in unitati de **timp** sau **spatiu**
- Timpul este elementul cel mai important
- Analiza modului de "rulare" permite cresterea eficientei algoritmilor.

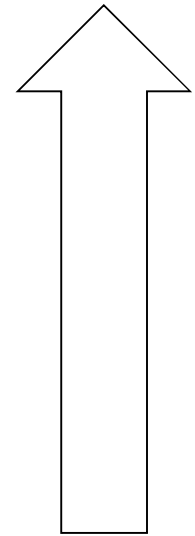
Exprimarea algoritmilor

Mai usor de
exprimat



- Limbajul curent
- Pseudocod
- Limbaj de programare de nivel înalt

Mai
precis



Pseudocod:

```
nume algoritm
      input
      output
corp algoritm
.....
end.
```

Principalele proprietati ale pseudocodului

- O facilitate de exprimare a algoritmilor
- Nu intra in detaliile de implementare
- Accent pe esenta algoritmului (exemplu):

Algorithm ArrayMax(A,n)

Input: Vector A ce stocheaza $n \geq 1$ intregi

Output: Cea mai mare valoare din A.

currentMax \leftarrow A[0]

for i \leftarrow 1 **to** n-1 **do**

if currentMax < A[i] **then**

 currentMax \leftarrow A[i]

return currentMax

Analiza algoritmilor (1)

- **Analiza:** predictia cu privire la resursele pe care un algoritm le necesita
- **Resurse:** memorie, largime banda comunicatii, **timp de lucru**

- La ce bun o analiza a algoritmilor?
 - Evalueaza performanta algoritmilor
 - Compara diferiti algoritmi

- Ce se analizeaza ?
 - Timpul de rulare, gradul de utilizare a memoriei
 - Bune practici / rele practici

Observatie: analiza algoritmilor compara algoritmi si nu programe !

Analiza algoritmilor (2)

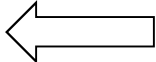
- Dacă fiecare linie (a unui algoritm) consuma un anumit buget (constant) de timp, întregul algoritm va consuma un buget (constant) de timp ?

Nu !

- Majoritatea algoritmilor au un număr de pași dependent de mărimea instanțelor (*instances*)
- Eficiența unui algoritm este întotdeauna funcție de dimensiunea problemei.
 - Se utilizează în general variabila N pentru reprezentarea dimensiunii problemei

Analiza algoritmilor (3)

Ce afecteaza timpul de rulare al unui algoritm?

- structura hw a sistemului de calcul
- compilatorul
- tipul algoritmului
- facilitatile pentru utilizator
- **numarul de intrari ale algoritmului** 
- efortul de programare
- etc...

Analiza algoritmilor poate fi facuta

- A priori (analiza teoretica) – ipoteza: efic alg depinde doar de nr intrari
- A posteriori (analiza empirica) – se analizeaza exact cum ruleaza un alg pe o masina

Complexitatea algoritmilor (1)

Complexitatea algoritmilor:

- Timp (numar de operatii de baza efectuate)
- Spatiu: evaluarea sp de memorie necesar rularii unui algoritm

□ **Cazul nefericit:**

- Functia definita de numarul *maxim* de pasi preluati din orice instanta de marime n

□ **Cazul fericit:**

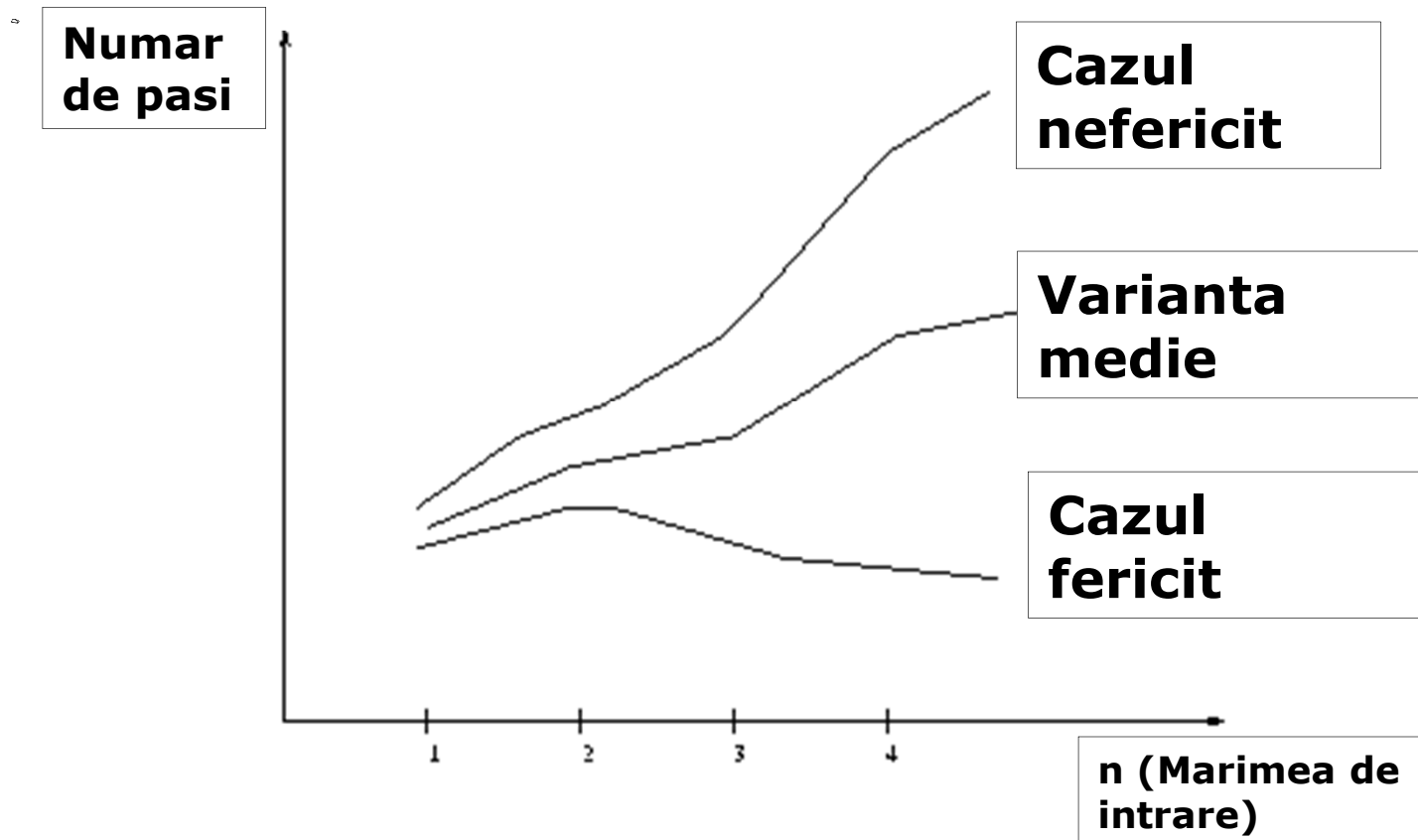
- Functia definita de numarul *minim* de pasi preluati din orice instanta de marime n

