Capitolul 3

Analiza complexităţii algoritmilor

Analiza complexităţii unui algoritm are ca scop estimarea volumului de *resurse de calcul* necesare pentru execuţia algoritmului. Prin resurse se înţelege:

- *Spaţiul de memorie* necesar pentru stocarea datelor pe care le prelucrează algoritmul.
- *Timpul* necesar pentru execuţia tuturor prelucrărilor specificate în algoritm.

Această analiză este utilă pentru a stabili dacă un algoritm utilizează un volum acceptabil de resurse pentru rezolvarea unei probleme. În caz contrar algoritmul, chiar dacă este corect, nu este considerat eficient şi nu poate fi aplicat în practică. Analiza complexităţii, numită şi analiza eficienţei algoritmilor, este utilizată şi în compararea algoritmilor cu scopul de a-l alege pe cel mai eficient (cel care foloseşte cele mai puţine resurse de calcul).

În majoritatea algoritmilor volumul resurselor necesare depinde de *dimensiunea* problemei de rezolvat. Aceasta este determinată de regulă de volumul datelor de intrare. În cazul cel mai general acesta este dat de numărul biţilor necesari reprezentării datelor. Dacă se prelucrează o valoare numerică, n (de exemplu, se verifică dacă n este număr prim) atunci ca dimensiune a problemei se consideră numărul de biţi utilizaţi în reprezentarea lui n, adică $\lfloor \log_2 n \rfloor + 1$. Dacă datele de prelucrat sunt organizate sub forma unor tablouri atunci dimensiunea problemei poate fi considerată ca fiind numărul de componente ale tablourilor (de exemplu la determinarea minimului dintr-un tablou cu n elemente sau în calculul valorii unui polinom de gradul n se consideră că dimensiunea problemei este n). Sunt situaţii în care volumul datelor de intrare este specificat prin mai multe valori (de exemplu în prelucrarea unei matrici cu m

linii şi n coloane). În aceste cazuri dimensiunea problemei este reprezentată de toate valorile respective (de exemplu, (m, n)).

Uneori la stabilirea dimensiunii unei probleme trebuie să se ţină cont şi de prelucrările ce vor fi efectuate asupra datelor. De exemplu, dacă prelucrarea unui text se efectuează la nivel de cuvinte atunci dimensiunea problemei va fi determinată de numărul cuvintelor, pe când dacă se efectuează la nivel de caractere atunci va fi determinată de numărul de caractere.

Spaţiul de memorare este influenţat de modul de reprezentare a datelor. De exemplu, o matrice cu elemente întregi având 100 de linii şi 100 de coloane din care doar 50 sunt nenule (matrice rară) poate fi reprezentată în una dintre variantele: (i) tablou bidimensional 100×100 (10000 de valori întregi); (ii) tablou unidimensional în care se reţin doar cele 50 de valori nenule şi indicii corespunzători (150 de valori întregi). Alegerea unui mod eficient de reprezentare a datelor poate influenţa complexitatea prelucrărilor. De exemplu algoritmul de adunare a două matrici rare devine mai complicat în cazul în care acestea sunt reprezentate prin tablouri unidimensionale. În general obţinerea unor algoritmi eficienţi din punct de vedere al timpului de execuţie necesită mărirea spaţiului de memorie alocat datelor şi reciproc.

Dintre cele două resurse de calcul, spaţiu şi timp, cea critică este timpul de execuţie. În continuare vom analiza doar dependenţa dintre timpul de execuţie (înţeles de cele mai multe ori ca număr de repetări ale unor operaţii) al unui algoritm şi dimensiunea problemei.

3.1 Timp de execuţie

În continuare vom nota cu $T(n)$ timpul de execuţie al unui algoritm destinat rezolvării unei probleme de dimensiune n. Pentru a estima timpul de execuţie trebuie stabilit un *model de calcul* şi o unitate de măsură. Vom considera un model de calcul (numit şi maşină de calcul cu acces aleator) caracterizat prin:

- Prelucrările se efectuează în mod secvenţial.
- Operaţiile *elementare* sunt efectuate în timp constant *indiferent* de valoarea operanzilor.
- Timpul de acces la informaţie nu depinde de poziţia acesteia (nu sunt diferenţe între prelucrarea primului element al unui tablou şi cea a oricărui alt element).

A stabili o unitate de măsură înseamnă a stabili care sunt operaţiile elementare şi a considera ca unitate de măsură timpul acestora de execuţie. În acest fel timpul de execuţie va fi exprimat prin numărul de operaţii elementare executate. Sunt considerate operaţii elementare cele aritmetice (adunare, scădere, înmulţire, împărţire), comparaţiile şi cele logice (negaţie, conjuncţie şi disjuncţie).

1: $S \leftarrow 0$			
2: $i \leftarrow 0$	Operaţie	Cost	Nr. repetări
3: **while** $i < n$ **do**	1	c_1	1
4: $\quad i \leftarrow i + 1$	2	c_2	1
5: $\quad S \leftarrow S + i$	3	c_3	$n + 1$
6: **end while**	4	c_4	n
7: **return** S	5	c_5	n

Tabelul 3.1: Analiza costurilor în cazul algoritmului de calcul a sumei primelor n numere naturale

Cum scopul calculului timpului de execuţie este de a permite compararea algoritmilor, uneori este suficient să se contorizeze doar anumite tipuri de operaţii elementare, numite *operaţii de bază* (de exemplu în cazul unui algoritm de căutare sau de sortare se pot contoriza doar operaţiile de comparare) şi/sau să se considere că timpul de execuţie a acestora este unitar (desi operaţiile de înmmulţire şi împărţire sunt mai costisitoare decât cele de adunare şi scădere în analiză se poate considera că ele au acelaşi cost).

Timpul de execuţie al întregului algoritm se obţine însumând timpii de execuţie ai prelucrărilor componente.

Exemplul 3.1 Considerăm problema calculului sumei $\sum_{i=1}^{n} i$. Dimensiunea acestei probleme poate fi considerată n. Algoritmul şi tabelul cu costurile corespunzătoare prelucrărilor sunt prezentate în Tabelul 3.1. Însumând timpii de execuţie ai prelucrărilor elementare se obţine: $T(n) = n(c_3 + c_4 + c_5) + c_1 + c_2 + c_3 = k_1 n + k_2$, adică timpul de execuţie depinde liniar de dimensiunea problemei. Costurile operaţiilor elementare influenţează doar constantele ce intervin în funcţia $T(n)$.

Calculul sumei poate fi realizat şi utilizând o prelucrare de tip **for** $i \leftarrow 1, n$ **do** $S \leftarrow S + i$ **endfor**. Costul gestiunii contorului (iniţializare, testare şi incrementare) fiind $c_2 + (n + 1)c_3 + nc_4$. În cazul în care toate prelucrările au cost unitar se obţine $2(n + 1)$ pentru costul gestiunii contorului unui ciclu **for** cu n iteraţii. Ipoteza costului unitar nu este în întregime corectă întrucât iniţializarea constă într-o simplă atribuire iar incrementarea într-o adunare şi o atribuire. Totuşi pentru a simplifica analiza o vom utiliza în continuare.

