Analiza Algoritmilor

CURS 2

Complexitatea Algoritmilor

- Complexitatea Algoritmilor
- Analiză algoritmi de căutare – algoritmi elementari
- Analiză algoritmi de sortare – algoritmi elementari
- Problemă rezolvată de un algoritm

Complexitatea Algoritmilor

Complexitatea unui algoritm se referă la **cantitatea de resurse consumate** la execuţie - adică timp de executie şi spaţiu de memorie.

Eficienţa unui algoritm se evaluează din două puncte de vedere:

- din punctul de vedere al **spaţiului de memorie** necesar pentru memorarea valorilor variabilelor care intervin în algoritm (complexitatea spaţiu);
- din punctul de vedere al **timpului de execuţie** (complexitate timp).

Complexitatea spaţiu depinde mult de tipurile de date şi de structurile de date folosite.

Complexitatea timp sau timpul necesar execuţiei programului depinde de numărul **de operaţii elementare** efectuate de algoritm.

Un algoritm efectuează trei operaţii de bază:

- intrare/ieşire
- atribuire
- decizie.

Complexitatea Algoritmilor

În general, operaţiile de intrare / ieşire sunt o constantă pentru algoritmi.

De exemplu: citim n întregi și afișăm pe cel mai mare dintre ei, indiferent de algoritmul ales, se execută n operații de intrare și una de ieșire. Prim urmare numărul de operații IO este constant indiferent de algoritm. Nu vom analiza aceste operații.

Între celelalte două operaţii (de atribuire şi de decizie) vom considera că una dintre ele este cea de bază şi vom estima de câte ori se execută aceasta.

O vom alege pe cea a carui număr de executări este mai usor de estimat, sau pe cea care necesită mai mult timp de execuţie.

Se poate măsura complexitatea **exact (cantititativ),** adică numărul de operaţii elementare sau se poate măsura **aproximativ (calitativ)**, rezultând clasa de complexitate din care face parte algoritmul.

Complexitatea Algoritmilor

Măsurarea cantitativă / exactă a complexităţii. Numărul de operaţii elementare

Vom calcula într-o funcție F(n) numărul de operații elementare executate.

Exemplu sortarea unui vector de n elemente:

```
pentru i ← 0, n-1 execută
    pentru j ← i+1, n execută
        dacă (v[i] > v[j]) atunci
            aux ← v[i]
            v[i] ← v[j]
            v[j] ← aux
```

Vom avea trei cazuri: cazul cel mai favorabil, cel mai defavorabil şi cazul mediu

Complexitatea Algoritmilor

Cazul favorabil este vectorul gata sortat, deci:

Ffavorabil(n) = (n-1) + (n-2) + ... + 3 + 2 + 1 = **(n-1)*n/2** - deoarece se efectuează numai comparaţia de (n-1)*n/2

Cazul cel mai defavorabil este vectorul sortat invers, deci:

Fdefavorabil(n) = 4 * ((n-1) + (n-2) + ... + 3 + 2 + 1) = **4*(n-1)*n/2** = **2(n-1)n** deoarece se execută atât condiția din if cât și cele trei operații de atribuire

Cazul mediu se considera media aritmetică a celorlalte două cazuri:

Fmediu(n) = (**(n-1)*n/2 + 4*(n-1)*n/2) /2 = 5(n-1)*n/4**

Astfel se calculează complexitatea exactă, însă calculele cresc cu cât algoritmul este mai elaborat. De aceea se folosește complexitatea aproximativă – aproximarea asimptotică a funcțiilor de complexitate exactă.

În cazul de față putem aproxima că toate cele trei funcții au o complexitate pătratică: n^2 și se notează $O(n^2)$.

Complexitatea Algoritmilor

Masurarea calitativă / aproximativă a complexităţii. Clase de complexităţi

Sunt folosite diferite notaţii care sunt utile pentru analiza performanței şi a complexităţii unui algoritm.

Aceste notaţii marginesc valorile unei funcţii f date cu ajutorul unor constante şi al altei funcţii.

Fie f și g două funcții definite pe numere întregi pozitive

Notaţii:

O (margine superioară – upper bound) – folosit pentru a descrie complexitatea timp

f(n) = O(g(n)), dacă există constantele c şi n_0 astfel încât f$(n) \leq c * g(n)$, pentru $n \geq n_0$

Pentru orice c, f(n) are un ordin de creștere cel mult egal cu cel al lui g(n)

Ω *(Omega)(margine inferioară – lower bound)*

f(n) = Ω(g(n)), dacă există constantele c şi n_0 astfel încât f$(n) \geq c * g(n)$ (sau g(n)=O(f(n))), pentru $n \geq n_0$

Pentru orice c, f(n) are un ordin de creștere cel puțin la fel de mare ca cel al lui g(n)

