5. Analiza eficientei algoritmilor

Vom dezvolta in acest capitol aparatul matematic necesar pentru analiza eficientei algoritmilor, incercand ca aceasta incursiune matematica sa nu fie excesiv de formala. Apoi, vom arata, pe baza unor exemple, cum poate fi analizat un algoritm. O atentie speciala o vom acorda tehnicilor de analiza a algoritmilor recursivi.

5.1 Notatia asimptotica

In Capitolul 1 am dat un inteles intuitiv situatiei cand un algoritm necesita un timp *in ordinul* unei anumite functii. Revenim acum cu o definitie riguroasa.

5.1.1 O notatie pentru "ordinul lui"

Fie **N** multimea numerelor naturale (pozitive sau zero) si **R** multimea numerelor reale. Notam prin $\mathbf{N}^+$ si $\mathbf{R}^+$ multimea numerelor naturale, respectiv reale, strict pozitive, si prin $\mathbf{R}^*$ multimea numerelor reale nenegative. Multimea {*true*, *false*} de constante booleene o notam cu **B**. Fie $f : \mathbf{N} \to \mathbf{R}^*$ o functie arbitrara. Definim multimea

$$O(f) = \{t : \mathbf{N} \to \mathbf{R}^* \mid (\exists c \in \mathbf{R}^+)\ (\exists n_0 \in \mathbf{N})\ (\forall n \geq n_0)\ [t(n) \leq cf(n)]\}$$

Cu alte cuvinte, $O(f)$ (se citeste "ordinul lui f") este multimea tuturor functiilor t marginite superior de un multiplu real pozitiv al lui f, pentru valori suficient de mari ale argumentului. Vom conveni sa spunem ca t este in ordinul lui f (sau, echivalent, t este in $O(f)$, sau $t \in O(f)$) chiar si atunci cand valoarea $f(n)$ este negativa sau nedefinita pentru anumite valori $n < n_0$. In mod similar, vom vorbi despre ordinul lui f chiar si atunci cand valoarea $t(n)$ este negativa sau nedefinita pentru un numar finit de valori ale lui n; in acest caz, vom alege n_0 suficient de mare, astfel incat, pentru $n \geq n_0$, acest lucru sa nu mai apara. De exemplu, vom vorbi despre ordinul lui $n/\log n$, chiar daca pentru $n = 0$ si $n = 1$ functia nu este definita. In loc de $t \in O(f)$, uneori este mai convenabil sa folosim notatia $t(n) \in O(f(n))$, subintelegand aici ca $t(n)$ si $f(n)$ sunt functii.

Fie un algoritm dat si fie o functie $t : \mathbf{N} \to \mathbf{R}^*$ astfel incat o anumita implementare a algoritmului sa necesite cel mult $t(n)$ unitati de timp pentru a rezolva un caz de marime n, $n \in \mathbf{N}$. Principiul invariantei (mentionat in Capitolul 1) ne asigura ca orice implementare a algoritmului necesita un timp in ordinul lui t. Mai mult, acest algoritm necesita un timp in ordinul lui f pentru orice functie $f : \mathbf{N} \to \mathbf{R}^*$ pentru care $t \in O(f)$. In particular, $t \in O(t)$. Vom cauta in general sa gasim cea mai simpla functie f, astfel incat $t \in O(f)$.

Proprietatile de baza ale lui $O(f)$ sunt date ca exercitii (Exercitiile 5.1–5.7) si este recomandabil sa le studiati inainte de a trece mai departe.

Notatia asimptotica defineste o relatie de ordine partiala intre functii si deci, intre eficienta relativa a diferitilor algoritmi care rezolva o anumita problema. Vom da in continuare o interpretare algebrica a notatiei asimptotice. Pentru oricare doua functii $f, g : \mathbf{N} \to \mathbf{R}^*$, definim urmatoarea relatie binara: $f \leq g$ daca $O(f) \subseteq O(g)$. Relatia "$\leq$" este o *relatie de ordine partiala* in multimea functiilor definite pe $\mathbf{N}$ si cu valori in $\mathbf{R}^*$ (Exercitiul 5.6). Definim si o *relatie de echivalenta*: $f \equiv g$ daca $O(f) = O(g)$.

In multimea $O(f)$ putem inlocui pe f cu orice alta functie echivalenta cu f. De exemplu, $\lg n \equiv \ln n \equiv \log n$ si avem $O(\lg n) = O(\ln n) = O(\log n)$. Notand cu $O(1)$ ordinul functiilor marginite superior de o constanta, obtinem ierarhia:

$$O(1) \subset O(\log n) \subset O(n) \subset O(n \log n) \subset O(n^2) \subset O(n^3) \subset O(2^n)$$

Aceasta ierarhie corespunde unei clasificari a algoritmilor dupa un criteriu al performantei. Pentru o problema data, dorim mereu sa obtinem un algoritm corespunzator unui ordin cat mai "la stanga". Astfel, este o mare realizare daca in locul unui algoritm exponential gasim un algoritm polinomial.

In Exercitiul 5.7 este data o metoda de simplificare a calculelor, in care apare notatia asimptotica. De exemplu,

$$n^3+3n^2+n+8 \in O(n^3+(3n^2+n+8)) = O(\max(n^3, 3n^2+n+8)) = O(n^3)$$

Ultima egalitate este adevarata, chiar daca $\max(n^3, 3n^2+n+8) \neq n^3$ pentru $0 \leq n \leq 3$, deoarece notatia asimptotica se aplica doar pentru n suficient de mare. De asemenea,

$$\begin{aligned} n^3-3n^2-n-8 \in O(n^3/2+(n^3/2-3n^2-n-8)) &= O(\max(n^3/2, n^3/2-3n^2-n-8)) \\ &= O(n^3/2) = O(n^3) \end{aligned}$$

chiar daca pentru $0 \leq n \leq 6$ polinomul este negativ. Exercitiul 5.8 trateaza cazul unui polinom oarecare.

Notatia $O(f)$ este folosita pentru a limita superior timpul necesar unui algoritm, masurand eficienta algoritmului respectiv. Uneori este util sa estimam si o limita inferioara a acestui timp. In acest scop, definim multimea

$$\Omega(f) = \{t : \mathbf{N} \to \mathbf{R}^* \mid (\exists c \in \mathbf{R}^+)\ (\exists n_0 \in \mathbf{N})\ (\forall n \geq n_0)\ [t(n) \geq cf(n)]\}$$

Exista o anumita dualitate intre notatiile $O(f)$ si $\Omega(f)$. Si anume, pentru doua functii oarecare $f, g : \mathbf{N} \to \mathbf{R}^*$, avem: $f \in O(g)$, daca si numai daca $g \in \Omega(f)$.

O situatie fericita este atunci cand timpul de executie al unui algoritm este limitat, atat inferior cat si superior, de cate un multiplu real pozitiv al aceleiasi functii. Introducem notatia

$$\Theta(f) = O(f) \cap \Omega(f)$$

numita *ordinul exact* al lui f. Pentru a compara ordinele a doua functii, notatia Θ nu este insa mai puternica decat notatia O, in sensul ca relatia $O(f) = O(g)$ este echivalenta cu $\Theta(f) = \Theta(g)$.

Se poate intampla ca timpul de executie al unui algoritm sa depinda simultan de mai multi parametri. Aceasta situatie este tipica pentru anumiti algoritmi care opereaza cu grafuri si in care timpul depinde atat de numarul de varfuri, cat si de numarul de muchii. Notatia asimptotica se generalizeaza in mod natural si pentru functii cu mai multe variabile. Astfel, pentru o functie arbitrara $f : \mathbf{N} \times \mathbf{N} \to \mathbf{R}^*$ definim

$$O(f) = \{t : \mathbf{N} \times \mathbf{N} \to \mathbf{R}^* \mid (\exists c \in \mathbf{R}^+)\ (\exists m_0, n_0 \in \mathbf{N})\ (\forall m \geq m_0)\ (\forall n \geq n_0)\ [t(m, n) \leq cf(m, n)]\}$$

Similar, se obtin si celelalte generalizari.