Exemplul 3.2 Considerăm problema determinării produsului a două matrici: A de dimensiune $m \times n$ şi B de dimensiune $n \times p$. În acest caz dimensiunea problemei este determinată de trei valori: (m, n, p). Algoritmul şi analiza costurilor este prezentată în Tabelul 3.2.

Costul prelucrărilor de pe liniile 1, 2 şi 4 reprezintă costul gestiunii contorului şi va fi tratat global. Presupunând că toate operaţiile aritmetice şi de comparare au cost unitar obţine timpul de execuţie $T(m, n, p) = 4mnp + 5mp + 4m + 2$.

```
1: for i ← 1, m do
2:    for j ← 1, p do
3:       c[i, j] ← 0
4:       for k ← 1, n do
5:          c[i, j] ← c[i, j] + a[i, k] ∗ b[k, j]
6:       end for
7:    end for
8: end for
9: return c[1..m, 1..p]
```

Op.	Cost	Nr. repet.
1	$2(m+1)$	1
2	$2(p+1)$	m
3	1	$m \cdot p$
4	$2(n+1)$	$m \cdot p$
5	2	$m \cdot p \cdot n$

Tabelul 3.2: Analiza costurilor în cazul produsului a două matrici

```
1: m ← x[1]
2: for i ← 2, n do
3:    if m > x[i] then
4:       m ← x[i]
5:    end if
6: end for
7: return m
```

Operaţie	Cost	Nr. repetări
1	1	1
2	$2n$	1
3	1	$n-1$
4	1	$\tau(n)$

Tabelul 3.3: Analiza costurilor în cazul algoritmului de determinare a minimului dintr-un tablou

În practică nu este necesară o analiză atât de detaliată ci este suficient să se identifice *operaţia dominantă* şi să se estimeze numărul de repetări ale acesteia. Prin operaţie dominantă se înţelege operaţia care contribuie cel mai mult la timpul de execuţie a algoritmului şi de regulă este operaţia ce apare în ciclul cel mai interior. În exemplul de mai sus ar putea fi considerată ca operaţie dominantă, operaţia de înmulţire. În acest caz costul execuţiei algoritmului ar fi $T(m, n, p) = mnp$.

Exemplul 3.3 Considerăm problema determinării valorii minime dintr-un tablou $x[1..n]$. Dimensiunea problemei este dată de numărul n de elemente ale tabloului. Prelucrările algoritmului şi costurile corespunzătoare sunt prezentate în Tabelul 3.3.

Spre deosebire de exemplele anterioare timpul de execuţie nu poate fi calculat explicit întrucât numărul de repetări ale prelucrării numerotate cu 4 depinde de valorile aflate în tablou. Dacă cea mai mică valoare din tablou se află chiar pe prima poziţie atunci prelucrarea 4 nu se efectuează nici o dată iar $\tau(n) = 0$. Acesta este considerat cazul cel mai *favorabil*.

Dacă, în schimb, elementele tabloului sunt în ordine strict descrescătoare atunci prelucrarea 4 se efectuează la fiecare iteraţie adică $\tau(n) = n - 1$. Acesta este cazul cel mai *defavorabil*.

```
1: gasit ← false
2: i ← 1
3: while (gasit =false) and i ≤ n do
4:    if v = x[i] then
5:       gasit ← true
6:    else
7:       i ← i + 1
8:    end if
9: end while
10: return gasit
```

Operaţie	Cost	Nr. repetări
1	1	1
2	1	1
3	3	$\tau_1(n)+1$
4	1	$\tau_1(n)$
5	1	$\tau_2(n)$
6	1	$\tau_3(n)$

Tabelul 3.4: Analiza costurilor în cazul căutării secvenţiale

Timpul de execuţie poate fi astfel încadrat între două limite: $3n \leq T(n) \leq 4n-1$. În acest caz este uşor de observat că se poate considera ca operaţie dominantă cea a comparării dintre valoarea variabilei m şi elementele tabloului. În acest caz costul algoritmului ar fi $T(n) = n-1$. In ambele situaţii dependenţa timpului de execuţie de dimensiunea problemei este liniară.

Exemplul 3.4 Considerăm problema căutării unei valori v într-un tablou $x[1..n]$. Dimensiunea problemei este din nou n iar o primă variantă a algoritmului este şi costurile corespunzătoare algoritmului este prezentată în Tabelul 3.4.

În cazul în care valoarea v se află în şir notăm cu k prima poziţie pe care se află. Se obţine:

$$\tau_1(n) = \begin{cases} k & \text{valoarea se află în şir} \\ n & \text{valoarea nu se află în şir} \end{cases} \qquad \text{deci } 1 \leq \tau_1(n) \leq n.$$

$$\tau_2(n) = \begin{cases} 1 & \text{valoarea se află în şir} \\ 0 & \text{valoarea nu se află în şir} \end{cases}.$$

$$\tau_3(n) = \begin{cases} k-1 & \text{valoarea se află în şir} \\ n & \text{valoarea nu se află în şir} \end{cases} \qquad \text{deci } 0 \leq \tau_3(n) \leq n.$$

Cazul cel mai favorabil este cel în care valoarea se află pe prima poziţie în tablou, caz în care $T(n) = 3(\tau_1(n)+1) + \tau_1(n) + \tau_2(n) + \tau_3(n) + 2 = 6 + 1 + 1 + 0 + 2 = 10$.

Cazul cel mai defavorabil este cel în care valoarea nu se află în tablou: $T(n) = 3(n+1) + n + 0 + n + 2 = 5(n+1)$.

O altă variantă a algoritmului de căutare este descrisa în Tabelul 3.5. Dacă există k astfel încât $x[k] = v$ atunci $\tau(n) = k-1$. In acest caz numărul total de operaţii este $4k+2$. În cazul cel mai favorabil numărul de operaţii este 6 iar în cazul cel mai defavorabil numărul de operaţii este $4n+2$.

```
1: i ← 1
2: while (x[i] ≠ v) and (i < n) do
3:    i ← i + 1
4: end while
5: if x[i] = v then
6:    gasit ←true
7: else
8:    gasit ←false
9: end if
10: return gasit
```

Operaţie	Cost	Nr. repetări
1	1	1
3	3	$\tau(n)+1$
4	1	$\tau(n)$
5-8	2	1

Tabelul 3.5: Analiza costurilor în cazul căutării secvenţiale

3.1.1 Analiza cazurilor extreme

Aşa cum rezultă din analiza algoritmului de căutare prezentat în Exemplul 3.4. numărul de operaţii executate depinde uneori nu doar de dimensiunea problemei ci şi de proprietăţile datelor de intrare. Astfel pentru instanţe diferite ale problemei se execută un număr diferit de prelucrări. În astfel de situaţii numărul de operaţii executate nu poate fi calculat exact, astfel că se calculează margini ale acestuia analizând cele două cazuri extreme: cazul cel mai *favorabil* şi cazul cel mai *defavorabil*.