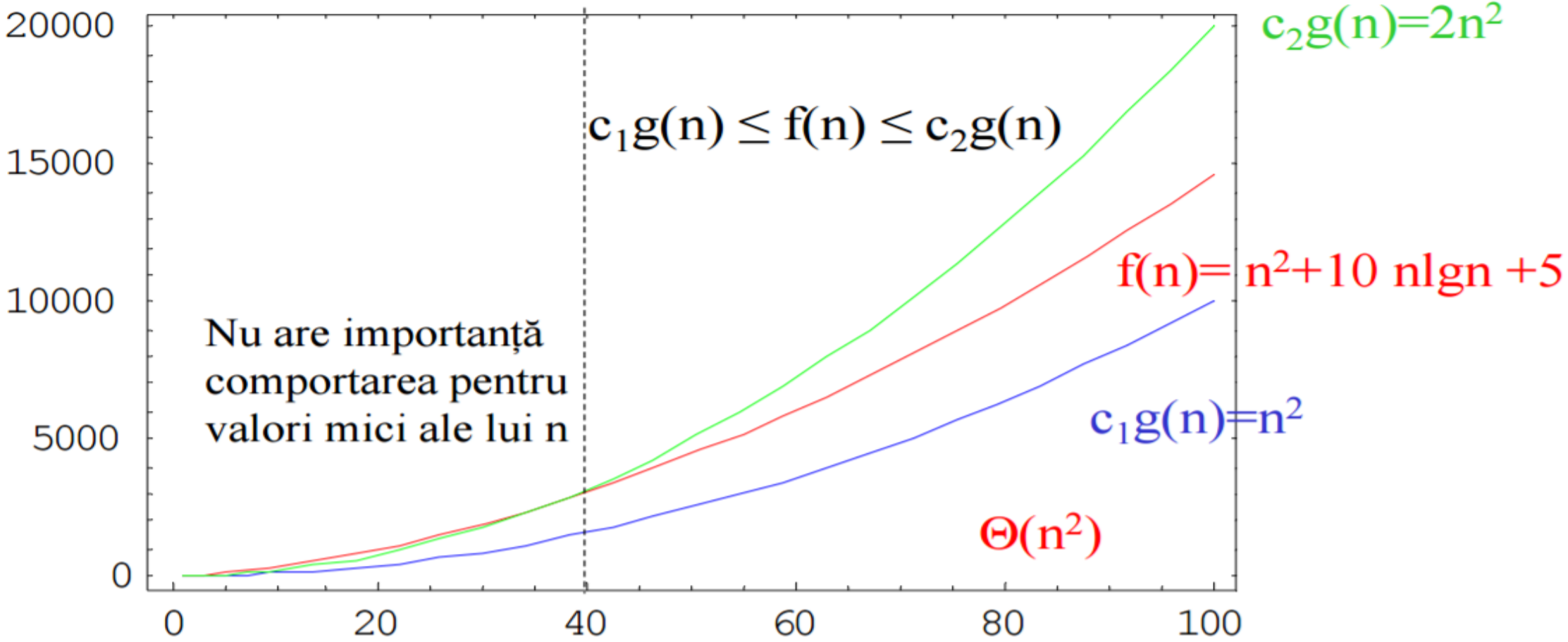
Θ (Theta)(categorie constantă – same order)

f(n) = Θ(g(n)), dacă și numai dacă f(n) = O(g(n)) și g(n) = O(f(n)), sau altfel spus, ambele funcții au aproximativ aceeași rată de creștere