5.1.2 Notatia asimptotica conditionata

Multi algoritmi sunt mai usor de analizat daca consideram initial cazuri a caror marime satisface anumite conditii, de exemplu sa fie puteri ale lui 2. In astfel de situatii, folosim *notatia asimptotica conditionata*. Fie $f : \mathbf{N} \to \mathbf{R}^*$ o functie arbitrara si fie $P : \mathbf{N} \to \mathbf{B}$ un predicat.

$$O(f \mid P) = \{t : \mathbf{N} \to \mathbf{R}^* \mid (\exists c \in \mathbf{R}^+)\ (\exists n_0 \in \mathbf{N})\ (\forall n \geq n_0)\ [P(n) \Rightarrow t(n) \leq cf(n)]\}$$

Notatia $O(f)$ este echivalenta cu $O(f \mid P)$, unde P este predicatul a carui valoare este mereu *true*. Similar, se obtin notatiile $\Omega(f \mid P)$ si $\Theta(f \mid P)$.

O functie $f : \mathbf{N} \to \mathbf{R}^*$ este *eventual nedescrescatoare*, daca exista un n_0, astfel incat pentru orice $n \geq n_0$ avem $f(n) \leq f(n+1)$, ceea ce implica prin inductie ca, pentru orice $n \geq n_0$ si orice $m \geq n$, avem $f(n) \leq f(m)$. Fie $b \geq 2$ un intreg oarecare. O functie eventual nedescrescatoare este *b-neteda* daca $f(bn) \in O(f(n))$. Orice functie care este b-neteda pentru un anumit $b \geq 2$ este, de asemenea, b-neteda

pentru orice $b \geq 2$ (demonstrati acest lucru!); din aceasta cauza, vom spune pur si simplu ca aceste functii sunt *netede*. Urmatoarea proprietate asambleaza aceste definitii, demonstrarea ei fiind lasata ca exercitiu.

Proprietatea 5.1 Fie $b \geq 2$ un intreg oarecare, $f : \mathbf{N} \to \mathbf{R}^*$ o functie neteda si $t : \mathbf{N} \to \mathbf{R}^*$ o functie eventual nedescrescatoare, astfel incat

$$t(n) \in X(f(n) \mid n \text{ este o putere a lui } b)$$

unde X poate fi O, Ω, sau Θ. Atunci, $t \in X(f)$. Mai mult, daca $t \in \Theta(f)$, atunci si functia t este neteda. _

Pentru a intelege utilitatea notatiei asimptotice conditionate, sa presupunem ca timpul de executie al unui algoritm este dat de ecuatia

$$t(n) = \begin{cases} a & \text{pentru } n = 1 \\ t(\lfloor n/2 \rfloor) + t(\lceil n/2 \rceil) + bn & \text{pentru } n \neq 1 \end{cases}$$

unde $a, b \in \mathbf{R}^+$ sunt constante arbitrare. Este dificil sa analizam direct aceasta ecuatie. Daca consideram doar cazurile cand n este o putere a lui 2, ecuatia devine

$$t(n) = \begin{cases} a & \text{pentru } n = 1 \\ 2t(n/2) + bn & \text{pentru } n > 1 \text{ o putere a lui 2} \end{cases}$$

Prin tehnicile pe care le vom invata la sfarsitul acestui capitol, ajungem la relatia

$$t(n) \in \Theta(n \log n \mid n \text{ este o putere a lui } 2)$$

Pentru a arata acum ca $t \in \Theta(n \log n)$, mai trebuie doar sa verificam daca t este eventual nedescrescatoare si daca $n \log n$ este neteda.

Prin inductie, vom demonstra ca $(\forall n \geq 1)\ [t(n) \leq t(n+1)]$. Pentru inceput, sa notam ca

$$t(1) = a \leq 2(a+b) = t(2)$$

Fie $n > 1$. Presupunem ca pentru orice $m < n$ avem $t(m) \leq t(m+1)$. In particular,

$$t(\lfloor n/2 \rfloor) \leq t(\lfloor (n+1)/2 \rfloor)$$
$$t(\lceil n/2 \rceil) \leq t(\lceil (n+1)/2 \rceil)$$

Atunci,

$$t(n) = t(\lfloor n/2 \rfloor) + t(\lceil n/2 \rceil) + bn \leq t(\lfloor (n+1)/2 \rfloor) + t(\lceil (n+1)/2 \rceil) + b(n+1) = t(n+1)$$

In fine, mai ramane sa aratam ca $n \log n$ este neteda. Functia $n \log n$ este eventual nedescrescatoare si

$$2n \log(2n) = 2n(\log 2 + \log n) = (2 \log 2)n + 2n \log n$$
$$\in O(n + n \log n) = O(\max(n, n \log n)) = O(n \log n)$$

De multe ori, timpul de executie al unui algoritm se exprima sub forma unor inegalitati de forma

$$t(n) \leq \begin{cases} t_1(n) & \text{pentru } n \leq n_0 \\ t(\lfloor n/2 \rfloor) + t(\lceil n/2 \rceil) + cn & \text{pentru } n > n_0 \end{cases}$$

si, simultan

$$t(n) \geq \begin{cases} t_2(n) & \text{pentru } n \leq n_0 \\ t(\lfloor n/2 \rfloor) + t(\lceil n/2 \rceil) + dn & \text{pentru } n > n_0 \end{cases}$$

pentru anumite constante $c, d \in \mathbf{R}^+$, $n_0 \in \mathbf{N}$ si pentru doua functii $t_1, t_2 : \mathbf{N} \to \mathbf{R}^+$. Notatia asimptotica ne permite sa scriem cele doua inegalitati astfel:

$$t(n) \in t(\lfloor n/2 \rfloor) + t(\lceil n/2 \rceil) + O(n)$$

respectiv

$$t(n) \in t(\lfloor n/2 \rfloor) + t(\lceil n/2 \rceil) + \Omega(n)$$

Aceste doua expresii pot fi scrise si concentrat:

$$t(n) \in t(\lfloor n/2 \rfloor) + t(\lceil n/2 \rceil) + \Theta(n)$$

Definim functia

$$f(n) = \begin{cases} 1 & \text{pentru } n = 1 \\ f(\lfloor n/2 \rfloor) + f(\lceil n/2 \rceil) + n & \text{pentru } n \neq 1 \end{cases}$$

Am vazut ca $f \in \Theta(n \log n)$. Ne intoarcem acum la functia t care satisface inegalitatile precedente. Prin inductie, se demonstreaza ca exista constantele $v \leq d$, $u \geq c$, astfel incat

$$v \leq t(n)/f(n) \leq u$$

pentru orice $n \in \mathbf{N}^+$. Deducem atunci

$$t \in \Theta(f) = \Theta(n \log n)$$

Aceasta tehnica de rezolvare a inegalitatilor initiale are doua avantaje. In primul rand, nu trebuie sa demonstram independent ca $t \in O(n \log n)$ si $t \in \Omega(n \log n)$. Apoi, mai important, ne permite sa restrangem analiza la situatia cand n este o putere a lui 2, aplicand apoi Proprietatea 5.1. Deoarece nu stim daca t este eventual nedescrescatoare, nu putem aplica Proprietatea 5.1 direct asupra inegalitatilor initiale.

