Curs 4: Analiza complexităţii algoritmilor
- Scopul analizei
- Timp de execuţie
- Ordin de creştere
- Notaţii asimptotice
- Etapele analizei complexităţii
- Analiza empirică a complexităţii algoritmilor
- Exerciţii

1 Scopul analizei complexităţii

Analiza complexităţii unui algoritm are ca scop stabilirea *resurselor* necesare pentru execuţia algoritmului pe o maşină de calcul. Prin resurse înţelegem:

- *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 $[\log_2 x] + 1$. 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.

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.

2 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 consideră 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 şi cea a ultimului element al unui tablou).

A stabili o unitate de măsură înseamnă a stabili care sunt operaţiile elementare şi a considera ca unitate de măsură timpul de execuţie a acestora. În acest fel timpul de execuţie va fi exprimat prin numărul de operaţii elementare executate. Operaţiile elementare sunt cele aritmetice (adunare, scădere, înmulţire, împărţire), comparaţiile şi cele logice (negaţie, conjuncţie şi disjuncţie). 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 analiza se poate considera că ele au acelaşi cost).

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

Exemplul 1. Considerăm problema calculului $\sum_{i=1}^{n} i$. Dimensiunea acestei probleme este n. Algoritmul de rezolvare poate fi descris prin:

```
suma(n)
1:    S ← 0
2:    i ← 1
3:    WHILE i <= n DO
4:        ||S ← S + i
5:        ||i ← i + 1
RETURN S
```

unde operaţiile ce sunt contorizate sunt numerotate. Timpul de execuţie a algoritmului poate fi determinat folosind tabelul de costuri:

Operaţie	Cost	Nr. repetări
1	c_1	1
2	c_2	1
3	c_3	$n+1$
4	c_4	n
5	c_5	n

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 $T(n)$.

Prelucrarea repetitivă de mai sus poate fi înlocuită cu `FOR` $i \leftarrow \overline{1,n}$ `DO` $S \leftarrow S + i$. Costul gestiunii contorului (iniţializare, testare şi incrementare) fiind $c_2 + (n+1)c_3 + nc_5$. 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 2. Considerăm problema determinării produsului a două matrici: A de dimensiuni $m \times n$ şi B de dimensiuni $n \times p$. În acest caz dimensiunea problemei este determinată de trei valori: (m, n, p). Algoritmul poate fi descris prin:

```
produs(a[1..m, 1..n], b[1..n, 1..p])
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]
RETURN c[1..m, 1..p]
```

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 tabelul costurilor devine:

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

Timpul de execuţie este în acest caz: $T(m, n, p) = 4mnp + 5mp + 4m + 2$.

În practică nu este necesară o analiză atât de detaliată ci este suficient să se identifice operaţia de bază şi să se estimeze numărul de repetări ale acesteia. Prin operaţie de bază se înţelege operaţia care contribuie cel mai mult la timpul de execuţie a algoritmului şi este operaţia ce apare în ciclul cel mai interior.

În exemplul de mai sus ar putea fi considerată ca operaţie de bază, operaţia de înmulţire. În acest caz costul execuţiei algoritmului ar fi $T(m, n, p) = mnp$.

Exemplul 3. Considerăm problema determinării valorii minime dintr-un tablou $x[1..n]$. Dimensiunea problemei este dată de numărul n de elemente ale tabloului. Algoritmul este:

```
minim(x[1..n])
1:    m ← x[1]
2:    FOR i ← 2,n DO
3:        ||IF m > x[i] THEN
4:        ||        m ← x[i]
RETURN m
```

Tabelul costurilor prelucrărilor este:

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

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 aflat 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*.

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 de bază cea a comparării dintre valoarea variabilei m şi elementele tabloului. În acest caz costul algoritmului ar fi $T(n) = n - 1$.