Complexitatea Algoritmilor

Notația Θ

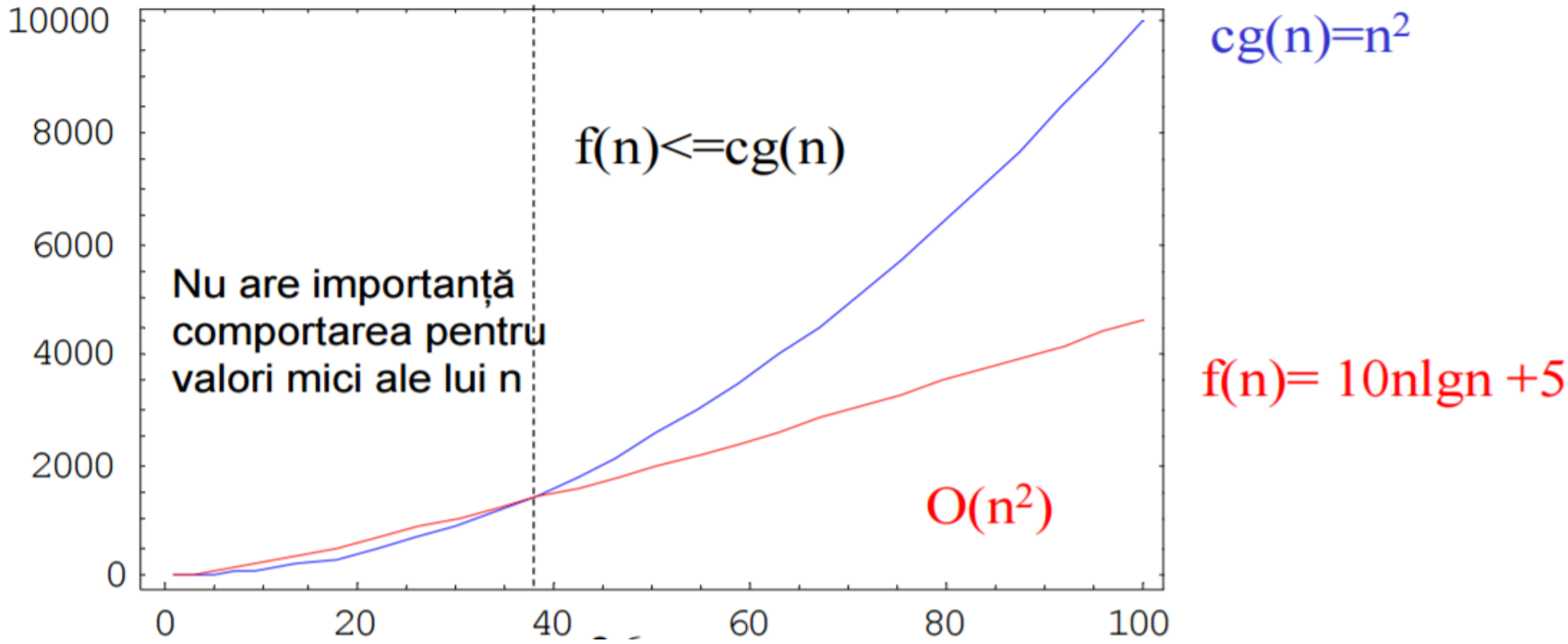
Ilustrare grafică. Pentru valori mari ale lui n, f(n) este mărginită, atât superior cât și inferior de g(n) înmulțit cu niște constante pozitive



Complexitatea Algoritmilor

Notatia O

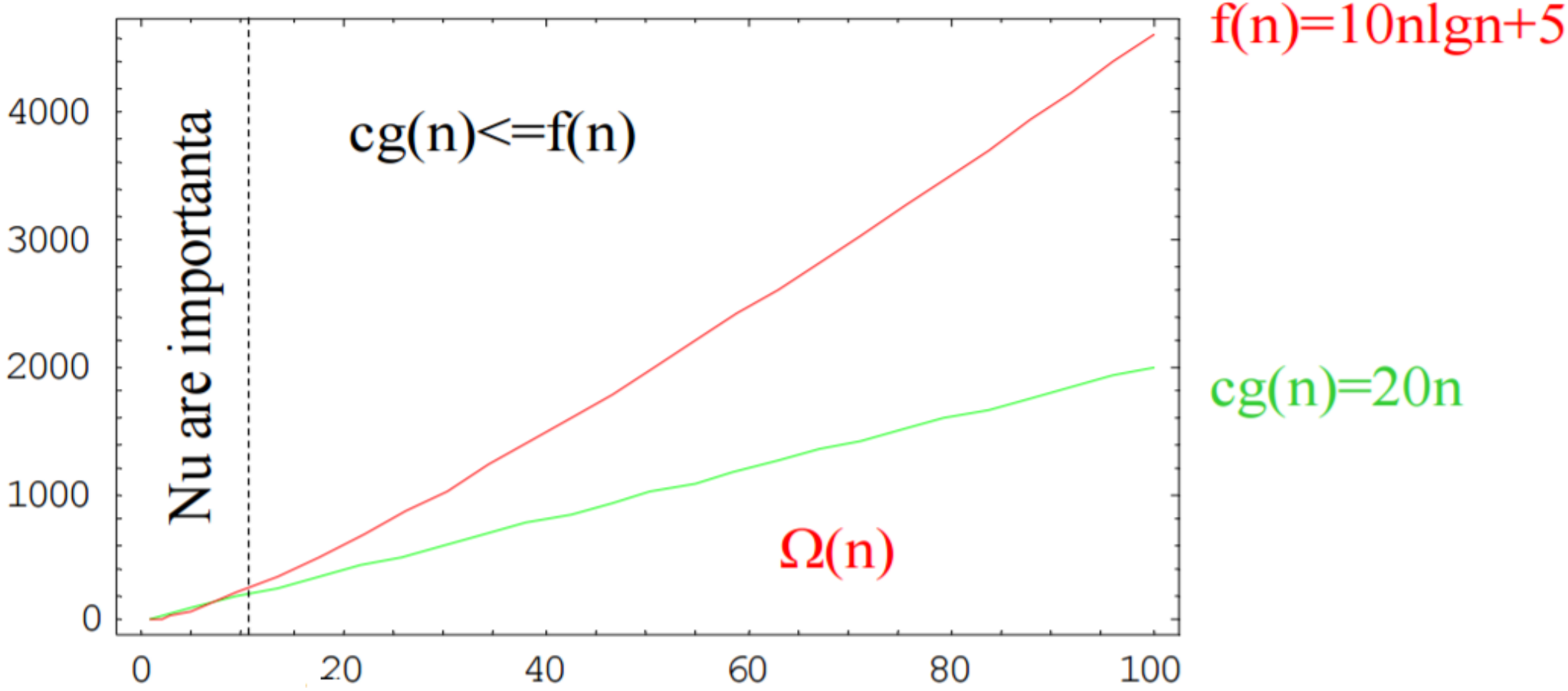
Ilustrare grafica. Pentru valori mari ale lui n, f(n) este marginită superior de g(n) multiplicată cu o constantă pozitivă