5.2 Tehnici de analiza a algoritmilor

Nu exista o formula generala pentru analiza eficientei unui algoritm. Este mai curand o chestiune de rationament, intuitie si experienta. Vom arata, pe baza exemplelor, cum se poate efectua o astfel de analiza.

5.2.1 Sortarea prin selectie

Consideram algoritmul *select* din Sectiunea 1.3. Timpul pentru o singura executie a buclei interioare poate fi marginit superior de o constanta a. In total, pentru un i dat, bucla interioara necesita un timp de cel mult $b+a(n-i)$ unitati, unde b este o constanta reprezentand timpul necesar pentru initializarea buclei. O singura executie a buclei exterioare are loc in cel mult $c+b+a(n-i)$ unitati de timp, unde c este o alta constanta. Algoritmul dureaza in total cel mult

$$d + \sum_{i=1}^{n-1} (c + b + a(n-i))$$

unitati de timp, d fiind din nou o constanta. Simplificam aceasta expresie si obtinem

$$(a/2)n^2 + (b+c-a/2)n + (d-c-b)$$

de unde deducem ca algoritmul necesita un timp in $O(n^2)$. O analiza similara asupra limitei inferioare arata ca timpul este de fapt in $\Theta(n^2)$. Nu este necesar sa consideram cazul cel mai nefavorabil sau cazul mediu, deoarece timpul de executie este independent de ordonarea prealabila a elementelor de sortat.

In acest prim exemplu am dat toate detaliile. De obicei, detalii ca initializarea buclei nu se vor considera explicit. Pentru cele mai multe situatii, este suficient sa alegem ca *barometru* o anumita instructiune din algoritm si sa numaram de cate ori se executa aceasta instructiune. In cazul nostru, putem alege ca barometru testul din bucla interioara, acest test executandu-se de $n(n-1)/2$ ori. Exercitiul 5.23 ne sugereaza ca astfel de simplificari trebuie facute cu discernamant.

5.2.2 Sortarea prin insertie

Timpul pentru algoritmul *insert* (Sectiunea 1.3) este dependent de ordonarea prealabila a elementelor de sortat. Vom folosi comparatia "$x < T[\,j]$" ca barometru.

Sa presupunem ca i este fixat si fie $x = T[i]$, ca in algoritm. Cel mai nefavorabil caz apare atunci cand $x < T[j]$ pentru fiecare j intre 1 si $i-1$, algoritmul facand in aceasta situatie $i-1$ comparatii. Acest lucru se intampla pentru fiecare valoare a lui i de la 2 la n, atunci cand tabloul T este initial ordonat descrescator. Numarul total de comparatii pentru cazul cel mai nefavorabil este

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

Vom estima acum timpul mediu necesar pentru un caz oarecare. Presupunem ca elementele tabloului T sunt distincte si ca orice permutare a lor are aceeasi probabilitate de aparitie. Atunci, daca $1 \le k \le i$, probabilitatea ca $T[i]$ sa fie cel de-al k-lea cel mai mare element dintre elementele $T[1]$, $T[2]$, …, $T[i]$ este $1/i$. Pentru un i fixat, conditia $T[i] < T[i-1]$ este falsa cu probabilitatea $1/i$, deci probabilitatea ca sa se execute comparatia "$x < T[j]$", o singura data inainte de iesirea din bucla **while**, este $1/i$. Comparatia "$x < T[j]$" se executa de exact doua ori tot cu probabilitatea $1/i$ etc. Probabilitatea ca sa se execute comparatia de exact $i-1$ ori este $2/i$, deoarece aceasta se intampla atat cand $x < T[1]$, cat si cand $T[1] \le x < T[2]$. Pentru un i fixat, numarul mediu de comparatii este

$$c_i = 1\cdot 1/i + 2\cdot 1/i + \ldots + (i-2)\cdot 1/i + (i-1)\cdot 2/i = (i+1)/2 - 1/i$$

Pentru a sorta n elemente, avem nevoie de $\sum_{i=2}^{n} c_i$ comparatii, ceea ce este egal cu

$$(n^2+3n)/4 - H_n \in \Theta(n^2)$$

unde prin $H_n = \sum_{i=1}^{n} i^{-1} \in \Theta(\log n)$ am notat al n-lea element al seriei armonice (Exercitiul 5.17).

Se observa ca algoritmul *insert* efectueaza pentru cazul mediu de doua ori mai putine comparatii decat pentru cazul cel mai nefavorabil. Totusi, in ambele situatii, numarul comparatiilor este in $\Theta(n^2)$.

Algoritmul necesita un timp in $\Omega(n^2)$, atat pentru cazul mediu, cat si pentru cel mai nefavorabil. Cu toate acestea, pentru cazul cel mai favorabil, cand initial tabloul este ordonat crescator, timpul este in $O(n)$. De fapt, in acest caz, timpul este si in $\Omega(n)$, deci este in $\Theta(n)$.

5.2.3 Heapsort

Vom analiza, pentru inceput, algoritmul *make-heap* din Sectiunea 3.4. Definim ca barometru instructiunile din bucla **repeat** a algoritmului *sift-down*. Fie m numarul

maxim de repetari al acestei bucle, cauzat de apelul lui *sift-down*(T, i), unde i este fixat. Notam cu j_t valoarea lui j dupa ce se executa atribuirea "$j \leftarrow k$" la a t-a repetare a buclei. Evident, $j_1 = i$. Daca $1 < t \le m$, la sfarsitul celei de-a $(t-1)$-a repetari a buclei, avem $j \ne k$ si $k \ge 2j$. In general, $j_t \ge 2j_{t-1}$ pentru $1 < t \le m$. Atunci,

$$n \ge j_m \ge 2j_{m-1} \ge 4j_{m-2} \ge \ldots \ge 2^{m-1} i$$

Rezulta $2^{m-1} \le n/i$, iar de aici obtinem relatia $m \le 1 + \lg(n/i)$.

Numarul total de executari ale buclei **repeat** la formarea unui heap este marginit superior de

$$\sum_{i=1}^{a} (1 + \lg(n/i)), \text{ unde } a = \lfloor n/2 \rfloor \qquad (*)$$

Pentru a simplifica aceasta expresie, sa observam ca pentru orice $k \ge 0$

$$\sum_{i=b}^{c} \lg(n/i) \le 2^k \lg(n/2^k), \text{ unde } b = 2^k \text{ si } c = 2^{k+1}-1$$

Descompunem expresia (*) in sectiuni corespunzatoare puterilor lui 2 si notam $d = \lfloor \lg(n/2) \rfloor$:

$$\sum_{i=1}^{a} \lg(n/i) \le \sum_{k=0}^{d} 2^k \lg(n/2^k) \le 2^{d+1} \lg(n/2^{d-1})$$

Demonstratia ultimei inegalitati rezulta din Exercitiul 5.26. Dar $d = \lfloor \lg(n/2) \rfloor$ implica $d+1 \le \lg n$ si $d-1 \ge \lg(n/8)$. Deci,

$$\sum_{i=1}^{a} \lg(n/i) \le 3n$$

Din (*) deducem ca $\lfloor n/2 \rfloor + 3n$ repetari ale buclei **repeat** sunt suficiente pentru a construi un heap, deci *make-heap* necesita un timp $t \in O(n)$. Pe de alta parte, deoarece orice algoritm pentru formarea unui heap trebuie sa utilizeze fiecare element din tablou cel putin o data, $t \in \Omega(n)$. Deci, $t \in \Theta(n)$. Puteti compara acest timp cu timpul necesar algoritmului *slow-make-heap* (Exercitiul 5.28).