□ **Varianta medie:**

- Functia definita de numarul *mediu* de pasi preluati din orice instanta de marime n

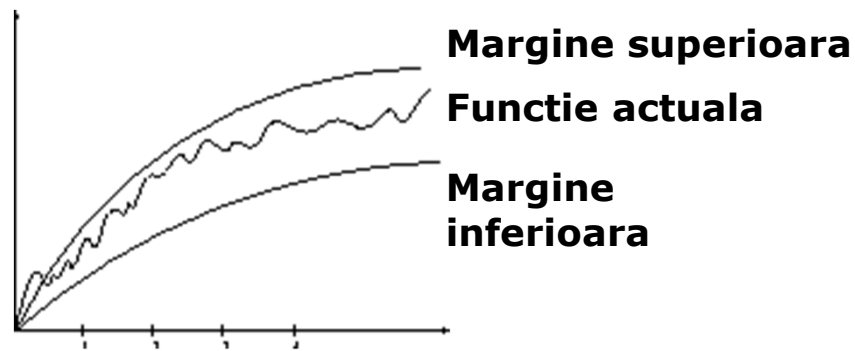
Observatie: limitele se refera la algoritm si **nu** la program.

Complexitatea algoritmilor (2)



Cum se face analiza ?

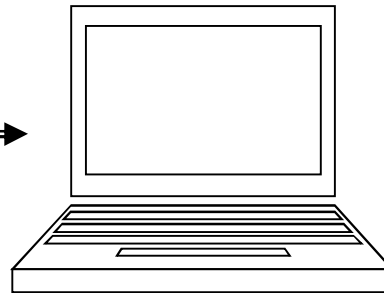
- In general, este greu de estimat timpul exact de rulare a unui algoritm
 - Cazul fericit depinde de intrare
 - Varianta medie este greu de calculat
 - Se recomanda focalizarea pe cazul nefericit:
 - Mai usor de calculat
 - Uzual – apropiat de timpul de lucru curent
- **Strategie:** se cauta sa se gaseasca marginile superioara / inferioara ale cazului nefericit.



Analiza timpului de rulare (1)

n date intrare

```
10001010101000111110001100011  
1010101010101010100100010101  
0100010000000000001111010100  
0111010
```

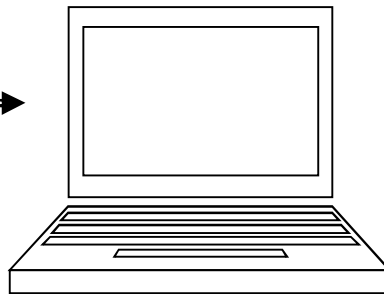


Intoarce rezultatul
in timpul $T1(n)$

Algoritm 1

n date intrare

```
10001010101000111110001100011  
1010101010101010100100010101  
0100010000000000001111010100  
0111010
```



