2. Noţiunea de algoritm. Analiza algoritmilor

2.1. Noţiunea de algoritm

- Termenul **algoritm** îşi are sorgintea în numele autorului persan **Abu Ja'far Mohamed Ibn Musa Al Khowarismi** care în anul 825 e.n. a redactat un tratat de matematică *(„Al-jabr wa'l muqabala”)* în care prezenta pentru prima dată metode de rezolvare generice pentru anumite categorii de probleme.

- ***Dicţionarul explicativ al limbii române*** [DEX75] oferă următoarea definiţie pentru noţiunea de **algoritm**:

    - *"Ansamblu de simboluri şi de operatori**,** folosiţi în matematică şi logică, permiţând găsirea în mod mecanic (prin calcul) a unor rezultate"*.

- Alte dicţionare oferă alte definiţii, spre exemplu:

    - *"Orice metodă de rezolvare a unei anumite probleme"*.

- În activitatea de programare vom conveni ca prin **algoritm** să înţelegem:

    - *O modalitate de rezolvare a unei probleme utilizând un sistem de calcul"*.

- Un **algoritm** este de fapt o **metodă** sau o **reţetă** pentru a obţine un rezultat dorit.

- Un algoritm constă din:

    - (1) Un **set de date iniţiale** care abstractizează contextul problemei de rezolvat.

    - (2) Un **set de relaţii de transformare** care sunt operate pe baza unor reguli al căror conţinut şi a căror succesiune reprezintă însăşi substanţa algoritmului.

    - (3) Un **set de rezultate preconizate** sau **informaţii finale,** care se obţin trecând de regulă printr-un şir de **informaţii** (rezultate) **intermediare**.

- Un algoritm se bucură de următoarele proprietăţi:

    - (1) **Generalitate** - un algoritm **nu** rezolvă doar o anume problemă ci o clasă generică de probleme de acelaşi tip;

    - (2) **Finitudine** - informaţia finală se obţine din cea iniţială trecând printr-un număr finit de transformări;

    - (3) **Unicitate** - transformările şi ordinea în care ele se aplică sunt univoc determinate de regulile algoritmului.

        - **Consecinţă:** Ori de câte ori se aplică acelaşi algoritm asupra aceluiaşi set de date iniţiale se obţin aceleaşi rezultate.

- **Scopul capitolului**: analiza performanţei algoritmilor.

2.2. Analiza algoritmilor

- La ce serveşte **analiza algoritmilor** ?
    - Permite **precizarea predictivă** a comportamentului algoritmului;
    - Prin analiză pot fi **comparaţi** diferiţi algoritmi şi poate fi astfel ierarhizată experienţa şi îndemânarea diverşilor producători [HS78].
- Analiza algoritmilor se bazează de regulă pe **ipoteze**:
    - (1) Sistemele de calcul sunt considerate **convenţionale** adică ele execută câte **o singură instrucţie** la un moment dat.
    - (2) **Timpul total** de execuţie al algoritmului rezultă din însumarea timpilor instrucţiilor individuale componente.
        - Desigur această ipoteză **nu** este valabilă la sistemele de calcul paralelele şi distribuite.
        - În astfel de cazuri aprecierea performanţelor se realizează de regulă prin **măsurători experimentale** şi **metode statistice**.
- În general, **analiza unui algoritm** se desfăşoară în două etape:
    - (1) **Analiza apriorică**
        - Constă în **aprecierea** din punct de vedere **temporal** a operaţiilor care se utilizează şi a costului lor relativ.
        - Conduce de regulă la stabilirea **expresiei** unei **funcţii** care **mărgineşte timpul de execuţie al algoritmului**
    - (2) **Testul ulterior**
        - Constă în stabilirea unui număr suficient de **seturi de date iniţiale** care să acopere practic **toate posibilităţile** de comportament ale algoritmului.
        - Presupune **testarea** comportamentul algoritmului pentru fiecare set în parte şi culegerea unor date specifice.
        - Se finalizează prin elaborarea unei serii de statistici referitoare la consumul de timp specific execuţiei algoritmului în cauză, adică **profilul algoritmului**.
- **Concluzie**:
    - **Analiza apriorică** are drept scop principal **determinarea teoretică** a **ordinului de mărime al timpului de execuţie al unui algoritm**
    - **Testul ulterior** are ca scop principal **determinarea efectivă** a acestui ordin prin stabilirea **profilului algoritmului**

2.3. Notaţii asimptotice

- **Ordinul de mărime** al timpului de execuţie al unui algoritm:
    - Reprezintă o măsură a eficienţei algoritmului.
    - Permite compararea relativă a variantelor de algoritmi.
- Studiul **eficienţei asimptotice** a unui algoritm, se realizează utilizând mărimi de intrare cu dimensiuni suficient de mari pentru a face relevant **ordinul de mărime** al **timpului de execuţie** al algoritmului.
- Interesează cu precădere **limita** la care tinde **timpul de execuţie al algoritmului** odată cu **creşterea nelimitată** a dimensiunii intrării.
- De regulă, un algoritm care este *"asimptotic mai eficient"* decât alţii, va constitui cea mai bună alegere şi pentru intrări de dimensiuni mici şi foarte mici.

2.3.1. Notaţia Θ (teta)

- Pentru aprecierea limitei ordinului de mărime al timpului de execuţie al unui algoritm se utilizează **notaţia Θ**.
- **Definiţie**. Fiind dată o funcţie $g(n)$, prin $\Theta(g(n))$ se desemnează o **mulţime de funcţii** definită astfel:

$\Theta(g(n))$ = { $f(n)$: există constantele pozitive c_1, c_2 şi n_0 astfel încât
$0 \le c_1 \cdot g(n) \le f(n) \le c_2 \cdot g(n)$ pentru $\forall\, n \ge n_0$ } [2.3.1.a]

- Se spune că o funcţie $f(n)$ aparţine mulţimii $\Theta(g(n))$, dacă există constantele pozitive c_1 si c_2, astfel încât ea poate fi "cuprinsă" (ca un *"sandwich"*) între $c_1 \cdot g(n)$ şi $c_2 \cdot g(n)$ pentru un n suficient de mare.