Exemplul 4. Considerăm problema căutării unei valori v într-un tablou $x[1..n]$. Dimensiunea problemei este din nou n iar o variantă a algoritmului este:

```
cautare(x[1..n], v)
1:    gasit ← FALSE
2:    i ← 1
3:    WHILE (gasit =FALSE) AND i ≤ n DO
4:        ||IF v = x[i]
5:        ||    THEN gasit ← TRUE
6:        ||    ELSE i ← i + 1
RETURN gasit
```

Considerând prelucrări de cost unitar se obţine:

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	$\tau_3(n)$

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: $T(n) = 3(n+1) + n + 0 + n + 2 = 5(n+1)$.

Importanţa celui mai defavorabil caz. 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 fixă. Pe de altă parte pentru anumiţi algoritmi cazul cel mai defavorabil este relativ frecvent.

În ceea ce priveşte analiza celui mai favorabil caz, aceasta furnizează o margine inferioară a timpului de execuţie şi poate fi 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 soluţie acceptabilă).

Timp mediu de execuţie. Uneori, cazurile extreme (cel mai defavorabil şi cel mai favorabil) se întâlnesc rar, astfel ca 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. Dacă $\nu(n)$ reprezintă numărul variantelor posibile, 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:

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

În cazul în care 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 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.

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:

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

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: $P(v$ se află în tablou$) = P(v$ nu se află în tablou$) = 0.5$ iar în cazul în care se află în tablou el se găseşte cu aceeaşi probabilitate pe oricare dintre cele n poziţii: $P(v$ se află pe poziţia k$) = 1/n$. În acest caz timpul mediu este:

$$T_m(n) = \frac{1}{2}\left(\frac{1}{n}\sum_{k=1}^{n} 5(k+1) + 5(n+1)\right) = \frac{5(3n+5)}{4}$$

Se observă că timpul mediu de execuţie depinde de ipotezele făcute asupra distribuţiei datelor de intrare şi că acesta 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 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.

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 $T(kn) = (ka)n + b$. În acest caz este vorba despre un ordin liniar de creştere. Dacă $T(n) = an^2 + bn + c$ $(a > 0)$ atunci $T(kn) = (k^2a)n^2 + (kb)n + c$, 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 se face analiza complexităţii pentru cazul când n este mare (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.

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

4 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.

Notaţia Θ. Pentru o funcţie $g : N \to R_+$, $\Theta(g(n))$ reprezintă mulţimea de funcţii:

$$\Theta(g(n)) = \{f : N \to R; \exists c_1, c_2 \in R_+^*, n_0 \in N \text{ astfel încât } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n), \forall n \geq n_0\} \tag{1}$$

Despre timpul de execuţie al unui algoritm, $T(n)$, se spune că este $\Theta(g(n))$ 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. Altfel spus $\lim_{n\to\infty} f(n)/g(n) = k$, k fiind o valoare strict pozitivă.

În figura 1 este ilustrată legătura dintre funcţiile f şi g (pentru $f(n) = n^2 + 5n\lg n + 10$ şi $g(n) = n^2$) când $f(n) \in \Theta(g(n))$ pentru diverse domenii de variaţie ale lui n ($n \in \{1, \ldots, 5\}$, $n \in \{1, \ldots, 50\}$ respectiv $n \in \{1, \ldots, 500\}$). Pentru marginea inferioară s-a folosit $c_1 = 1$ iar pentru cea superioară $c_2 = 4$. In acest caz se observă din grafic că este suficient să considerăm $n_0 = 3$. Constanta $c_2 = 4$ conduce la o margine superioară relaxată. În realitate orice valoarea supraunitară poate fi considerată (cu cât c_2 este mai mică cu atât n_0 va fi mai mare).