Cazul favorabil corespunde acelor instanţe ale problemei pentru care numărul de operaţii efectuate este cel mai mic. Analiza în cazul cel mai favorabil permite identificarea unei limite inferioare a timpului de execuţie. Această analiză este utilă pentru a identifica algoritmi ineficienţi (dacă un algoritm are un cost mare în cel mai favorabil caz, atunci el nu poate fi considerat o variantă acceptabilă). Sunt situaţii în care frecvenţa instanţelor corespunzătoare celui mai favorabil caz sau apropiate acestuia este mare. In astfel de situatii estimările obţinute în cel mai favorabil caz sunt furnizează informaţii utile despre algoritm. De exemplu algoritmul de sortare prin inserţie, analizat în Capitolul 4, se comportă bine în cazul în care tabloul este deja sortat, iar această comportare rămâne valabilă şi pentru tablouri "aproape" sortate.

Cazul cel mai defavorabil corespunde instanţelor pentru care numărul de operaţii efectuate este maxim. Analiza acestui caz furnizează o limită superioară a timpului de execuţie . În aprecierea şi compararea algoritmilor interesează în special cel mai defavorabil caz deoarece furnizează cel mai mare timp de execuţie relativ la *orice* date de intrare de dimensiune dată. Pe de altă parte pentru anumiţi algoritmi cazul cel mai defavorabil este relativ frecvent.

3.1.2 Analiza cazului mediu

Uneori, cazurile extreme (cel mai defavorabil şi cel mai favorabil) se întâlnesc rar, astfel că analiza acestor cazuri nu furnizează suficientă informaţie despre

algoritm.

În aceste situaţii este utilă o altă măsură a complexităţii algoritmilor şi anume *timpul mediu de execuţie*. Acesta reprezintă o valoare medie a timpilor de execuţie calculată în raport cu distribuţia de probabilitate corespunzătoare spaţiului datelor de intrare. Stabilirea acestei distribuţii de probabilitate presupune împărţirea mulţimii instanţelor posibile ale problemei în clase astfel încât pentru instanţele din aceeaşi clasă numărul de operaţii efectuate să fie acelaşi.

Dacă $\nu(n)$ reprezintă numărul cazurilor posibile (clase de instanţe pentru care algoritmul efectuează acelaşi număr de operaţii), P_k este probabilitatea de apariţie a cazului k iar $T_k(n)$ este timpul de execuţie corespunzător cazului k atunci timpul mediu este dat de relaţia:

$$T_m(n) = \sum_{k=1}^{\nu(n)} T_k(n) P_k.$$

Dacă toate cazurile sunt echiprobabile ($P_k = 1/\nu(n)$) se obţine $T_m(n) = \sum_{k=1}^{\nu(n)} T_k(n)/\nu(n)$.

Exemplu. Considerăm din nou problema căutării unei valori v într-un tablou $x[1..n]$ (exemplul 3.4). Pentru a simplifica analiza vom considera că elementele tabloului sunt distincte. Pentru a calcula timpul mediu de execuţie trebuie să facem ipoteze asupra distribuţiei datelor de intrare.

Dificultatea principală în stabilirea timpului mediu constă în stabilirea distribuţiei corespunzătoare spaţiului datelor de intrare. Pentru exemplul în discuţie s-ar putea lua în considerare şi ipoteza că probabilitatea ca valoarea v să se afle în tablou este $P(v \text{ se află în tablou}) = p$, iar probabilitatea ca valoarea să nu se afle în tablou este $P(v \text{ nu se află în tablou}) = 1-p$. In plus considerăm că în cazul în care se află în tablou elementul căutat se găseşte cu aceeaşi probabilitate pe oricare dintre cele n poziţii: $P(v \text{ se află pe poziţia k}) = 1/n$. În acest caz timpul mediu este:

$$T_m(n) = \frac{p}{n} \sum_{k=1}^{n} 5(k+1) + 5(1-p)(n+1) = \frac{(10-5p)n + (5p+10)}{2}$$

Dacă şansa ca elementul să se afle în şir coincide cu cea ca el să nu se afle atunci se obţine $T_m(n) = (15n+25)/4$.

O altă ipoteză asupra distribuţiei de probabilitate corespunzătoare diferitelor instanţe ale problemei este să considerăm că valoarea v se poate afla pe oricare dintre poziţiile din tablou sau în afara acestuia cu aceeaşi probabilitate. Cum numărul cazurilor posibile este $\nu(n) = n+1$ (n cazuri în care valoarea se află în cadrul tabloului şi unul în care v nu se află în tablou) rezultă că probabilitatea fiecărui caz este $1/(n+1)$. Cum timpul corespunzător cazului în care v se află pe poziţia k este $T_k = 5(k+1)$ iar cel corespunzător cazului în care valoarea

nu se află în tablou este $T_{n+1} = 5(n+1)$ rezultă că timpul mediu de execuţie este:

$$T_m(n) = \frac{1}{n+1}(\sum_{k=1}^{n} 5(k+1) + 5(n+1)) = \frac{5(n^2+5n+2)}{2(n+1)}$$

Această ipoteză este naturală în cazul în care se analizează o variantă a algoritmului caracterizată prin faptul că se adaugă la tablou un element care coincide cu valoarea căutată. Aceasta variantă se bazează pe *tehnica fanionului* şi este descrisă în algoritmul 3.1. Algoritmul returnează poziţia pe care se află valoarea căutată. Dacă poziţia returnată este $n+1$ atunci înseamnă că valoarea căutată nu se află în tabloul iniţial.

Algoritmul 3.1 Căutare secvenţială folosind tehnica fanionului

```
1: x[n+1] ← v
2: i ← 1
3: while x[i] ≠ v do
4:     i ← i + 1
5: end while
6: return i
```

Pentru cazul în care valoarea căutată se află pe poziţia k ($k \in \{1, \ldots, n+1\}$) numărul de operaţii efectuate este $T_k(n) = 2k + 1$. Astfel timpul mediu este $T_m(n) = \left(\sum_{k=1}^{n+1}(2k+1)\right)/(n+1) = n+3$.

Timpul mediu de execuţie depinde de ipotezele făcute asupra distribuţiei datelor de intrare şi în general nu este o simplă medie aritmetică a timpilor corespunzători cazurilor extreme (cel mai favorabil respectiv cel mai defavorabil).