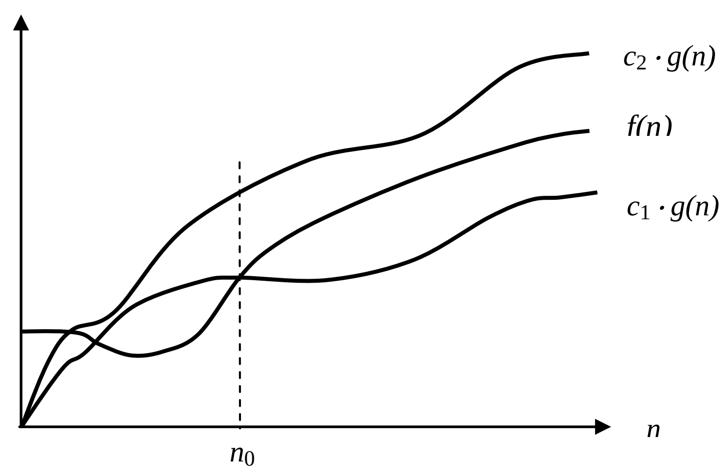

Fig.2.3.a. Reprezentarea lui $f(n) = \Theta(g(n))$

- Deşi $\Theta(g(n))$ reprezintă o mulţime de funcţii, se utilizează uzual notaţia $f(n) = \Theta(g(n))$, cu semnificaţia "$f(n)$ este $\Theta(g(n))$", indicând astfel faptul că $f(n)$ este membru al lui $\Theta(g(n)$ sau că $f(n) \in \Theta(g(n))$ [2.3.1.b].

$f(n) = \Theta(g(n))$ [2.3.1.b]

- Cu alte cuvinte pentru orice $n > n_0$, *f(n)* este egală cu *g(n)* în **interiorul** unui factor constant.

- Se spune ca *g(n)* este o **margine asimptotică strânsă** *("asymptotically tight bound")* a lui *f(n)*.

- Definiţia lui Θ necesită ca fiecare membru a lui Θ*(g(n))* să fie **asimptotic pozitiv**, deci *f(n)* să fie **pozitiv** pentru valori suficient de mari ale lui *n*.

- În **practică**, determinarea lui Θ în cazul unei **expresii polinomiale**, se realizează de regulă luând în considerare **termenii de ordinul cel mai mare** şi neglijând restul termenilor.

- Această afirmaţie poate fi **demonstrată intuitiv**, utilizând definiţia formală a lui Θ în a arăta că

$$\frac{1}{2}\cdot n^2 - 3\cdot n = \Theta(n^2)$$

  - Pentru aceasta, constantele c_1, c_2 şi n_0 trebuiesc determinate astfel încât, pentru orice $n \geq n_0$ să fie valabilă relaţia:

$$c_1\cdot n^2 \leq \frac{1}{2}\cdot n^2 - 3\cdot n \leq c_2\cdot n^2 .$$

  - Se împart membrii inegalităţii cu n^2 şi se obţine

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$$

  - Inegalitatea din dreapta este valabilă pentru orice $n \geq 1$ dacă îl alegem pe $c_2 \geq 1/2$.

  - Inegalitatea din stânga este valabilă pentru orice valoare a lui $n \geq 7$ dacă se alege $c_1 \leq 1/14$.

  - Astfel, alegând $c_1 = 1/14$, $c_2 = 1/2$ şi $n_0 = 7$ se poate verifica simplu că

$$\frac{1}{2}\cdot n^2 - 3\cdot n = \Theta(n^2)$$

- -

- **Exemplul 2.3.1.** Stabilirea funcţiei *g(n)* pentru o **funcţie polinomială**.

- Se porneşte de la observaţia ca termenii de ordin inferior ai unei funcţii asimptotice pozitive pot fi neglijaţi.

  - Dacă se alege pentru c_1 o valoare care este cu puţin inferioară coeficientului termenului celui mai semnificativ,

  - Dacă se alege pentru c_2 o valoare uşor superioară aceluiaşi coeficient,

  - Inegalităţile impuse de notaţia Θ sunt satisfăcute.

- Coeficientul termenului celui mai semnificativ poate fi în continuare omis, întrucât el modifică pe c_1 şi pe c_2 doar cu un factor constant egal cu coeficientul.

- Fie spre **exemplu** funcţia polinomială:

$$f(n) = a \cdot n^2 + b \cdot n + c \qquad a > 0$$

  - Neglijând termenii de ordin inferior lui 2, se ajunge la concluzia ca $f(n) = \Theta(n^2)$.

  - Acest lucru poate fi demonstrat şi **formal** alegând pentru coeficienţi următoarele valori:

$$c_1 = \frac{a}{4} \quad , \quad c_2 = \frac{7a}{4} \quad \text{şi} \quad n_0 = 2 \cdot max\left(\left(\frac{|b|}{a}\right), \sqrt{\frac{|c|}{a}}\right) \qquad \text{[2.3.1.c]}$$

- În baza aceloraşi considerente, pentru orice **funcţie polinomială** $p(n)$ de ordinul n unde a_i este constant şi $a^d > 0$, este valabilă afirmaţia $p(n) = \Theta(n^d)$ ([2.3.1.d]).

Dacă $p(n) = \sum_{i=0}^{d} a_i \cdot n^i$ unde a_i = constant şi $a^d > 0$

atunci $p(n) = \Theta(n^d)$ [2.3.1.d]

- Deoarece o funcţie polinominală de grad zero este o **constantă**, despre orice funcţie constantă se poate spune ca este $\Theta(n^0)$ sau $\Theta(1)$.

- Deşi acesta este un abuz de interpretare (deoarece n nu tinde la infinit), prin **convenţie** $\Theta(1)$ desemnează fie o ***constantă*** fie o ***funcţie constantă*** în raport cu o variabilă.

