Corectitudine şi complexitate

Sunt prezintate notaţiile uzuale de măsurare a complexităţii algoritmilor şi modul în care acestea pot fi folosite în formule şi, 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 performanţă dinamică, de comportament, dintre cei mai importanţi sunt consumul de resurse, mai ales de timp şi spaţiu de memorie. Spunem că un algoritm este cu atât mai complex cu cât consumul său de timp, respectiv de spaţiu de memorie, este mai mare, iar analiza complexităţii are scopul expres de a determina ordinul de mărime şi variaţia acestui consum în funcţie de doi parametri esenţiali: (a) operaţiile critice efectuate de algoritm şi (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 mulţi algoritmi de sortare operaţia critică este compararea cheilor secvenţei sortate, indiferent de reprezentarea secvenţei ca vector, ca listă sau ca fişier. În cazul algoritmilor pentru grafuri, operaţia critică poate fi străbaterea arcelor sau vizitarea nodurilor. Aceste operaţii critice constituie baza analizei algoritmilor respectivi. În cele ce urmează, considerăm că operaţiile elementare ale unui algoritm au cost unitar. Implicit, considerăm că problemele analizate satisfac această convenţie.

Dimensiunea problemei este definită ca acel parametru sau acei parametri de care depinde frecvenţa execuţiei operaţiilor critice ale algoritmilor de rezolvare. Teoretic, dimensiunea unei probleme este măsurată ca lungime a şirului 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 secvenţei sortate). În cazul prelucrării grafurilor, dimensiunea este numărul de arce sau/şi de noduri ale grafului. În schimb, pentru problemele cu caracter numeric, unde valoarile datelor sunt modificate, trebuie avut în vedere numărul de biţi folosiţi pentru reprezentarea valorilor respective.

Pentru a compara, chiar şi calitativ, algoritmii din punctul de vedere al performanţelor sunt necesare notaţii ale complexităţii. Notaţiile sunt utile şi din considerentul simplificării procesului de analiză a complexităţii. În foarte multe cazuri, analiza exactă este dificilă şi îndeobşte inutilă. Nu este nevoie de o valoare exactă a cantităţii 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 operaţiilor critice efectuate este direct proporţional cu dimensiunea problemei rezolvate. Totodată, este necesar ca precizia măsurării performanţelor să fie cu atât mai mare cu cât dimensiunea problemei rezolvate este mai mare. Notaţiile de complexitate folosite în cele ce urmează reflectă valoarea asimptotică a performanţelor algoritmilor, fiind cu atât mai relevante cu cât dimensiunea problemelor creşte.

2 Notaţii de complexitate

Considerăm că funcţiile de complexitate, care calculează exact performanţa algoritmilor din punct de vedere cantitativ – al resurselor consumate –, sunt de variabilă întreagă cu valori reale pozitive: $f{:}N \to R_+$, unde N este mulţimea numerelor naturale, iar R_+ este mulţimea realilor pozitivi. Parametrul unei asemenea funcţii corespunde dimensiunii problemei, iar valoarea reprezintă consumul de resurse.

Definiţia 1. Fie $g{:}N \to R_+$ o funcţie reprezentativă de complexitate. Clasa $O(g(n))$ a algoritmilor conţine exclusiv algoritmi care sunt caracterizaţi de funcţii de complexitate ce fac parte din mulţimea:

$$O(g(n)) = \{f{:}N \to R_+ \mid \exists c \in R_+, n_0 \in N \mid c>0 \bullet (\forall n \in N \mid n \ge n_0 \bullet f(n) \le c\ g(n))\}$$

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 funcţiilor din $O(g(n))$.

Definiţia 2. Fie $g{:}N \to R_+$ o funcţie reprezentativă de complexitate. Clasa $\Omega(g(n))$ a algoritmilor conţine exclusiv algoritmi care sunt caracterizaţi de funcţii de complexitate ce fac parte din mulţimea:

$$\Omega(g(n)) = \{f{:}N \to R_+ \mid \exists c \in R_+, n_0 \in N \mid c>0 \bullet (\forall n \in N \mid n \ge n_0 \bullet f(n) \le c\ g(n))\}$$

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 funcţiilor din $O(g(n))$.

