

## Corectitudine și complexitate

Sunt prezentate notațiile uzuale de măsurare a complexității algoritmilor și modul în care acestea pot fi folosite în formule și, mai ales, deduse. De asemenea, sunt discutate tehnici de verificare a corectitudinii algoritmilor.

### 1 Calitatea algoritmilor

Calitatea unui algoritm poate fi caracterizată obiectiv în raport cu factori de performanță dinamică, de comportament, dintre cei mai importanți sunt consumul de resurse, mai ales de timp și spațiu de memorie. Spunem că un algoritm este cu atât mai complex cu cât consumul său de timp, respectiv de spațiu de memorie, este mai mare, iar analiza complexității are scopul expres de a determina ordinul de mărime și variația acestui consum în funcție de doi parametri esențiali: (a) operațiile critice efectuate de algoritm și (b) dimensiunea (talia) problemei rezolvate.

De obicei, orice algoritm are una sau mai multe prelucrări critice, ce impun un consum mare de resurse, fie prin natura lor, fie prin faptul că sunt executate frecvent în cursul aplicării algoritmului. De exemplu, în mulți algoritmi de sortare operația critică este compararea cheilor secvenței sortate, indiferent de reprezentarea secvenței ca vector, ca listă sau ca fișier. În cazul algoritmilor pentru grafuri, operația critică poate fi străbaterea arcelor sau vizitarea nodurilor. Aceste operații critice constituie baza analizei algoritmilor respectivi. În cele ce urmează, considerăm că operațiile elementare ale unui algoritm au cost unitar. Implicit, considerăm că problemele analizate satisfac această convenție.

Dimensiunea problemei este definită ca acel parametru sau acei parametri de care depinde frecvența execuției operațiilor critice ale algoritmilor de rezolvare. Teoretic, dimensiunea unei probleme este măsurată ca lungime a șirului de simboluri ce reprezintă datele problemei, simbolurile fiind considerate atomice din punctul de vedere al prelucrărilor efectuate de algoritm. De exemplu, pentru sortarea prin comparare de chei dimensiunea este numărul elementelor sortate (lungimea secvenței sortate). În cazul prelucrării grafurilor, dimensiunea este numărul de arce sau/și de noduri ale grafului. În schimb, pentru problemele cu caracter numeric, unde valorile datelor sunt modificate, trebuie avut în vedere numărul de biți folosiți pentru reprezentarea valorilor respective.

Pentru a compara, chiar și calitativ, algoritmii din punctul de vedere al performanțelor sunt necesare notații ale complexității. Notațiile sunt utile și din considerentul simplificării procesului de analiză a complexității. În foarte multe cazuri, analiza exactă este dificilă și îndeobște inutilă. Nu este nevoie de o valoare exactă a cantității de resurse consumate de algoritm, ci doar de o estimare a acestei măsuri

astfel încât algoritmul să poată fi caracterizat calitativ. Bunăoară, spunem că un algoritm este în timp liniar, sau că are complexitate liniară, dacă numărul operațiilor critice efectuate este direct proporțional cu dimensiunea problemei rezolvate. Totodată, este necesar ca precizia măsurării performanțelor să fie cu atât mai mare cu cât dimensiunea problemei rezolvate este mai mare. Notațiile de complexitate folosite în cele ce urmează reflectă valoarea asimptotică a performanțelor algoritmilor, fiind cu atât mai relevante cu cât dimensiunea problemelor crește.

## 2 Notații de complexitate

Considerăm că funcțiile de complexitate, care calculează exact performanța algoritmilor din punct de vedere cantitativ – al resurselor consumate –, sunt de variabilă întreagă cu valori reale pozitive:  $f: \mathbb{N} \rightarrow \mathbb{R}_+$ , unde  $\mathbb{N}$  este mulțimea numerelor naturale, iar  $\mathbb{R}_+$  este mulțimea realilor pozitivi. Parametrul unei asemenea funcții corespunde dimensiunii problemei, iar valoarea reprezintă consumul de resurse.

**Definiția 1.** Fie  $g: \mathbb{N} \rightarrow \mathbb{R}_+$  o funcție reprezentativă de complexitate. Clasa  $O(g(n))$  a algoritmilor conține exclusiv algoritmi care sunt caracterizați de funcții de complexitate ce fac parte din mulțimea:

$$O(g(n)) = \{f: \mathbb{N} \rightarrow \mathbb{R}_+ \mid \exists c \in \mathbb{R}_+, n_0 \in \mathbb{N} \mid c > 0 \bullet (\forall n \in \mathbb{N} \mid n \geq n_0 \bullet f(n) \leq c \cdot g(n))\}$$

Pentru valori mari ale dimensiunii problemei rezolvate, comportarea unui algoritm din clasa  $O(g(n))$  nu este mai slabă decât estimarea  $g(n)$ . Spunem că  $g(n)$  este limita asimptotică superioară a funcțiilor din  $O(g(n))$ .

**Definiția 2.** Fie  $g: \mathbb{N} \rightarrow \mathbb{R}_+$  o funcție reprezentativă de complexitate. Clasa  $\Omega(g(n))$  a algoritmilor conține exclusiv algoritmi care sunt caracterizați de funcții de complexitate ce fac parte din mulțimea:

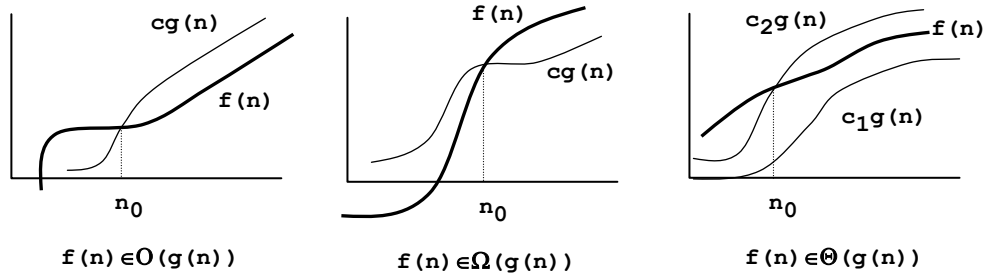
$$\Omega(g(n)) = \{f: \mathbb{N} \rightarrow \mathbb{R}_+ \mid \exists c \in \mathbb{R}_+, n_0 \in \mathbb{N} \mid c > 0 \bullet (\forall n \in \mathbb{N} \mid n \geq n_0 \bullet f(n) \geq c \cdot g(n))\}$$

Pentru valori mari ale dimensiunii problemei rezolvate, comportarea unui algoritm din clasa  $\Omega(g(n))$  este mai slabă decât estimarea  $g(n)$ . Spunem că  $g(n)$  este limita asimptotică inferioară a funcțiilor din  $O(g(n))$ .

**Definiția 3.** Fie  $g: \mathbb{N} \rightarrow \mathbb{R}_+$  o funcție reprezentativă de complexitate. Clasa  $\Theta(g(n))$  a algoritmilor conține exclusiv algoritmi care sunt caracterizați de funcții de complexitate ce fac parte din mulțimea:

$$\Theta(g(n)) = \{f: \mathbb{N} \rightarrow \mathbb{R}_+ \mid \exists c_1, c_2 \in \mathbb{R}_+, n_0 \in \mathbb{N} \mid c_1 > 0 \wedge c_2 > 0 \bullet (\forall n \in \mathbb{N} \mid n \geq n_0 \bullet c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n))\}$$

Pentru valori mari ale dimensiunii problemei rezolvate, comportarea unui algoritm din clasa  $\Theta(g(n))$  este caracterizată aproximativ de  $g(n)$ .



**Figura 1** Interpretarea grafică a notațiilor de complexitate

Interpretarea grafică a definițiilor de mai sus este dată în figura 1. În ceea ce privește convenția de notare, tradițional se scrie  $f(n) = X(g(n))$ , cu  $X$  dintre  $O$ ,  $\Omega$ ,  $\Theta$ , în loc de  $f(n) \in X(g(n))$ .