2.3.2. Notaţia O (O mare)

- Notaţia O desemnează **marginea asimptotică superioară** a unei funcţii. Pentru o funcţie dată $f(n)$, se defineşte $O(g(n))$ ca şi mulţimea de funcţii:

$O(g(n))$ = { $f(n)$: există constantele pozitive c şi n_0 astfel încât
$0 \le f(n) \le c \cdot g(n)$ pentru $\forall\, n \ge n_0$ } [2.3.2.a]

- Notaţia O se utilizează pentru a desemna o **margine superioară** a unei funcţii în **interiorul** unui **factor constant**.

- Pentru toate valorile n superioare lui n_0, valoarea funcţiei $f(n)$ este pe sau dedesubtul lui $g(n)$ (fig.2.3.b).

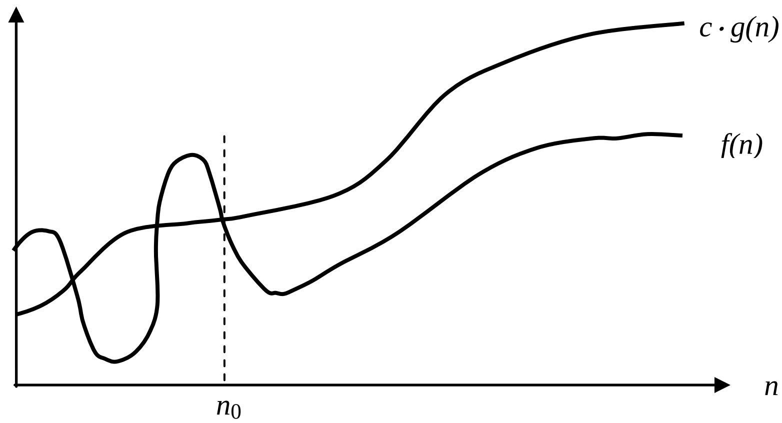

Fig.2.3.b. Reprezentarea lui $f(n)$ = O$(g(n))$

- Pentru a preciza că o funcţie $f(n)$ este membră a lui O$(g(n))$ se utilizează notaţia $f(n)$ = O$(g(n))$.

- Faptul că $f(n)$ este $\Theta(g(n))$ implică că $f(n)$ = O$(g(n))$ deoarece notaţia Θ este mai puternică decât notaţia O. Formal acest lucru se precizează prin relaţia [2.3.2.b].

$\Theta(g(n)) \subseteq O(g(n))$ [2.3.2.b]

- În consecinţă, deoarece s-a demonstrat faptul că orice funcţie pătratică $a \cdot n^2 + b \cdot n + c$, $a > 0$ este $\Theta(n^2)$, în baza relaţiei [2.3.2.b] rezultă ca aceasta funcţie este implicit şi O(n^2).

    - Este surprinzător faptul că din aceleaşi considerente, **funcţia liniară** $a \cdot n + b$ este de asemenea O(n^2), lucru uşor de verificat dacă se alege $c = a + |b|$ şi $n_0 = 1$.

- **Notaţia O** este de obicei cea mai utilizată în aprecierea **timpului de execuţie al algoritmilor** respectiv a **performanţei** acestora**.**

    - Uneori ea poate fi estimată direct din **inspectarea structurii algoritmului**, spre exemplu existenţa unei bucle duble conduce de regulă, la o margine de ordinul O(n^2)

- Deoarece notaţia O descrie o **margine superioară** şi atunci când este utilizată, ea mărgineşte **cazul cel mai defavorabil** de execuţie al unui algoritm.

    - Prin implicaţie, ea **mărgineşte superior** comportamentul algoritmului în aceeaşi măsură pentru **orice** altă intrare.

- In cazul notaţiei Θ lucrurile diferă.

    - Spre exemplu, daca $\Theta(n^2)$ mărgineşte superior cel mai defavorabil timp de execuţie al unui algoritm de sortare prin inserţie, aceasta **nu** înseamnă că $\Theta(n^2)$ mărgineşte **orice intrare**.

    - Astfel, dacă intrarea este gata sortată, algoritmul durează un timp proporţional cu $\Theta(n)$.

- **Tehnic** vorbind, afirmaţia că *timpul de execuţie al unui algoritm de sortare este* O(n^2) constituie un **abuz de limbaj.**

    - **Corect**: indiferent de configuraţia intrării de dimensiune *n*, pentru orice valoare a lui *n*, timpul de execuţie al algoritmului pentru setul corespunzător de intrări este O(n^2).

- Cele mai obişnuite ordine de mărime ale notaţiei O se află în relaţiile de ordine prezentate în [2.3.2.c], iar reprezentarea lor grafică la scară logaritmică apare în fig.2.3.c.

$$O(1) < O(\log n) < O(n) < O(n \cdot \log n) < O(n^2) < O(n^3) <$$
$$< O(2^n) < O(10^n) < O(n!) < O(n^n) \qquad [2.3.2.c]$$