Definiţia 3. Fie $g{:}N \to R_+$ o funcţie reprezentativă de complexitate. Clasa $\Theta(g(n))$ a algoritmilor conţine exclusiv algoritmi care sunt caracterizaţi de funcţii de complexitate ce fac parte din mulţimea:

$$\Theta(g(n)) = \{f{:}N \to R_+ \mid \exists c_1, c_2 \in R_+, n_0 \in N \mid c_1>0 \wedge c_2>0 \bullet (\forall n \in N \mid n \ge n_0 \bullet c_1\ g(n) \le f(n) \le c_2\ g(n))\}$$

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

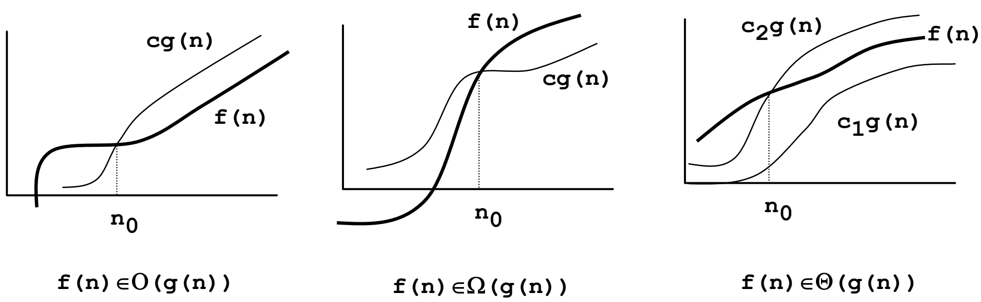

Figura 1 Interpretarea grafică a notaţiilor de complexitate

Interpretarea grafică a definiţiilor de mai sus este dată în figura 1. În ceea ce priveşte convenţia de notare, tradiţional se scrie $f(n) = X(g(n))$, cu X dintre O, Ω, Θ, în loc de $f(n) \in X(g(n))$.

Notaţiile O, Ω şi Θ sunt folosite pentru a aproxima asimptotic "strâns" complexitatea unui algoritm. Complexitatea estimată diferă faţă 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 faţă de cea exactă. De exemplu, pentru funcţia de complexitate $f(n)=n^2$, aproximarea asimptotică $f(n)=O(n^3)$ este largă. Pentru a diferenţia aproximările asimptotice "largi" faţă de cele "strânse" se folosesc notaţiile o şi ω.

Definiţia 1'. Fie $g:N\to R_+$ o funcţie reprezentativă de complexitate. Clasa $o(g(n))$ a algoritmilor conţine exclusiv algoritmi care sunt caracterizaţi de funcţii de complexitate ce fac parte din mulţimea:

$$o(g(n)) = \{f:N\to R_+ \mid \forall c\in R_+ \mid c>0 \bullet (\exists n_0\in N \bullet (\forall n\in N \mid n\geq n_0 \bullet f(n)\leq c\ g(n))\}$$

Definiţia 2'. Fie $g:N\to R_+$ o funcţie reprezentativă de complexitate. Clasa $\omega(g(n))$ a algoritmilor conţine exclusiv algoritmi care sunt caracterizaţi de funcţii de complexitate ce fac parte din mulţimea:

$$\omega(g(n)) = \{f:N\to R_+ \mid \forall c\in R_+ \mid c>0 \bullet (\exists n_0\in N \bullet (\forall n\in N \mid n\geq n_0 \bullet c\ g(n)\leq f(n))\}$$

Diferenţa dintre $o(g(n))$ şi $O(g(n))$, respectiv dintre $\omega(g(n))$ şi $\Omega(g(n))$, este vizibilă. Astfel, în cazul $O(g(n))$ şi $\Omega(g(n))$ constantele c şi n_0 sunt fixate, în timp ce pentru $o(g(n))$ şi $\omega(g(n))$ constanta c poate fi aleasă arbitrar. Bunăoară, în cazul aproximării $f(n)= o(n^3)$, cu $f(n)=n^2$, oricare ar fi constanta $c\in R_+$, $c>0$, există într-adevăr $n_0 \in \mathbf{N}$ astfel încât $0 \le n^2 \le c\ n^3$, pentru $n \ge n_0$. Constanta n_0 trebuie să satisfacă inegalitatea $n_0 \ge \frac{1}{c}$. Astfel, pentru $c \ge 1$ avem $n_0 \ge 1$. Pentru $c = \frac{1}{k}$, $k > 1$, avem $n_0 \ge k$.