Datorită dificultăţilor ce pot interveni în estimarea timpului mediu şi datorită faptului că în multe situaţii acesta diferă de timpul în cazul cel mai defavorabil doar prin valori ale constantelor implicate, de regulă analiza se referă la estimarea timpului în cazul cel mai defavorabil. Timpul mediu are semnificaţie atunci când pentru problema în studiu cazul cel mai defavorabil apare rar.

3.2 Ordin de creştere

Pentru a aprecia eficienţa unui algoritm nu este necesară cunoaşterea expresiei detaliate a timpului de execuţie. Mai degrabă interesează modul în care timpul de execuţie creşte o dată cu creşterea dimensiunii problemei. O măsură utilă în acest sens este *ordinul de creştere*. Acesta este determinat de *termenul dominant* din expresia timpului de execuţie. Când dimensiunea problemei este mare valoarea termenului dominant depăşeşte semnificativ valorile celorlalţi termeni astfel că aceştia din urmă pot fi neglijaţi. Pentru a stabili ordinul de

creştere al timpului de execuţie al unui algoritm este suficient să se contorizeze operaţia dominantă (cea mai costisitoare şi mai frecvent executată).
Exemple. Dacă $T(n) = an + b$ $(a > 0)$ când dimensiunea problemei creşte de k ori şi termenul dominant din timpul de execuţie creşte de acelaşi număr de ori căci $T(kn) = k \cdot (an) + b$. În acest caz este vorba despre un ordin liniar de creştere. Dacă $T(n) = an^2+bn+c$ $(a > 0)$ atunci $T(kn) = k^2 \cdot (an^2)+k \cdot (bn)+c$, deci termenul dominant creşte de k^2 ori, motiv pentru care spunem că este un ordin pătratic de creştere.

Dacă $T(n) = a\lg n$ atunci $T(kn) = a\lg n + a\lg k$, adică termenul dominant nu se modifică, timpul de execuţie crescând cu o constanta (în raport cu n). În relaţiile anterioare şi in toate cele ce vor urma prin lg notăm logaritmul în baza 2 (întrucât trecerea de la o bază la alta este echivalentă cu înmulţirea cu o constantă ce depinde doar de bazele implicate, iar în stabilirea ordinului de creştere se ignoră constantele, în analiza eficienţei baza logaritmilor nu este relevantă). Un ordin logaritmic de creştere reprezintă o comportare bună. Dacă în schimb $T(n) = a2^n$ atunci $T(kn) = a(2^n)^k$ adică ordinul de creştere este exponenţial.

Întrucât problema eficienţei devine critică pentru probleme de dimensiuni mari analiza eficienţei se efectuează pentru valori mari ale lui n (teoretic se consideră că $n \to \infty$), în felul acesta luându-se în considerare doar comportarea termenului dominant. Acest tip de analiză se numeşte *analiză asimptotică*. În cadrul analizei asimptotice se consideră că un algoritm este mai eficient decât altul dacă ordinul de creştere al timpului de execuţie al primului este mai mic decât cel al celui de-al doilea.

Relaţia între ordinele de creştere are semnificaţie doar pentru dimensiuni mari ale problemei. Dacă considerăm timpii $T_1(n) = 10n + 10$ şi $T_2(n) = n^2$ atunci se observă cu uşurintă că $T_1(n) > T_2(n)$ pentru $n \leq 10$, deşi ordinul de creştere al lui T_1 este evident mai mic decât cel al lui T_2 (Figura 3.1).

Prin urmare un algoritm asimptotic mai eficient decât altul reprezintă varianta cea mai bună doar în cazul problemelor de dimensiuni mari.

3.3 Notaţii asimptotice

Pentru a uşura analiza asimptotică şi pentru a permite gruparea algoritmilor în clase în funcţie de ordinul de creştere a timpului de execuţie (făcând abstracţie de eventualele constante ce intervin în expresiile detaliate ale timpilor de execuţie) s-au introdus nişte clase de funcţii şi notaţii asociate.

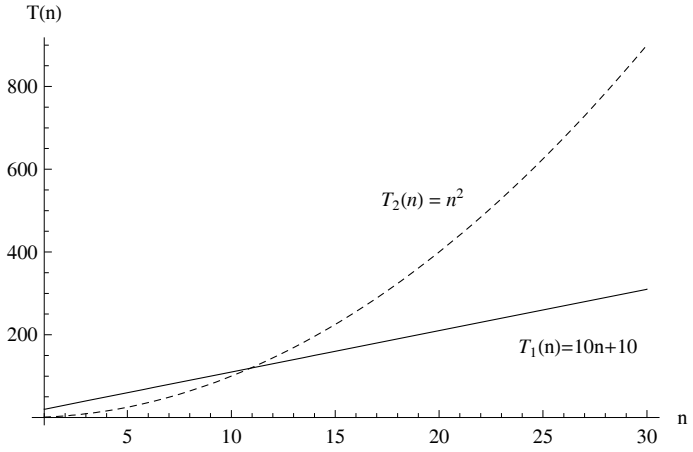

Figura 3.1: Ordin liniar şi ordin pătratic de creştere

3.3.1 Notaţia Θ.

Definiţia 3.1 *Pentru o funcţie* $g : \mathbb{N} \to \mathbb{R}_+$, $\Theta(g(n))$ *reprezintă mulţimea de funcţii:*

$$\begin{aligned} \Theta(g(n)) \quad = \quad & \{f : \mathbb{N} \to \mathbb{R}_+; \exists c_1, c_2 \in \mathbb{R}_+^*, n_0 \in \mathbb{N} \text{ astfel încât} \\ & c_1 g(n) \leq f(n) \leq c_2 g(n), \forall n \geq n_0\} \end{aligned} \tag{3.1}$$

Despre timpul de execuţie al unui algoritm, $T(n)$, se spune că este de ordinul $\Theta(g(n))$ dacă $T(n) \in \Theta(g(n))$. Prin abuz de notaţie în algoritmică se obişnuieşte să se scrie $T(n) = \Theta(g(n))$. Din punct de vedere intuitiv faptul că $f(n) \in \Theta(g(n))$ înseamnă că $f(n)$ şi $g(n)$ sunt asimptotic echivalente, adică au acelaşi ordin de creştere. Altfel spus $\lim_{n\to\infty} f(n)/g(n) = k$, k fiind o valoare finită strict pozitivă.

În figura 3.2 este ilustrată această idee folosind reprezentările grafice ale funcţiilor $f(n) = n^2 + 5n\lg n + 10$ şi $g(n) = c_i n^2$, $c_i \in \{1, 2\}$, pentru diverse domenii de variaţie ale lui n ($n \in \{1, \ldots, 5\}$, $n \in \{1, \ldots, 50\}$ respectiv $n \in \{1, \ldots, 500\}$) şi valorile $c_1 = 1$ şi $c_2 = 4$. Pentru aceste valori ale constantelor c_1 şi c_2 se observă din grafic că este suficient să considerăm $n_0 = 3$ pentru a arăta că $f \in \Theta(g(n))$. Constanta $c_2 = 4$ conduce la o margine superioară relaxată. În realitate orice valoarea supraunitară poate fi considerată, însă cu cât c_2 este mai mică cu atât n_0 va fi mai mare.