Pentru cel mai nefavorabil caz, *sift-down*($T[1 .. i-1]$, 1) necesita un timp in $O(\log n)$ (Exercitiul 5.27). Tinand cont si de faptul ca algoritmul *make-heap* este liniar, rezulta ca timpul pentru algoritmul *heapsort* pentru cazul cel mai nefavorabil este in $O(n \log n)$. Mai mult, timpul de executie pentru *heapsort* este de fapt in $\Theta(n \log n)$, atat pentru cazul cel mai nefavorabil, cat si pentru cazul mediu.

Algoritmii de sortare prezentati pana acum au o caracteristica comuna: se bazeaza numai pe comparatii intre elementele tabloului *T*. Din aceasta cauza, ii vom numi algoritmi de sortare prin comparatie. Vom cunoaste si alti algoritmi de acest tip: *bubblesort*, *quicksort*, *mergesort*. Sa observam ca, pentru cel mai nefavorabil caz, orice algoritm de sortare prin comparatie necesita un timp in $\Omega(n \log n)$ (Exercitiul 5.30). Pentru cel mai nefavorabil caz, algoritmul *heapsort* este deci optim (in limitele unei constante multiplicative). Acelasi lucru se intampla si cu *mergesort*.

5.2.4 Turnurile din Hanoi

Matematicianul francez Éduard Lucas a propus in 1883 o problema care a devenit apoi celebra, mai ales datorita faptului ca a prezentat-o sub forma unei legende. Se spune ca Brahma a fixat pe Pamant trei tije de diamant si pe una din ele a pus in ordine crescatoare 64 de discuri de aur de dimensiuni diferite, astfel incat discul cel mai mare era jos. Brahma a creat si o manastire, iar sarcina calugarilor era sa mute toate discurile pe o alta tija. Singura operatiune permisa era mutarea a cate unui singur disc de pe o tija pe alta, astfel incat niciodata sa nu se puna un disc mai mare peste unul mai mic. Legenda spune ca sfarsitul lumii va fi atunci cand calugarii vor savarsi lucrarea. Aceasta se dovedeste a fi o previziune extrem de optimista asupra sfarsitului lumii. Presupunand ca in fiecare secunda se muta un disc si lucrand fara intrerupere, cele 64 de discuri nu pot fi mutate nici in 500 de miliarde de ani de la inceputul actiunii!

Observam ca pentru a muta cele mai mici n discuri de pe tija i pe tija j (unde $1 \le i \le 3$, $1 \le j \le 3$, $i \ne j$, $n \ge 1$), transferam cele mai mici $n-1$ discuri de pe tija i pe tija $6-i-j$, apoi transferam discul n de pe tija i pe tija j, iar apoi retransferam cele $n-1$ discuri de pe tija $6-i-j$ pe tija j. Cu alte cuvinte, reducem problema mutarii a n discuri la problema mutarii a $n-1$ discuri. Urmatoarea procedura descrie acest algoritm recursiv.

```
procedure Hanoi(n, i, j)
    {muta cele mai mici n discuri de pe tija i pe tija j}
    if n > 0 then   Hanoi(n–1, i, 6–i–j)
                    write i "→" j
                    Hanoi(n–1, 6–i–j, j)
```

Pentru rezolvarea problemei initiale, facem apelul *Hanoi*(64, 1, 2).

Consideram instructiunea **write** ca barometru. Timpul necesar algoritmului este exprimat prin urmatoarea recurenta:

$$t(n) = \begin{cases} 1 & \text{pentru } n = 1 \\ 2t(n-1) + 1 & \text{pentru } n > 1 \end{cases}$$

Vom demonstra in Sectiunea 5.2 ca $t(n) = 2^n-1$. Rezulta $t \in \Theta(2^n)$.

Acest algoritm este optim, in sensul ca este imposibil sa mutam n discuri de pe o tija pe alta cu mai putin de 2^n-1 operatii. Implementarea in oricare limbaj de programare care admite exprimarea recursiva se poate face aproape in mod direct.

5.3 Analiza algoritmilor recursivi

Am vazut in exemplul precedent cat de puternica si, in acelasi timp, cat de eleganta este recursivitatea in elaborarea unui algoritm. Nu vom face o introducere in recursivitate si nici o prezentare a metodelor de eliminare a ei. Cel mai important castig al exprimarii recursive este faptul ca ea este naturala si compacta, fara sa ascunda esenta algoritmului prin detaliile de implementare. Pe de alta parte, apelurile recursive trebuie folosite cu discernamant, deoarece solicita si ele resursele calculatorului (timp si memorie). Analiza unui algoritm recursiv implica rezolvarea unui sistem de recurente. Vom vedea in continuare cum pot fi rezolvate astfel de recurente. Incepem cu tehnica cea mai banala.

5.3.1 Metoda iteratiei

Cu putina experienta si intuitie, putem rezolva de multe ori astfel de recurente prin *metoda iteratiei*: se executa primii pasi, se intuieste forma generala, iar apoi se demonstreaza prin inductie matematica ca forma este corecta. Sa consideram de exemplu recurenta problemei turnurilor din Hanoi. Pentru un anumit $n > 1$ obtinem succesiv

$$t(n) = 2t(n-1) + 1 = 2^2 t(n-2) + 2 + 1 = \ldots = 2^{n-1} t(1) + \sum_{i=0}^{n-2} 2^i$$

Rezulta $t(n) = 2^n-1$. Prin inductie matematica se demonstreaza acum cu usurinta ca aceasta forma generala este corecta.

5.3.2 Inductia constructiva

Inductia matematica este folosita de obicei ca tehnica de demonstrare a unei asertiuni deja enuntate. Vom vedea in aceasta sectiune ca inductia matematica poate fi utilizata cu succes si in descoperirea enuntului asertiunii. Aplicand aceasta tehnica, putem simultan sa demonstram o asertiune doar partial specificata si sa descoperim specificatiile care lipsesc si datorita carora asertiunea este corecta. Vom vedea ca aceasta tehnica a *inductiei constructive* este utila pentru

rezolvarea anumitor recurente care apar in contextul analizei algoritmilor. Incepem cu un exemplu.

Fie functia $f : \mathbf{N} \to \mathbf{N}$, definita prin recurenta

$$f(n) = \begin{cases} 0 & \text{pentru } n = 0 \\ f(n-1) + n & \text{pentru } n > 0 \end{cases}$$

Sa presupunem pentru moment ca nu stim ca $f(n) = n(n+1)/2$ si sa cautam o astfel de formula. Avem

$$f(n) = \sum_{i=0}^{n} i \leq \sum_{i=0}^{n} n = n^2$$

si deci, $f(n) \in O(n^2)$. Aceasta ne sugereaza sa formulam *ipoteza inductiei specificate partial IISP*(n) conform careia f este de forma $f(n) = an^2+bn+c$. Aceasta ipoteza este partiala, in sensul ca a, b si c nu sunt inca cunoscute. Tehnica inductiei constructive consta in a demonstra prin inductie matematica aceasta ipoteza incompleta si a determina in acelasi timp valorile constantelor necunoscute a, b si c.

Presupunem ca $IISP(n-1)$ este adevarata pentru un anumit $n \geq 1$. Atunci,

$$f(n) = a(n-1)^2+b(n-1)+c+n = an^2+(1+b-2a)n+(a-b+c)$$

Daca dorim sa aratam ca $IISP(n)$ este adevarata, trebuie sa aratam ca $f(n) = an^2+bn+c$. Prin identificarea coeficientilor puterilor lui n, obtinem ecuatiile $1+b-2a = b$ si $a-b+c = c$, cu solutia $a = b = 1/2$, c putand fi oarecare. Avem acum o ipoteza mai completa, pe care o numim tot $IISP(n)$: $f(n) = n^2/2+n/2+c$. Am aratat ca, daca $IISP(n-1)$ este adevarata pentru un anumit $n \geq 1$, atunci este adevarata si $IISP(n)$. Ramane sa aratam ca este adevarata si $IISP(0)$. Trebuie sa aratam ca $f(0) = a\cdot 0+b\cdot 0+c = c$. Stim ca $f(0) = 0$, deci $IISP(0)$ este adevarata pentru $c = 0$. In concluzie, am demonstrat ca $f(n) = n^2/2+n/2$ pentru orice n.