Notațiile  $O$ ,  $\Omega$  și  $\Theta$  sunt folosite pentru a aproxima asimptotic "strâns" complexitatea unui algoritm. Complexitatea estimată diferă față de cea exactă doar cu un factor constant. Există cazuri în care aproximarea asimptotică este "largă", complexitatea estimată diferind dincolo de limitele unui factor constant față de cea exactă. De exemplu, pentru funcția de complexitate  $f(n) = n^2$ , aproximarea asimptotică  $f(n) = O(n^3)$  este largă. Pentru a diferenția aproximările asimptotice "largi" față de cele "strânse" se folosesc notațiile  $o$  și  $\omega$ .

**Definiția 1'.** Fie  $g: \mathbb{N} \rightarrow \mathbb{R}_+$  o funcție reprezentativă de complexitate. Clasa  $o(g(n))$  a algoritmilor conține exclusiv algoritmi care sunt caracterizați de funcții de complexitate ce fac parte din mulțimea:

$$o(g(n)) = \{f: \mathbb{N} \rightarrow \mathbb{R}_+ \mid \forall c \in \mathbb{R}_+ \mid c > 0 \bullet (\exists n_0 \in \mathbb{N} \bullet (\forall n \in \mathbb{N} \mid n \geq n_0 \bullet f(n) < c \cdot g(n)))\}$$

**Definiția 2'.** Fie  $g: \mathbb{N} \rightarrow \mathbb{R}_+$  o funcție reprezentativă de complexitate. Clasa  $\omega(g(n))$  a algoritmilor conține exclusiv algoritmi care sunt caracterizați de funcții de complexitate ce fac parte din mulțimea:

$$\omega(g(n)) = \{f: \mathbb{N} \rightarrow \mathbb{R}_+ \mid \forall c \in \mathbb{R}_+ \mid c > 0 \bullet (\exists n_0 \in \mathbb{N} \bullet (\forall n \in \mathbb{N} \mid n \geq n_0 \bullet f(n) > c \cdot g(n)))\}$$

Diferența dintre  $o(g(n))$  și  $O(g(n))$ , respectiv dintre  $\omega(g(n))$  și  $\Omega(g(n))$ , este vizibilă. Astfel, în cazul  $O(g(n))$  și  $\Omega(g(n))$  constantele  $c$  și  $n_0$  sunt fixate în raport cu funcția de complexitate aproximată asimptotic, în timp ce pentru  $o(g(n))$  și  $\omega(g(n))$  constanta  $c$  poate fi aleasă arbitrar. Bunăoară, în cazul aproximării  $f(n) = o(n^3)$ , cu  $f(n) = n^2$ , oricare ar fi constanta  $c \in \mathbb{R}_+$ ,  $c > 0$ , există  $n_0 \in \mathbb{N}$  astfel încât  $n^2 < c \cdot n^3$ , pentru

$n \geq n_0$ . Constanta  $n_0$  trebuie să satisfacă inegalitatea  $n_0 > \frac{1}{c}$ . Astfel, pentru  $c \geq 1$  avem  $n_0 > 1$ . Pentru  $c = \frac{1}{k}$ ,  $k > 1$ , avem  $n_0 > k$ .

Larghețea aproximărilor asimptotice  $o$  și  $\omega$  derivă și din proprietățile următoare, induse de definițiile (1') și (2'):

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0 \text{ în cazul } f(n) = o(g(n))$$

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty \text{ în cazul } f(n) = \omega(g(n))$$

Conform  $f(n) = o(g(n))$ , să alegem la întâmplare  $n_1 \in \mathbb{N}$  și fie  $c_1 \in \mathbb{R}_+ \setminus \{0\}$  constanta cea mai mică astfel încât  $f(n) < c_1 g(n)$ , pentru  $n \geq n_1$ . De asemenea, să alegem cea mai mare constantă posibilă  $c_2 \in \mathbb{R}_+ \setminus \{0\}$  și  $n_2 \in \mathbb{N}$  astfel încât să fie satisfăcute restricțiile:

- $c_1 > c_2$
- $f(n) < c_2 g(n)$ , pentru  $n \geq n_2$

Presupunem  $n_1 \geq n_2$ . Avem  $f(n) < c_2 g(n)$ , pentru  $n \geq n_1$ , ceea ce contrazice alegerea constantei  $c_1$ . Deci  $n_1 < n_2$ . Putem alege șirul monoton descrescător de constante  $c_1, c_2, \dots, c_i, \dots$  și șirul monoton crescător  $n_1, n_2, \dots, n_i, \dots$  astfel încât  $f(n) < c_i g(n)$ , pentru  $n \geq n_i$ ,  $i \geq 1$ . Raportul  $\frac{f(n_i)}{g(n_i)} < c_i$  este descrescător către 0

prin creșterea valorii  $n_i$ , deci  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$ . Folosind proprietatea  $f(n) = o(g(n)) \Leftrightarrow$

$g(n) = \omega(f(n))$ , obținem  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty$  pentru  $f(n) = \omega(f(n))$ .

Evident,  $f(n) = o(g(n))$  implică  $f(n) = O(g(n))$ , iar  $f(n) = \omega(g(n))$  implică  $f(n) = \Omega(g(n))$ . Reciproca nu este adevărată. De exemplu,  $n^2 = O(n^2)$ , dar  $n^2 \neq o(n^2)$ . Într-adevăr, dacă  $n^2 = o(n^2)$ , atunci ar trebui ca pentru orice constantă  $c \in \mathbb{R}_+$ ,  $c > 0$ , să existe  $n_0 \in \mathbb{N}$  astfel încât  $0 \leq n^2 < c n^2$ , pentru  $n \geq n_0$ . Dar, pentru  $c < 1$  inegalitatea este imposibilă, deci  $n_0$  nu există pentru o asemenea constantă.

Pentru a sublinia diferența esențială între notațiile  $o$  și  $O$ , respectiv între  $\omega$  și  $\Omega$ , anume modul de alegere a constantei  $c$ , definițiile notațiilor  $o$  și  $\omega$  pot fi rescrise:

$$o'(g(n)) = \{f: \mathbb{N} \rightarrow \mathbb{R}_+ \mid \forall c \in \mathbb{R}_+ \mid c > 0 \bullet (\exists n_0 \in \mathbb{N} \bullet (\forall n \in \mathbb{N} \mid n \geq n_0 \bullet f(n) \leq c g(n)))\}$$

$$\omega'(g(n)) = \{f: \mathbb{N} \rightarrow \mathbb{R}_+ \mid \forall c \in \mathbb{R}_+ \mid c > 0 \bullet (\exists n_0 \in \mathbb{N} \bullet (\forall n \in \mathbb{N} \mid n \geq n_0 \bullet f(n) \geq c g(n)))\}$$

Se arată ușor că  $o'(g(n)) = o(g(n))$  și  $\omega'(g(n)) = \omega(g(n))$ . De exemplu, pentru  $o'(g(n)) = o(g(n))$ , alegem la întâmplare o funcție  $f(n) \in o'(g(n))$  și o constantă  $c > 0$ . Pentru  $c' < c$  avem  $f(n) \leq c'g(n) < c g(n)$ , deci  $f(n) \in o(g(n))$ . Pentru că  $f$  a fost aleasă la întâmplare, rezultă  $o'(g(n)) \subseteq o(g(n))$ . Similar, pentru o funcție  $f(g(n)) \in o(g(n))$  aleasă la întâmplare avem  $f(n) < c g(n)$ ,  $n \geq n_0$ , pentru orice  $c > 0$ , și o constantă  $n_0$  aleasă în raport cu  $c$ . Putem scrie,  $f(n) \leq c g(n)$ ,  $n \geq n_0$ , deci  $f(n) \in o'(g(n))$ . Rezultă  $o(g(n)) \subseteq o'(g(n))$ . În final  $o'(g(n)) = o(g(n))$ .