Complexitatea Algoritmilor

Notația Ω

Ilustrare grafică. Pentru valori mari ale lui n, funcția f(n) este marginită inferior de g(n) multiplicată eventual de o constantă pozitivă



Complexitatea Algoritmilor

Masurarea calitativă / aproximativă a complexităţii. Clase de complexităţi

Exemplul anterior:

f(n) = 5*n2/4 – 5*n/4 = 1.25*n2 - 1.25*n

În cazul de față termenul 1.25*n poate fi neglijat deoarece este mult mai mic decât primul termen și nu va afecta timpul de execuție semnificativ.

Putem deci aproxima ca rezultatul $f(n)=1.25*n^2$ iar dacă comparăm algoritmul cu alt algoritmul care rezolvă aceeași problemă nici constanta nu are o semnificație mare.

De exemplu:

Considerăm doi algoritmi, unul cu timpul de execuție n^2 și celălalt cu timpul de execuție $0.001*n^3$. Pentru valorile n > 1000 primul algoritm este mai rapid. În general, putem spune că indiferent de constantă, un algoritm cu timpul de execuție proporțional cu n^2 este mai bun decât unul cu timpul de execuție proporțional cu n^3 pentru aproape toate intrările.

Conform exemplului nostru, f(n)=O(n2) prin urmare garantează că timpul de execuție este cel mult pătratic.

Se spune: Complexitatea algoritmului este O(n2) sau simplu, Algoritmul este O(n2).

Complexitatea Algoritmilor

Masurarea calitativă / aproximativă a complexităţii. Clase de complexităţi

Complexitatea algoritmilor este deci o funcţie g(n) care limitează superior numărul de operaţii necesare pentru dimensiunea n a problemei. Există două interpretări ale limitei superioare:

- complexitatea în cazul cel mai defavorabil: timpul de execuţie pentru orice dimensiune dată va fi mai mic sau egal decât limita superioară
- timpul de execuţie pentu orice dimensiune dată va fi media numărului de operaţii pentru toate instanţele posibile ale problemei.

Deoarece este dificil să se estimeze o comportare statistică ce depinde de dimensiunea intrării, de cele mai multe ori este folosită prima interpretare, cea a cazului cel mai defavorabil (O).

Complexitatea Algoritmilor

Masurarea calitativă / aproximativă a complexităţii. Clase de complexităţi

În majoritatea cazurilor, complexitatea lui f(n) este aproximată de familia sa O(g(n)), unde g(n) este una dintre functiile:

- **n (complexitate liniară)**
- **Log(n) (complexitate logaritmică)**
- **n^a , cu a>=2 (complexitate polinomială)**
- **a^n (complexitate exponențială)**
- **n! (complexitate factorială)**

O(g(n)) este deci clasa de complexitate cu care se aproximează complexitatea unui algoritm.

Complexitatea Algoritmilor

Masurarea calitativă / aproximativă a complexităţii. Clase de complexităţi

Pentru **n** suficient de mare au loc inegalitaţile:

$$\log(n) < n < n*\log(n) < n^2 < n^3 < 2^n$$

prin urmare:

$$O(\log(n)) < O(n) < O(n*\log(n)) < O(n^2) < O(n^3) < O(2^n)$$

În situaţia în care găsim mai mulţi algoritmi pentru rezolvarea unei probleme vom alege dintre toţi algoritmii pe cel cu **ordinul de complexitate mai mic**.

Complexitatea Algoritmilor

Reguli generale de estimare a complexităţii

Cicluri

Timpul de execuţie al unui ciclu este cel mult timpul de execuţie al instrucţiunilor din interiorul ciclului înmulţit cu numărul de iteraţii. De regulă se estimează ca o structură repetitivă este de ordinul **O(n).**

Cicluri imbricate

Analiza se realizează din interior spre exterior. Timpul de execuţie al instrucţiunilor din interiorul unui grup de cicluri imbricate este dat de timpul de execuţie al instrucţiunilor înmulţit cu **produsul numărului de iteraţii ale tuturor ciclurilor**. Cu alte cuvinte, dacă avem două cicluri imbricate (**for** în **for** spre exemplu) putem aproxima complexitatea la $\mathbf{O(n^2)}$.