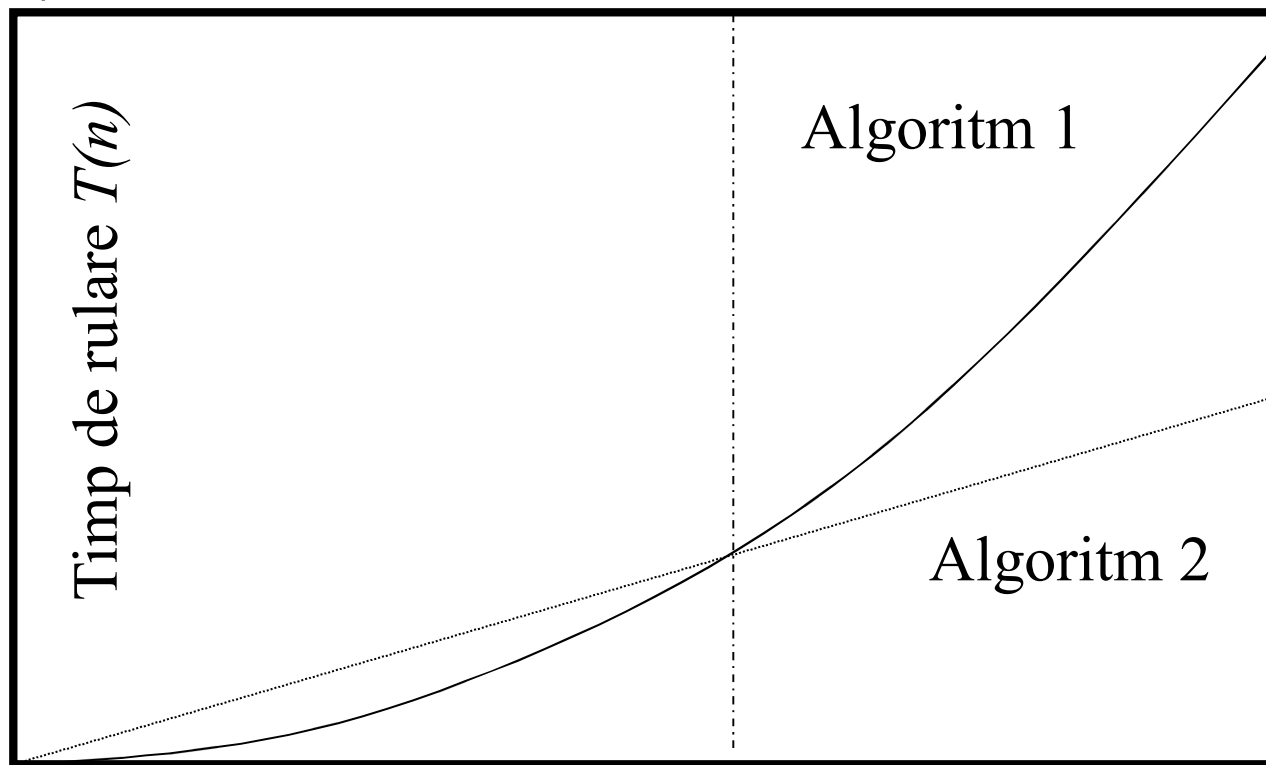
Intoarce
rezultatul
in timpul $T2(n)$

Algoritm 2

Intrebare: un algoritm este dependent de masina?

NU: analiza timpului de rulare trebuie facuta independent de masina

Analiza timpului de rulare (2)



n_0

Numar date de intrare n
Dimensiunea problemei

Compararea algoritmilor

- Stabilirea unei ordini relative între diferenti algoritmi în raport cu ratele lor de creștere.
- Ratele de creștere sunt funcții dependente (în general) de numărul de intrări, n .

Analiza asimptotica

- Analiza asimptotica (**AA**) a unui algoritm ce descrie eficienta relativa a acestuia in conditiile in care n este foarte mare.
- Atunci cand numarul datelor de intrare este mic, precizia scade
- d.e.: pentru marimi mari ale lui n , algoritmul 1 creste mult mai repede decat algoritmul 2.

De remarcat ca aici se compara algoritmi. In practica, atunci cand se scriu programe de mici dimensiuni, analiza asimptotica nu mai este importanta.

O comparatie simpla

- Fie 3 algoritmi de sortare a unor liste:
 - $f(n) = n \log_2 n$
 - $g(n) = n^2$
 - $h(n) = n^3$
- **Ipoteza:** fiecare pas presupune un timp de calcul de 1 microsecunda (10^{-6} sec)

<i>n</i>	<i>n log n</i>	<i>n²</i>	<i>n³</i>
10	33.2	100	1000
100	664	10000	1s
1000	9966	1s	16min
100000	1.7s	2.8 ore	31.7 ani

O alta comparatie

FIGURE 2.1 The growth rate of all terms of function $f(n) = n^2 + 100n + \log_{10}n + 1,000$.

n	f(n)	n ²		100n		log ₁₀ n		1,000	
	Value	Value	%	Value	%	Value	%	Value	%
1	1,101	1	0.1	100	9.1	0	0.0	1,000	90.83
10	2,101	100	4.76	1,000	47.6	1	0.05	1,000	47.60
100	21,002	10,000	47.6	10,000	47.6	2	0.001	1,000	4.76
1,000	1,101,003	1,000,000	90.8	100,000	9.1	3	0.0003	1,000	0.09
10,000	101,001,004	100,000,000	99.0	1,000,000	0.99	4	0.0	1,000	0.001
100,000	10,010,001,005	10,000,000,000	99.9	10,000,000	0.099	5	0.0	1,000	0.00

O abordare teoretica

□ “Big-Oh”