Figura 1 lasă să transpară câteva proprietăți utile ale notațiilor  $O$ ,  $\Omega$  și  $\Theta$ . Astfel diagrama notației  $\Theta$  sugerează proprietatea:

$$f(n) = \Theta(g(n)) \Leftrightarrow f(n) = O(g(n)) \text{ și } f(n) = \Omega(g(n))$$

De asemenea, notațiile de complexitate satisfac condițiile impuse unor relații de ordine sau de echivalență peste mulțimea funcțiilor  $N \rightarrow \mathbb{R}_+$ .

**Propoziția 1** Notațiile  $O$ ,  $\Omega$ ,  $\Theta$ ,  $o$  și  $\omega$  au următoarele proprietăți:

Tranzitivitatea:

$$f(n) = O(h(n)) \text{ și } h(n) = O(g(n)) \Rightarrow f(n) = O(g(n))$$

$$f(n) = \Omega(h(n)) \text{ și } h(n) = \Omega(g(n)) \Rightarrow f(n) = \Omega(g(n))$$

$$f(n) = \Theta(h(n)) \text{ și } h(n) = \Theta(g(n)) \Rightarrow f(n) = \Theta(g(n))$$

$$f(n) = o(h(n)) \text{ și } h(n) = o(g(n)) \Rightarrow f(n) = o(g(n))$$

$$f(n) = \omega(h(n)) \text{ și } h(n) = \omega(g(n)) \Rightarrow f(n) = \omega(g(n))$$

$$\text{Reflexivitatea: } f(n) = O(f(n)), f(n) = \Omega(f(n)), f(n) = \Theta(f(n))$$

$$\text{Simetria: } f(n) = \Theta(g(n)) \Leftrightarrow g(n) = \Theta(f(n))$$

Antisimetria:

$$f(n) = O(g(n)) \Leftrightarrow g(n) = \Omega(f(n))$$

$$f(n) = o(g(n)) \Leftrightarrow g(n) = \omega(f(n))$$

$$\text{Proiecția } \Theta: f(n) = \Theta(g(n)) \Leftrightarrow f(n) = O(g(n)) \text{ și } f(n) = \Omega(g(n))$$

Demonstrațiile sunt simple, fiind lăsate ca exercițiu.

Notațiile  $O$  și  $\Omega$  pot fi asimilate cu relații de ordine parțială peste mulțimea funcțiilor  $N \rightarrow \mathbb{R}_+$ .  $O$  este oarecum similar relației  $\leq$ , iar  $\Omega$  este asemănătoare cu  $\geq$ . În schimb  $\Theta$  joacă rolul unei relații de echivalență, aidoma relației  $=$ , peste mulțimea  $N \rightarrow \mathbb{R}_+$ . Notația  $o$  este asemănătoare relației de ordine  $<$ , iar  $\omega$  joacă rolul relației de ordine  $>$ . Este ușor de observat că relațiile definite de  $O$ ,  $\Omega$ ,  $\Theta$ ,  $o$  și  $\omega$  sunt parțiale. Există funcții  $f$  și  $g$ , cum ar fi  $f(x) = 1$  și  $g(x) = 2 \sin(x)$ , între care nu există nici o relație conformă cu vreuna din notațiile de complexitate. Faptul că  $O$ ,  $\Omega$ ,  $\Theta$  pot fi văzute ca relații de ordine sau de echivalență, fie și parțiale, ajută compararea algoritmilor caracterizați de funcții din clasele respective de complexitate. De exemplu, putem arăta ca algoritmi din clasa de complexitate  $\Theta(n^2)$  sunt, ca performanțe, exact cei din clasa  $\Theta(n^2 + n)$ . Cu alte cuvinte,  $\Theta(n^2 + n) = \Theta(n^2)$ .

Fie doi algoritmi aleși la întâmplare din cele două clase, unul caracterizat de o funcție de complexitate  $f(n) = \Theta(n^2)$  și altul cu o funcție de complexitate  $g(n) = \Theta(n^2+n)$ .

1. Să arătăm că  $n^2 = \Theta(n^2+n)$ . Conform definiției (3), trebuie să existe constantele  $c_1, c_2$  și  $n_0$  astfel încât  $0 \leq c_1(n^2+n) \leq n^2 \leq c_2(n^2+n)$ , pentru  $n \geq n_0$ . Rezultă,  $c_2 = 1$ ,  $c_1 = 0.5$  și  $n_0 = 1$ .

2. Conform tranzitivității,  $f(n) = \Theta(n^2)$  și  $n^2 = \Theta(n^2+n)$  implică  $f(n) = \Theta(n^2+n)$ .

3. Conform simetriei  $n^2 = \Theta(n^2+n)$  implică  $n^2+n = \Theta(n^2)$  și pentru că  $g(n) = \Theta(n^2+n)$ , prin tranzitivitate, rezultă  $g(n) = \Theta(n^2)$ .

4. Conform (1) și (3) rezultă că orice funcție din clasa de complexitate  $\Theta(n^2)$  este în clasa de complexitate  $\Theta(n^2+n)$  și, reciproc, orice funcție din  $\Theta(n^2+n)$  este în  $\Theta(n^2)$ , deci  $\Theta(n^2) \subseteq \Theta(n^2+n)$  și  $\Theta(n^2+n) \subseteq \Theta(n^2)$ . Obținem  $\Theta(n^2+n) = \Theta(n^2)$ .

Proprietatea demonstrată poate fi generalizată pentru orice polinom în  $n$ :  $\Theta(c_k n^k + c_{k-1} n^{k-1} + \dots + c_1 n + c_0) = \Theta(n^k)$ , cu  $k \geq 0$  și  $c_k > 0$ .

### 3 Utilizarea notațiilor asimptotice

Notațiile  $O$ ,  $\Omega$  și  $\Theta$  lucrează cu funcții de variabilă întreagă. Dar, deseori, în cursul analizei algoritmilor, funcțiile de complexitate sunt descrise de ecuații cu termeni care nu sunt întregi. De exemplu, recurența  $T(n) = 2T(n/2) + n$  conține termeni  $n/2$ , întregi doar pentru  $n = 2^k$ . În astfel de situații considerăm că un asemenea termen, fie el  $x$ , este rotunjit la întregul imediat superior  $\lceil x \rceil$  sau trunchiat la întregul imediat inferior  $\lfloor x \rfloor$ . În majoritatea cazurilor această convenție nu modifică clasa de complexitate a algoritmului analizat.