Complexitatea Algoritmilor

Reguli generale de estimare a complexităţii

Structuri secvenţiale

- În acest caz timpii de execuţie se adună, ceea ce înseamnă că maximul lor contează, adică gradul lui **n** va fi dat de gradul cel mai mare.

Structuri decizionale

- Timpul de execuţie al instrucţiunii decizionale este cel mult timpul de execuţie al testului plus maximul dintre timpii de rulare pe ramura *ATUNCI*, respectiv *ALTFEL*.

Dacă există apeluri de funcţii, acestea trebuie analizate primele.

Complexitatea Algoritmilor

Estimarea complexității

Se citeşte o matrice de la tastatură cu n linii şi n coloane. Se cere să se realizeze un program care afişează suma elementelor de pe diagonala principală.

Avem două variante de algoritmi

ALGORITM NEEFICIENT

```
citeşte n
//citire matrice
pentru i ← 1, n execută
  pentru j ← 1, n execută
    citeşte a[i][j]

S ← 0
pentru i ← 1, n execută
  pentru j ← 1, n execută
    dacă (i = j) atunci
          s ← s + a[i][j]
scrie s
```

ALGORITM EFICIENT

```
citeşte n
//citire matrice
pentru i ← 1, n execută
  pentru j ← 1, n execută
        citeşte a[i][j]

S ← 0
pentru i ← 1, n execută
    s ← s + a[i][i]

scrie s
```

Complexitatea Algoritmilor

Estimarea complexității

Ambii algoritmi citesc matricea folosind două structuri **for (pentru)** imbricate, care are complexitatea $O(n^2)$.

În continuare secvența de program pentru calculul sumei are complexitate diferită de la un caz la altul.

Algoritmul din stânga parcurge apoi toată toată matricea şi doar în cazul în care i=j adună elementul curent din matrice la sumă.

Adunarea se execută de n*n ori -> **$O(n^2)$**.

Ce resultă pentru n=100?

Algoritmul din dreapta parcurge doar diagonala principală (indicii sunt egali), rezultă că acest for are complexitatea O(n).

Pentru n=100 de căte ori se execută?

Complexitatea Algoritmilor

Estimarea complexității

Fie următorul algoritm:

```
int j=0;
for (int i=0; i<N; i++) {
  while (       (j < N-1)    &&    ( A[i]-A[j]  >  D )       )
        j++;
  if (A[i]-A[j] == D) return 1;
}
```

Care este complexitatea algoritmului?

Complexitatea Algoritmilor

Estimarea complexității

```
1.      int result=0;
2.      for (int i=0; i<N; i++)
3.              for (int j=i; j<N; j++) {
4.                      for (int k=0; k<M; k++) {
5.                              int x=0;
6.                              while (x<N) { result++; x+=3; }
7.                      }
8.              for (int k=0; k<2*M; k++)
9.                      if (k%7 == 4) result++;
10. }
```

Complexitatea O ?

Complexitatea Algoritmilor

Estimarea complexității

6. while (x<N) { result++; x+=3; }

X de la 0 la N, iar pasul de incrementare este 3 -> cel mult N/3 +1 -> O(N)

Complexitatea Algoritmilor

Estimarea complexității

```
4.          for (int k=0; k<M; k++) {
5.                  int x=0;
6.                  while (x<N) { result++; x+=3; }
7.          }
```

For se execută de cel mult m ori (k este incrementat de m ori) -> O(M)

While se execută la fiecare incrementare a lui k -> complexitatea pt liniile 4-7 este O(MN)

Complexitatea Algoritmilor

Estimarea complexității

```
8.          for (int k=0; k<2*M; k++)
9.                  if (k%7 == 4) result++;
```

K se execută de cel mult 2*M ori -> O(M) pentru liniile 8-9

Complexitatea Algoritmilor

Estimarea complexității

```
4.                  for (int k=0; k<M; k++) {
5.                          int x=0;
6.                          while (x<N) { result++; x+=3; }
7.                  }
8.          for (int k=0; k<2*M; k++)
9.                  if (k%7 == 4) result++;
```

Pentru liniile 4-9 se adună complexitățile celor două structuri: O(MN + M) -> O(MN)

Complexitatea Algoritmilor

Estimarea complexității

```
2.      for (int i=0; i<N; i++)
3.              for (int j=i; j<N; j++)  {

        liniile de la 4-9  cu complexitatea O(MN)
}
```

Pentru liniile 2-3 există cel mult N(N+1)/2 valori pentru combinația i,j -> $O(N^2)$

Liniile 4-9 se vor executa de $O(N^2)$ prin urmare -> $O(N^2 * MN) = O(MN^3)$

Complexitatea Algoritmilor

Estimarea complexității

1. Care este complexitatea pentru algoritmul de calcul al maximului dintr-un șir?
(propuneți algoritmul și faceți analiza. Cazul favorabil și nevaforabil)

2. Sortarea prin inserție - care este complexitatea?

```
for i=2 .. N
        val=a[i]
        poz=i
        while a[poz-1]>val
                a[poz]=a[poz-1]
                poz=poz-1
        a[poz]=val
```

(Cazul favorabil: șirul este deja ordonat. Caz nefavorabil?)