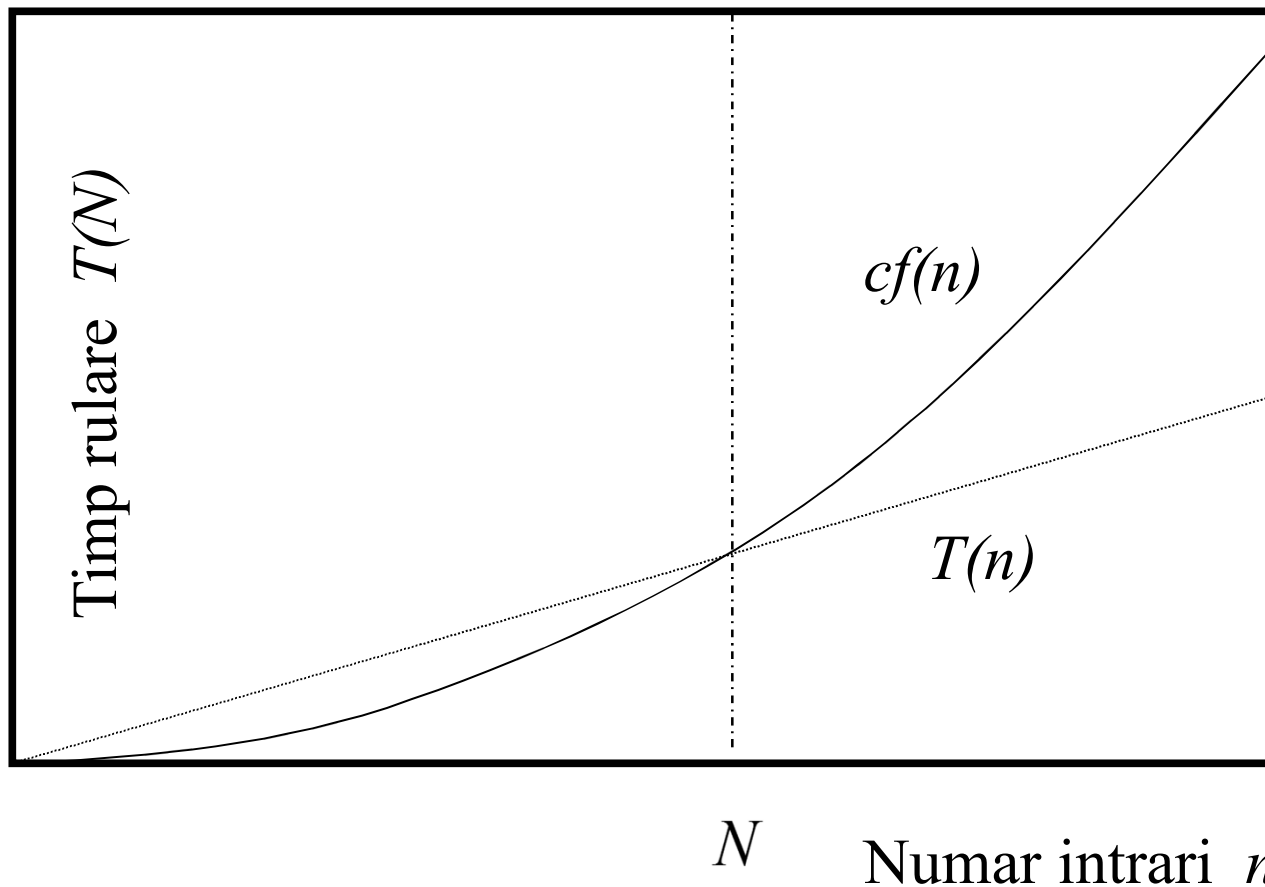
□ **Definitie:**

$T(n) = O(f(n))$ daca exista doua constante pozitive c si N a.i. $T(n) \leq c f(n)$ cand $n \geq N$

□ Cu alte cuvinte: functia $T(n)$ are o viteza de crestere mai mica decat $f(n)$; astfel $f(n)$ este limita superioara a lui $T(n)$.

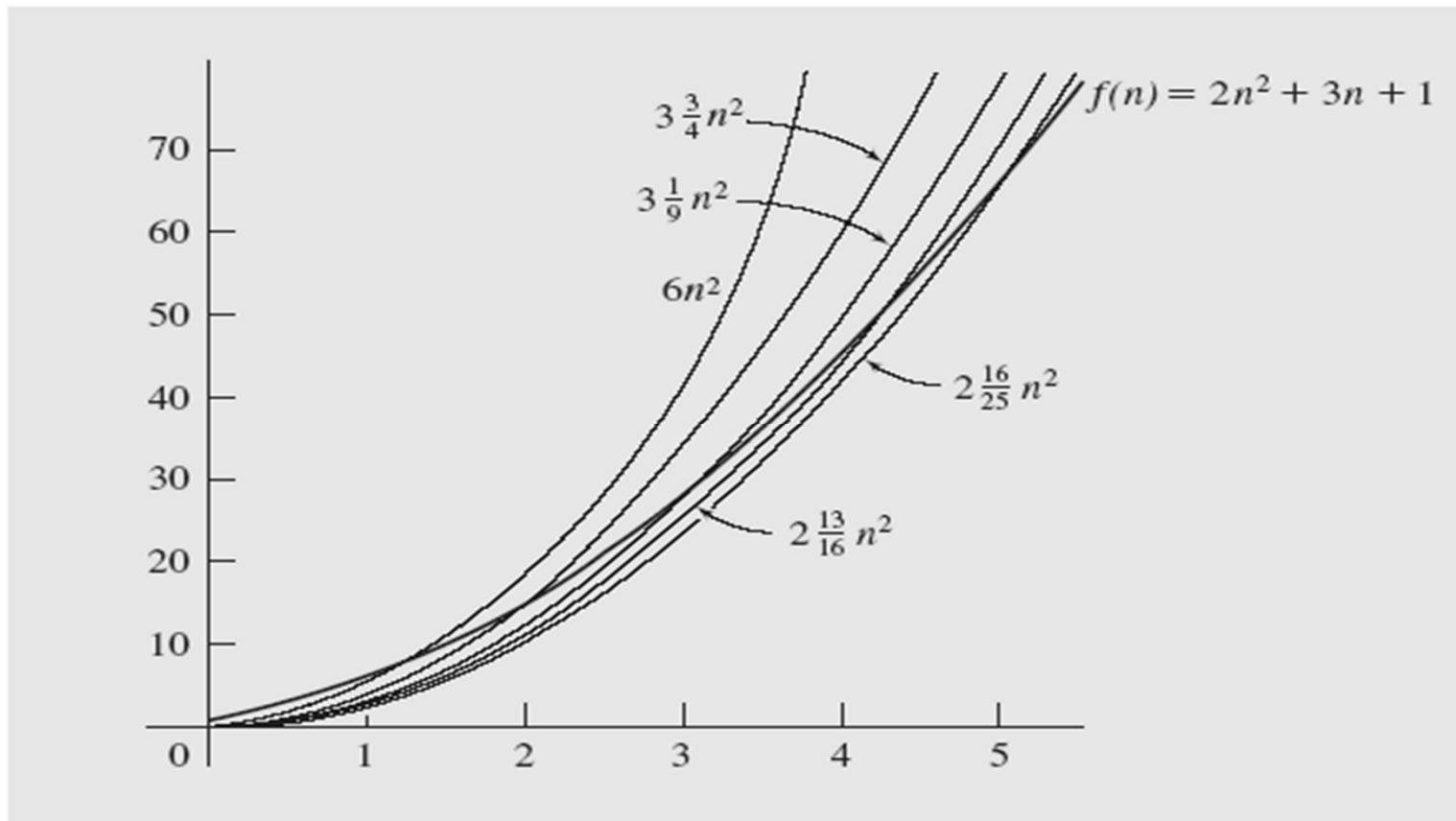
***Big-Oh* (limita superioara)**

$T(n) = O(f(n))$ daca exista doua constante pozitive c si N a.i. $T(n) \leq c f(n)$ cand $n \geq N$