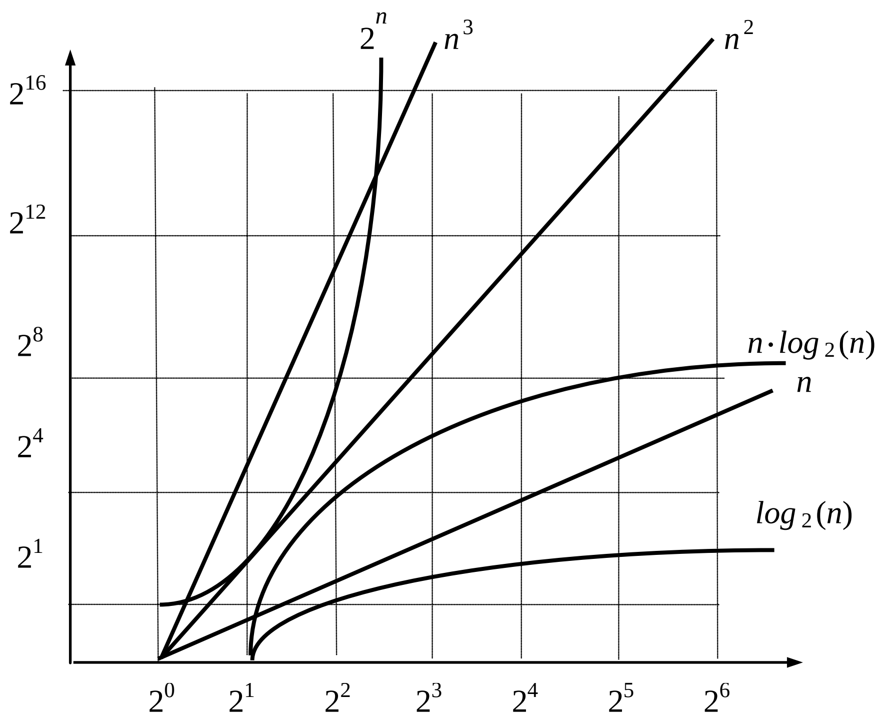

Fig. 2.3.c. Ordine de mărime ale notaţiei O

- În acelaşi context în [2.3.2.d] se prezintă câteva **sume de întregi** utile care sunt frecvent utilizate în **calculul complexităţii algoritmilor**.

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

$$\sum_{i=1}^{n} i^2 = \frac{n \cdot (n+1) \cdot (2 \cdot n+1)}{6} = O(n^3)$$

$$\sum_{i=1}^{n} i^3 = \frac{n^2 \cdot (n^2+1)}{4} = O(n^4) \qquad [2.3.2.d]$$

$$\sum_{i=1}^{n} i^k = \frac{n^{k+1}}{k+1} + \cdots = O\left(\frac{n^{k+1}}{k+1}\right) = O(n^{k+1})$$

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

2.3.3. Aritmetica ordinală a notaţiei O

- În vederea aprecierii complexităţii algoritmilor în raport cu notaţia O, a fost dezvoltată o **aritmetică ordinală** specifică care se bazează pe o serie de **reguli formale** [De89].

- Aceste reguli sunt prezentate în continuare.

  - **Regula 1**. $O(k) < O(n)$ pentru $\forall\, k$.

  - **Regula 2**. Ignorarea constantelor: $k \cdot f(n) = O(f(n))$ pentru $\forall\, k$ şi $\forall\, f$ sau $O(k \cdot f) = O(f)$

  - **Regula 3**. Tranzitivitate: dacă $f(n) \rightarrow O(g(n))$ şi $g(n) \rightarrow O(h(n))$ atunci $f(n) \rightarrow O(h(n))$

- -

Demonstraţie:

- $f(n) = O(g(n)) \Rightarrow \exists\ c_1, n_1 \quad f(n) \leq c_1 \cdot g(n)$ pentru $\forall\, n > n_1$

- $g(n) = O(h(n)) \Rightarrow \exists\ c_2, n_2 \quad g(n) \leq c_2 \cdot h(n)$ pentru $\forall\, n > n_2$

- Se alege $n_3 = max\{n_1, n_2\}$.

- În relaţia $f(n) \leq c_1 \cdot g(n)$ se înlocuieşte $g(n)$ cu expresia de mai sus. Se obţine relaţia:

$$f(n) \leq c_1 \cdot (c_2 \cdot h(n))$$

- Alegând $c_3 = c_1 \cdot c_2$ se obţine $f(n) \leq c_3 \cdot h(n)$ pentru $\forall n > n_3$ deci $f(n) = O(h(n))$.

- -

  - **Regula 4**. $f(n) + g(n) = O(max\{f(n), g(n)\})$

  - **Regula 5**. Dacă $f_1(n) = O(g_1(n))$ şi $f_2(n) = O(g_2(n))$
    atunci $f_1(n) \cdot f_2(n) = O(g_1(n) \cdot g_2(n))$

- Utilizând aceste reguli se poate calcula o **estimare** a lui O pentru o expresie aritmetică dată, **fără** a avea valori explicite pentru *k* şi *n*.

- -

- **Exemplul 2.3.3.** Se consideră expresia

$$8 \cdot n \cdot log(n) + 4 \cdot n^{3/2}$$

  şi se cere o estimare a sa în termenii notaţiei O.

- Se procedează după cum urmează:

  - $log(n) = O(n^{1/2})$ deoarece logaritmul unui număr este mai mic decât orice putere pozitivă a numărului în baza următoarei **teoreme**:

    - Fie $a > 0$ şi $a \neq 1$;

    - Pentru $\forall$ exponent $d > 0$ există un număr N astfel încât pentru $\forall\, x > N$ avem $log_a x < x^d$.

  - $n \cdot log(n) = O(n \cdot n^{1/2}) = O(n^{3/2})$ (Regula 5).

- $8 \cdot n \cdot log(n) = O(n^{3/2})$ (Regula 2).

- $4 \cdot n^{3/2} = O(n^{3/2})$ (Regula 2).

- $8 \cdot n \cdot log(n) + 4 \cdot n^{3/2} = O(max\{8 \cdot n \cdot log(n), 4 \cdot n^{3/2}\}) =$
  $= O(max\{n^{3/2}, n^{3/2}\}) = O(n^{3/2})$ (Regula 4).

- Rezultă în urma estimării că expresia $8 \cdot n \cdot log(n) + 4 \cdot n^{3/2}$ este $O(n^{3/2})$.

2.3.4. Notaţia Ω (Omega mare)

- Notaţia Ω (omega mare) precizează o **margine asimptotică inferioară**.