Complexitatea Algoritmilor

Estimarea complexității

3. Problema celebrității

Celebritate – o persoană cunoscută de toată lumea dar nu cunoaște pe nimeni

Dat un digraf cu n vârfuri se verifică dacă există un vârf cu grad exterior 0 si grad interior n-1.

Avem o matrice de adiacență cu relațiile dintre persoane:

$$A_{i,j} = \begin{cases} 1, & dacă\ persoana\ i\ cunoaște\ persoana\ j \\ 0, & altfel \end{cases}$$

Se calculează pentru fiecare persoană p câte persoane cunoaște (out) și câte persoane cunosc persoana p (in).

Dacă out = 0 și in = n-1 atunci p este o celebritate

Complexitatea Algoritmilor

Estimarea complexității

3. Problema celebrității - care este complexitatea?

```
celebritate=0
pentru p=1 .. n
        in=0,  out=0
        pentru j=1 .. n
                in=in+a[j][p]
                out=out+a[p][j]

        dacă in=n-1 și out = 0 celebritate=p

Dacă celebritate=0 scrie  'Nu exista celebritati !'
altfel scrie  p, ' este o celebritate.'
```

Complexitatea Algoritmilor

Estimarea complexității

Algoritmul poate fi îmbunătățit dacă considerăm relațiile dintre x și y:

A[x][y] = 0 -> y nu poate fi celebritate (x nu cunoaște pe y)

A[x][y] = 1 -> x nu poate fi celebritate (x cunoaște pe y)

```
candidat=1;
pentru i=2 .. n
        daca a[candidat][i]=1        candidat=i
out=0;
in=0;
pentru i=1 .. N
        in=in+a[i][candidat]
        out=out+a[candidat][i]
daca out=0 si  in=n-1 scrie candidat, ’ este o celebritate .’
altfel scrie  ’Nu exista celebritati.’
```

Care este complexitatea acum?

Complexitatea Algoritmilor

O comparație a ordinelor de creștere

Tipuri de dependență a timpului de execuție în raport cu dimensiunea problemei:

n	$\log_2 n$	$n\log_2 n$	n^2	2^n	$n!$
10	3.3	33	100	1024	3628800
100	6.6	664	10000	10^{30}	10^{157}
1000	10	9965	1000000	10^{301}	10^{2567}
10000	13	132877	100000000	10^{3010}	10^{35659}

Complexitatea Algoritmilor

Analiza empirică a eficienței algoritmilor

Uneori analiza teoretică a eficienței este dificilă; în aceste cazuri poate fi utilă analiza empirică.

Analiza empirică poate fi utilizată pentru:

- Formularea unei ipoteze inițiale privind eficiența algoritmului
- Compararea eficienței mai multor algoritmi destinați rezolvării aceleiași probleme
- Analiza eficienței unei implementări a algoritmului (pe o anumită mașină)
- Verificarea acurateții unei afirmații privind eficiența algoritmului

Complexitatea Algoritmilor

Structura generală a analizei empirice a eficienței algoritmilor

1. Se stabilește scopul analizei
2. Se alege o măsură a eficienței (de exemplu, numărul de execuții ale unor operații sau timpul necesar execuției unor pași de prelucrare)
3. Se stabilesc caracteristicile setului de date de intrare ce va fi utilizat (dimensiune, domeniu de valori …)
4. Se implementează algoritmul sau în cazul in care algoritmul este deja implementat se adaugă instrucțiunile necesare efectuării analizei (contoare, funcții de înregistrare a timpului necesar execuției etc)
5. Se generează datele de intrare
6. Se execută programul pentru fiecare dată de intrare și se înregistrează rezultatele
7. Se analizează rezultatele obținute

Complexitatea Algoritmilor

Structura generală a analizei empirice a eficienței algoritmilor

Măsura eficienței: este aleasă în funcție de scopul analizei

- Dacă scopul este să se identifice clasa de eficiență atunci se poate folosi numărul de operații care se execută
- Dacă scopul este să se analizeze/compare implementarea unui algoritm pe o anumita mașină de calcul atunci o măsura adecvată ar fi timpul fizic