Example

9

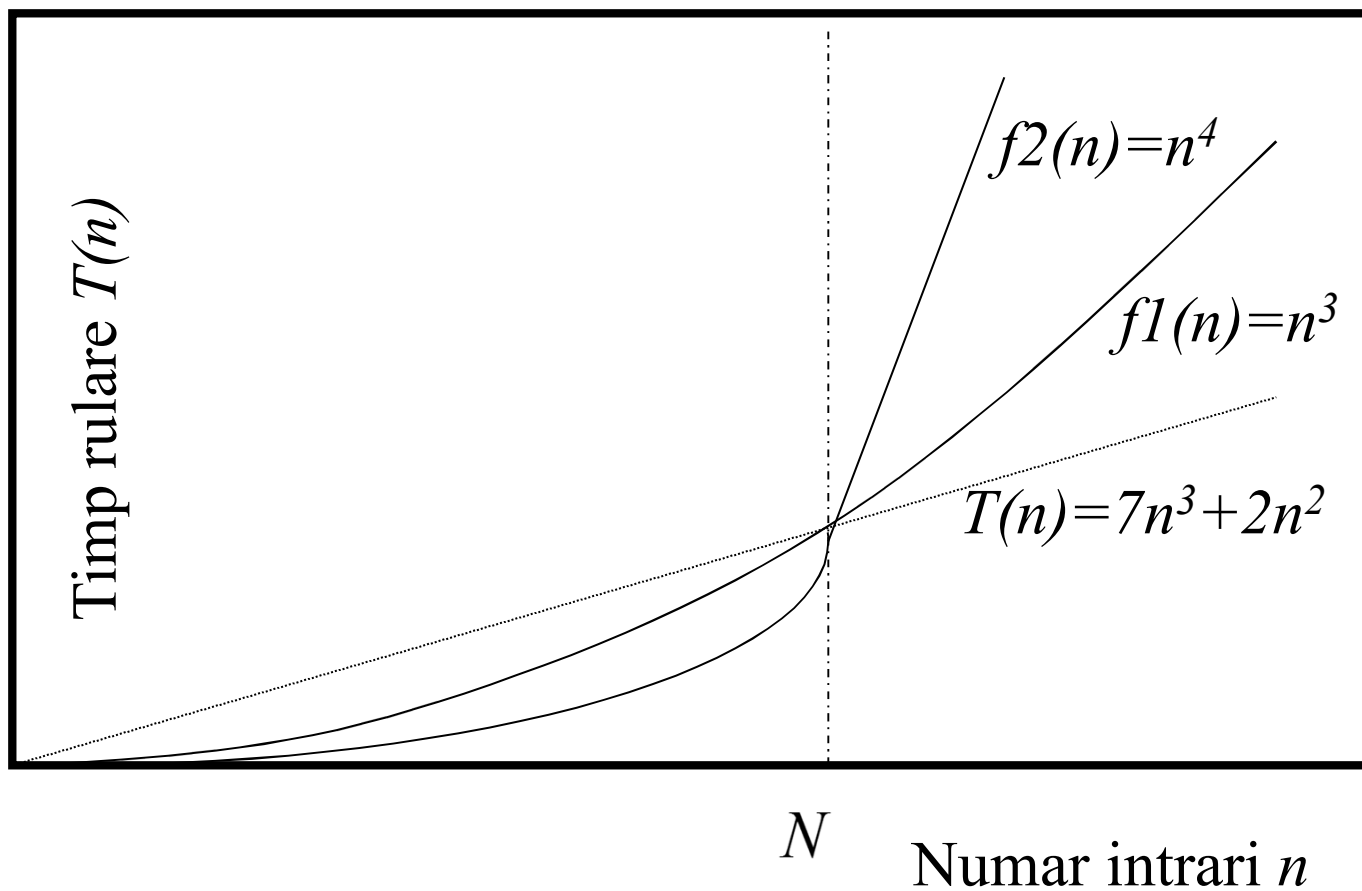


Alt exemplu

- Sa se arate ca $7n^3 + 2n^2 = O(n^3)$
 - Daca $7n^3 + 2n^2 < 7n^3 + 2n^3 = 9n^3$ (*pentru* $n \geq 1$)
 - Atunci $7n^3 + 2n^2 = O(n^3)$ unde $c = 9$ si $N = 1$

- Similar, se poate arata ca $7n^3 + 2n^2 = O(n^4)$

Marginire superioara



Big Omega

Definitie:

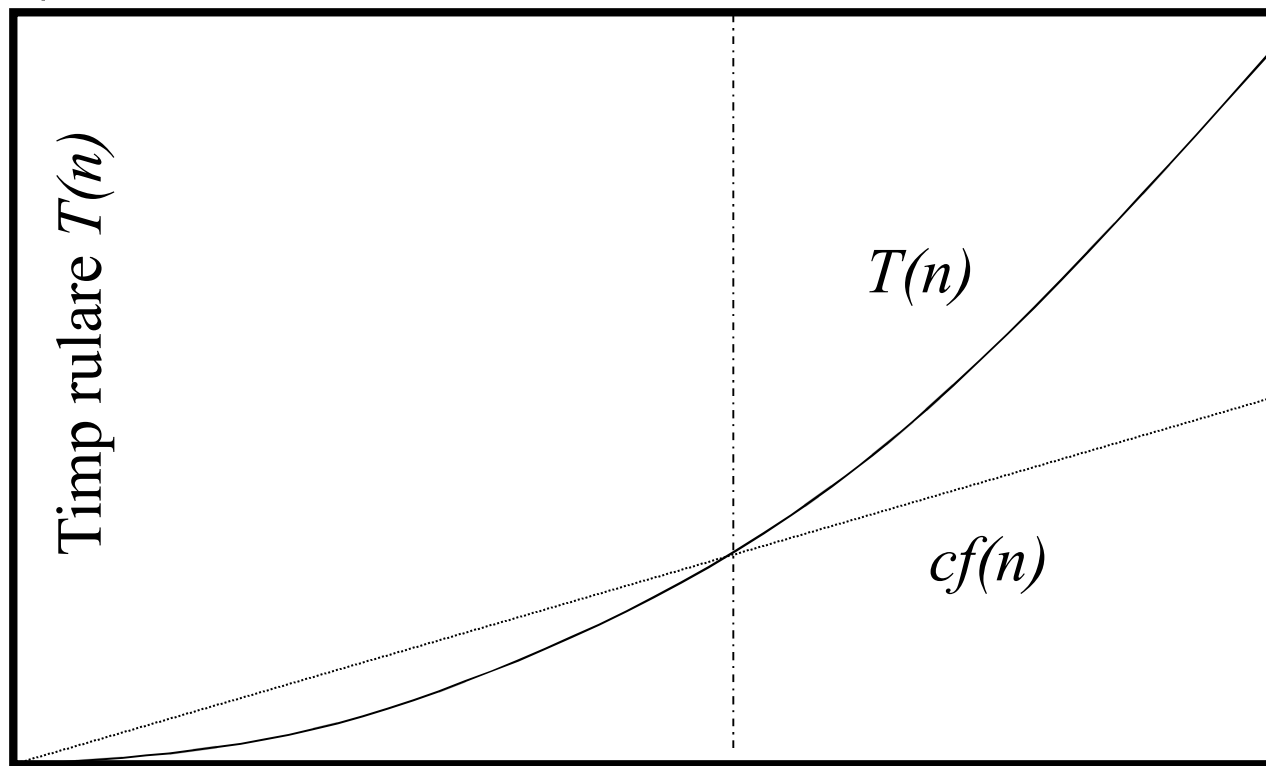
$T(n) = \Omega(f(n))$ daca exista doua constante pozitive c si N a.i. $T(n) \geq c f(n)$ daca $n \geq N$