- Pentru o funcţie dată *f(n)*, prin *Ω(g(n))* se precizează mulţimea funcţiilor

---

$\Omega(g(n)) = \{f(n)$: există constantele pozitive c si n_0 astfel încât
$0 \le c \cdot g(n) \le f(n)$ pentru $\forall\ n \ge n_0\}$ [2.3.4.a]

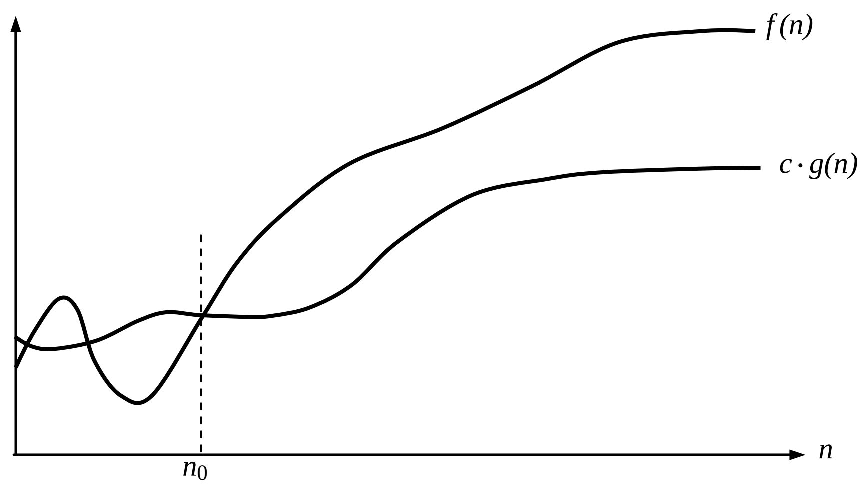

Fig. 2.3.d. Reprezentarea lui *f(n)=Ω(g)n))*

- **Teoremă**: Pentru oricare două funcţii $f(n)$ şi $g(n)$, $f(n) = \Theta(g(n))$
  dacă şi numai dacă $f(n) = O(g(n))$ şi $f(n) = \Omega(g(n))$. [2.3.4.b]

- Spre **exemplu** s-a demonstrat că $a \cdot n^2 + b \cdot n + c = \Theta(n^2)$ pentru orice valori ale constantelor *a, b, c* cu $a > 0$. De aici rezultă imediat că $a \cdot n^2 + b \cdot n + c = \Omega(n^2)$.

- Deoarece notaţia Ω descrie o **limită inferioară**, atunci când este utilizată pentru a mărgini cazul cel mai favorabil de execuţie al unui algoritm, prin implicaţie ea **mărgineşte inferior** orice intrare arbitrară a algoritmului.

2.3.5. Utilizarea notaţiilor asimptotice Θ, O şi Ω

- **Exemplul a).** Se consideră formula $2 \cdot n^2 + 3 \cdot n + 1 = 2 \cdot n^2 + \Theta(n)$. Cum se interpretează această formulă ?

  - În general, când într-o formulă apare o notaţie asimptotică se consideră că ea ţine locul unei funcţii care **nu** se nominalizează explicit.

  - Astfel în exemplul de mai sus avem:

    $$2 \cdot n^2 + 3 \cdot n + 1 = 2 \cdot n^2 + f(n) \text{ , unde } f(n) \in \Theta(n).$$

  - Se observă faptul că în acest caz $f(n) = 3 \cdot n + 1$, care este într-adevăr $\Theta(n)$.

- **Exemplul b).** Utilizarea notaţiei asimptotice poate fi de folos în eliminarea detaliilor neesenţiale ale unei ecuaţii.

  - Spre exemplu fie relaţia recursivă:

    $$T(n) = 2 \cdot T(n/2) + \Theta(n)$$

  - Dacă prezintă interes doar comportamentul asimptotic al lui $T(n)$, nu e necesar să fie specificaţi toţi termenii de ordin inferior.

  - Ei sunt incluşi într-o **funcţie anonimă**, precizată prin $\Theta(n)$.

- **Exemplul c).** Ecuaţia $2 \cdot n^2 + \Theta(n) = \Theta(n^2)$ se interpretează astfel: indiferent de maniera în care se alege funcţia anonimă din partea stângă a ecuaţiei, există o modalitate de a alege funcţia anonimă din partea dreaptă, astfel încât semnul "=" să indice o ecuaţie adevărată.

  - În cazul de faţă, pentru orice funcţie $f(n) \in \Theta(n)$, există o funcţie $g(n) \in \Theta(n^2)$, astfel încât $2 \cdot n^2 + f(n) = g(n)$ pentru orice n.

  - De fapt membrul drept al ecuaţiei evidenţiază un grad mai redus de detaliere decât cel stâng.

- **Exemplul d).** Relaţiile de acest tip pot fi înlănţuite, spre exemplu:

  $$2 \cdot n^2 + 3 \cdot n + 1 = 2 \cdot n^2 + \Theta(n) = \Theta(n^2)$$

  - Prima ecuaţie precizează faptul că există o funcţie $f(n) \in \Theta(n)$ astfel încât $2 \cdot n^2 + 3 \cdot n + 1 = 2 \cdot n^2 + f(n)$ pentru toţi *n*.

  - A doua ecuaţie precizează că pentru orice $g(n) \in \Theta(n)$ există o funcţie $h(n) \in \Theta(n^2)$ astfel încât $2 \cdot n^2 + g(n) = h(n)$ pentru toţi *n*.

  - Această interpretare implică $2 \cdot n^2 + 3 \cdot n + 1 = \Theta(n^2)$ ceea ce de fapt sugerează intuitiv lanţul de egalităţi.

2.3.6. Notaţia o (o mic)

- Marginea asimptotică superioară desemnată prin notaţia O, poate fi din punct de vedere asimptotic *strânsă* sau *lejeră (laxă)*.

- Pentru desemnarea unei **margini asimptotice lejere** se utilizează notaţia o (o mic).