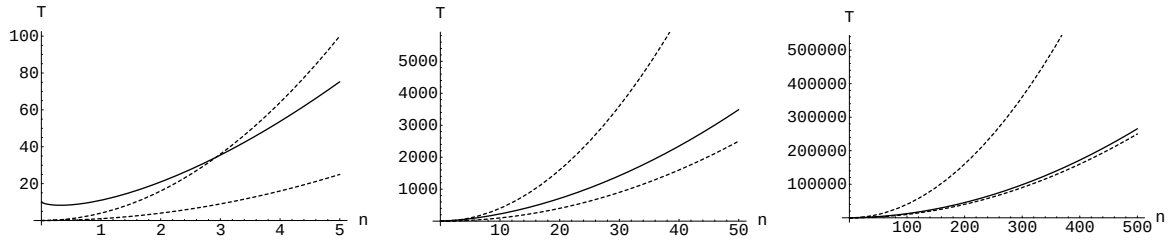

Figura 1: Graficele funcţiilor f (linie continuă), $c_1 g$, $c_2 g$ (linie punctată)

Exemplu. Pentru exemplul 1 din secţiunea 2 (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 $T(n) = \Theta(n)$.

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

În general, se poate demonstra că dacă $T(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0$, $a_k > 0$ atunci $T(n) = \Theta(n^k)$. Într-adevăr, din $\lim_{n\to\infty} T(n)/n^k = a_k$ rezultă că există $\varepsilon > 0$ şi $n_0(\varepsilon)$ 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 obţine inegalităţile din (1).

În cazul exemplului 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 $T(n) \neq \Theta(n)$ 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 $T(n) = \Theta(1)$ (timp de execuţie constant).

Notaţia O. Pentru o funcţie $g : N \to R_+$, $O(g(n))$ reprezintă mulţimea de funcţii:

$$O(g(n)) = \{f : N \to R; \exists c \in R_+^*, n_0 \in N \text{ astfel încât } 0 \leq f(n) \leq cg(n), \forall n \geq n_0\} \tag{2}$$

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 $f(n) \in O(g(n))$ înseamnă că $f(n)$ creşte asimptotic cel mult la fel de repede ca $g(n)$. Altfel spus $\lim_{n\to\infty} f(n)/g(n) \leq k$, k fiind o valoare strict pozitivă.

Din definiţiile lui Θ şi O rezultă că $\Theta(g(n)) \subset O(g(n))$ incluziunea fiind strictă (vezi exemplul 4 din secţiunea 2). Prin urmare dacă $T(n) = \Theta(g(n))$ atunci $T(n) = O(g(n))$.

Folosind definiţia lui O se verifică uşor că dacă $g_1(n) < g_2(n)$ pentru $n \geq n_0$ iar $f(n) \in O(g_1(n))$ atunci $f(n) \in O(g_2(n))$. Prin urmare dacă $T(n) = O(n)$ atunci $T(n) = O(n^d)$ pentru orice $d \geq 1$. În general, dacă $T(n) \leq a_k n^k + a_{k-1}n^{k-1} + \ldots + a_1 n + a_0$, $a_k > 0$ atunci $T(n) = O(n^d)$ pentru orice $d \geq k$. Evident la analiza unui algoritm este util să se pună în evidenţa cea mai mică margine superioară. Astfel pentru algoritmul din exemplul 4 vom spune ca are ordinul de complexitate $O(n)$ şi nu $O(n^2)$ (chiar dacă afirmaţia ar fi corectă din punctul de vedere al definiţiei).

Notaţia Ω. Pentru o funcţie $g : N \to R_+$, $\Omega(g(n))$ reprezintă mulţimea de funcţii:

$$\Omega(g(n)) = \{f : N \to R; \exists c \in R_+^*, n_0 \in N \text{ astfel încât } 0 \leq cg(n) \leq f(n), \forall n \geq n_0\} \qquad (3)$$

Este o măsură a ordinului de creştere a timpului de execuţie în cazul cel mai favorabil. 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 (limita poate fi chiar infinită).