Complexitatea Algoritmilor

Structura generală a analizei empirice a eficienței algoritmilor

Set de date de intrare. Trebuie generate diferite categorii de date de intrare pentru a surprinde diferitele cazuri de funcționare ale algoritmului

Câteva reguli de generare a datelor de intrare:

- Datele de intrare trebuie să fie de diferite dimensiuni și cu valori cat mai variate
- Setul de test trebuie să conțină date cât mai arbitrare (nu doar excepții)

Complexitatea Algoritmilor

Structura generală a analizei empirice a eficienței algoritmilor

Implementarea algoritmului. De regulă este necesară introducerea unor prelucrări de monitorizare

- Variabile contor (în cazul in care eficiența este estimată folosind numărul de execuții ale unor operații)

- Apelul unor funcții specifice care returnează ora curentă (în cazul în care măsura eficienței este timpul fizic)

Complexitatea Algoritmilor - Algoritmi de căutare

Căutare secvenţială

Ideea:

Verificăm pe rând elementele din șir până când găsim elementul căutat – poziția

Exemplu:

sir = 1, 7, 3, 5, 2

x = 3

Rezultat: 3 (poziția de la 1)

sir = 1, 7, 3, 5, 2

x = 4

Rezultat: nu s-a gasit (sau -1 sau null?)

Complexitatea Algoritmilor

Căutare secvenţială

poz=0;

i=1;

Cât timp (i<=v_n) and (poz=0) do

dacă (v_i = x) atunci poz=i

altfel i=i+1;

returnez poz;

Complexitatea?

Complexitatea Algoritmilor

Căutare binară (doar pentru șir ordonat)

Ideea:

Se determină în ce relaţie se află elementul aflat în mijlocul șirului cu elementul ce se caută. În urma acestei verificări căutarea se continuă doar într-o jumătate a șirului. În acest mod, prin înjumătăţiri succesive se micşorează volumul șirului rămas pentru căutare.

Complexitatea Algoritmilor

Căutare binară (doar pentru șir ordonat)

{v este ordonat crescator}

{rezultatul e pozitia pe care apare x sau pe care ar trebui inserat x}

mij, inf, sup, ind

inf=1, sup=n

ind=0

execută

mij=(inf+sup)/2

dacă v_{mij} = k atunci ind=1

altfel dacă v_{mij} < k atunci inf=mij+1

altfel sup=mij-1

până când inf <= sup si ind=0

dacă ind=1 return mij

altfel return 0

Complexitatea?

Complexitatea Algoritmilor - Algoritmi de sortare

Metoda bulelor

Ideea:

Compară două câte două elemente consecutive iar în cazul în care acestea nu se află în relaţia dorită, ele vor fi interschimbate. Procesul de comparare se va încheia în momentul în care toate perechile de elemente consecutive sunt în relaţia de ordine dorită.

Complexitatea Algoritmilor

Metoda bulelor

execută

ordonat=true;

pentru i=2 , v_n

dacă $v_{i-1} > v_i$ atunci

$t=v_{i-1}$

$v_{i-1} =v_i$

$v_i =t$

ordonat=false;

până când ordonat;

Complexitatea?

Complexitatea Algoritmilor

Metoda inserției

Ideea:

Traversăm elementele, inserăm elementul curent pe poziția corectă în subșirul care este deja ordonat. În acest fel elementele care au fost deja procesate sunt în ordinea corectă. După ce am traversat tot șirul toate elementele vor fi sortate.

Complexitatea Algoritmilor

Metoda inserției

Pentru i=2, v_n

ind=i-1;

$x=v_i$

cât timp ind>0 și $x<v_{ind}$

$v_{ind+1} = v_{ind}$

ind=ind -1

Complexitate ?

Complexitatea Algoritmilor

Metoda selecției

Ideea:

Se determină poziţia elementului cu valoare minimă (respectiv maximă), după care acesta se va interschimba cu primul element.

Acest procedeu se repetă pentru subșirul rămas, până când mai rămâne doar elementul maxim.

Complexitatea Algoritmilor

Metoda selecției

pentru i=1, v_{n-1}

ind=i;

for j=i+1, v_n

if $v_j < v_{ind}$ atunci ind=j;

if i<ind atunci

$t=v_i$

$v_i = v_{ind}$

$v_{ind} = t$

Complexitatea?

Complexitatea Algoritmilor

Metoda numărării

Ideea:

Se da un sir de elemente distincte. Pentru fiecare element numarăm elementele care sunt mai mici, astfel aflăm poziția elementului în șirul ordonat.

Complexitatea Algoritmilor

Metoda numărării

v1=v

pentr i=1 , v_n

//numar cate elemente sunt mai mici decat v_i

k=0;

$x=v_i$

for j=1 , v_n

dacă $v_j < x$ atunci k=k+1;

$v_{k+1} = x$

Complexitatea?