5.3.3 Recurente liniare omogene

Exista, din fericire, si tehnici care pot fi folosite aproape automat pentru a rezolva anumite clase de recurente. Vom incepe prin a considera ecuatii recurente liniare omogene, adica de forma

$$a_0 t_n + a_1 t_{n-1} + \ldots + a_k t_{n-k} = 0 \qquad (*)$$

unde t_i sunt valorile pe care le cautam, iar coeficientii a_i sunt constante.

Conform intuitiei, vom cauta solutii de forma

$$t_n = x^n$$

unde x este o constanta (deocamdata necunoscuta). Incercam aceasta solutie in (*) si obtinem

$$a_0x^n + a_1x^{n-1} + \ldots + a_kx^{n-k} = 0$$

Solutiile acestei ecuatii sunt fie solutia triviala $x = 0$, care nu ne intereseaza, fie solutiile ecuatiei

$$a_0x^k + a_1x^{k-1} + \ldots + a_k = 0$$

care este ecuatia caracteristica a recurentei (*).

Presupunand deocamdata ca cele k radacini r_1, r_2, ..., r_k ale acestei ecuatii caracteristice sunt distincte, orice combinatie liniara

$$t_n = \sum_{i=1}^{k} c_i r_i^n$$

este o solutie a recurentei (*), unde constantele c_1, c_2, ..., c_k sunt determinate de conditiile initiale. Este remarcabil ca (*) are *numai* solutii de aceasta forma.

Sa exemplificam prin recurenta care defineste sirul lui Fibonacci (din Sectiunea 1.6.4):

$$t_n = t_{n-1} + t_{n-2} \qquad n \geq 2$$

iar $t_0 = 0$, $t_1 = 1$. Putem sa rescriem aceasta recurenta sub forma

$$t_n - t_{n-1} - t_{n-2} = 0$$

care are ecuatia caracteristica

$$x^2 - x - 1 = 0$$

cu radacinile $r_{1,2} = (1 \pm \sqrt{5})/2$. Solutia generala are forma

$$t_n = c_1r_1^n + c_2r_2^n$$

Impunand conditiile initiale, obtinem

$$\begin{aligned} c_1 + c_2 &= 0 \qquad n = 0 \\ r_1c_1 + r_2c_2 &= 1 \qquad n = 1 \end{aligned}$$

de unde determinam

$$c_{1,2} = \pm 1/\sqrt{5}$$

Deci, $t_n = 1/\sqrt{5}(r_1^n - r_2^n)$. Observam ca $r_1 = \phi = (1+\sqrt{5})/2$, $r_2 = -\phi^{-1}$ si obtinem

$$t_n = 1/\sqrt{5}\,(\phi^n - (-\phi)^{-n})$$

care este cunoscuta relatie a lui de Moivre, descoperita la inceputul secolului XVI. Nu prezinta nici o dificultate sa aratam acum ca timpul pentru algoritmul *fib*1 (din Sectiunea 1.6.4) este in $\Theta(\phi^n)$.

Ce facem insa atunci cand radacinile ecuatiei caracteristice nu sunt distincte? Se poate arata ca, daca r este o radacina de multiplicitate m a ecuatiei caracteristice, atunci $t_n = r^n$, $t_n = nr^n$, $t_n = n^2r^n$, ..., $t_n = n^{m-1}r^n$ sunt solutii pentru (*). Solutia generala pentru o astfel de recurenta este atunci o combinatie liniara a acestor termeni si a termenilor proveniti de la celelalte radacini ale ecuatiei caracteristice. Din nou, sunt de determinat exact k constante din conditiile initiale.

Vom da din nou un exemplu. Fie recurenta

$$t_n = 5t_{n-1} - 8t_{n-2} + 4t_{n-3} \qquad n \geq 3$$

iar $t_0 = 0$, $t_1 = 1$, $t_2 = 2$. Ecuatia caracteristica are radacinile 1 (de multiplicitate 1) si 2 (de multiplicitate 2). Solutia generala este:

$$t_n = c_1 1^n + c_2 2^n + c_3 n 2^n$$

Din conditiile initiale, obtinem $c_1 = -2$, $c_2 = 2$, $c_3 = -1/2$.

5.3.4 Recurente liniare neomogene

Consideram acum recurente de urmatoarea forma mai generala

$$a_0 t_n + a_1 t_{n-1} + \ldots + a_k t_{n-k} = b^n p(n) \qquad (**)$$

unde b este o constanta, iar $p(n)$ este un polinom in n de grad d. Ideea generala este ca, prin manipulari convenabile, sa reducem un astfel de caz la o forma omogena.

De exemplu, o astfel de recurenta poate fi:

$$t_n - 2t_{n-1} = 3^n$$

In acest caz, $b = 3$ si $p(n) = 1$, un polinom de grad 0. O simpla manipulare ne permite sa reducem acest exemplu la forma (*). Inmultim recurenta cu 3, obtinand

$$3t_n - 6t_{n-1} = 3^{n+1}$$

Inlocuind pe n cu n+1 in recurenta initiala, avem

$$t_{n+1} - 2t_n = 3^{n+1}$$

In fine, scadem aceste doua ecuatii

$$t_{n+1} - 5t_n + 6t_{n-1} = 0$$

Am obtinut o recurenta omogena pe care o putem rezolva ca in sectiunea precedenta. Ecuatia caracteristica este:

$$x^2 - 5x + 6 = 0$$

adica $(x-2)(x-3) = 0$.

Intuitiv, observam ca factorul $(x-2)$ corespunde partii stangi a recurentei initiale, in timp ce factorul $(x-3)$ a aparut ca rezultat al manipularilor efectuate, pentru a scapa de parte dreapta.

Generalizand acest procedeu, se poate arata ca, pentru a rezolva (**), este suficient sa luam urmatoarea ecuatie caracteristica:

$$(a_0x^k + a_1x^{k-1} + \ldots + a_k)(x-b)^{d+1} = 0$$

Odata ce s-a obtinut aceasta ecuatie, se procedeaza ca in cazul omogen.

Vom rezolva acum recurenta corespunzatoare problemei turnurilor din Hanoi:

$$t_n = 2t_{n-1} + 1 \qquad n \geq 1$$

iar $t_0 = 0$. Rescriem recurenta astfel

$$t_n - 2t_{n-1} = 1$$

care este de forma (**) cu $b = 1$ si $p(n) = 1$, un polinom de grad 0. Ecuatia caracteristica este atunci $(x-2)(x-1) = 0$, cu solutiile 1 si 2. Solutia generala a recurentei este:

$$t_n = c_1 1^n + c_2 2^n$$

Avem nevoie de doua conditii initiale. Stim ca $t_0 = 0$; pentru a gasi cea de-a doua conditie calculam

$$t_1 = 2t_0 + 1$$

Din conditiile initiale, obtinem

$$t_n = 2^n - 1$$

Daca ne intereseaza doar ordinul lui t_n, nu este necesar sa calculam efectiv constantele in solutia generala. Daca stim ca $t_n = c_1 1^n + c_2 2^n$, rezulta $t_n \in O(2^n)$. Din faptul ca numarul de mutari a unor discuri nu poate fi negativ sau constant, deoarece avem in mod evident $t_n \geq n$, deducem ca $c_2 > 0$. Avem atunci $t_n \in \Omega(2^n)$ si deci, $t_n \in \Theta(2^n)$. Putem obtine chiar ceva mai mult. Substituind solutia generala inapoi in recurenta initiala, gasim

$$1 = t_n - 2t_{n-1} = c_1 + c_2 2^n - 2(c_1 + c_2 2^{n-1}) = -c_1$$

Indiferent de conditia initiala, c_1 este deci -1.