$\text{o}(g(n)) = \{ f(n)$: pentru **orice** constantă pozitivă $c > 0$ există o constantă $n_0 > 0$ astfel încât $0 \le f(n) < c \cdot g(n)$ pentru $\forall\ n \ge n_0 \}$ [2.3.6.a]

---

* Principala diferenţă dintre notaţiile O şi o rezidă în faptul că în cazul $f(n) = O(g(n))$, marginea $0 \le f(n) \le c \cdot g(n)$ este valabilă pentru **anumite constante** $c > 0$, în timp ce $f(n) = \text{o}(g(n))$, marginea $0 \le g(n) < c \cdot g(n)$ este valabilă pentru **orice constantă** $c > 0$.

* Intuitiv, în notaţia o, funcţia $f(n)$ devine nesemnificativă în raport cu $g(n)$ când n tinde la infinit [2.3.6.b].

$$f(n) = o(g(n)) \ \text{ implică } \ \lim_{n \to \infty} \frac{f(n)}{g(n)} = 0 \qquad [2.3.6.b]$$

2.3.7. Notaţia ω (omega)

* Prin analogie, notaţia ω este pentru notaţia Ω ceea ce este o pentru O.

* Cu alte cuvinte notaţia ω precizează o **margine asimptotică inferioară lejeră**.

$\omega(g(n)) = \{ f(n)$: pentru **orice** constantă pozitivă $c > 0$ există o constantă $n_0 > 0$ astfel încât $0 \le c \cdot g(n) < f(n)$ pentru $\forall\ n \ge n_0 \}$ [2.3.7.a]

---

* **Proprietate**: Relaţia $f(n) = \omega(g(n))$ implică [2.3.7.b], adică $f(n)$ devine în mod arbitrar semnificativă în raport cu $g(n)$ atunci când n tinde la infinit.

$$f(n) = \omega(g(n)) \ \text{ implică } \ \lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty \qquad [2.3.7.b]$$

2.3.8. Proprietăţi ale notaţiilor asimptotice

* Presupunând că $f(n)$ şi $g(n)$ sunt **asimptotic pozitive**, multe din proprietăţile relaţionale ale **numerelor reale** se aplică similar **comparaţiilor asimptotice**.

---

a) **Tranzitivitate**:

$f(n) = \Theta(g(n))$ şi $g(n) = \Theta(h(n))$ implică $f(n) = \Theta(h(n))$

$f(n) = O(g(n))$ şi $g(n) = O(h(n))$ implică $f(n) = O(h(n))$

$f(n) = \Omega(g(n))$ şi $g(n) = \Omega(h(n))$ implică $f(n) = \Omega(h(n))$

$f(n) = \text{o}(g(n))$ şi $g(n) = \text{o}(h(n))$ implică $f(n) = \text{o}(h(n))$

$f(n) = \omega(g(n))$ şi $g(n) = \omega(h(n))$ implică $f(n) = \omega(h(n))$

b) **Reflexivitate**:

$f(n) = \Theta(f(n))$;

$f(n) = O(f(n))$; [2.3.8.a]

$f(n) = \Omega(f(n))$.

c) **Simetrie**:

$f(n) = \Theta(g(n))$ dacă şi numai dacă $g(n) = \Theta(f(n))$

d) **Simetrie transpusă**:

$f(n) = O(g(n))$ dacă şi numai dacă $g(n) = \Omega(f(n))$

$f(n) = o(g(n))$ dacă şi numai dacă $g(n) = \omega(f(n))$

---

* **Analogia** dintre **comparaţia asimptotică a două funcţii** *f* şi *g* şi **compararea a două numere reale** *a* şi *b* este prezentată în [2.3.8.b].

$$
\begin{array}{lcl}
f(n) = O(g(n)) & \approx & a \le b \\
f(n) = \Omega(g(n)) & \approx & a \ge b \\
f(n) = \Theta(g(n)) & \approx & a = b \\
f(n) = o(g(n)) & \approx & a < b \\
f(n) = \omega(g(n)) & \approx & a > b
\end{array}
\qquad [2.3.8.b]
$$

---

* O proprietate specifică **numerelor reale**, care **nu** se aplică **comparaţiei asimptotice** este **trihotomia** ("*trichotomy*").

    * Conform acestei proprietăţi între două numere reale oarecare *a* şi *b* **există în mod obligatoriu exact una** din următoarele relaţii:

    $a < b$; $a = b$; $a > b$.

    * Dacă **oricare două numere reale** pot fi comparate conform relaţiei de mai sus, pentru **două funcţii** $f(n)$ şi $g(n)$ este însă posibil ca să **nu** fie valabilă nici $f(n) = O(g(n))$, nici $f(n) = \Omega(g(n))$ şi nici $f(n) = \Theta(g(n))$.

2.4. Aprecierea timpului de execuţie

* Cu ajutorul **notaţiei O mare** se poate aprecia **ordinul timpul de execuţie** al unui algoritm.

* Se face sublinierea aprecierea se referă atât la **ordinul timpului de execuţie** al unui **algoritm abstract** şi cel al unui **program** real rezultat din implementarea respectivului algoritm.

* **Timpul efectiv de execuţie** al unui **program**, depinde de mai mulţi factori cum ar fi:

    * (1) Dimensiunea şi natura datelor de intrare.

    * (2) Caracteristicile sistemului de calcul pe care se rulează programul.

    * (3) Eficienţa codului produs de compilator.

* Notaţia O mare permite **eliminarea** factorilor care **nu** pot fi controlaţi, cum ar fi spre exemplu (2) şi (3) enumeraţi mai sus, concentrându-se asupra comportării algoritmului **independent** de program.

* În general un algoritm a cărui complexitate temporală este $O(n^2)$ va rula ca şi program în $O(n^2)$ unităţi de timp indiferent de limbajul sau sistemul de calcul utilizat.

* (1) În aprecierea timpului de execuţie se porneşte de la **ipoteza simplificatoare** deja enunţată, că fiecare **instrucţie** utilizează în medie aceeaşi cantitate de timp.

* Instrucţiunile care **nu** pot fi încadrate în această medie de timp sunt:

- Instrucţiunea **IF**
- Secvenţele repetitive (buclele)
- Apelurile de proceduri şi funcţii.