Se observă din definiţii că $\Theta(g(n)) \subset \Omega(g(n))$ incluziunea fiind strictă (în exemplul 4 din secţiunea 2 avem $T(n) = \Omega(1)$ - corespunde cazului în care valoarea se găseşte pe prima poziţie - însă $T(n) \neq \Theta(1)$ întrucât în cazul cel mai defavorabil timpul de execuţie depinde de n).

Relaţia dintre cele trei clase de funcţii este: $\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$.

Analiza asimptotică a structurilor fundamentale. 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 $A_1, \ldots, A_k$ şi fiecare dintre acestea are ordinul de complexitate $O(g_i(n))$. Atunci structura va avea ordinul de complexitate $O(\max\{g_1(n), \ldots, g_k(n)\})$.

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

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 $O(n)$. 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, $O(n^2)$. 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  Θ(1)     {prelucrare de cost constant }
```

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

Clase de complexitate. În ierarhizarea algoritmilor după ordinul de complexitate se ţine cont de următoarele proprietăţi ale funcţiilor ce intervin cel mai frecvent în analiză:

$$\lim_{n\to\infty}\frac{(\lg n)^b}{n^k}=0,\quad \lim_{n\to\infty}\frac{n^k}{a^n}=0,\quad \lim_{n\to\infty}\frac{a^n}{n^n}=0,\quad \lim_{n\to\infty}\frac{a^n}{n!}=0\qquad (a>1) \tag{4}$$

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

Algoritmii din clasa celor de complexitate exponenţială sunt aplicabili doare pentru probleme de dimensiune mică.

5 Etapele analizei complexităţii

În analiza complexităţii unui algoritm se parcurge următoarele etape:

1. Se stabileşte dimensiunea problemei.
2. Se identifică operaţia de bază.
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.

6 Analiza empirică a complexităţii algoritmilor

Motivaţie. Analiza matematică a complexităţii 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 matematică a complexităţii 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 un anumit calculator.

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 (număr de execuţii ale unei/unor operaţii sau timp 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 plaja tuturor dimensiunilor care vor apare in practică. 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 tabloul 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).

7 Exerciţii

1. Să se stabilească ordinul de complexitate pentru următorul algoritm ce prelucrează date de volum n:

```
FOR i ← 1, n DO
    ||... (Θ(1))
    ||FOR j ← 1, 2i DO
        ||... (Θ(1))
        ||k ← j (Θ(1))
        ||WHILE k ≥ 0 DO
            ||... (Θ(1))
            ||k ← k − 1
```

2. Să se stabilească ordinul de complexitate pentru următorul algoritm ce prelucrează date de volum n:

```
h ← 1
WHILE h ≤ n DO
    ||... (Θ(1))
    ||h ← 2 * h
```

3. Să se stabilească ordinul de complexitate pentru următorul algoritm ce prelucrează date de volum n:

```
calcul (x[0..n-1])
k ← 0
FOR i ← 0, n − 1 DO
    ||FOR j ← 0, n − 1 DO
        ||y[k] ← x[i] * x[j]
        ||k ← k + 1
RETURN y[1..k]
```

4. Se consideră următorii doi algoritmi pentru calculul puterii unui număr:

```
putere1(x,n)
p ← 1
FOR i ← 1, n DO
    ||np ← 0
    ||FOR j ← 1, x DO
        ||np ← np + p
    ||p ← np
RETURN(p)
```

```
putere2(x,n)
p ← 1
FOR i ← 1, n DO
    ||p ← p * x
RETURN(p)
```

Să se stabilească ordinele de complexitate considerând că toate operaţiile aritmetice au acelaşi cost.

5. Să se arate că $\sum_{i=1}^{n} i^k \in \Theta(n^{k+1})$.

6. Ce relaţie există între $\Theta(\log_a n)$ şi $\Theta(\log_b n)$?

7. Propuneţi un algoritm de complexitate $\Theta(n^2)$ şi unul de complexitate $\Theta(n)$ pentru evaluarea unui polinom de grad n.