5.3.5 Schimbarea variabilei

Uneori, printr-o schimbare de variabila, putem rezolva recurente mult mai complicate. In exemplele care urmeaza, vom nota cu $T(n)$ termenul general al recurentei si cu t_k termenul noii recurente obtinute printr-o schimbare de variabila. Presupunem pentru inceput ca n este o putere a lui 2.

Un prim exemplu este recurenta

$$T(n) = 4T(n/2) + n \qquad n > 1$$

in care inlocuim pe n cu 2^k, notam $t_k = T(2^k) = T(n)$ si obtinem

$$t_k = 4t_{k-1} + 2^k$$

Ecuatia caracteristica a acestei recurente liniare este

$$(x-4)(x-2) = 0$$

si deci, $t_k = c_1 4^k + c_2 2^k$. Inlocuim la loc pe k cu $\lg n$

$$T(n) = c_1 n^2 + c_2 n$$

Rezulta

$$T(n) \in O(n^2 \mid n \text{ este o putere a lui } 2)$$

Un al doilea exemplu il reprezinta ecuatia

$$T(n) = 4T(n/2) + n^2 \qquad n > 1$$

Procedand la fel, ajungem la recurenta

$$t_k = 4t_{k-1} + 4^k$$

cu ecuatia caracteristica

$$(x-4)^2 = 0$$

si solutia generala $t_k = c_1 4^2 + c_2 k 4^2$. Atunci,

$$T(n) = c_1 n^2 + c_2 n^2 \lg n$$

si obtinem

$$T(n) \in O(n^2 \log n \mid n \text{ este o putere a lui 2})$$

In fine, sa consideram si exemplul

$$T(n) = 3T(n/2) + cn \qquad n > 1$$

c fiind o constanta. Obtinem succesiv

$$T(2^k) = 3T(2^{k-1}) + c2^k$$

$$t_k = 3t_{k-1} + c2^k$$

cu ecuatia caracteristica

$$(x-3)(x-2) = 0$$

$$t_k = c_1 3^k + c_2 2^k$$

$$T(n) = c_1 3^{\lg n} + c_2 n$$

si, deoarece

$$a^{\lg b} = b^{\lg a}$$

obtinem

$$T(n) = c_1 n^{\lg 3} + c_2 n$$

deci,

$$T(n) \in O(n^{\lg 3} \mid n \text{ este o putere a lui 2})$$

In toate aceste exemple am folosit notatia asimptotica conditionata. Pentru a arata ca rezultatele obtinute sunt adevarate pentru orice n, este suficient sa adaugam conditia ca $T(n)$ sa fie eventual nedescrescatoare. Aceasta, datorita Proprietatii 5.1 si a faptului ca functiile n^2, $n \log n$ si $n^{\lg 3}$ sunt netede.

Putem enunta acum o proprietate care este utila ca reteta pentru analiza algoritmilor cu recursivitati de forma celor din exemplele precedente.

Proprietatea, a carei demonstrare o lasam ca exercitiu, ne va fi foarte utila la analiza algoritmilor divide et impera din Capitolul 7.

Proprietatea 5.2 Fie $T : \mathbf{N} \to \mathbf{R}^+$ o functie eventual nedescrescatoare

$$T(n) = aT(n/b) + cn^k \qquad n > n_0$$

unde: $n_0 \geq 1$, $b \geq 2$ si $k \geq 0$ sunt intregi; a si c sunt numere reale pozitive; n/n_0 este o putere a lui b. Atunci avem

$$T(n) \in \begin{cases} \Theta(n^k) & \text{pentru } a < b^k \\ \Theta(n^k \log n) & \text{pentru } a = b^k \\ \Theta(n^{\log_b a}) & \text{pentru } a > b^k \end{cases}$$

–

5.4 Exercitii

5.1 Care din urmatoarele afirmatii sunt adevarate?

i) $n^2 \in O(n^3)$

ii) $n^3 \in O(n^2)$

iii) $2^{n+1} \in O(2^n)$

iv) $(n+1)! \in O(n!)$

v) pentru orice functie $f : \mathbf{N} \to \mathbf{R}^*$, $f \in O(n) \Rightarrow [f^2 \in O(n^2)]$

vi) pentru orice functie $f : \mathbf{N} \to \mathbf{R}^*$, $f \in O(n) \Rightarrow [2^f \in O(2^n)]$

5.2 Presupunand ca f este strict pozitiva pe $\mathbf{N}$, demonstrati ca definitia lui $O(f)$ este echivalenta cu urmatoarea definitie:

$$O(f) = \{t : \mathbf{N} \to \mathbf{R}^* \mid (\exists c \in \mathbf{R}^+)\ (\forall n \in \mathbf{N})\ [t(n) \leq cf(n)]\}$$

5.3 Demonstrati ca relatia "$\in O$" este *tranzitiva*: daca $f \in O(g)$ si $g \in O(h)$, atunci $f \in O(h)$. Deduceti de aici ca daca $g \in O(h)$, atunci $O(g) \subseteq O(h)$.

5.4 Pentru oricare doua functii $f, g : \mathbf{N} \to \mathbf{R}^*$, demonstrati ca:

i) $O(f) = O(g) \quad \Leftrightarrow \quad f \in O(g)$ si $g \in O(f)$

ii) $O(f) \subset O(g) \quad \Leftrightarrow \quad f \in O(g)$ si $g \notin O(f)$

5.5 Gasiti doua functii $f, g : \mathbf{N} \to \mathbf{R}^*$, astfel incat $f \notin O(g)$ si $g \notin O(f)$.

Indicatie: $f(n) = n$, $g(n) = n^{1+\sin n}$

5.6 Pentru oricare doua functii $f, g : \mathbf{N} \to \mathbf{R}^*$ definim urmatoarea relatie binara: $f \leq g$ daca $O(f) \subseteq O(g)$. Demonstrati ca relatia "$\leq$" este o relatie de ordine partiala in multimea functiilor definite pe $\mathbf{N}$ si cu valori in $\mathbf{R}^*$.

Indicatie: Trebuie aratat ca relatia este partiala, reflexiva, tranzitiva si antisimetrica. Tineti cont de Exercitiul 5.5.

5.7 Pentru oricare doua functii $f, g : \mathbf{N} \to \mathbf{R}^*$ demonstrati ca

$$O(f + g) = O(\max(f, g))$$

unde suma si maximul se iau punctual.

5.8 Fie $f(n) = a_m n^m + \ldots + a_1 n + a_0$ un polinom de grad m, cu $a_m > 0$. Aratati ca $f \in O(n^m)$.

5.9 $O(n^2) = O(n^3 + (n^2 - n^3)) = O(\max(n^3, n^2 - n^3)) = O(n^3)$

Unde este eroarea?

5.10 Gasiti eroarea in urmatorul lant de relatii:

$$\sum_{i=1}^{n} i = 1+2+\ldots+n \in O(1+2+\ldots+n) = O(\max(1, 2, \ldots, n)) = O(n)$$

5.11 Fie $f, g : \mathbf{N} \to \mathbf{R}^+$. Demonstrati ca:

i) $\lim_{n\to\infty} f(n) / g(n) \in \mathbf{R}^+ \quad \Rightarrow \quad O(f) = O(g)$

ii) $\lim_{n\to\infty} f(n) / g(n) = 0 \quad \Rightarrow \quad O(f) \subset O(g)$

Observatie: Implicatiile inverse nu sunt in general adevarate, deoarece se poate intampla ca limitele sa nu existe.