- Presupunând pentru moment că apelurile de proceduri şi funcţii se ignoră, se adoptă prin **convenţie** următoarele simplificări:

- (2) Se presupune că o **instrucţiune IF** va consuma întotdeauna timpul necesar execuţiei **ramurii celei mai lungi**, dacă nu există raţiuni contrare justificate;

- (3) Se presupune că întotdeauna instrucţiunile din interiorul unei **bucle** se vor executa de **numărul maxim de ori** permis de condiţia de control.

- **Exemplul 2.4.a.**

- Se consideră o procedură care:
  - (1) Caută într-un tablou `a` cu `n` elemente, un element egal cu `cheie.`
  - (2) Returnează în variabila `unde` ultima locaţie în care este găsită cheia sau zero în caz contrar [2.4.a].

```
------------------------------------------------------------
PROCEDURE CautareLiniara(n,cheie: integer,
                 a: TablouNumeric; VAR unde: integer);
VAR indice: integer;
  BEGIN
    unde: = 0
    FOR indice:= 1 TO n DO
      IF a[indice]= cheie THEN                    [2.4.a]
        unde:= indice
  END; {CautareLiniara}
------------------------------------------------------------
```

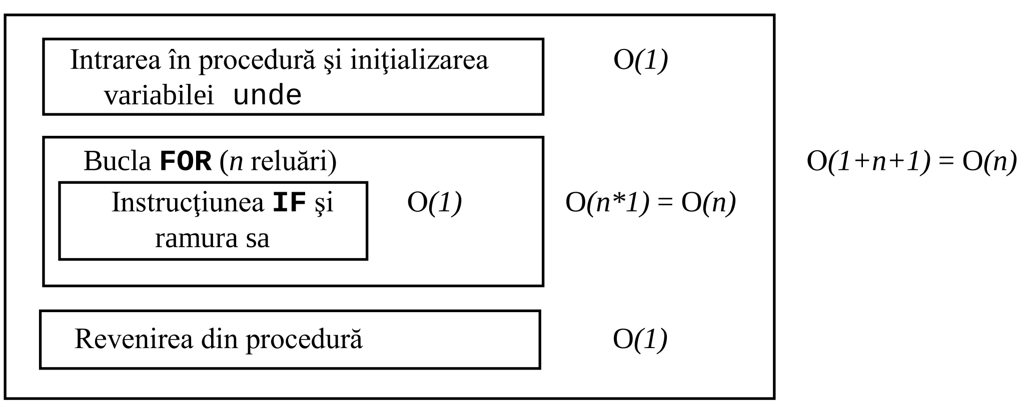

Fig. 2.4.a. Schema temporală a algoritmului [2.4.a]

- În figura 2.4.a apare **schema temporală** a algoritmului împreună cu estimarea O mare corespunzătoare.

- În analiză au fost introduse şi operaţiile de intrare şi revenire din procedură care **nu** apar ca părţi explicite ale rutinei.

    - Pe viitor apelul şi revenirea din procedură vor fi omise din analiza temporală deoarece ele contribuie cu un timp constant la execuţie şi nu influenţează estimarea O mare.

- **Exemplul 2.4.b**. Se consideră următoarea porţiune de cod [2.4.b].

```
-------------------------------------------------------
FUNCTION SumProd(n: integer): integer;
  VAR rezultat,k,i: integer;
  BEGIN
    rezultat:= 0;
    FOR k:= 1 TO n                                [2.4.b]
      FOR  i:= 1 TO k
        rezultat:= rezultat+k*i;
    SumProd:= rezultat
  END;{SumProd}
-------------------------------------------------------
```

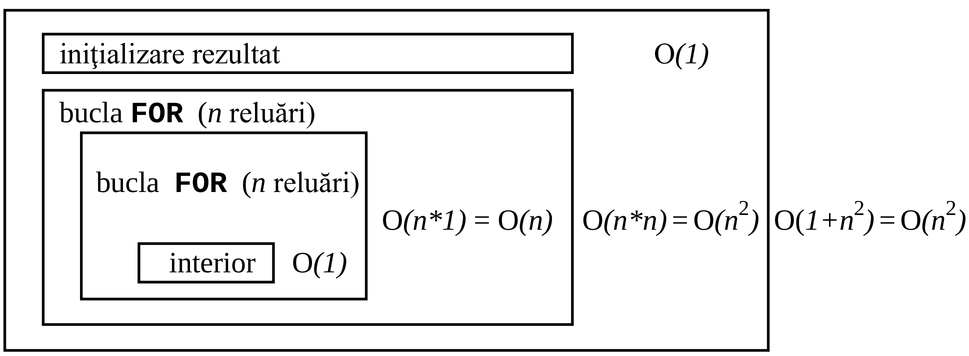

Fig. 2.4.b. Schema temporală a algoritmului din secvenţa [2.4.b]

- La analiza complexităţii temporale se face presupunerea că ambele bucle se reiau de **numărul maxim** de ori posibil

    - În realitate acest lucru **nu** este adevărat deoarece bucla interioară se execută cu limita $k \le n$ (şi nu cu limita n)

- Se va **demonstra** că prin această simplificare estimarea temporală finală **nu** este afectată.

- La o analiză mai aprofundată se ţine cont de faptul că bucla **FOR** interioară se execută prima oară o dată, a doua oară de 2 ori ş.a.m.d, a k-oară de k ori.