De asemenea, măsurile asimptotice de complexitate pot să apară în formule care aproximează comportarea unui algoritm. Un exemplu este funcția de complexitate caracterizată de recurența  $T(n) = 2T(n/2) + \Theta(n)$  sau calculul în care apare funcția:

$$F(n) = \frac{1 + \Theta\left(\frac{1}{kn}\right)}{1 + \Theta\left(\frac{1}{n}\right)}, \text{ pentru } k > 0.$$

Termenii  $\Theta\left(\frac{1}{kn}\right)$  și  $\Theta\left(\frac{1}{n}\right)$  înlocuiesc funcții anonime, a căror formulă exactă nu este importantă, importantă fiind doar aproximarea lor asimptotică. Mai mult, în astfel de cazuri se pot efectua calcule cu clasele de complexitate, de fapt cu funcții anonime, reprezentante ale claselor respective. Ca exemplu de calcul cu măsuri de complexitate să demonstrăm următoarea proprietate.

$$\text{Propoziția 2} \quad F(n) = \frac{1 + \Theta\left(\frac{1}{kn}\right)}{1 + \Theta\left(\frac{1}{n}\right)} = \Theta(1) \quad (1)$$

a) Să notăm  $f(n)$  funcția reprezentată de  $\Theta\left(\frac{1}{kn}\right)$  și cu  $g(n)$  funcția desemnată de  $\Theta\left(\frac{1}{n}\right)$ . Membrul stâng din ecuația (1) devine  $F(n) = \frac{1 + f(n)}{1 + g(n)}$  și, conform definiției notației  $\Theta$ , avem:

$$0 \leq a_1 \frac{1}{kn} \leq f(n) \leq a_2 \frac{1}{kn}, \text{ pentru } n \geq n_a \geq 1 \quad (2)$$

$$0 \leq b_1 \frac{1}{n} \leq g(n) \leq b_2 \frac{1}{n}, \text{ pentru } n \geq n_b \geq 1 \quad (3)$$

cu constantele  $a_1, a_2, b_1, b_2$  strict pozitive.

b) Să notăm  $n_0 = \max(n_a, n_b)$ . Din inecuațiile (2) și (3) rezultă:

$$f(n) \leq \frac{a_2}{kn} \text{ și } g(n) \geq \frac{b_1}{n}, \text{ deci } F(n) \leq \frac{1 + \frac{a_2}{kn}}{1 + \frac{b_1}{n}} \leq 1 + \frac{a_2}{k}, \text{ pentru } n \geq n_0$$

$$f(n) \geq \frac{a_1}{kn} \text{ și } g(n) \leq \frac{b_2}{n}, \text{ deci } F(n) \geq \frac{1 + \frac{a_1}{kn}}{1 + \frac{b_2}{n}} \geq \frac{1}{1 + b_2}, \text{ pentru } n \geq n_0$$

Deci  $F(n) = \Theta(1)$ . ■

Există multe proprietăți utile care servesc simplificării ecuațiilor în care apar măsuri de complexitate. Menționăm doar câteva echivalențe între clase de complexitate pentru notația  $\Theta$ , dar care sunt valabile și pentru  $\Omega$  și  $O$ . Probarea lor este simplă, fiind lăsată ca exercițiu.

**Propoziția 3** Există următoarele proprietăți.

$\Theta(k f(n)) = \Theta(f(n))$ , pentru orice constantă pozitivă  $k$ ;

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

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

$\Theta(c_k n^k + c_{k-1} n^{k-1} + \dots + c_0) = \Theta(n^k)$ , pentru  $k \geq 0$  și  $c_k > 0$ .

De exemplu, notând  $\log_2$  cu  $lg$  și ținând cont că pentru o funcție  $f: \mathbb{N} \rightarrow \mathbb{R}_+$  monoton crescătoare există inegalitatea  $\int_{m-1}^n f(x) dx \leq \sum_{k=m}^n f(k) \leq \int_m^{n+1} f(x) dx$ , avem:

$$\sum_{k=1}^n O(lg(k)) = O \sum_{k=1}^n lg(k) = O \left( \int_1^{n+1} lg(k) dk \right) = O(n lg(n))$$

Lucrul cu notațiile de complexitate este delicat și trebuie să țină seama de faptul că notațiile nu reprezintă valori ci mulțimi de funcții și că, atunci când apar în formule, stau în locul unor funcții anonime. Mai jos este ilustrat un exemplu edificator de raționament greșit, preluat din [CLRS 01].

Să demonstrăm prin inducție după  $n$  că  $F(n) = \sum_{k=1}^n k = O(n)$ , pentru  $n \geq 1$ .

**Caz de bază.** Pentru  $n = 1$  rezultă banal  $F(1) = O(1)$ .

**Pas de inducție.** Ipoteza inductivă este  $F(n) = O(n)$ . Să demonstrăm că  $F(n+1) = O(n+1)$ . Avem  $F(n+1) = \sum_{k=1}^{n+1} k = \sum_{k=1}^n k + (n+1) = O(n) + (n+1) = O(n+1)$ .

Incorect, pentru că  $\sum_{k=1}^n k = O(n^2)$ . Ce este greșit în demonstrație?

Eroarea este la pasul de inducție, unde constanta implicată de notația  $O(n)$  din ipoteza inductivă este alta decât cea impusă de  $O(n+1)$  pentru rezultatul obținut. Într-adevăr,  $F(n) = O(n)$  implică  $F(n) \leq c n$ , pentru  $n \geq n_0$ . Prin urmare, cu atât mai mult trebuie să avem  $F(n+1) \leq c(n+1)$ , pentru  $n \geq n_0$ . Dar rezultatul obținut este altul.

$$\sum_{k=1}^{n+1} k = \sum_{k=1}^n k + (n+1) \leq c n + n+1 \leq \frac{(c+1)}{\uparrow} (n+1), \text{ pentru } n \geq n_0.$$

constantă diferită de  $c$

Conform definiției notației  $O$ , pasul de inducție ar trebui să probeze că aceeași constantă ascunsă de  $F(n) = O(n)$  este validă și pentru  $F(n+1) = O(n+1)$  ceea ce este imposibil, pentru că aici constanta variază odată cu  $n$ . Deci,  $F(n)$  nu poate fi în clasa de complexitate  $O(n)$ .

## 4 Corectitudinea algoritmilor

Proprietatea esențială a unui algoritm este *corectitudinea* rezultatelor obținute în raport cu datele problemei rezolvate. În general, un algoritm  $Alg$  cu date de tip  $I$  și rezultate de tip  $O$  este dezvoltat pentru a prelucra date  $d \in I$  considerate corecte, adică date care satisfac un predicat  $P_I: I \rightarrow \{0, 1\}$  numit *asertiune de intrare* a algoritmului. Similar, rezultatele algoritmului pentru datele  $d$ , rezultate notate  $r = Alg(d)$ ,  $r \in O$ , sunt considerate corecte dacă satisfac un predicat  $P_O: I \times O \rightarrow \{0, 1\}$  numit *asertiune de ieșire* a algoritmului. Cele două aserțiuni caracterizează problema rezolvată. Astfel, pentru algoritmul de calcul al factorialului unui număr natural  $n$ ,

```
fact(n) { if (n = 1) return 1; return n * fact(n-1) }
```

cu  $I = \mathbb{N}$  și  $O = \mathbb{N}$ , aserțiunile sunt:  $P_I(n) =_{\text{def}} n \geq 1$  și  $P_O(n, r) =_{\text{def}} r = \prod_{k=1}^n k$ .



#### 4.1 Corectitudine parțială și corectitudine totală

Intuitiv, un algoritm este corect dacă produce rezultate care satisfac aserțiunea de ieșire în cazul unor date valide, conforme aserțiunii de intrare.

**Definiția 4** Fie  $\text{Alg}$  un algoritm cu datele  $\mathbf{I}$  și rezultatele  $\mathbf{O}$ , iar  $P_{\mathbf{I}}$  și  $P_{\mathbf{O}}$  aserțiuni de intrare, respectiv de ieșire, definite în raport cu  $\text{Alg}$ .

- Algoritmul este *parțial corect* în raport cu aserțiunile  $P_{\mathbf{I}}$  și  $P_{\mathbf{O}}$  dacă există proprietatea:

$$\forall d \in \mathbf{I} \bullet (P_{\mathbf{I}}(d) \wedge \text{se\_termină Alg}(d)) \Rightarrow P_{\mathbf{O}}(d, \text{Alg}(d))$$

- Algoritmul este *total corect* în raport cu aserțiunile  $P_{\mathbf{I}}$  și  $P_{\mathbf{O}}$  dacă există proprietatea:

$$\forall d \in \mathbf{I} \bullet P_{\mathbf{I}}(d) \Rightarrow (\text{se\_termină Alg}(d) \wedge P_{\mathbf{O}}(d, \text{Alg}(d)))$$

După cum se observă, corectitudinea parțială nu implică terminarea algoritmului, ci doar satisfacerea aserțiunii de ieșire. Astfel, algoritmul

```
plus(x,y) {if(x = 0) return y; return 1+plus(x-1,y)}
```

cu  $\mathbf{I} = \mathbb{Z} \times \mathbb{Z}$  și  $\mathbf{O} = \mathbb{Z}$ , unde  $\mathbb{Z}$  este mulțimea întregilor, este doar parțial corect în raport cu aserțiunile:  $P_{\mathbf{I}}(x,y) =_{\text{def}} 1$  și  $P_{\mathbf{O}}((x,y), r) =_{\text{def}} (r = x+y)$ .