5.12 Folosind regula lui l'Hôspital si Exercitiile 5.4, 5.11, aratati ca

$$\log n \in O(\sqrt{n}), \quad \text{dar} \quad \sqrt{n} \notin O(\log n)$$

Indicatie: Prelungim domeniile functiilor pe $\mathbf{R}^+$, pe care sunt derivabile si aplicam regula lui l'Hôspital pentru $\log n/\sqrt{n}$.

5.13 Pentru oricare $f, g : \mathbf{N} \to \mathbf{R}^*$, demonstrati ca:

$$f \in O(g) \quad \Leftrightarrow \quad g \in \Omega(f)$$

5.14 Aratati ca $f \in \Theta(g)$ daca si numai daca

$$(\exists c, d \in \mathbf{R}^+)\ (\exists n_0 \in \mathbf{N})\ (\forall n \geq n_0)\ [cg(n) \leq f(n) \leq dg(n)]$$

5.15 Demonstrati ca urmatoarele propozitii sunt echivalente, pentru oricare doua functii $f, g : \mathbf{N} \to \mathbf{R}^*$.

i) $O(f) = O(g)$

ii) $\Theta(f) = \Theta(g)$

iii) $f \in \Theta(g)$

5.16 Continuand Exercitiul 5.11, aratati ca pentru oricare doua functii $f, g : \mathbf{N} \to \mathbf{R}^+$ avem:

i) $\lim_{n\to\infty} f(n)/g(n) \in \mathbf{R}^+ \quad \Rightarrow \quad f \in \Theta(g)$

ii) $\lim_{n\to\infty} f(n)/g(n) = 0 \quad \Rightarrow \quad f \in O(g)$ dar $f \notin \Theta(g)$

iii) $\lim_{n\to\infty} f(n)/g(n) = +\infty \quad \Rightarrow \quad f \in \Omega(g)$ dar $f \notin \Theta(g)$

5.17 Demonstrati urmatoarele afirmatii:

i) $\log_a n \in \Theta(\log_b n)$ pentru oricare $a, b > 1$

ii) $\sum_{i=1}^{n} i^k \in \Theta(n^{k+1})$ pentru oricare $k \in \mathbf{N}$

iii) $\sum_{i=1}^{n} i^{-1} \in \Theta(\log n)$

iv) $\log n! \in \Theta(n \log n)$

Indicatie: La punctul *iii)* se tine cont de relatia:

$$\sum_{i=1}^{\infty} i^{-1} = \ln n + \gamma + 1/2n - 1/12n^2 + \ldots$$

unde $\gamma = 0{,}5772\ldots$ este constanta lui Euler.

La punctul *iv)*, din $n! < n^n$, rezulta $\log n! \in O(n \log n)$. Sa aratam acum, ca $\log n! \in \Omega(n \log n)$. Pentru $0 \le i \le n-1$ este adevarata relatia

$$(n-i)(i+1) \ge n$$

Deoarece

$$(n!)^2 = (n\cdot 1)\ ((n-1)\cdot 2)\ ((n-2)\cdot 3)\cdot\ldots\cdot(2\cdot(n-1))\ (1\cdot n) \ge n^n$$

rezulta $2 \log n! \ge n \log n$ si deci $\log n! \in \Omega(n \log n)$.

Punctul *iv)* se poate demonstra si altfel, considerand aproximarea lui Stirling:

$$n! \in \sqrt{2\pi n}\ (n/e)^n (1+\Theta(1/n))$$

unde $e = 1{,}71828\ldots$.

5.18 Aratati ca timpul de executie al unui algoritm este in $\Theta(g)$, $g : \mathbf{N} \to \mathbf{R}^*$, daca si numai daca: timpul este in $O(g)$ pentru cazul cel mai nefavorabil si in $\Omega(g)$ pentru cazul cel mai favorabil.

5.19 Pentru oricare doua functii $f, g : \mathbf{N} \to \mathbf{R}^*$ demonstrati ca

$$\Theta(f) + \Theta(g) = \Theta(f+g) = \Theta(\max(f, g)) = \max(\Theta(f), \Theta(g))$$

unde suma si maximul se iau punctual.

5.20 Demonstrati Proprietatea 5.1. Aratati pe baza unor contraexemple ca cele doua conditii "$t(n)$ este eventual nedescrescatoare" si "$f(bn) \in O(f(n))$" sunt necesare.

5.21 Analizati eficienta urmatorilor patru algoritmi:

```
for i ← 1 to n do
    for j ←1 to 5 do
        {operatie elementara}
```

```
for i ← 1 to n do
    for j ← 1 to i+1 do
        {operatie elementara}
```

```
for i ← 1 to n do
    for j ← 1 to 6 do
        for k ←1 to n do
            {operatie elementara}
```

```
for i ← 1 to n do
    for j ← 1 to i do
        for k ← 1 to n do
            {operatie elementara}
```

5.22 Construiti un algoritm cu timpul in $\Theta(n \log n)$.

5.23 Fie urmatorul algoritm

```
k ← 0
  for i ← 1 to n do
    for j ← 1 to T[i] do
      k ← k+T[ j]
```

unde T este un tablou de n intregi nenegativi. In ce ordin este timpul de executie al algoritmului?

Solutie: Fie s suma elementelor lui T. Daca alegem ca barometru instructiunea "$k \leftarrow k+T[\,j]$", calculam ca ea se executa de s ori. Deci, am putea deduce ca timpul este in ordinul exact al lui s. Un exemplu simplu ne va convinge ca am gresit. Presupunem ca $T[i] = 1$, atunci cand i este un patrat perfect, si $T[i] = 0$, in rest. In acest caz, $s = \lfloor \sqrt{n} \rfloor$. Totusi, algoritmul necesita timp in ordinul lui $\Omega(n)$, deoarece fiecare element al lui T este considerat cel putin o data. Nu am tinut cont de urmatoarea regula simpla: putem neglija timpul necesar initializarii si controlului unei bucle, dar cu conditia sa includem "ceva" de fiecare data cand se executa bucla.

Iata acum analiza detailata a algoritmului. Fie a timpul necesar pentru o executare a buclei interioare, inclusiv partea de control. Executarea completa a buclei interioare, pentru un i dat, necesita $b+aT[i]$ unitati de timp, unde constanta b reprezinta timpul pentru initializarea buclei. Acest timp nu este zero, cand $T[i] = 0$. Timpul pentru o executare a buclei exterioare este $c+b+aT[i]$, c fiind o noua constanta. In fine, intregul algoritm necesita $d + \sum_{i=1}^{n} (c + b + aT[i])$ unitati de timp, unde d este o alta constanta. Simplificand, obtinem $(c+b)n+as+d$. Timpul $t(n, s)$ depinde deci de doi parametri independenti n si s. Avem: $t \in \Theta(n+s)$ sau, tinand cont de Exercitiul 5.19, $t \in \Theta(\max(n, s))$.

5.24 Pentru un tablou $T[1 \,..\, n]$, fie urmatorul algoritm de sortare:

```
for i ← n downto 1 do
  for j ← 2 to i do
    if T[ j−1] > T[ j] then interschimba T[ j−1] si T[ j]
```

Aceasta tehnica de sortare se numeste *metoda bulelor* (*bubble sort*).

i) Analizati eficienta algoritmului, luand ca barometru testul din bucla interioara.

ii) Modificati algoritmul, astfel incat, daca pentru un anumit i nu are loc nici o interschimbare, atunci algoritmul se opreste. Analizati eficienta noului algoritm.