Larghețea aproximărilor asimptotice o şi ω derivă şi din proprietățile următoare, induse de definițiile (1') şi (2'):

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

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

Conform $f(n)=o(g(n))$, să alegem la întâmplare $n_1 \in N$ şi fie $c_1 \in R_+ \setminus \{0\}$ constanta cea mai mică astfel încât $f(n) \le c_1\ g(n)$, pentru $n \ge n_1$. De asemenea, să alegem cea mai mare constantă posibilă $c_2 \in R_+ \setminus \{0\}$ şi cel mai mic număr $n_2 \in N$ astfel încât să fie satisfăcure restricțiile:

- $c_1 > c_2$
- $f(n) \le c_2\ g(n)$, pentru $n \ge n_2$

Să presupunem că $n_1 \ge n_2$. Atunci avem $f(n) \le c_2\ g(n)$, pentru $n \ge n_1$, cea ce contrazice alegerea constantei c_1. Rezultă $n_1 < n_2$. Inseamnă că putem alege şirul monoton descrescător de constante $c_1, c_2, \ldots, c_i, \ldots$ şi şirul monoton crescător $n_1, n_2, \ldots, n_i, \ldots$ astfel încât $f(n) \le c_i\ g(n))$, pentru $n > n_i$ şi $i \ge 1$. În consecință, raportul $\frac{f(n_i)}{g(n_i)} = c_i$ este monoton descrescător odată cu creşterea valorii n_i. Deoarece constantele c_i sunt pozitive, rezultă $\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$. Folosind proprietatea $f(n)=o(g(n)) \Leftrightarrow g(n)=\omega(f(n))$, rezultă $\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$.

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 $n^2 \ne o(n^2)$. Într-adevăr, dacă $n^2=o(n^2)$, atunci ar trebui ca pentru orice constantă $c \in \mathbf{R}_+$, $c>0$, să existe $n_0 \in N$ astfel încât $0 \le n^2 \le c\ n^2$, pentru $n \ge n_0$. Dar, pentru $c<1$ inegalitatea este imposibilă, deci n_0 nu există pentru o asemenea constantă.

Figura 1 lasă să transpară câteva proprietăți utile ale notațiilor O, Ω şi Θ. Astfel diagrama notației Θ sugerează proprietatea:

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

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

Propoziția 1 Notațiile O, Ω, Θ, o şi ω au următoarele proprietăți:

Tranzitivitatea:

$f(n) = O(h(n))$ şi $h(n) = O(g(n)) \Rightarrow f(n) = O(g(n))$

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

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

$f(n) = o(h(n))$ şi $h(n) = o(g(n)) \Rightarrow f(n) = o(g(n))$

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

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

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

Antisimetria:

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

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

Proiecția Θ: $f(n) = \Theta(g(n)) \Leftrightarrow f(n) = O(g(n))$ şi $f(n) = \Omega(g(n))$

Demonstrațiile sunt simple, fiind lăsate ca exercițiu.

Notațiile O şi Ω pot fi asimilate cu relații de ordine parțială peste mulțimea funcțiilor $N\to R_+$. O este oarecum similar relației $\le$, iar Ω este asemănătoare cu $\ge$. În schimb Θ joacă rolul unei relații de echivalență, aidoma relației $=$, peste mulțimea $N\to R_+$. Notația o este asemănătoare relației de ordine $<$, iar ω joacă rolul relației de ordine $>$. Este uşor de observat că relațiile definite de O, Ω, Θ, o şi ω sunt parțiale. Există funcții f şi g, cum ar fi $f(x) = 1$ şi $g(x) = 2 \sin(x)$, între care nu există nici o relație conformă cu vreuna din notațiile de complexitate. Faptul că O, Ω, Θ pot fi văzute ca relații de ordine sau de echivalență, fie şi parțiale, ajută compararea algoritmilor caracterizați de funcții din clasele respective de complexitate. De exemplu, putem arăta ca algoritmii 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 aleşi la întâmplare din cele două clase, unul caracterizat de o funcție de complexitate $f(n) = \Theta(n^2)$ şi 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 şi n_0 astfel încât $0 \le c_1(n^2+n) \le n^2 \le c_2(n^2+n)$, pentru $n \ge n_0$. Rezultă, $c_2 = 1$, $c_1 = 0.5$ şi $n_0 = 1$.