Apelul `plus(-1,2)` nu se termină deși, cu ipoteza "evaluarea algoritmului `plus` se termină forțat după  $k > 0$  pași și, în momentul terminării, `plus(x-k,y) = x-k+y`", aserțiunea de ieșire  $P_{\mathbf{O}}$  este satisfăcută. Aici aserțiunea de intrare nu corespunde problemei rezolvate: adunarea a doi întregi, dintre care primul trebuie să fie pozitiv. În schimb, algoritmul este total corect pentru aserțiunile:

$$P_{\mathbf{I}}(x,y) =_{\text{def}} (x \geq 0) \text{ și } P_{\mathbf{O}}((x,y), r) =_{\text{def}} (r = x+y)$$

Algoritmul se termină dacă se aplică asupra unor parametri care satisfac  $P_{\mathbf{I}}$ , iar aserțiunea de ieșire este verificată de rezultat și de datele algoritmului<sup>1</sup>.

Analiza corectitudinii unui algoritm nu are sens fără existența unor aserțiuni de intrare și de ieșire care să specifice proprietățile datelor corecte ale problemei și proprietățile rezultatelor. Din perspectiva corectitudinii parțiale a unui algoritm, vom înțelege prin verificare a corectitudinii demonstrarea satisfacerii aserțiunii de ieșire a algoritmului pentru orice date care satisfac aserțiunea de intrare. În ceea ce privește terminarea, fie o vom considera ca o ipoteză, fie o vom demonstra intuitiv. De asemenea, uneori, aserțiunile de intrare și de ieșire vor fi implicite, rezultând indirect din textul problemei rezolvate de algoritm.

---

<sup>1</sup> Demonstrația este prin inducție matematică după  $x$ .

## 4.2 Verificarea inductivă a corectitudinii parțiale

Verificarea corectitudinii este activitatea cea mai dificilă și cea mai puțin "standardizată" a procesului de proiectare a algoritmilor. Deși demonstrarea corectitudinii unui algoritm este puternic influențată de natura problemei rezolvate, pot fi folosite ca suport metodele generale de verificare cum ar fi reducerea la absurd, deducția naturală și inducția. În cele ce urmează, acordăm o atenție deosebită inducției, datorită frecvenței cu care apare în procesul de analiză a algoritmilor atât în ceea ce privește corectitudinea cât și complexitatea lor.

### Inducția structurală

O metodă de verificare a corectitudinii parțiale a unui algoritm este *inducția structurală*, o generalizare a inducției matematice pentru tipuri de date. Deseori, un algoritm poate fi privit ca un transformator al valorilor unui tip de date  $\tau$ . Algoritmul, sau părți ale algoritmului, acceptă valori de tip  $\tau$  care satisfac o proprietate  $P$  și produc rezultate de tip  $\tau$  care satisfac proprietatea<sup>2</sup>  $P$ . Prin urmare, algoritmul poate fi considerat un constructor al valorilor de tip  $\tau$  ce satisfac  $P$ . Este cazul în care verificarea corectitudinii algoritmului se poate efectua prin inducție structurală.

Pentru un tip de date  $\tau$  și o proprietate  $P$  a valorilor lui  $\tau$ , proprietate asociată unui algoritm, inducția structurală verifică dacă mulțimea de adevăr a proprietății  $P$  este închisă în raport cu constructorii tipului  $\tau$ . Deoarece toate valorile lui  $\tau$  (eventual o infinitate) pot fi formate cu constructorii tipului, se verifică dacă  $P$  este satisfăcută de tipul  $\tau$ , deci dacă algoritmul caracterizat de  $P$  este corect. Formal, proprietatea  $P$  este satisfăcută de tipul  $\tau$  (sau, echivalent,  $\forall x \in \tau \bullet P(x)$ ) dacă:

- a) Orice constructor nular  $\sigma$  al lui  $\tau$  satisface  $P$ . Deci constantele din  $\tau$  satisfac  $P$ . Notăm  $\Sigma_0$  mulțimea constructorilor nulari ai lui  $\tau$ .  $\Sigma_0 =_{\text{def}} \{\sigma : \rightarrow \tau\}$
- b) Pentru orice constructor  $\sigma : \text{Dom}_\sigma \rightarrow \tau$ , astfel încât valorile din  $\text{Dom}_\sigma$  nu conțin valori de tip  $\tau$ , valoarea  $\sigma(v)$ ,  $v \in \text{Dom}_\sigma$ , satisface  $P$ . Cu alte cuvinte,  $P$  este satisfăcută de orice valoare a lui  $\tau$  construită exclusiv cu valori externe tipului. Numim  $\Sigma_e$  mulțimea constructorilor externi ai lui  $\tau$ .  $\Sigma_e =_{\text{def}} \{\sigma : \text{Dom}_\sigma \rightarrow \tau \mid \tau \text{ nu apare în } \text{Dom}_\sigma\}$
- c) Pentru orice constructor  $\sigma : \text{Dom}_\sigma \rightarrow \tau$ , astfel încât valorile din  $\text{Dom}_\sigma$  conțin valori de tip  $\tau$ , valoarea  $y = \sigma(\dots, x, \dots)$  satisface proprietatea  $P$ , considerând că orice valoare  $x \in \tau$  care apare în  $y$  satisface  $P$ . Deci  $P$  este satisfăcută de valorile simbolice ale lui  $\tau$  formate recursiv cu valori ale lui  $\tau$ , corecte în raport cu  $P$ . Numim  $\Sigma_\tau$  mulțimea constructorilor nenulari ai lui  $\tau$ .  $\Sigma_\tau =_{\text{def}} \{\sigma : \text{Dom}_\sigma \rightarrow \tau \mid \tau \text{ apare în } \text{Dom}_\sigma\}$

<sup>2</sup> Folosirea inducției, pentru a demonstra corectitudinea unui algoritm  $\text{Alg} : I \rightarrow O$  în raport cu aserțiunile  $P_I$  și  $P_O$ , impune formularea unei proprietăți  $P(i)$ ,  $i \in I$ . Această proprietate combină cele două aserțiuni și, în general, are forma  $P(i) =_{\text{def}} P_I(i) \Rightarrow P_O(i, \text{Alg}(i))$

Inducția structurală poate fi descrisă prin intermediul schemei de mai jos, care pune în evidență elementele principale: cazurile de bază și pasul inducției.

$$\begin{array}{c}
 \begin{array}{c} \text{cazuri de bază} \\ | \\ \forall \sigma \in \Sigma_0 \bullet P(\sigma) \\ \forall \sigma \in \Sigma_e, v \in \text{Dom}_\sigma \bullet P(\sigma(v)) \end{array}
 \qquad
 \begin{array}{c} \text{ipoteză inductivă} \\ | \\ [P(x)] \\ \hline P(\sigma(\dots, x, \dots)) \end{array}
 \qquad
 \begin{array}{c} \text{pas inducție} \\ \vdash \\ \forall \sigma \in \Sigma_t, x \in t \end{array}
 \end{array}
 \quad
 \frac{}{\forall x \in t \bullet P(x)}$$

Inducția structurală folosește dimensiunea valorilor simbolice ale tipului  $t$ . Dimensiunea unei valori  $x \in t$  este adâncimea maximă a aparițiilor constructorilor lui  $t$  în cadrul formulei ce reprezintă valoarea simbolică  $x$ . De exemplu, dimensiunea valorii  $\text{insert}(1, \text{insert}(3, \text{newset}))$  a unei mulțimi de întregi este 2. Dacă definim  $\text{dim}: t \rightarrow \mathbb{N}$ , funcția care calculează dimensiunea valorilor simbolice ale lui  $t$ , atunci se observă că pasul de inducție din schema inducției structurale folosește în ipoteza inductivă valori  $x \in t$  cu  $\text{dim}(x) < \text{dim}(\sigma(\dots, x, \dots))$ . Astfel, inducția este efectuată în raport cu dimensiunea valorilor simbolice ale tipului  $t$ . Observația sugerează că inducția structurală este o particularizare a unei scheme mai generale de inducție, numită inducție bine formată.