Exemplu. Pentru Exemplul 3.1 (calcul sumă) s-a obţinut $T(n) = k_1 n + k_2$ ($k_1 > 0$, $k_2 > 0$) prin urmare pentru $c_1 = k_1$, $c_2 = k_1 + 1$ şi $n_0 > k_2$ se obţine că $c_1 n \leq T(n) \leq c_2 n$ pentru $n \geq n_0$, adică $T(n) \in \Theta(n)$.

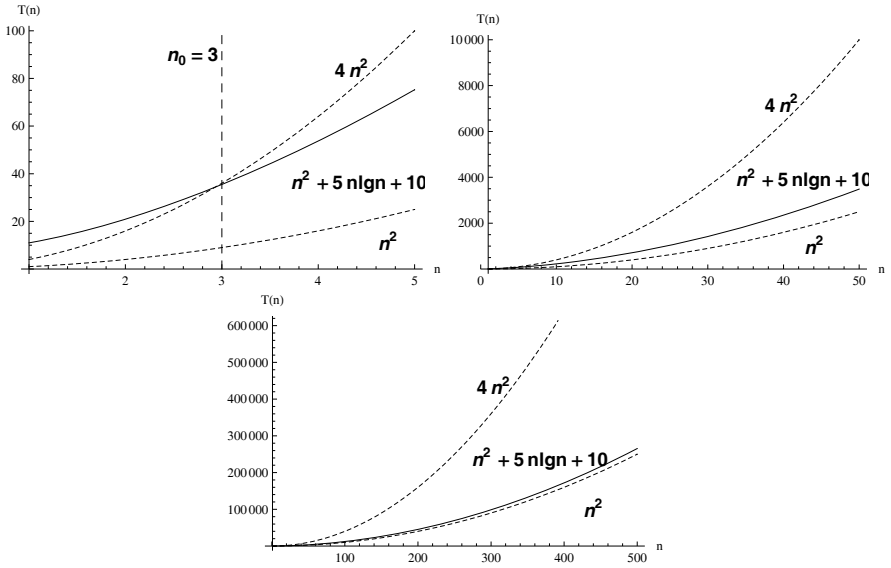

Figura 3.2: Graficele funcţiilor f (linie continuă), c_1g,c_2g (linie punctată)

Pentru Exemplul 3.3 (determinare minim) s-a obţinut că $3n \leq T(n) \leq 4n-1$ prin urmare $T(n) \in \Theta(n)$ (este suficient să se considere $c_1 = 3$, $c_2 = 4$ şi $n_0 = 1$).

Propoziţia 3.1 *Notaţia* Θ *are următoarele proprietăţi:*

(i) Dacă $T(n) = a_k n^k + a_{k-1}n^{k-1} + \ldots + a_1 n + a_0$, $a_k > 0$ *atunci* $T(n) \in \Theta(n^k)$.

(ii) $\Theta(\log_a(n)) = \Theta(\log_b(n))$ *pentru orice valori reale pozitive a şi b (diferite de* 1*).*

(iii) $f(n) \in \Theta(f(n))$ *(reflexivitate).*

(iv) Dacă $f(n) \in \Theta(g(n))$ *atunci* $g(n) \in \Theta(f(n))$ *(simetrie).*

(v) Dacă $f(n) \in \Theta(g(n))$ *şi* $g(n) \in \Theta(h(n))$ *atunci* $f(n) \in \Theta(h(n))$ *(tranzitivitate).*

(vi) $\Theta(f(n) + g(n)) = \Theta(\max\{f(n), g(n)\})$.

Demonstraţie. (i) Într-adevăr, din $\lim_{n\to\infty} T(n)/n^k = a_k$ rezultă că pentru orice $\varepsilon > 0$ există $n_0(\varepsilon)$ cu proprietatea că $|T(n)/n^k - a_k| \leq \varepsilon$ pentru orice $n \geq n_0(\varepsilon)$. Deci

$$a_k - \varepsilon \leq \frac{T(n)}{n^k} \leq a_k + \varepsilon, \quad \forall n \geq n_0(\varepsilon)$$

adică pentru $c_1 = a_k - \varepsilon$, $c_2 = a_k + \varepsilon$ şi $n_0 = n_0(\varepsilon)$ se obţin inegalităţile din (3.1).
(ii) Proprietatea rezultă din definiţie şi din faptul că $\log_a(n) = \log_b(n)/\log_b(a)$.
(iii)-(vi) Toate proprietăţile rezultă simplu din definiţie.

□

Proprietatea (ii) sugerează faptul că în analiza eficienţei nu are importanţă baza logaritmului, motiv pentru care în continuare logaritmul va fi specificat generic prin lg fără a se face referire la baza lui (implicit se consideră baza 2). Proprietăţile (iii)-(v) sugerează că notaţia Θ permite definirea unei clase de echivalenţă ($f(n)$ şi $g(n)$ sunt echivalente dacă $f(n) \in \Theta(g(n))$). Clasele de echivalenţă corespunzătoare sunt numite *clase de complexitate*.

Notaţia Θ se foloseşte atunci când se poate determina explicit expresia timpului de execuţie sau timpii de execuţie corespunzători cazurilor extreme au acelaşi ordin de creştere.

În cazul Exemplului 3.4 (problema căutării) s-a obţinut că $10 \leq T(n) \leq 5(n+1)$ ceea ce sugerează că există cazuri (de exemplu, valoarea este găsită pe prima poziţie) în care numărul de operaţii efectuate nu depinde de dimensiunea problemei. În această situaţie $T(n) \notin \Theta(n)$ deoarece nu poate fi găsit un c_1 şi un n_0 astfel încât $c_1 n \leq 10$ pentru orice $n \geq n_0$. În astfel de situaţii se analizează comportarea asimptotică a timpului în cazul cel mai defavorabil. În situaţia în care pentru toate datele de intrare timpul de execuţie nu depinde de volumul acestora (de exemplu în cazul algoritmului de determinare a valorii minime dintr-un şir ordonat crescător) atunci se notează $T(n) \in \Theta(1)$ (timp de execuţie constant).