5.25 Fie urmatorul algoritm

```
for i ← 0 to n do
    j ← i
    while j ≠ 0 do j ← j div 2
```

Gasiti ordinul exact al timpului de executie.

5.26 Demonstrati ca pentru oricare intregi pozitivi n si d

$$\sum_{k=0}^{d} 2^k \lg(n / 2^k) = 2^{d+1} \lg(n / 2^{d-1}) - 2 - \lg n$$

Solutie:

$$\sum_{k=0}^{d} 2^k \lg(n / 2^k) = (2^{d+1} - 1) \lg n - \sum_{k=0}^{d} (2^k k)$$

Mai ramane sa aratati ca

$$\sum_{k=0}^{d} (2^k k) = (d-1)2^{d+1} + 2$$

5.27 Analizati algoritmii *percolate* si *sift-down* pentru cel mai nefavorabil caz, presupunand ca opereaza asupra unui heap cu n elemente.

Indicatie: In cazul cel mai nefavorabil, algoritmii *percolate* si *sift-down* necesita un timp in ordinul exact al inaltimii arborelui complet care reprezinta heap-ul, adica in $\Theta(\lfloor \lg n \rfloor) = \Theta(\log n)$.

5.28 Analizati algoritmul *slow-make-heap* pentru cel mai nefavorabil caz.

Solutie: Pentru *slow-make-heap*, cazul cel mai nefavorabil este atunci cand, initial, T este ordonat crescator. La pasul i, se apeleaza *percolate*($T[1 .. i]$, i), care efectueaza $\lfloor \lg i \rfloor$ comparatii intre elemente ale lui T. Numarul total de comparatii este atunci

$$C(n) \leq (n-1) \lfloor \lg n \rfloor \in O(n \log n)$$

Pe de alta parte, avem

$$C(n) = \sum_{i=2}^{n} \lfloor \lg i \rfloor > \sum_{i=2}^{n} (\lg i - 1) = \lg n! - (n-1)$$

In Exercitiul 5.17 am aratat ca lg $n! \in \Omega(n \log n)$. Rezulta $C(n) \in \Omega(n \log n)$ si timpul este deci in $\Theta(n \log n)$.

5.29 Aratati ca, pentru cel mai nefavorabil caz, timpul de executie al algoritmului *heapsort* este si in $\Omega(n \log n)$, deci in $\Theta(n \log n)$.

5.30 Demonstrati ca, pentru cel mai nefavorabil caz, orice algoritm de sortare prin comparatie necesita un timp in $\Omega(n \log n)$. In particular, obtinem astfel, pe alta cale, rezultatul din Exercitiul 5.29.

Solutie: Orice sortare prin comparatie poate fi interpretata ca o parcurgere a unui arbore binar de decizie, prin care se stabileste ordinea relativa a elementelor de sortat. Intr-un arbore binar de decizie, fiecare varf neterminal semnifica o comparatie intre doua elemente ale tabloului T si fiecare varf terminal reprezinta o permutare a elementelor lui T. Executarea unui algoritm de sortare corespunde parcurgerii unui drum de la radacina arborelui de decizie catre un varf terminal. La fiecare varf neterminal se efectueaza o comparatie intre doua elemente $T[i]$ si $T[j]$: daca $T[i] \leq T[j]$ se continua cu comparatiile din subarborele stang, iar in caz contrar cu cele din subarborele drept. Cand se ajunge la un varf terminal, inseamna ca algoritmul de sortare a reusit sa stabileasca ordinea elementelor din T.

Fiecare din cele $n!$ permutari a celor n elemente trebuie sa apara ca varf terminal in arborele de decizie. Vom lua ca barometru comparatia intre doua elemente ale tabloului T. Inaltimea h a arborelui de decizie corespunde numarului de comparatii pentru cel mai nefavorabil caz. Deoarece cautam limita inferioara a timpului, ne intereseaza doar algoritmii cei mai performanti de sortare, deci putem presupune ca numarul de varfuri este minim, adica $n!$. Avem: $n! \leq 2^h$ (demonstrati acest lucru!), adica $h \geq \lg n!$. Considerand si relatia $\log n! \in \Omega(n \log n)$ (vezi Exercitiul 5.17), rezulta ca timpul de executie pentru orice algoritm de sortare prin comparatie este, in cazul cel mai nefavorabil, in $\Omega(n \log n)$.

5.31 Analizati algoritmul *heapsort* pentru cel mai favorabil caz. Care este cel mai favorabil caz?

5.32 Analizati algoritmii *fib*2 si *fib*3 din Sectiunea 1.6.4.

Solutie:

i) Se deduce imediat ca timpul pentru *fib*2 este in $\Theta(n)$.

ii) Pentru a analiza algoritmul *fib*3, luam ca barometru instructiunile din bucla **while**. Fie n_t valoarea lui n la sfarsitul executarii celei de-a t-a bucle. In particular, $n_1 = \lfloor n/2 \rfloor$. Daca $2 \leq t \leq m$, atunci

$$n_t = \lfloor n_{t-1}/2 \rfloor \leq n_{t-1}/2$$

Deci,

$$n_t \leq n_{t-1}/2 \leq \ldots \leq n/2^t$$

Fie $m = 1 + \lfloor \lg n \rfloor$. Deducem:

$$n_m \leq n/2^m < 1$$

Dar, $n_m \in \mathbf{N}$, si deci, $n_m = 0$, care este conditia de iesire din bucla. Cu alte cuvinte, bucla este executata de cel mult m ori, timpul lui *fib*3 fiind in $O(\log n)$. Aratati ca timpul este de fapt in $\Theta(\log n)$.

La analiza acestor doi algoritmi, am presupus implicit ca operatiile efectuate sunt independente de marimea operanzilor. Astfel, timpul necesar adunarii a doua numere este independent de marimea numerelor si este marginit superior de o constanta. Daca nu mai consideram aceasta ipoteza, atunci analiza se complica.

5.33 Rezolvati recurenta $t_n - 3t_{n-1} - 4t_{n-2} = 0$, unde $n \geq 2$, iar $t_0 = 0$, $t_1 = 1$.

5.34 Care este ordinul timpului de executie pentru un algoritm recursiv cu recurenta $t_n = 2t_{n-1} + n$.

Indicatie: Se ajunge la ecuatia caracteristica $(x-2)(x-1)^2 = 0$, iar solutia generala este $t_n = c_1 2^n + c_2 1^n + c_3 n 1^n$. Rezulta $t \in O(2^n)$.

Substituind solutia generala inapoi in recurenta, obtinem ca, indiferent de conditia initiala, $c_2 = -2$ si $c_3 = -1$. Atunci, toate solutiile interesante ale recurentei trebuie sa aiba $c_1 > 0$ si ele sunt toate in $\Omega(2^n)$, deci in $\Theta(2^n)$.

5.35 Scrieti o varianta recursiva a algoritmului de sortare prin insertie si determinati ordinul timpului de executie pentru cel mai nefavorabil caz.

Indicatie: Pentru a sorta $T[1 .. n]$, sortam recursiv $T[1 .. n-1]$ si inseram $T[n]$ in tabloul sortat $T[1 .. n-1]$.

5.36 Determinati prin schimbare de variabila ordinul timpului de executie pentru un algoritm cu recurenta $T(n) = 2T(n/2) + n \lg n$, unde $n > 1$ este o putere a lui 2.

Indicatie: $T(n) \in O(n \log^2 n \mid n$ este o putere a lui 2)

5.37 Demonstrati Proprietatea 5.2, folosind tehnica schimbarii de variabila.