Problemă rezolvată de un algoritm

Atunci când vorbim de rezolvarea unei probleme cu ajutorul unui algoritm vorbim de date de intrare – input și de date de ieșire – output.

Spunem că un algoritm A rezolvă o problemă P dacă pentru orice date de intrare execuția lui A dintr-o configurație inițială se termină într-o configurație finală ce are ca rezultat datele de ieșire.

O problemă P este **rezolvabilă** dacă există un algoritm A care rezolvă P.

O problemă P este **nerezolvabilă** dacă NU există un algoritm A care rezolvă P.

O problemă de **decizie** este o problemă P la care soluția este de tipul ”da” sau ”nu” – true sau false.

O problemă **decidabilă** este o problemă de decizie rezolvabilă.

O problemă **nedecidabilă** este o problemă de decizie nerezolvabilă.

Problemă rezolvată de un algoritm

De exemplu problema opririi (Halting Problem) este o problemă **nedecidabilă**:

"***Există un algoritm Turing-echivalent A care primind ca parametru de intrare specificația (e.g. codul sursă) al unui program arbitrar să decidă într-un număr finit de pași dacă programul a cărui specificație a primit-o ca parametru se încheie într-un număr finit de pași sau rulează la infinit?***„

Răspunsul este nu, pentru orice specificație de cod sursă (orice limbaj), adică nu putem ști despre **orice program** dacă el se va opri odată sau nu.

Teorema a fost demonstrată de Alan Turing in 1936 prin faptul că putem da unei mașini Turing H specificațiile unei mașini H1 Turing despre care să spunem dacă se oprește sau nu, însă mașina H1 fiind tot o mașină Turing putem să o descriem pe ea însăși, adică H primește de fapt la intrare propriile specificații.

Demonstrație bazată pe paradoxul mincinosului de la greci: “Eu mint”.

Problemă rezolvată de un algoritm

Există algoritmi determiniști pentru care plecând de la un set de date anume rezultatul este unic ori decâte ori este executat algoritmul (dată o mașină algoritmul trece prin aceeași succesiune de stări) (mașina Turing).

Există și algoritmi nedeterminiști care, spre deosebire de cei determiniști acceptă și următoarele instrucțiuni:

- succes și failure, când algoritmul se termină cu succes sau cu eșec
- choice(A), unde A este o mulțime finită de valori, iar funcția întoarce un element din A ales arbitrar

Există operații al căror rezultat nu este unic definit ci are valori într-o mulțime finită de posibilități.

Etape în execuția unui algoritm nedeterminist:

- choise – generează o copie pentru fiecare element din mulțimea A – partea nedeterministă a algoritmului. Are complexitatea O(1)
- testare – fiecare copie generată este testată dacă e o soluție corectă

Un algoritm nedeterminist se termină cu eșec dacă nu există o modalitate de a efectua alegeri care să conducă la o instrucțiune de succes.

Problemă rezolvată de un algoritm

Exemple de algoritmi nedeterminiști:

Exemplul 1: - se verifică dacă elementul x apare sau nu în mulțimea A

Algoritmul determinist este de ordin O(n) iar cel nedeterminist este O(1)

i -> choise() // generarea poziției elementului căutat

dacă a_i = x atunci   //testarea elementului de căutat

scrie i

scrie success

altfel failure

Problemă rezolvată de un algoritm

Exemple de algoritmi nedeterminiști:

Exemplul 2: ordonarea crescătoare a unui vector A

Complexitatea în cazul cel mai nefavorabil este $O(n^2)$ iar pentru cel nedeterminist este O(n).

Se copie elementele vectorului A într-un vector auxiliar B după care se verifică dacă elementele lui B sunt ordonate crescător

Pentru i -> 1 la n

j -> choice() //generarea poziția elementului vectorului B

dacă b_j = NULL atunci b_j -> a_i // testare dacă elementul b_j nu e ocupat

altfel failure

pentru i -> 1 la n-1

dacă $b_i > b_{i+1}$ atunci failure //testare dacă vectorul nu e sortat

scrie b ; succes //dacă testul nu a dat fail atunci avem succes

Problemă rezolvată de un algoritm

Exemple de algoritmi nedeterminiști:

Exemplul 3: Problema celor N regine. Avem o tablă de șah n x n și o așezare a n piese de tip regină a.î. Nicio regină nu atacă o altă regină.

Complexitatea este $O(n^2)$

Ataca (VectorSolutii, i, j)

pentru k = 0 , (i-1)

dacă $VectorSolutii_k = j$ sau $VectorSolutii_k - j = k - i$ sau $VectorSolutii_k - j = i - k$

return true

return false

Regine (n, VectorSolutii)

pentru i = 0, n

j = choise() //generare linia reginei de pe coloana i

dacă *Ataca(VectorSolutii*, i, j) failure //testare dacă se atacă

$VectorSolutii_i = j$

succes //dacă testul nu generează failure atunci avem succes