3.3.2 Notaţia $\mathcal{O}$.

Definiţia 3.2 *Pentru o funcţie $g : \mathbb{N} \to \mathbb{R}_+$, $\mathcal{O}(g(n))$ reprezintă mulţimea de funcţii:*

$$\begin{aligned} \mathcal{O}(g(n)) \quad = \quad & \{f : \mathbb{N} \to \mathbb{R}_+; \exists c \in \mathbb{R}_+^*, n_0 \in \mathbb{N} \text{ astfel încât} \\ & 0 \leq f(n) \leq cg(n), \forall n \geq n_0\} \end{aligned} \tag{3.2}$$

Această clasă de funcţii permite descrierea comportării unui algoritm în cazul cel mai defavorabil fără a se face referire la celelalte situaţii. Întrucât de regulă interesează comportarea algoritmului pentru date arbitrare de intrare este suficient să specificăm o margine superioară pentru timpul de execuţie. Intuitiv, faptul că $f(n) \in \mathcal{O}(g(n))$ înseamnă că $f(n)$ creşte asimptotic cel mult la fel de repede ca $g(n)$. Altfel spus $\lim_{n\to\infty} f(n)/g(n) = k$, k fiind o valoare pozitivă, dar nu neapărat nenulă.

Folosind definiţia se pot demonstra următoarele proprietăţi ale notatiei $\mathcal{O}$.

Propoziţia 3.2 *Notaţia $\mathcal{O}$ are următoarele proprietăţi:*

(i) Dacă $T(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0$, $a_k > 0$ *atunci* $T(n) \in \mathcal{O}(n^p)$ *pentru orice* $p \geq k$.

(ii) $f(n) \in \mathcal{O}(f(n))$ *(reflexivitate).*

(iii) Dacă $f(n) \in \mathcal{O}(g(n))$ *şi* $g(n) \in \mathcal{O}(h(n))$ *atunci* $f(n) \in \mathcal{O}(h(n))$ *(tranzitivitate).*

(iv) $\mathcal{O}(f(n) + g(n)) = \mathcal{O}(\max\{f(n), g(n)\})$.

(v) $\Theta(g(n)) \subset \mathcal{O}(g(n))$.

După cum ilustrează Exemplul 3.4, incluziunea de la proprietatea (v), adică $\Theta(g(n)) \subset \mathcal{O}(g(n))$, este strictă.

Folosind definiţia lui $\mathcal{O}$ se verifică uşor că dacă $g_1(n) < g_2(n)$ pentru $n \geq n_0$ iar $f(n) \in \mathcal{O}(g_1(n))$ atunci $f(n) \in \mathcal{O}(g_2(n))$. Prin urmare dacă $T(n) \in \mathcal{O}(n)$ atunci $T(n) \in \mathcal{O}(n^d)$ pentru orice $d \geq 1$. Evident la analiza unui algoritm este util să se pună în evidenţa cea mai mică margine superioară. Astfel pentru algoritmul din Exemplul 3.4 vom spune ca are ordinul de complexitate $\mathcal{O}(n)$ şi nu $\mathcal{O}(n^2)$ (chiar dacă afirmaţia ar fi corectă din punctul de vedere al definiţiei).

Notaţia *o.* Dacă în Definiţia 3.2 în locul inegalităţii $f(n) \leq cg(n)$ se specifică inegalitatea strictă $f(n) < cg(n)$ care are loc pentru orice constantă pozitivă c atunci se obţine clasa $o(g(n))$. Aceasta este echivalent cu faptul că $\lim_{n\to\infty} f(n)/g(n) = 0$. Cu această notaţie putem scrie că $3n - 1 \in o(n^2)$ dar $n^2 + 3n - 1 \notin o(n^2)$ (deşi $n^2 + 3n - 1 \in \mathcal{O}(n^2)$). Această notaţie este mai puţin frecvent folosită în practică decât $\mathcal{O}$.

3.3.3 Notaţia Ω.

Definiţia 3.3 *Pentru o funcţie* $g : \mathbb{N} \to \mathbb{R}_+$, $\Omega(g(n))$ *reprezintă mulţimea de funcţii:*

$$\begin{aligned} \Omega(g(n)) \quad = \quad & \{f : \mathbb{N} \to \mathbb{R}_+; \exists c \in \mathbb{R}_+^*, n_0 \in \mathbb{N} \text{ astfel încât} \\ & cg(n) \leq f(n), \forall n \geq n_0\} \end{aligned} \tag{3.3}$$

Notaţia Ω se foloseşte pentru a exprima eficienţa algoritmului pornind de la timpul de execuţie corespunzător celui mai favorabil caz. Intuitiv, faptul că $f(n) \in \Omega(g(n))$ înseamnă că $f(n)$ creşte asimptotic cel puţin la fel de repede ca $g(n)$, adică $\lim_{n\to\infty} f(n)/g(n) \geq k$, k fiind o valoare strict pozitivă dar nu neapărat finită (limita poate fi chiar infinită).

Propoziţia 3.3 *Notaţia* Ω *are următoarele proprietăţi:*

(i) Dacă $T(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0$, $a_k > 0$ *atunci* $T(n) \in \mathcal{O}(n^p)$ *pentru orice* $p \leq k$.

(ii) $f(n) \in \Omega(f(n))$ *(reflexivitate).*

(iii) Dacă $f(n) \in \Omega(g(n))$ *şi* $g(n) \in \Omega(h(n))$ *atunci* $f(n) \in \Omega(h(n))$ *(tranzitivitate).*

(iv) $\Omega(f(n) + g(n)) = \Omega(\max\{f(n), g(n)\})$.

(v) $\Theta(g(n)) \subset \Omega(g(n))$.

(vi) Dacă $f(n) \in \mathcal{O}(g(n))$ *atunci* $g(n) \in \Omega(f(n))$ *şi reciproc.*

Aşa cum rezultă din Exemplul 3.4 incluziunea $\Theta(g(n)) \subset \Omega(g(n))$ este strictă: $T(n) \in \Omega(1)$ - corespunde cazului în care valoarea se găseşte pe prima poziţie - însă $T(n) \notin \Theta(1)$ întrucât în cazul cel mai defavorabil timpul de execuţie depinde de n. Din proprietăţile (v) din Propoziţiile 3.2 şi 3.3 rezultă că $\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$.

Notaţia ω**.** Dacă în Definiţia 3.3 în locul inegalităţii $cg(n) \leq f(n)$ se specifică inegalitatea strictă $cg(n) < f(n)$ care are loc pentru orice constantă pozitivă c atunci se obţine clasa $\omega(g(n))$. Aceasta este echivalent cu faptul că $\lim_{n\to\infty} f(n)/g(n) = \infty$. Cu această notaţie putem scrie că $3n^2 - 1 \in \omega(n)$ dar $3n - 1 \notin \omega(n)$ (deşi $3n - 1 \in \Omega(n)$).

3.3.4 Analiza asimptotică a principalelor structuri de prelucrare

Considerăm problema determinării ordinului de complexitate în cazul cel mai defavorabil pentru structurile algoritmice: *secvenţială*, *alternativă* şi *repetitivă*.

Presupunem că structura secvenţială este constituită din prelucrările A_1, ..., A_k şi fiecare dintre acestea are ordinul de complexitate $\mathcal{O}(g_i(n))$. Atunci structura va avea ordinul de complexitate $\mathcal{O}(g_1(n) + \ldots + g_k(n)) = \mathcal{O}(\max\{g_1(n), \ldots, g_k(n)\})$.