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

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

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

Proprietatea demonstrată poate fi generalizată pentru orice polinom în n: $\Theta(c_kn^k+ c_{k-1}n^{k-1}+...c_1n^1+c_0) = \Theta(n^k)$, cu $k \ge 0$ şi $c_k > 0$.

3 Utilizarea notațiilor asimptotice

Notațiile O, Ω şi Θ 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 $T(n)= 2\ T(n/2)+ n$ conține termeni $n/2$, întregi doar pentru $n = 2^k$. In 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 $T(n)= 2\ T(n/2)+ \Theta(n)$ 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 $\Theta\left(\frac{1}{kn}\right)$ şi $\Theta\left(\frac{1}{n}\right)$ 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.

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

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

$$0 \le a_1 \frac{1}{kn} \le f(n) \le a_2 \frac{1}{kn}, \text{ pentru } n \ge n_a \ge 1 \qquad (2)$$

$$0 \le b_1 \frac{1}{n} \le g(n) \le b_2 \frac{1}{n}, \text{ pentru } n \ge n_b \ge 1 \qquad (3)$$

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) şi (3) rezultă:

$$f(n) \le \frac{a_2}{kn} \text{ şi } g(n) \ge \frac{b_1}{n}, \text{ deci } F(n) \le \frac{1+\frac{a_2}{kn}}{1+\frac{b_1}{n}} \le 1 + \frac{a_2}{k}, \text{ pentru } n \ge n_0$$

$$f(n) \ge \frac{a_1}{kn} \text{ şi } g(n) \le \frac{b_2}{n}, \text{ deci } F(n) \ge \frac{1+\frac{a_1}{kn}}{1+\frac{b_2}{n}} \ge \frac{1}{1+b_2}, \text{ pentru } n \ge n_0$$

Deci $F(n) = \Theta(1)$. ■

Există multe proprietăţi utile care servesc simplificării ecuaţiilor în care apar măsuri de complexitate. Menţionăm doar câteva echivalenţe între clase de complexitate pentru notaţia Θ, dar care sunt valabile şi pentru Ω şi O. Probarea lor este simplă, fiind lăsată ca exerciţiu.

Propoziţia 3 Există următoarele proprietăţi.

$\Theta(k\ f(n)) = \Theta(f(n))$, pentru orice constantă pozitivă k;
$\Theta(f(n)) + \Theta(g(n)) = \Theta(f(n)+g(n))$;
$\Theta(f(n)) * \Theta(g(n)) = \Theta(f(n)*g(n))$;
$\Theta(c_k n^k + c_{k-1} n^{k-1} + \ldots + c_0) = \Theta(n^k)$, pentru $k \ge 0$ şi $c_k > 0$.

De exemplu, notând $\log_2$ cu $\lg$ şi ţinând cont că pentru o funcţie $f:N\to R_+$ monoton crescătoare există inegalitatea $\int_{m-1}^{n} f(x)dx \le \sum_{k=m}^{n} f(x) \le \int_{m}^{n+1} f(x)dx$, 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 notaţiile de complexitate este delicat şi trebuie să ţină seama de faptul că notaţiile nu reprezintă valori ci mulţimi de funcţii şi că, atunci când apar în formule, stau în locul unor funcţii anonime. Mai jos este ilustrat un exemplu edificator de raţionament greşit, preluat din [CLRS 01].

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

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

Pas de inducţie. 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 greşit în demonstraţie?