- Cu alte cuvinte: $T(n)$ creste cu viteza mai mare decat $f(n)$; astfel $f(n)$ este limita inferioara a lui $T(n)$.

Proprietatile lui *Big-O*

- **P 1.** (tranzitivitate) Daca $f(n)$ este $O(g(n))$ si $g(n)$ este $O(h(n))$, atunci $f(n)$ este $O(h(n))$, (i.e. $O(O(g(n)))$ este $O(g(n))$.)
- **P 2.** Daca $f(n)$ este $O(h(n))$ si $g(n)$ este $O(h(n))$, atunci $f(n) + g(n)$ este $O(h(n))$.
- **P 3.** Functia an^k este $O(n^k)$.
- **P 4.** Functia n^k este $O(n^{k+j})$ pentru orice j pozitiv.
- **P 5.** Daca $f(n) = cg(n)$, atunci $f(n)$ este $O(g(n))$.
- **P 6.** Functia $\log_a n$ este $O(\log_b n)$ pentru orice numere pozitive a si $b \neq 1$
- **P 7.** $\log_a n$ este $O(\lg n)$ pentru orice numar pozitiv $a \neq 1$, unde $\lg n = \log_2 n$.

***Big Omega* - inferior**



N

Numar itemi intrare n

Exemple *Big-Omega*

□ Sa se arate ca

$$2n + 5n^2 = \Omega(n^2)$$

□ daca

$$2n + 5n^2 > 5n^2 > 1n^2 \text{ (pentru } n \geq 1)$$

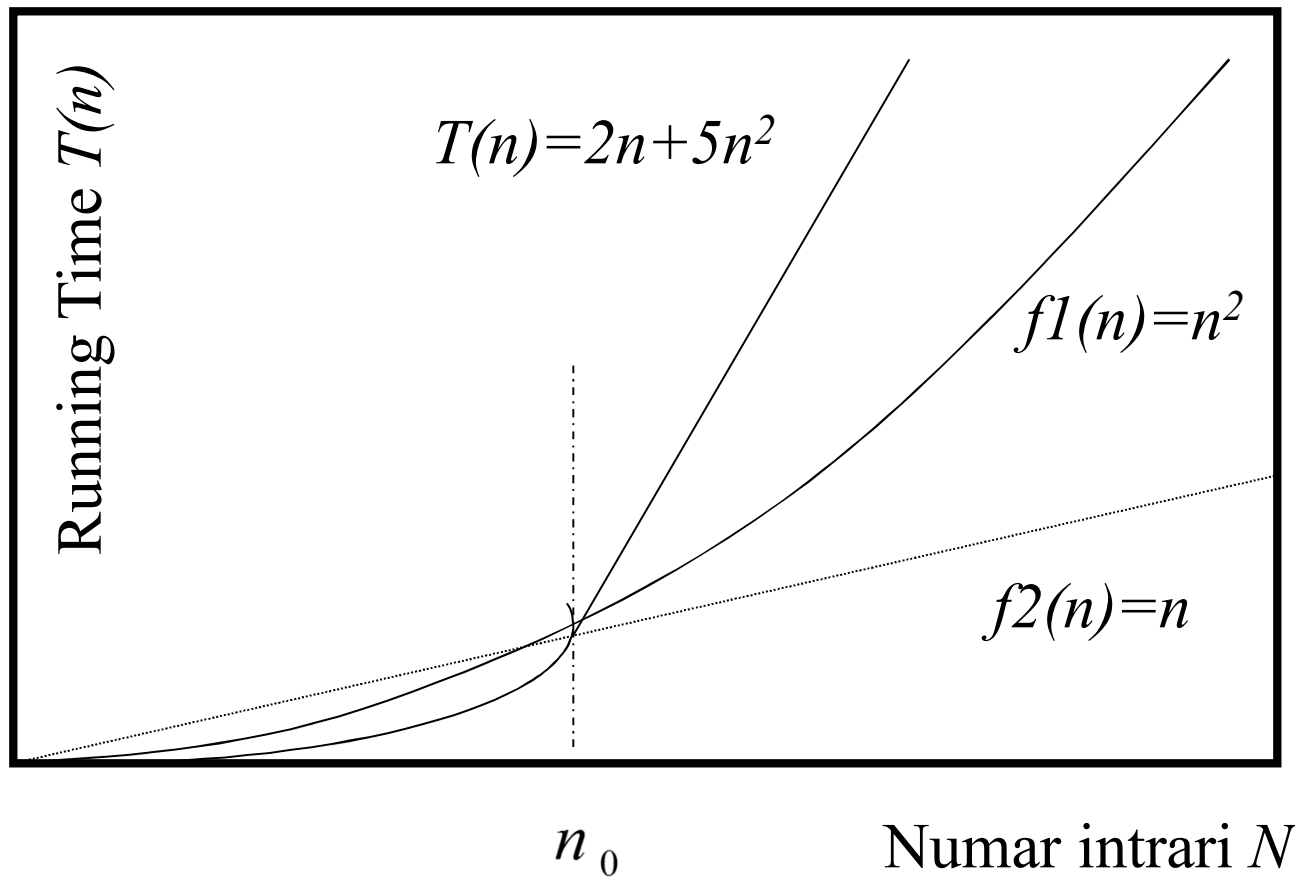
□ atunci

$$2n + 5n^2 = \Omega(n^2) \text{ cu } c = 1 \text{ si } N = 1$$

□ Similar, se poate arata ca

$$2n + 5n^2 = \Omega(n)$$

Marginire inferioara



Big Theta

- **Definitie:**

$T(n) = \Theta(f(n))$ daca si numai daca

$T(n) = O(f(n))$ si $T(n) = \Omega(f(n))$

- Cu alte cuvinte: $T(n)$ creste cu **aceeasi rata (de crestere)** ca si $f(n)$.

- Varianta:

$T(n) = \Theta(f(n))$ daca exista constantele pozitive

c, d , si N a.i. $cf(n) \leq T(n) \leq df(n)$

cand $n \geq N$

Exemple *Big Theta*

- Dacă două funcții f și g sunt proportionale, atunci

$$f(N) = \Theta(g(n))$$

- Dacă $\log_A n = \log_B n / \log_B A$

- Atunci: $\log_A(n) = \Theta(\log_B n)$

- **Nota:** baza logaritmului este irelevantă.

Little oh

□ **Definitie:**

$$T(n) = o(f(n)) \text{ d.d.}$$

$$T(n) = O(f(n)) \text{ si } T(n) \neq \Theta(f(n))$$

- Cu alte cuvinte functia $T(n)$ creste cu **o rata (de crestere) strict mai mica decat** $f(n)$.

O ierarhie a ratelor de crestere

$$c < \log n < \log^2 n < \log^k n < n < n \log n < n^2 < n^3 < 2^n < 3^n < n! < n^n$$

- ❑ Constanta: Foarte rapida.
- ❑ Logaritmic: De asemenea f. rapida. Tipica pentru multi algoritmi ce utilizeaza arbori (binari).
- ❑ Timp linear: Tipica pentru algoritmii rapizi ce ruleaza pe calculatoare cu un singur procesor.
- ❑ Poli-logaritmic ($n \log n$): Tipica pentru “the best sorting algorithms”. Poate fi o buna solutie de implemenatare.