Dacă evaluarea condiţiei unei structuri alternative are cost constant iar prelucrările corespunzătoare celor două variante au ordinele de complexitate $\mathcal{O}(g_1(n))$ respectiv $\mathcal{O}(g_2(n))$ atunci costul structurii alternative va fi $\mathcal{O}(\max\{g_1(n), g_2(n)\})$.

În cazul unei structuri repetitive pentru a determina ordinul de complexitate în cazul cel mai defavorabil se consideră numărul maxim de iteraţii. Dacă acesta este n, iar dacă în corpul ciclului prelucrările sunt de cost constant, atunci se obţine ordinul $\mathcal{O}(n)$. În cazul unui ciclu dublu, dacă atât pentru ciclul interior cât şi pentru cel exterior limitele variază între 1 şi n atunci se obţine de regulă o complexitate pătratică, $\mathcal{O}(n^2)$. Dacă însă limitele ciclului interior se modifică este posibil să se obţină un alt ordin. Să considerăm următoarea prelucrare:

```
m ← 1
for i ← 1, n do
  m ← 3 * m
  for j ← 1, m do
    prelucrare de ordin Θ(1)      //prelucrare de cost constant
  end for
end for
```

Cum pentru fiecare valoarea a lui i se obţine $m = 3^i$ rezultă că timpul de execuţie este de forma $T(n) = 1 + \sum_{i=1}^{n}(3^i + 1) \in \Theta(3^n)$.

3.3.5 Clase de complexitate

Majoritatea algoritmilor întâlniţi în practică se încadrează în una dintre clasele menţionate în Tabelul 3.6. Ordinele de complexitate menţionate în tabel corespund celui mai defavorabil caz.

Complexitate	Ordin	Exemplu
logaritmică	$\mathcal{O}(\lg n)$	căutare binară
liniară	$\mathcal{O}(n)$	căutare secvenţială
	$\mathcal{O}(n \lg n)$	sortare prin interclasare
pătratică	$\mathcal{O}(n^2)$	sortare prin inserţie
cubică	$\mathcal{O}(n^3)$	produsul a două matrici pătratice de ordin n
exponenţială	$\mathcal{O}(2^n)$	prelucrarea tuturor submulţimilor unei mulţimi cu n elemente
factorială	$\mathcal{O}(n!)$	prelucrarea tuturor permutărilor unei mulţimi cu n elemente

Tabelul 3.6: Clase de complexitate şi exemple de algoritmi reprezentativi

În ierarhizarea algoritmilor după ordinul de complexitate sunt utile relaţiile (3.4) cunoscute din matematică.

$$\lim_{n\to\infty} \frac{(\lg n)^b}{n^k} = 0, \quad \lim_{n\to\infty} \frac{n^k}{a^n} = 0, \quad \lim_{n\to\infty} \frac{a^n}{n^n} = 0, \quad \lim_{n\to\infty} \frac{a^n}{n!} = 0 \qquad (a > 1) \tag{3.4}$$

Algoritmii aplicabili pentru probleme de dimensiune mare sunt doar cei din clasa $\mathcal{O}(n^k)$ ($k \ll n$ constantă) cunoscuţi sub numele de algoritmi *polinomiali*. Algoritmii de complexitate exponenţială sunt aplicabili doar pentru probleme de dimensiune mică.

Pentru stabilirea clasei (ordinului) de complexitate a unui algoritm se parcurg următoarele etape:

1. Se stabileşte dimensiunea problemei.
2. Se identifică operaţia de bază (operaţia dominantă).
3. Se verifică dacă numărul de execuţii ale operaţiei de bază depinde doar de dimensiunea problemei. Dacă da, se determină acest număr. Dacă nu, se analizează cazul cel mai favorabil, cazul cel mai defavorabil şi (dacă este posibil) cazul mediu.
4. Se stabileşte clasa de complexitate căruia îi aparţine algoritmul.

3.4 Analiza empirică

Motivaţie. Analiza teoretică a eficienţei algoritmilor poate fi dificilă în cazul unor algoritmi care nu sunt simpli, mai ales dacă este vorba de analiza cazului mediu. O alternativă la analiza teoretică a eficienţei o reprezintă *analiza empirică*.

Aceasta poate fi utilă pentru: (i) a obţine informaţii preliminare privind clasa de complexitate a unui algoritm; (ii) pentru a compara eficienţa a doi (sau mai mulţi) algoritmi destinaţi rezolvării aceleiaşi probleme; (iii) pentru a compara eficienţa mai multor implementări ale aceluiaşi algoritm; (iv) pentru a obţine informaţii privind eficienţa implementării unui algoritm pe o anumită maşină de calcul; (v) pentru a identifica porţiunile cele mai costisitoare din cadrul programului (*profilare*).

Etapele analizei empirice. În analiza empirică a unui algoritm se parcurg de regulă următoarele etape:

1. Se stabileşte scopul analizei.
2. Se alege metrica de eficienţă ce va fi utilizată (numărul de execuţii ale unei/unor operaţii sau timpul de execuţie a întregului algoritm sau a unei porţiuni din algoritm).
3. Se stabilesc proprietăţile datelor de intrare în raport cu care se face analiza (dimensiunea datelor sau proprietăţi specifice).
4. Se implementează algoritmul într-un limbaj de programare.
5. Se generează mai multe seturi de date de intrare.
6. Se execută programul pentru fiecare set de date de intrare.
7. Se analizează datele obţinute.

Alegerea măsurii de eficienţă depinde de scopul analizei. Dacă, de exemplu, se urmăreşte obţinerea unor informaţii privind clasa de complexitate sau chiar verificarea acurateţei unei estimări teoretice atunci este adecvată utilizarea numărului de operaţii efectuate. Dacă însă scopul este evaluarea comportării implementării unui algoritm atunci este potrivit timpul de execuţie.

Pentru a efectua o analiză empirică nu este suficient un singur set de date de intrare ci mai multe, care să pună în evidenţă diferitele caracteristici ale algoritmului. În general este bine să se aleagă date de diferite dimensiuni astfel încât să fie acoperită o plajă cât mai largă de dimensiuni. Pe de altă parte are importanţă şi analiza diferitelor valori sau configuraţii ale datelor de intrare. Dacă se analizează un algoritm care verifică dacă un număr este prim sau nu şi testarea se face doar pentru numere ce nu sunt prime sau doar pentru numere care sunt prime atunci nu se va obţine un rezultat relevant. Acelaşi lucru se poate întâmpla pentru un algoritm a cărui comportare depinde de gradul de sortare a unui tablou (dacă se aleg fie doar tablouri aproape sortate după criteriul dorit fie tablouri ordonate în sens invers analiza nu va fi relevantă).