### Inducția bine formată

Fie  $\prec$  o relație peste o mulțime  $x$ , astfel încât orice șir  $\dots \prec x_n \prec \dots \prec x_2 \prec x_1$  cu elemente din  $x$  poate fi extins la stânga cu un număr finit de elemente din  $x$ . Se spune că relația  $\prec$  este *bine formată*. De exemplu, relația  $<$  definită peste mulțimea numerelor naturale este bine formată. În schimb, relația  $<$  pentru mulțimea numerelor întregi nu este bine formată. Alegând o funcție totală  $f: x \rightarrow \mathbb{N}$ , putem interpreta  $f$  ca *funcție de măsurare* a elementelor din mulțimea  $x$  și putem defini relația bine formată  $\prec_f$  peste  $x$ :

$$\forall a, b \in x \bullet (a \prec_f b \Leftrightarrow f(a) < f(b))$$

De exemplu, relația  $\prec_{\text{length}}$ , definită pe mulțimea listelor, este bine formată. La fel, pentru tipul mulțime finită, relația  $\prec_{\text{card}}$  (cardinalul mulțimii) este bine formată.

Folosind conceptul de relație bine formată, principiul inducției poate fi formulat într-o variantă generală, numită *inducție bine formată*. Inducția completă, inducția matematică și cea structurală sunt particularizări ale inducției bine formate.

$$\begin{array}{c}
 \text{ipoteză inductivă} \\
 [\forall y \in x \mid y \prec_f x \bullet P(y)] \vdash \\
 \forall x \in x \quad \frac{}{P(x)} \quad \text{pas inducție} \\
 \hline
 \forall x \in x \bullet P(x)
 \end{array}$$

Pentru un element  $x \in X$  oarecare, pasul de inducție referitor la  $x$  pornește de la ipoteza inductivă  $\forall y \in X \mid y \prec_f x \bullet P(y)$  pentru a demonstra proprietatea  $P(x)$ . Cu alte cuvinte, cu ipoteza că  $P$  este verificată de toate elementele mulțimii  $X$  care au măsură inferioară lui  $x$ , trebuie demonstrat că și  $x$  satisface proprietatea  $P$ . În particular, pentru  $x_q \in X$  astfel încât nu există  $y \in X$ ,  $y \prec_f x_q$ , ipoteza inductivă este adevărată banal. În acest caz, numit *de bază*, trebuie verificată proprietatea  $P(x_q)$ .

Argumentul corectitudinii inducției bine formate se obține prin contradicție. Dacă presupunem că schema de inducție funcționează pentru a demonstra o proprietate  $P$  și totuși proprietatea este falsă, înseamnă că există o valoare  $x_1$  astfel încât  $\neg P(x_1)$ . În acest caz, pasul de inducție este aplicat banal pentru  $x_1$ , adică implicația  $[\forall y \in X \mid y \prec_f x_1 \bullet P(y)] \Rightarrow P(x_1)$  este adevărată pentru că premiza  $\forall y \in X \mid y \prec_f x_1 \bullet P(y)$  este falsă, deci  $\exists x_2 \in X \mid x_2 \prec_f x_1 \bullet \neg P(x_2)$ . Repetând raționamentul ajungem la concluzia că există un șir  $\dots \prec x_n \prec \dots \prec x_2 \prec x_1$  infinit, ceea ce contrazice ipoteza: relația  $\prec_f$  este bine formată.

### Particularizări ale inducției bine formate

1. În cazul în care mulțimea  $X$  este mulțimea  $N$  a numerelor naturale, iar  $\prec_f$  devine  $<$ , inducția bine formată este particularizată la *inducția completă*, cu schema:

$$\frac{\forall n \in N \mid \frac{[\forall i \in N \mid i < n \bullet P(i)]}{P(n)} \text{ pas inducție}}{\forall i \in N \bullet P(i)}$$

2. Dacă  $X$  este  $N$ , iar relația  $\prec_f$  este particularizată la  $x \prec_f y \Leftrightarrow y = x+1$ , pasul de inducție de sparge în două părți: una pentru numere strict pozitive, care au predecesor, și cealaltă pentru 0.  $P(0)$  trebuie demonstrată independent, în lipsa oricărei ipoteze inductive, și constituie cazul de bază al inducției. Inducția bine formată se reduce la *inducția matematică*, reprezentată de schema:

$$\frac{\begin{array}{c} \text{caz de bază} \\ | \\ P(0) \end{array} \quad \frac{[P(i)]}{P(i+1)} \text{ pas inducție}}{\forall i \in N \bullet P(i)}$$

Similar particularizării  $x \prec_f y \Leftrightarrow y = x+1$ , relația  $\prec_f$  poate fi  $x \prec_f y \Leftrightarrow y = g(x)$ , cu  $g: N \rightarrow N$  strict crescătoare. Considerând impusă o valoare  $n_0$  ca limită inferioară a argumentului lui  $g$ , se formează șirul  $n_0 < n_1 < \dots < n_i < n_{i+1} \dots$ , unde  $n_{k+1} = g(n_k)$ ,  $k=0, 1, \dots$ , iar schema inducției bine formate devine:

$$\frac{\begin{array}{c} P(n_0) \\ \hline \end{array} \quad \frac{\begin{array}{c} [P(n_i)] \\ \hline P(g(n_i)) \end{array}}{\hline} \quad \forall n, k \in \mathbb{N} \mid n = g^k(n_0) \bullet P(n)$$

unde  $g^0(n_0) = n_0$ , iar  $g^k$ ,  $k > 0$ , reprezintă compunerea de  $k$  ori a funcției  $g$  cu ea înseși. Rezultatul inducției poate fi generalizat la  $\forall n \in \mathbb{N} \bullet P(n)$ . O particularizare a relației bine formate care a condus la schema de mai sus este  $x \prec_f y \Leftrightarrow x = g(y)$  unde  $g: \mathbb{N} \rightarrow \mathbb{N}$  este strict descrescătoare. Un asemenea caz este cel folosit în evaluarea complexității unui algoritm prin metoda substituției. Schema inducției devine:

$$\frac{\begin{array}{c} P(n_0) \\ \hline \end{array} \quad \frac{\begin{array}{c} [P(g(n_i))] \\ \hline P(n_i) \end{array}}{\hline} \quad \forall n \in \mathbb{N} \bullet P(n)$$

3. Pentru un tip de date  $t$  să definim măsura  $f(x)$ ,  $x \in t$ , ca adâncimea maximă a aparițiilor constructorilor tipului  $t$  în cadrul valorii simbolice<sup>3</sup>  $x$ . Relația  $\prec_f$  poate fi particularizată la  $x_i \prec_f x \Leftrightarrow x = \sigma(\dots, x_i, \dots)$ , cu  $x_i \in t$ , pentru  $\sigma$  un constructor nenul al lui  $t$ . Ca și la inducția matematică, pasul de inducție se despică în două părți: una pentru valori  $x$  care pot fi formate din valori  $x_i \prec_f x$ , iar cealaltă pentru valori  $x$  care corespund constructorilor nulari ai tipului (constante ale lui  $t$ ) sau constructorilor aplicați asupra valorilor externe tipului  $t$ . Aceste ultime două cazuri corespund cazurilor de bază ale inducției. Se obține schema *inducției structurale*.