Eroarea este la pasul de inducţie, unde constanta implicată de notaţia $O(n)$ din ipoteza inductivă este alta decât cea impusă de $O(n+1)$ pentru rezultatul obţinut. Cu alte cuvinte, constanta nu este constantă! Într-adevăr,

$\sum_{k=1}^{n} k = O(n)$ implică $0 \le \sum_{k=1}^{n} k \le c\ n$, pentru $n \ge 1$

constantă diferită de c

$$0 \le \sum_{k=1}^{n+1} k = \sum_{k=1}^{n} k + (n+1) \le \underline{(c+1)}n + 1 \le \underline{(c+1)}(n+1)$$

Conform definiției notației O, pasul de inducție ar trebui să probeze că aceeaşi constantă ascunsă de $f(n)=O(n)$ este validă şi 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 şi rezultate de tip O este dezvoltat pentru a prelucra date $d \in I$ considerate corecte, adică date care satisfac un predicat $P_I: I \to \{0,1\}$ numit *aserțiune 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 $P_O: I \times O \to \{0,1\}$ numit *aserțiune de ieşire* a algoritmului. Cele două aserțiuni caracterizează problema rezolvată. De exemplu, pentru algoritmul de calcul al factorialului unui număr natural n,

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

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

4.1 Corectitudine parțială şi corectitudine totală

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

Definiția 4 Fie **Alg** un algoritm cu datele I şi rezultatele O, iar P_I şi P_O aserțiuni de intrare, respectiv de ieşire, definite în raport cu **Alg**.

- Algoritmul este *parțial corect* în raport cu aserțiunile P_I şi P_O dacă există proprietatea:

$$\forall d \in I \bullet (P_I(d) \wedge \text{se\_termină } Alg(d)) \Rightarrow P_O(d, Alg(d))$$

- Algoritmul este *total corect* în raport cu aserțiunile P_I şi P_O dacă există proprietatea:

$$\forall d \in I \bullet P_I(d) \Rightarrow (\text{se\_termină } Alg(d) \wedge P_O(d, Alg(d)))$$

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

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

cu `I = Z×Z` şi `O = Z`, unde `Z` este mulțimea întregilor, este doar parțial corect în raport cu aserțiunile: $P_I(x,y) =_{def} 1$ şi $P_O((x,y),r) =_{def} (r = x+y)$.

Apelul `plus(-1,2)` nu se termină deşi, cu ipoteza "evaluarea algoritmului `plus` se termină forțat după $k>0$ paşi şi, în momentul terminării, `plus(x-k,y) = x-k+y`", aserțiunea de ieşire P_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:

$$P_I(x,y) =_{def} (x \geq 0) \text{ şi } P_O((x,y),r) =_{def} (r = x+y)$$

Algoritmul se termină dacă se aplică asupra unor parametri care satisfac P_I, iar aserțiunea de ieşire este verificată de rezultat şi de datele algoritmului[1].

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

4.2 Verificarea inductivă a corectitudinii parțiale

Verificarea corectitudinii este activitatea cea mai dificilă şi cea mai puțin "standardizată" a procesului de proiectare a algoritmilor. Deşi 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 şi 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 priveşte corectitudinea cât şi 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 `t`. Algoritmul, sau părți ale algoritmului, acceptă valori de tip `t` care satisfac o proprietate `P` şi

[1] Demonstrația este prin inducție matematică după `x`.

produc rezultate de tip t care satisfac proprietatea[2] P. Prin urmare, algoritmul poate fi considerat un constructor al valorilor de tip t ce satisfac P. Este cazul în care verificarea corectitudinii algoritmului se poate efectua prin inducție structurală.

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

a) Orice constructor nular σ al lui t satisface P. Deci constantele din t satisfac P. Notăm Σ_0 mulțimea constructorilor nulari ai lui t. $\Sigma_0 =_{def} \{\sigma{:}\to t\}$

b) Pentru orice constructor $\sigma{:}Dom_\sigma \to t$, astfel încât valorile din Dom_σ nu conțin valori de tip t, valoarea $\sigma(v)$, $v \in Dom_\sigma$, satisface P. Cu alte cuvinte, P este satisfăcută de orice valoare a lui t construită exclusiv cu valori externe tipului. Numim Σ_e mulțimea constructorilor externi ai lui t. $\Sigma_e =_{def} \{\sigma{:}Dom_\sigma \to t \mid t \text{ nu apare în } Dom_\sigma\}$

c) Pentru orice constructor $\sigma{:}Dom_\sigma \to t$, astfel încât valorile din Dom_σ conțin valori de tip t, valoarea $y = \sigma(\ldots,x,\ldots)$ satisface proprietatea P, considerând că orice valoare $x \in t$ care apare în y satisface P. Deci P este satisfăcută de valorile simbolice ale lui t formate recursiv cu valori ale lui t, corecte în raport cu P. Numim Σ_t mulțimea constructorilor nenulari ai lui t. $\Sigma_t =_{def} \{\sigma{:}Dom_\sigma \to t \mid t \text{ apare în } Dom_\sigma\}$