    - În consecinţă **numărul exact de reluari** este cel precizat de expresia de mai jos şi după cum se observă el este $O(n^2)$

$$1 + 2 + 3 + ... + n \;=\; \frac{n(n+1)}{2} = O(n^2) \qquad [2.4.c]$$

- În mod analog în cadrul secvenţei [2.4.d] se ajunge la estimarea: $O(n \cdot n \cdot n) = O(n^3)$.

```
FOR i:= 1 TO n DO
  FOR j:= 1 TO i DO
    FOR k:= 1 TO i DO                                      [2.4.d]
     Secvenţă {O(1)};
```

- Calculul **exact** al numărului de reluări conduce la valoarea precizată de relaţia [2.4.e], ţinând cont de faptul că cele două bucle interioare se execută de i^2 ori la fiecare reluare a buclei exterioare **FOR**.

$$1^2 + 2^2 + 3^2 + ... + n^2 = \frac{n \cdot (n+1) \cdot (2 \cdot n+1)}{6} = O(n^3) \qquad [2.4.e]$$

Exemplul 2.4.c. se referă tot la o buclă multiplicativă.

```
m:= 1
FOR i:= 1 TO n DO
  BEGIN
    m:= m *2;
    FOR  j:= 1  TO m DO                                    [2.4.f]
     Secventa    {O(1)}
  END; {FOR}
```

- În situaţia în care bucla exterioară se execută pentru valorile 1, 2, 3, ... ale lui `i`, bucla interioară iterează de 2, 4, 8, ... ori conducând la un timp de execuţie de ordinul $O(2+4+8+ ... +2^n)$.

$$2 + 4 + 8 + ... + 2^n = \frac{2 \cdot (2^n - 1)}{2-1} = O(2^{n+1}) \qquad [2.4.g]$$

  - Această estimare este complet diferită de estimarea $O(n^2)$ pentru care există tentaţia de a fi avansată şi care este valabilă pentru buclele nemultiplicative.

2.5. Profilarea unui algoritm

- Presupunem că un algoritm a fost conceput, implementat, testat şi depanat pe un sistem de calcul ţintă.

- Ne interesează de regulă **profilul performanţei** sale, adică **timpii precişi** de execuţie ai algoritmului pentru diferite seturi de date, eventual pe diferite sisteme ţintă.

- Pentru aceasta sistemul de calcul ţintă trebuie să fie dotat cu un **ceas intern** şi cu **funcţii** sistem de acces la acest ceas.

- După cum s-a mai precizat, determinarea profilului performanţei face parte din **testul ulterior** al unui algoritm şi are drept scop determinarea precisă a **ordinul de mărime** al **timpului de execuţie al algoritmului**.

- Informaţiile rezultate sunt utilizate de regulă pentru a valida sau invalida, respectiv pentru a nuanţa rezultatele **estimării apriorice**.

- Se presupune un algoritm implementat în forma unui program numit `Algoritm(X: Intrare,Y: Iesire)` unde `X` este intrarea iar `Y` ieşirea.

- Pentru a construi **profilul algoritmului** este necesar să fie concepute:

- (1) Seturile de **date de intrare** a căror dimensiune creşte între anumite limite, pentru a studia comportamentul algoritmului în raport cu **dimensiunea intrării.**

- (2) Seturile de **date de intrare** care în principiu se referă la **cazurile extreme** de comportament.

- (3) O **procedură** cu ajutorul căreia poate fi construit profilul algoritmului în baza seturilor de date anterior amintite.

```
------------------------------------------------------------
procedure Profil;
{procedura construieşte profilul procedurii Algoritm(X:
Intrare,Y: Ieşire)}
begin
    *initializează procedura Algoritm;
    *afiseaza("Testul lui Algoritm. Timpii în milisecunde");
     repeat
        *citeste(SetDeDate);                        [2.5.a]
        *afiseaza("Un nou set de date:", SetDeDate);
        *apel TIME(t);  {atribuie lui t valoarea curentă
                         a ceasului sistem}
        *apel Algoritm(SetDeDate, Rezultate);
        *apel TIME(t1);
        *afiseaza("TimpExecuţie=", t - t1);
     until sfârşit(SetDeDate)
end {Profil}
------------------------------------------------------------
```

- Procedura **Profil** poate fi utilizată în mai multe **scopuri** funcţie de obiectivele urmărite.

    - (1) Evidenţierea **performanţei intrinseci** a unui algoritm precizat.

        - Pentru aceasta se aleg, aşa cum s-a mai precizat, seturi de date cu dimensiuni din ce în ce mai mari.

        - Rezultatul va fi **profilul** algoritmului.

        - Pentru deplina conturare a profilului se testează de asemenea şi cazurile de **comportament extrem** ale algoritmului respectiv cazul cel mai **favorabil** şi cel mai **defavorabil**.

    - (2) Evidenţierea **performanţei relative** a doi sau mai mulţi **algoritmi diferiţi** care îndeplinesc aceeaşi sarcină.

        - În acest scop se execută procedura **Profil** pentru fiecare din algoritmi în parte, cu aceleaşi seturi de date inţiale, pe un acelaşi sistem de calcul.

        - Compararea profilelor rezultate permite **ierarhizarea performanţelor** algoritmilor analizaţi.

    - (3) Evidenţierea **performanţei relative** a două sau mai multe **sisteme de calcul.**

        - În acest scop se rulează procedura `Profil` pentru un acelaşi algoritm, cu aceleaşi date iniţiale pe sistemele de calcul ţintă supuse analizei.

        - Compararea profilelor rezultate permite **ierarhizarea performanţelor** sistemelor de calcul analizate.

        - De regulă, pe piaţa sistemelor de calcul se utilizează în acest scop algoritmi consacraţi, în anumite cazuri standardizaţi, cunoscuţi sub denumirea de "**bench marks**" în baza cărora sunt evidenţiate performanţele diferitelor arhitecturi de sisteme de calcul.

- Trebuie însă acordată o **atenţie specială** procedurii `TIME` a căror rezultate în anumite circumstanţe pot fi nerevelevante.

    - Astfel, în cazul sistemelor **time-sharing**, al **sistemelor complexe** care lucrează cu întreruperi sau al **sistemelor multi-procesor**, timpul furnizat de această funcţie poate fi complet needificator.

    - În astfel de cazuri, una din soluţii este aceea de a executa programul de un număr suficient de ori şi de a apela la **metode statistice** pentru evidenţierea performanţelor algoritmului testat.