- ❑ Polinomial: Cand o problema de dimensiune n poate fi rezolvata intr'un timp n^k unde k este o constanta. Pentru valori mici ale lui n ($n \leq 3$) is OK.
- ❑ Exponential: utilizeaza timpul k^n unde k este o constanta. Algoritmii ce cresc dupa o asemenea regula sunt recomandati doar pentru probleme de dimensiuni mici.

Analogie cu numerele reale

$$\square f(n) = O(g(n)) \cong f \leq g$$

$$\square f(n) = \Omega(g(n)) \cong f \geq g$$

$$\square f(n) = \Theta(g(n)) \cong f = g$$

$$\square f(n) = o(g(n)) \cong f < g$$

$$\square f(n) = \omega(g(n)) \cong f > g$$

Reguli generale (1)

Daca $T_1(n) = O(f(n))$ si $T_2(n) = O(g(n))$, atunci

(a) $T_1(n) + T_2(n) = \max(O(f(n)), O(g(n)))$

(b) $T_1(n) * T_2(n) = O(f(n)) * O(g(n))$

Exemple: Algorithm A:

Pas 1: Run algorithm A1 that takes $O(n^3)$ time

Pas 2: Run algorithm A2 that takes $O(n^2)$ time

$$T_A(n) = T_{A1}(n) + T_{A2}(n) = O(n^3) + O(n^2)$$

$$= \max(O(n^3), O(n^2)) = O(n^3)$$

Reguli generale (2)

Daca $T(n)$ este un polinom de grad k , atunci

$$T(n) = \Theta(n^k)$$

Exemple:

$$T(n) = n^8 + 3n^5 + 4n^2 + 6 = \Theta(n^8)$$

$\log^k(n) = O(n)$ pentru orice constanta k .

Exemplu

Fie $f(n) = n \log n$ si $g(n) = n^{1.5}$.

Care dintre functii va avea o crestere mai rapida?

$$n \log n$$



$$\log n$$



$$\log^2 n = O(n)$$



$$n^{1.5}$$



$$n^{0.5}$$



$$n$$



$g(n)$ creste mai rapid decat $f(n)$.

O alta tehnica...

Totdeauna va putea fi determinată rata relativă de creștere pentru două funcții $f(n)$ și $g(n)$ calculând

$L = \lim_{n \rightarrow \infty} f(n)/g(n)$, (regula lui l'Hospital poate fi utilizată).

1) If $L = 0$, then $f(n) = o(g(n))$

2) If $L = c$, then $f(n) = \Theta(g(n))$

3) If $L = \infty$, then $g(n) = o(f(n))$

Exemplu

Fie $f(n) = 7n^2 + n$ si $g(n) = n^2$.

Atunci $\lim_{n \rightarrow \infty} f(n)/g(n)$

$$= \lim_{n \rightarrow \infty} 7 + 1/n$$

$$= 7 + \lim_{n \rightarrow \infty} 1/n = 7.$$

$$\Rightarrow f(n) = \Theta(n^2).$$

Modelul (1)

- Pentru analiza algoritmilor (dpsv formal) este necesar un model computational.
- Modelul propus cuprinde setul standard de instructiuni: adunare, inmultire, comparare si asignare.
- **Ipoteze:**
 - Dureaza o unitate de timp pentru realizarea unei operatii simple. Acest lucru nu este complet realist, deoarece exista operatii diferite ce dureaza intervale diferite de timp.
 - Exista o memorie infinita. Astfel, nu vor exista erori datorita limitarilor de memorie.