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

cazuri de bază | ipoteză inductivă | pas inducție

$$\frac{\begin{array}{l}\forall\sigma\in\Sigma_0 \bullet P(\sigma)\\ \forall\sigma\in\Sigma_e, v\in Dom_\sigma \bullet P(\sigma(v))\end{array} \qquad \dfrac{[P(x)]}{P(\sigma(\ldots,x,\ldots))}\ \forall\sigma\in\Sigma_t, x\in t}{\forall x\in t \bullet P(x)}$$

Inducția structurală foloseşte dimensiunea valorilor simbolice ale tipului t. Dimensiunea unei valori $x \in t$ este văzută ca adâncimea maximă a aparițiilor constructorilor lui t în cadrul formulei ce reprezintă valoarea simbolică x. De exemplu, dimensiunea valorii `insert(1,insert(3,newset))` a unei mulțimi de întregi este 2. Dacă definim $dim{:}t \to N$, funcția care calculează dimensiunea valorilor simbolice ale lui t, atunci se observă că pasul de inducție din schema inducției structurale foloseşte în ipoteza inductivă valori $x \in t$ cu $dim(x) < dim(\sigma(\ldots,x,\ldots))$. Astfel, se poate

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

considera că inducția este efectuată în raport cu dimensiunea valorilor simbolice ale tipului `t`. Această observație 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 şir $\ldots \prec x_n \prec \ldots \prec x_2 \prec x_1$ 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 $f:X \to N$, putem interpreta f ca *funcție de măsurare* a elementelor din X şi putem defini relația bine formată $\prec_f$ peste X prin:

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

De exemplu, relația $\prec_{length}$, definită pe mulțimea listelor, este bine formată. La fel, pentru tipul mulțime finită, relația $\prec_{card}$ (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 şi cea structurală sunt particularizări ale inducției bine formate.

ipoteză inductivă

$$\frac{\forall x \in X \quad \dfrac{[\forall y \in X \mid y \prec_f x \bullet P(y)] \lrcorner}{P(x)} \quad \text{pas inducție}}{\forall x \in X \bullet P(x)}$$

Pentru un element $x \in X$ oarecare, pasul de inducție referitor la x porneşte 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 şi 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 şi totuşi există o valoare x_1 astfel încât $\neg P(x_1)$ atunci înseamnă că 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 există $x_2 \prec_f x_1$ şi $\neg P(x_2)$. Repetând raționamentul ajungem la concluzia că există un şir $\ldots \prec x_n \prec \ldots \prec x_2 \prec x_1$ 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 $\mathbb{N}$ a numerelor naturale, iar $\prec_f$ devine <, inducția bine formată este particularizată la *inducția completă*, cu schema:

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

2. Dacă x este $\mathbb{N}$, 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, şi cealaltă pentru 0. $P(0)$ trebuie demonstrată independent, în lipsa oricărei ipoteze inductive, şi constituie cazul de bază al inducției. Inducția bine formată se reduce la *inducția matematică*, reprezentată de schema:

$$\cfrac{\overset{\text{caz de bază}}{\underset{P(0)}{|}}\qquad \cfrac{[P(i)]}{P(i+1)}\ \text{pas inducție}}{\forall i\in \mathbb{N} \bullet P(i)}$$

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

$$\cfrac{P(n_0)\qquad \cfrac{[P(n_i)]}{P(g(n_i))}}{\forall n,k \in \mathbb{N} \mid n = g^k(n_0) \bullet P(n)}$$

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

$$\cfrac{P(n_0)\qquad \cfrac{[P(g(n_i))]}{P(n_i)}}{\forall n \in \mathbb{N} \bullet P(n)}$$

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[3] `x`. Relația $\prec_f$ poate fi particularizată la $x_i \prec_f x \Leftrightarrow x = \sigma(\ldots, x_i, \ldots)$, cu $x_i \in t$, pentru σ un constructor nenular al lui `t`. Ca şi 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!` şi `P(n)` proprietatea care îl caracterizează.