### Exemple de utilizare a inducției

**Exemplul 1.** Fie *fact* algoritmul recursiv de calcul al  $n!$  și  $P(n)$  proprietatea care îi caracterizează corectitudinea<sup>4</sup>.

`fact(n) { return n = 1 ? 1 : n * fact(n-1) }`

$\forall n \in \mathbb{N} \mid n \geq 1 \bullet P(n)$ , unde  $P(n) =_{\text{def}} \text{fact}(n) = \prod_{k=1}^n k$

<sup>3</sup> Valoarea  $\sigma(x_1, \dots, x_n)$  poate fi privită ca un arbore cu rădăcina  $\sigma \in \Sigma_t$  (adâncime 0) și cu descendenții corespunzând valorilor  $x_k$ ,  $k=1, n$ . Nodurile interne ale arborelui conțin constructori nenulari, iar frunzele conțin constructori nulari (din  $\Sigma_0$ ) sau valori externe tipului  $t$ .

<sup>4</sup> O formulare echivalentă a proprietății de verificat este  $\forall n \in \mathbb{N} \bullet P_I(n) \Rightarrow P_O(n, r)$ , unde  $P_I(n) =_{\text{def}} n \geq 1$  și  $P_O(n, r) =_{\text{def}} \text{fact}(n) = r$ ,  $r = \prod_{k=1}^n k$ . Se văd clar aserțiunile de intrare și de ieșire ale algoritmului.

Algoritmul este corect dacă  $\text{fact}(n) = \prod_{k=1}^n k$ , pentru orice  $n \in \mathbb{N} \setminus \{0\}$ .

Demonstrația este prin inducție matematică după  $n$ , pentru  $n \geq 1$ .

**Caz de bază.** Trebuie verificată proprietatea  $P(1)$ , deci trebuie verificată egalitatea  $\text{fact}(1) = \prod_{k=1}^1 k$ . Conform algoritmului rezultă imediat  $\text{fact}(1) = 1 = \prod_{k=1}^1 k$ .

**Pas de inducție.** Ipoteza inductivă consideră adevărată proprietatea  $P(n)$ ,  $n \geq 1$ . Trebuie verificată proprietatea  $P(n+1)$ , adică  $\text{fact}(n+1) = \prod_{k=1}^{n+1} k$ .

Conform algoritmului avem  $\text{fact}(n+1) = (n+1) * \text{fact}(n)$ . Conform ipotezei

$$\text{inductive } \text{fact}(n) = \prod_{k=1}^n k. \text{ Rezultă } \text{fact}(n+1) = (n+1) \prod_{k=1}^n k = \prod_{k=1}^{n+1} k \quad \blacksquare$$

**Exemplul 2.** Pentru a observa legătura strânsă dintre inducția structurală și cea matematică, să rescriem funcția  $\text{fact}$  din exemplul 1, considerând drept constructori de bază ai valorilor din  $N_1 = \mathbb{N} \setminus \{0\}$  funcțiile  $\text{succ}(n) = n+1$  și 1.

- (1)  $\text{fact}(1) = 1$
- (2)  $\text{fact}(\text{succ}(n)) = \text{succ}(n) * \text{fact}(n)$

Funcția  $\text{fact}: N_1 \rightarrow N_1$  este privită ca operator al tipului de date  $N_1$  și putem demonstra proprietatea  $\forall n \in N_1 \bullet P(n)$  prin inducție structurală.

**Caz de bază:** pentru constructorul nular 1.

Proprietatea  $P(1)$  este  $\text{fact}(1) = \prod_{k=1}^1 k \rightarrow 1 = 1$  și este adevărată banal.

**Pas de inducție:** pentru constructorul de bază  $\text{succ}$ .

Fie  $n \in N_1$ , ales la întâmplare. Ca ipoteză inductivă, proprietatea  $P(n)$  este adevărată. Trebuie să verificăm dacă  $P(\text{succ}(n))$  este adevărată. Avem:

$$\begin{aligned} \text{fact}(\text{succ}(n)) & \rightarrow \text{succ}(n) * \text{fact}(n) \\ & \rightarrow \text{succ} \text{ și ipoteza inductivă } \rightarrow (n+1) \prod_{k=1}^n k = \prod_{k=1}^{n+1} k \rightarrow \text{succ} \rightarrow \prod_{k=1}^{\text{succ}(n)} k \end{aligned}$$

Deci  $P(\text{succ}(n))$  este adevărată.  $\blacksquare$

**Exemplul 3.** Fie tipul de date  $\tau_{\text{Tree}}$ , desemnând arborii binari cu chei aparținând unui tip  $\tau$  oarecare. Un arbore  $\tau_{\text{Tree}}$  poate fi construit cu operatorii:

```
empty:  $\rightarrow T \text{ Tree}$ 
node:  $(T \text{ Tree}) \times T \times (T \text{ Tree}) \rightarrow T \text{ Tree}$ 
```

Arborele vid este `empty`, iar `node( $t_1, k, t_2$ )` este un arbore a cărui rădăcină conține cheia  $k \in T$  și are ca succesori  $t_1 \in (T \text{ Tree})$  (succesor stânga) și  $t_2 \in (T \text{ Tree})$  (succesor dreapta).

Fie algoritmul<sup>5</sup> `mirror:  $T \text{ Tree} \rightarrow T \text{ Tree}$`  astfel încât `mirror( $t$ )` este reflectarea în oglindă a arborelui  $t$ .

```
mirror( $t$ ) {
  switch( $t$ ) {
    case empty: return empty;
    case node( $t_1, k, t_2$ ):
      return node(mirror( $t_2$ ),  $k$ , mirror( $t_1$ ))
  }
}
```

Să utilizăm inducția structurală pentru a verifica proprietatea  $\forall t \in T \text{ Tree} \bullet P(t)$ , unde<sup>6</sup>  $P(t) =_{\text{def}} \text{mirror}(\text{mirror}(t)) = t$ .

**Caz de bază.** Trebuie verificată proprietatea  $P(\text{empty})$ . Conform algoritmului `mirror` avem:

`mirror(mirror(empty)) = mirror(empty) = empty.`

**Pas de inducție.** Fie  $t = \text{node}(t_1, k, t_2)$ ,  $t_1 \in T \text{ Tree}$ ,  $t_2 \in T \text{ Tree}$  și  $k \in T$  fiind valori alese la întâmplare. Considerăm că proprietățile  $P(t_1)$  și  $P(t_2)$  sunt adevărate (ipoteza inductivă). Trebuie arătat că proprietatea  $P(t)$  este adevărată. Conform algoritmului `mirror` avem:

```
mirror(mirror( $t$ )) =
  mirror(mirror(node( $t_1, k, t_2$ ))) =
  mirror(node(mirror( $t_2$ ),  $k$ , mirror( $t_1$ ))) =
  node(mirror(mirror( $t_1$ )),  $k$ , mirror(mirror( $t_2$ )))
```

Conform ipotezei inductive: `mirror(mirror( $t_1$ )) =  $t_1$`   
`mirror(mirror( $t_2$ )) =  $t_2$`

Rezultă:

```
mirror(mirror( $t$ )) =
  node(mirror(mirror( $t_1$ )),  $k$ , mirror(mirror( $t_2$ ))) =
  node( $t_1, k, t_2$ ) =  $t$  ■
```

<sup>5</sup> Există limbaje de programare în care algoritmul `mirror` este codificat așa cum este sugerat aici: folosind un `switch` în raport cu structura valorii selector.