Modelul (2)

- Cel mai important lucru de analizat îl reprezintă timpul de rulare:
 - Asta nu înseamnă că va fi făcut un model al compilatorului sau al calculatorului !
 - Accentul va fi pus pe algoritm (și nu neapărat pe program !) și pe intrările acestuia. Tipic: mărimea intrării (n) va fi luată în considerare.

Modelul (3)

- Se definesc doua functii, $T_{avg}(n)$ si $T_{worst}(n)$ ca fiind timpul mediu si timpul maxim de rulare pentru un algoritm
- In general, timpul maxim de rulare este un indicator absolut necesar: indica o “margine” pentru toate intrarile.
- Este mult mai dificil de calculat timpul mediu de rulare.
 - Este si greu de definit: d.e. ce inseamna “intrare medie” pentru un algoritm ?

Reguli generale (1)

□ Bucle

- Timpul de rulare a unei bucle “for” este cel mult egal cu timpul de rulare al instructiunilor din interiorul buclei “for” (inclusiv teste) inmultit cu numarul de iteratii.

```
for (i = 1; i <= n; i++) {  
    sum = sum + i;  
}
```

- Exemplul de mai sus este $O(n)$.

Reguli generale (2)

- Bucle imbricate

- In cazul buclerlor imbricate: timpul de rulare al instructiunilor x produsul marimii tuturor buclelor.

```
for (i = 1; i <= n; i++) {  
    for (j = 1; j <= m; j++) {  
        sum = sum + i + j;  
    }  
}
```

$$3mn = O(mn)$$

- Exemplul de mai sus este $O(mn)$.

Reguli generale (3)

□ O întrebare:

```
for (i = 1; i <= n; i++) {  
    for (j = 1; j <= m; j++) {  
        for (k = 1; k <= p; k++) {  
            sum = sum + i + j + k;  
        }  
    }  
}
```

□ $4pmn = O(pmn)$.

Reguli generale (4)

- Instructiuni consecutive
 - Se aduna; conteaza valoarea maxima.

```
for (i = 1; i <= n; i++) {  
    sum = sum + i;  
}  
for (i = 1; i <= n; i++) {  
    for (j = 1; j <= n; j++) {  
        sum = sum + i + j;  
    }  
}
```

← $O(n)$

← $O(n^2)$

- In acest caz: $O(n^2 + n) = O(n^2)$.

Reguli generale (5)

□ O intrebare

```
for (i = 1; i <= n; i++) {  
    for (j = 1; j <= n; j++) {  
        sum = sum + i + j;  
    }  
}  
sum = sum / n;  
for (i = 1; i <= n; i++) {  
    sum = sum + i;  
}  
for (j = 1; j <= n; j++) {  
    sum = sum + j*j;  
}
```

□ $n^2 + 1 + n + n = O(n^2 + 2n + 1) = O(n^2)$.

Reguli generale (6)

□ If (test) s1 else s2

- Timpul de rulare va fi \leq timpul de rulare al testului + timpul de rulare cel mai lung in raport cu s1 sau s2.

```
if (test == 1) {  
    for (i = 1; i <= n; i++) {  
        sum = sum + i;  
    }  
}  
else for (i = 1; i <= n; i++) {  
    for (j = 1; j <= n; j++) {  
        sum = sum + i + j;  
    }  
}
```

- Timp de rulare = $1 + \max(n, n^2) = O(n^2)$.

Reguli generale (7)

```
for (i = 1; i <= n; i++) {  
    for (j = 1; j <= n; j++) {  
        for (k = 1; k <= n; k++) {  
            sum = sum + i + j + k;  
        }  
    }  
}
```

```
if (test == 1) {  
    for (i = 1; i <= n; i++) {  
        for (j = 1; j <= n; j++) {  
            sum = sum + i;  
        }  
    }  
}  
else for (i = 1; i <= n; i++) {  
    sum = sum + i + j;  
}
```

□ Timpul de rulare
= $O(n^3) + O(n^2)$
= $O(n^3)$.

Reguli generale (6)

□ Recursivitate:

- In cazul apelului de functii, acestea trebuiesc analizate mai intai:

```
long factorial (int n) {  
    if (n <= 1)  
        return 1;  
    else  
        return n * factorial(n-1);  
}
```

Time Units to Compute

1 pentru test.

1 pentru inmultire.

ce se poate spune despre apelul functiei?

- Timpul de rulare pentru $factorial(n) = T(n) = 2 + T(n-1)$
 $= 4 + T(n-2) = 6 + T(n-3) = \dots = 2n = O(n)$.

Exemplu de recursivitate

```
long fib (int n) {  
    if (n <= 1)  
        return 1;  
    else  
        return fib(n-1) + fib(n-2);  
}
```

Unitati de timp

1 pentru test.

1 pentru adunare.

ce se poate spune despre apelul functiei'

- Timpul de rulare $fib(n) = T(n) = T(n-1) + T(n-2) + 2$.
Poate fi estimat $T(n)$?

Analiza sirului *Fibonacci*

Fie $F(n)$ numarul n al sirului Fibonacci.

Se poate demonstra :

$$(1) T(n) \geq F(n) \text{ si}$$

$$(2) F(n) \geq (3/2)^n.$$

$$\text{Astfel } T(n) \geq (3/2)^n,$$

ceea ce inseamna ca timpul de rulare creste exponential.

Ceea ce NU e bine!

Demonstratia prin inductie

□ **Inductia:**

- Se demonstreaza adevarul teoremei pentru cazul trivial.
 - Se presupune teorema adevarata pentru cazul k .
 - Se demonstreaza pentru cazul $k+1$.
 - Rezulta: **ADEVARAT** pentru orice k .
-

Demostratie $T(n) \geq F(n)$

Prin inductie :

Cazul trivial : $T(1) = 1 \geq F(1) = 1$,

$$T(2) = 3 \geq F(2) = 1.$$

Ipoteza inductiva : $T(n-1) \geq F(n-1)$ si

$$T(n-2) \geq F(n-2).$$

Rezulta :

$$T(n-1) + T(n-2) \geq F(n-1) + F(n-2) \text{ si}$$

$$T(n) \geq F(n-1) + F(n-2) = F(n)$$