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

$$P(n) =_{def} fact(n) = \prod_{k=1}^{n} k$$

Algoritmul este corect dacă proprietatea $\forall n \in N \mid n \geq 1 \bullet P(n)$ este satisfăcută[4]. Mai precis, $fact(n) = \prod_{k=1}^{n} k$, pentru orice $n \in N \setminus \{0\}$. Demonstrația este prin inducție matematică după `n`.

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

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

Conform algoritmului avem $fact(n+1) = (n+1)*fact(n)$. Conform ipotezei inductive $fact(n) = \prod_{k=1}^{n} k$. Rezultă $fact(n+1) = (n+1)\prod_{k=1}^{n} k = \prod_{k=1}^{n+1} k$ ■

---

[3] Valoarea $\sigma(x_1, \ldots, x_n)$ poate fi privită ca un arbore cu rădăcina $\sigma \in \Sigma_t$ (adâncime `0`) şi cu descendenții corespunzând valorilor x_k, $k=1,n$. Nodurile interne ale arborelui conțin constructori nenulari, iar frunzele conțin constructori nulari (din Σ_0) sau valori externe tipului `t`.

[4] O formulare echivalentă a proprietății de verificat este $\forall n \in N \bullet P_I(n) \Rightarrow P_O(n,r)$, unde $P_I(n) =_{def} n \geq 1$ şi $P_O(n,r) =_{def} fact(n)=r$, $r = \prod_{k=1}^{n} k$. Se văd clar aserțiunile de intrare şi de ieşire ale algoritmului.

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

```
(1) fact(1)=1
(2) fact(succ(n)) = succ(n)*fact(n)
```

Funcția $\texttt{fact}:N_1 \to N_1$ este privită ca operator al tipului de date N_1 şi putem demonstra propriatea $\forall n \in N_1 \bullet P(n)$ prin inducție structurală.

Caz de bază: pentru constructorul nular `1`.

Proprietatea `P(1)` este $\texttt{fact(1)} = \prod_{k=1}^{1} k \xrightarrow{1} 1 = 1$ şi este adevărată banal.

Pas de inducție: pentru constructorul de bază `succ`.

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

$$\texttt{fact(succ(n))}$$

$$\xrightarrow{2} \texttt{succ(n) fact(n)}$$

$$\xrightarrow{\texttt{succ}\text{ şi ipoteza inductivă}} (n+1) \prod_{k=1}^{n} k = \prod_{k=1}^{n+1} k \xrightarrow{\texttt{succ}} \prod_{k=1}^{\texttt{succ(n)}} k$$

Deci `P(succ(n))` este adevărată. ■

Exemplul 3. Fie tipul de date `T Tree`, desemnând arborii binari cu chei aparținând unui tip `T` oarecare. Un arbore `T Tree` poate fi construit cu operatorii:

$$\texttt{empty}: \to \texttt{T Tree}$$
$$\texttt{node}: (\texttt{T Tree}) \times \texttt{T} \times (\texttt{T Tree}) \to \texttt{T Tree}$$

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

Fie algoritmul[5] $\texttt{mirror}:\texttt{T Tree} \to \texttt{T Tree}$ astfel încât `mirror(t)` este reflectarea în oglindă a arborelui `t`.

```
mirror(t) {
  switch(t) {
    case empty: return empty;
    case node(t1,k,t2):
              return node(mirror(t2),k,mirror(t1))
  }
}
```

[5] Există limbaje de programare în care algoritmul `mirror` este codificat aşa cum este sugerat aici: folosind un `switch` în raport cu structura valorii selector.