În vederea analizei empirice la implementarea algoritmului într-un limbaj de programare vor trebui introduse secvenţe al căror scop este monitorizarea execuţiei. Dacă metrica de eficienţă este numărul de execuţii ale unei operaţii atunci se utilizează un contor care se incrementează după fiecare execuţie a operaţiei respective. Dacă metrica este timpul de execuţie atunci trebuie înregistrat momentul intrării în secvenţa analizată şi momentul ieşirii. Majoritatea limbajelor de programare oferă funcţii de măsurare a timpului scurs între două momente. Este important, în special în cazul în care pe calculator sunt active mai multe taskuri, să se contorizeze doar timpul afectat execuţiei programului analizat. În special dacă este vorba de măsurarea timpului este indicat să se ruleze programul de test de mai multe ori şi să se calculeze valoarea medie a timpilor.

La generarea seturilor de date de intrare scopul urmărit este să se obţină date tipice rulărilor uzuale (să nu fie doar excepţii). În acest scop adesea datele se generează în manieră aleatoare. În realitate este vorba de o pseudo-aleatoritate întrucât este simulată prin tehnici cu caracter determinist.

După execuţia programului pentru datele de test se înregistrează rezultatele, iar în scopul analizei fie se calculează mărimi sintetice (media, abaterea standard etc.), fie se reprezintă grafic perechi de puncte de forma (dimensiune problema, măsură de eficienţă).

3.5 Analiza amortizată

Să considerăm problema numărării în baza 2 de la 0 până la $n = 2^k - 1$ considerând valoarea binară curentă stocată într-un tablou $b[0..k-1]$ ($b[0]$ reprezintă bitul cel mai puţin semnificativ iar $b[n]$ reprezintă bitul cel mai semnificativ). Algoritmul determinării valorii n prin numărare constă în aplicarea

repetată a unui algoritm de incrementare în baza 2 (Algoritmul 4).

Algoritmul 3.2 Incrementare binară

```
increm(integer b[0..k − 1])
integer i
1: i ← 0
2: while i < k and b[i] = 1 do
3:     b[i] ← 0
4:     i ← i + 1
5: end while
6: b[i] ← 1
7: return  b[0..k − 1]
```

```
numărare(integer n, k)
integer b[0..k − 1],j
1: b[0..k − 1] ← 0; write b[0..k − 1]
2: for j ← 1, n do
3:     b[0..k − 1] ← increm(b[0..k − 1])
4:     write b[0..k − 1]
5: end for
```

Se pune problema determinării ordinului de complexitate în cazul cel mai defavorabil. Dimensiunea problemei este determinată de valorile k şi n iar operaţia dominantă este cea de schimbare a valorii unei cifre binare: $b[i]$ se transformă din 1 în 0 (linia 3 a algoritmului `increm`) sau din 0 în 1 (linia 6 a algoritmului `increm`). Dacă se analizează eficienţa algoritmului `increm` independent de contextul general al problemei atunci se observă uşor că în cazul cel mai defavorabil numărul de modificări ale unei cifre binare este k. Deci algoritmul `increm` aparţine lui $\mathcal{O}(k)$. Cum algoritmul `numărare` apelează algoritmul `increm` de exact n ori rezultă că în cazul cel mai defavorabil se efectuează kn operaţii pentru a se număra de la 1 la n.

j	$b[0..k-1]$				Nr.	j	$b[0..k-1]$				Nr.
	b_3	b_2	b_1	b_0	operaţii		b_3	b_2	b_1	b_0	operaţii
0	0	0	0	0	0	8	**1**	**0**	**0**	**0**	15
1	0	0	0	**1**	1	9	1	0	0	**1**	16
2	0	0	**1**	**0**	3	10	1	0	**1**	**0**	18
3	0	0	1	**1**	4	11	1	0	1	**1**	19
4	0	**1**	**0**	**0**	7	12	1	**1**	**0**	**0**	22
5	0	1	0	**1**	8	13	1	1	0	**1**	23
6	0	1	**1**	**0**	10	14	1	1	**1**	**0**	25
7	0	1	1	**1**	11	15	1	1	1	**1**	26

Tabelul 3.7: Numărul de operaţii de transformare a unei cifre binare în cazul numărării de la 0 la 15. Cifrele marcate sunt modificate la iteraţia j în cadrul algoritmului `numărare`

Dacă însă se urmăreşte numărul de operaţii efectuate pentru $k = 4$ şi $n = 15$ (Tabelul 3.7) se observă că numărul de operaţii nu depăşeşte $2n$. Aceasta înseamnă că marginea superioară obţinută aplicând analiza clasică este prea largă. Se observă că cifra de pe poziţia 0 se modifică la fiecare iteraţie, cifra de pe poziţia 1 din în două iteraţii, cea de pe poziţia 2 din patru în patru iteraţii

ş.a.m.d. Prin urmare numărul de operaţii efectuate este

$$\sum_{i=0}^{k-1} \lfloor \frac{n}{2^i} \rfloor \leq n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n$$

Deci la fiecare dintre cele n iteraţii din algoritmul `numărare` se efectuează în cazul cel mai defavorabil un număr mediu de 2 modificăi ale cifrelor binare. Aceasta înseamna că în contextul utilizării în algoritmul `numărare`, algoritmul de incrementare (care luat independent este de complexitate $\mathcal{O}(k)$) poate fi considerat de complexitate $\mathcal{O}(n)/n$ ceea ce înseamnă că se poate considera ca aparţine lui $\mathcal{O}(1)$. Un cost astfel determinat este numit cost *amortizat* iar analiza bazată pe astfel de costuri se numeşte analiză amortizată. Metoda folosită mai sus este cunoscută sub numele de analiză bazată pe *agregare*. In această metodă costul amortizat este considerat acelaşi pentru toate operaţiile (setarea unei cifre binare pe 1 respectiv setarea pe 0 au acelaşi cost). O altă variantă este cea care acordă costuri diferite pentru aceste operaţii. Întrucât numărul de operaţii este mai mare cu cât sunt mai multe cifre de 1 pe primele poziţii se asignează un cost mai mare setării unei cifre pe 1 decât setării unei cifre pe 0. De exemplu se consideră că setarea unei cifre pe 1 este cotată cu costul 2. La setarea efectivă a unei cifre pe 1 se contorizează costul efectiv scazând 1 din valoarea cotată, 2. Restul joacă rolul unui credit care este folosit la setarea unei cifre pe 0. In felul acesta costul unui apel al funcţiei `increm` poate fi considerat egal cu costul setării unei cifre pe 1 adică 2. Ideea este inspirată de amortizarea costurilor în sistemele economice, de unde provine şi denumirea.

Analiza amortizată se bazează, ca şi analiza cazului mediu, tot pe noţiunea de cost mediu, dar nu în sens statistic, nefiind astfel necesară stabilirea unei distribuţii de probabilitate pentru diferitele clase de instanţe ale problemei. Pentru detalii suplimentare privind analiza amortizată pot fi consultate [3], [6], [11].