<sup>6</sup> Aserțiunea de intrare este  $P_I(t) =_{\text{def}} 1$  (întotdeauna adevărată). Proprietatea de demonstrat se reduce la aserțiunea de ieșire  $P_O(t, r) =_{\text{def}} r = t$ , unde  $r = \text{mirror}(\text{mirror}(t))$ .

### 4.3 Verificarea deductivă a corectitudinii algoritmilor

Verificarea prin inducție este potrivită mai ales pentru algoritmi de construcție sau modificare a structurilor de date, deci atunci când structura este privită ca un șir de operații aplicate asupra unor date primare. Există însă situații în care este necesară probarea unei proprietăți intrinseci unei structuri de date, independent de modul de construcție al acesteia, sau unui proces de calcul. În astfel de cazuri o soluție alternativă constă în aplicarea deducției: direct sau prin reducere la absurd. În cazul deducției directe, proprietatea rezultă în mod natural din demonstrație. În cazul reducerii la absurd, se consideră că proprietatea  $P$  nu este adevărată. Altfel spus, se încearcă probarea negației  $\neg P$  urmărindu-se obținerea unei contradicții. În virtutea principiului terțiului exclus - propoziția  $P \vee \neg P$  este întotdeauna adevărată - rezultă că dacă  $\neg P$  este falsă atunci în mod necesar  $P$  este adevărată.

#### Demonstrație prin reducere la absurd

**Propoziție.** Dacă într-un graf  $G$  drumul  $\gamma = a \dots b \dots c$  are cost minim, iar funcția de cost  $w$  este aditivă, adică  $w(a \dots b \dots c) = w(a \dots b) + w(b \dots c)$ , atunci drumurile  $a \dots b$  și  $b \dots c$  au cost minim.

Demonstrație prin reducere la absurd. Să considerăm că cel puțin unul dintre drumurile  $\alpha_1 = a \dots b$  și  $\beta_1 = b \dots c$  din  $\gamma$  nu este de cost minim. Fie  $\alpha_2 = a \dots b$  și  $\beta_2 = b \dots c$  drumurile de cost minim, astfel încât  $w(\alpha_1) + w(\beta_1) > w(\alpha_2) + w(\beta_2)$ . Deci  $w(\gamma) > w(\alpha_2) + w(\beta_2)$  ceea ce contrazice ipoteza  $\gamma$  este drum de cost minim. ■

Deseori inducția este combinată cu demonstrația deductivă atât pentru verificarea corectitudinii, cât și în procesul de analiză a complexității algoritmilor. Exemplele din secțiunile următoare susțin această afirmație. Un caz aparte constă în utilizarea deducției în procesul de dezvoltare a algoritmilor. Exemplul de mai jos arată cum poate fi construit deductiv un algoritm pentru calculul rădăcinii pătrate întregi a unui întreg pozitiv.

#### Dezvoltarea unui algoritm prin deducție

Fie  $n$  un număr natural. Definim rădăcina pătrată întreagă a lui  $n$  ca fiind numărul natural  $\text{isqr}(n)$  astfel încât:

$$(\text{isqr}(n))^2 \leq n < (\text{isqr}(n)+1)^2$$

Să construim un algoritm de calcul al lui  $\text{isqr}(n)$ . Considerăm  $n > 0$  și fie  $q = \lfloor n/4 \rfloor$ <sup>7</sup> câtul împărțirii lui  $n$  la 4. Notăm  $k = \text{isqr}(q)$ . Conform definiției lui  $\text{isqr}$  avem

$$k^2 \leq q < (k+1)^2 \quad (1)$$

<sup>7</sup> Notăția  $\lfloor x \rfloor$  desemnează cel mai mare întreg care nu este mai mare ca  $x$ .



Deoarece  $q$  și  $(k+1)^2$  sunt numere întregi, inegalitatea  $q < (k+1)^2$  poate fi rescrisă

$$q+1 \leq (k+1)^2 \text{ sau } 4q+4 \leq (2k+2)^2 \text{ sau } 4q+3 < (2k+2)^2$$

Pentru că  $4q \leq n \leq 4q+3$ , din inegalitatea (1) și din ultima inegalitate de mai sus obținem

$$(2k)^2 \leq n < (2k+2)^2 \quad (2)$$

Conform inegalității (2) și definiției lui  $\text{isqr}$ , există două valori posibile pentru  $\text{isqr}(n)$ :  $2k$  sau  $2k+1$ . Doar una este validă.

**Cazul 1.**  $n < (2k+1)^2$  Din (2) avem  $(2k)^2 \leq n < (2k+1)^2$  și din definiția lui  $\text{isqr}(n)$  rezultă

$$\text{isqr}(n) = 2k, \text{ adică } \text{isqr}(n) = 2 \text{ isqr}(\lfloor n/4 \rfloor)$$

**Cazul 2.**  $n \geq (2k+1)^2$  Din (2) avem  $(2k+1)^2 \leq n < (2k+1+1)^2$  și din definiția lui  $\text{isqr}(n)$  rezultă

$$\text{isqr}(n) = 2k+1, \text{ adică } \text{isqr}(n) = 2 \text{ isqr}(\lfloor n/4 \rfloor) + 1$$

Algoritmul pentru calculul  $\text{isqr}(n)$  derivă din cele două cazuri de mai sus, pentru  $n > 0$ , și din  $\text{isqr}(0)=0$ . Algoritmul codificat în C este:

```
int increase(int n, int k2) {
    return n < (k2+1)*(k2+1) ? k2 : k2+1;}

int isqr(int n) {
    return n == 0 ? 0 : increase(n, 2*isqr(n/4));}
```

Complexitatea algoritmului poate fi dedusă ușor. Dacă  $p$  este numărul de pași executați pentru a calcula  $\text{isqr}(n)$  pentru  $n>0$ , algoritmul se oprește atunci când parametrul lui  $\text{isqr}$  devine 0, deci cu un pas după ce egalitatea  $n/4^p=1$  este satisfăcută. Numărul de pași de calcul efectiv (fără a considera pasul pentru care  $n$  devine 0) este  $p = \log_4(n)$  pentru  $n > 0$ .

Deoarece numărul de operații pentru un pas este limitat, se poate considera că timpul necesar execuției unui pas este  $c_1 \leq T_{\text{pas}} \leq c_2$ , unde  $c_1$  și  $c_2$  sunt constante reale strict pozitive. Timpul  $T(n)$  necesar terminării calculului lui  $\text{isqr}(n)$  este:

$$T(n) = p T_{\text{pas}}$$

$$c_1 p \leq T(n) \leq c_2 p$$

$$c_1 \log_4(n) \leq T(n) \leq c_2 \log_4(n)$$

$$T(n) = \Theta(\log_4(n)) = \Theta(\lg(n)/2) = \Theta(\lg(n)), \text{ pentru } n > 0.$$

Modul în care complexitatea  $T(n)$  a fost dedusă este mai mult intuitiv, dar suficient de clar. A abordare formală pornește de la recurența  $T(n) = T(\lfloor n/4 \rfloor) + \Theta(1)$  a cărei rezolvare conduce la rezultatul de mai sus.

Complexitatea  $\Theta(\lg(n))$  este dedusă în funcție de valoarea lui  $n$  și este corectă dacă operațiile cu numere durează  $\Theta(1)$ , deci pentru valori relativ mici ale lui  $n$ . În cazul valorilor mari ale lui  $n$ , este plauzibil să considerăm că o operație cu numere durează  $\Theta(k)$ , unde  $k$  este lungimea codului binar al lui  $n$ . În acest caz, dimensiunea datelor calculului `isqr` este  $k$ , iar la fiecare pas din cei  $\lg_4(n) = \Theta(\lg(2^k))$  pași sunt executate calcule ce durează  $\Theta(k)$ . Complexitatea algoritmului devine  $\Theta(k^2)$ .