Să utilizăm inducția structurală pentru a verifica proprietatea $\forall$`t`$\in$`Tree` • `P(t)`, unde[6]

$$\texttt{P(t)} =_{def} \texttt{mirror(mirror(t)) = t}$$

Caz de bază. Trebuie verificată proprietatea `P(empty)`. Conform algoritmului `mirror` avem:

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

Pas de inducție. Fie `t = node(`t_1`,k,`t_2`)`, $t_1\in$`T Tree`, $t_2\in$`T Tree` şi `k`$\in$`T` fiind valori alese la întâmplare. Considerăm că proprietățile `P(`t_1`)` şi `P(t2)` sunt adevărate (ipoteza inductivă). Trebuie arătat că proprietatea `P(t)` este adevărată. Conform algoritmului `mirror` avem:

```
mirror(mirror(t)) =
     mirror(mirror(node(t1,k,t2))) =
     mirror(node(mirror(t2),k,mirror(t1)) =
     node(mirror(mirror(t1)),k,mirror(mirror(t2)))
```

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

Rezultă:

```
mirror(mirror(t)) =
   node(mirror(mirror(t1)),k,mirror(mirror(t2))) =
   node(t1,k,t2) = t ■
```

4.3 Verificarea deductivă a corectitudinii algoritmilor

Verificarea prin inducție este potrivită mai ales pentru algoritmii de construcție sau modificare a structurilor de date, deci atunci când structura este privită ca un şir 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 negatei $\neg$`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.

---

[6] Asertiunea de intrare este $\texttt{P}_\texttt{I}\texttt{(t)} =_{def} \texttt{1}$ (întotdeauna adevărată). Proprietatea de demonstrat se reduce la aserțiunea de ieşire $\texttt{P}_\texttt{O}\texttt{(t,r)} =_{def} \texttt{r = t}$, unde `r= mirror (mirror(t))`.

Demonstrație prin reducere la absurd

Teoremă. Dacă într-un graf `G` drumul γ = `a..b..c` are cost minim, iar funcția de cost `w` este aditivă, adică `w(a..b..c) = w(a..b)+w(b..c)`, atunci drumurile `a..b` şi `b..c` au cost minim.

Demonstrație prin reducere la absurd. Să considerăm că cel puțin unul dintre drumurile α_1 = `a..b` şi β_1 = `b..c` din γ nu este de cost minim. Fie α_2 = `a..b` şi β_2 = `b..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 γ este drum de cost minim. ■

Deseori inducția este combinată cu demonstrația deductivă atât pentru verificarea corectitudinii, cât şi 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 `isqr(n)` astfel încât:

$$(\mathrm{isqr}(n))^2 \le n < (\mathrm{isqr}(n)+1)^2$$

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

$$k^2 \le q < (k+1)^2 \qquad (1)$$

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

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

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

$$(2k)^2 \le n < (2k+2)^2 \qquad (2)$$

Conform inegalității (2) şi definiției lui `isqr`, există două valori posibile pentru `isqr(n)`: `2k` sau `2k+1`. Doar una este validă.

Cazul 1. $n < (2k+1)^2$ Din (2) avem $(2k)^2 \le n < (2k+1)^2$ şi din definiția lui `isqr(n)` rezultă

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

---

[7] Notația $\lfloor x \rfloor$ desemnează cel mai mare întreg care nu este mai mare ca `x`.

Cazul 2. $n \geq (2k+1)^2$ Din (2) avem $(2k+1)^2 \leq n < (2k+1+1)^2$ şi din definiţia lui `isqr(n)` rezultă

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

Algoritmul pentru calculul `isqr(n)` derivă din cele două cazuri de mai sus, pentru $n > 0$, şi din `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 uşor. Dacă p este numărul de paşi executaţi pentru a calcula `isqr(n)` pentru $n>0$, algoritmul se opreşte atunci când parametrul lui `isqr` devine 0, deci cu un pas după ce egalitatea $n/4^p=1$ este satisfăcută. Numărul de paşi de calcul efectiv (fără a considera pasul pentru care n devine 0) este $p = \log_4(n)$ pentru $n > 0$.

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

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

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

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

Modul în care complexitatea $T(n)$ a fost dedusă este mai mult intuitiv, dar suficient de clar. A abordare formală porneşte de la recurenţa

$$T(n) = T(\lfloor n/4 \rfloor) + \Theta(1)$$

a cărei rezolvare conduce la rezultatul de mai sus.

Complexitatea $\Theta(\lg(n))$ este dedusă în funcţie de valoarea lui n şi este corectă dacă operaţiile 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 operaţie 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 $\log_4(n)=\Theta(\lg(2^k))$ paşi sunt executate calcule ce durează $\Theta(k)$. Complexitatea algoritmului devine $\Theta(k^2)$.