

## ***De la problemă la algoritm***

Procesul dezvoltării unui algoritm, pornind de la specificația unei probleme, impune atât verificarea corectitudinii și analiza detaliată a complexității algoritmului, cât și analiza problemei din perspectivă computațională, în scopul determinării "durității" procesului de rezolvare. Etapele menționate, esențiale pentru garantarea calității și performanțelor soluției, constituie obiectivul întregii cărți și sunt prezentate succint în acest capitol introductiv.

### ***1. Aspecte ale construcției unui algoritm***

Rutina profesională ne îndeamnă, uneori, să considerăm algoritmi ca punct de plecare pe drumul rezolvării unei probleme și să subestimăm calea ce trebuie străbătută de la formularea problemei până la construcția unui algoritm de rezolvare corect și performant. În cazurile nebanale, drumul este dificil și impune utilizarea unei baze logistice impresionante din domenii cum ar fi calculabilitatea, complexitatea și, nu în ultimul rând, specificarea problemelor. Dificultatea transpare și dacă ne gândim că descrierea unei probleme este declarativă, în timp ce, în marea majoritate a cazurilor, algoritmul corespunzător are un caracter imperativ. Descrierea definește proprietățile soluției problemei, în timp ce algoritmul o construiește (o calculează). Etapele drumului de la problemă la algoritm și principalele elemente ce trebuie clarificate în cursul acestor etape sunt enumerate succint mai jos.

- a) Problema este rezolvabilă mecanic? Altfel spus, există o procedură efectivă care calculează soluția problemei?
- b) Există o formulare suficient de clară, completă și neambiguă a problemei? Eventual, în afara unei descrieri efectuate în limbaj natural, poate fi utilă o specificație formală folosind un limbaj de specificare sau de modelare.
- c) Problema acceptă algoritmi tractabili, anume algoritmi care pot fi executați în timp polinomial în raport cu volumul datelor? Altfel spus, din ce clasă de complexitate face parte problema?
- d) Algoritmul construit corespunde specificației problemei (este corect)? Care sunt diferențele în raport cu specificația, dacă există.
- e) Ce cantitate de resurse de calcul, în special spațiu de memorie și timp de calcul, este consumată de algoritm. Care algoritm este mai performant pentru problema dată, în eventualitatea existenței mai multor algoritmi de rezolvare?

În afara aspectelor menționate, drumul de la problemă la algoritm este dificil și datorită deciziei delicate a alegerii tipului de limbaj de programare adecvat rezolvării problemei și aplicației în care este integrată problema, alegere care influențează procesul de dezvoltare a algoritmului. Ultima etapă, anume cea a codificării algoritmului într-un limbaj de programare particular, pare simplă pe lângă celelalte etape premergătoare în drumul de la problemă la algoritm.

Exemplele următoare ilustrează câțiva din pașii transformării unei probleme, specificată formal, într-un algoritm. Ca preambul, detalierea acestor pași impune precizarea sensului termenilor: problemă, procedură efectivă, algoritm, rezolvare mecanică și complexitate.

## Probleme

Intuitiv, conform [More98], putem defini o problemă ca fiind raportul tensionat, deseori conflictual, dintre starea de fapt așa cum este percepută în domeniul studiat și starea dorită. Cu alte cuvinte, putem vedea o problemă ca fiind constituită dintr-o stare inițială și o stare finală, în care dorim să se ajungă, în domeniul studiat. Totuși, această definiție este prea largă din perspectiva rezolvabilității mecanice, folosind o mașină de calcul. Bunăoară, anunțul matrimonial

Tânăra 90/60/95, sensibilă, fără obligații, fără vicii, 23 ani, caut domn serios și prosper pentru relație stabilă. 0745.xxx.xxx dimineața.

constituie o problemă reală și interesantă, dar a cărei rezolvare este imprecisă și are un grad mare de subiectivism. Convenim să restricționăm problemele de interes, din perspectiva domeniului studiat în carte, conform definiției:

- Prin *problemă* înțelegem o mulțime de nedeterminări (întrebări) referitoare la proprietățile dinamice sau structurale ale unor entități – procese sau obiecte –, nedeterminări care acceptă variante de înlăturare (răspunsuri) precise, finite ca reprezentare și care pot fi riguros (matematic) justificate. Entitățile formează universul problemei, partea universului problemei cunoscută apriori reprezintă datele problemei, iar variantele de înlăturare a nedeterminărilor constituie soluțiile problemei.

Se observă că excludem "problemele" ale căror soluții sunt incerte, în sensul că soluțiile sunt credibile pe baza bunului simț, intuiției, unor convingeri sociale sau etice, etc., deși acestea sunt problemele cu care ne confruntăm adesea<sup>1</sup>. Mai mult, deliberat, pentru a evita speculațiile, limităm problemele investigate la mulțimea celor reducibile la următoarele tipuri de abstractizări echivalente:

1. calculul efectiv al unei funcții și
2. deciderea unei mulțimi.

În primul caz, problema este formulată ca o relație funcțională între date și soluții. A rezolva problema înseamnă a calcula efectiv funcția, unde termenul efectiv

---

<sup>1</sup> În schimb acceptăm probleme ale căror soluții sunt incerte în limitele unor probabilități riguros controlate sau probleme cu soluții aproximative în limite impuse.

subliniază folosirea unei mașini de calcul. Chestiunea "problema este rezolvabilă mecanic?" este transformată în "funcția care găsește soluția este efectiv calculabilă?". De exemplu problema simplă "fiind dat un număr natural  $x$  să se calculeze rădăcina pătrată întreagă" poate fi rezolvată folosind funcția  $isqr: \mathbb{N} \rightarrow \mathbb{N}$ , unde  $\mathbb{N}$  reprezintă mulțimea numerelor naturale (inclusiv 0). Intuitiv, funcția  $isqr$  poate fi calculată, deci problema este rezolvabilă. Dacă notăm cu  $\lfloor n/4 \rfloor$  câtul împărțirii  $n/4$ , un calcul posibil este:

$$isqr(n) = \begin{cases} 0, & \text{dacă } n=0 \\ 2 \cdot isqr(\lfloor n/4 \rfloor), & \text{dacă } n < (2\lfloor n/4 \rfloor + 1)^2 \\ 2 \cdot isqr(\lfloor n/4 \rfloor) + 1, & \text{dacă } n \geq (2\lfloor n/4 \rfloor + 1)^2 \end{cases}$$

În cel de al doilea caz, problema (nimită *de decizie*) este formulată ca proces de decizie a apartenenței unui element la o mulțime dată. Rezolvarea constă în a decide dacă datele problemei fac parte din mulțimea acelor date pentru care problema are soluție. Întrebarea "problema este rezolvabilă?" este transformată în "să se decidă dacă datele problemei fac parte din mulțimea de adevăr a unui predicat care specifică proprietățile soluțiilor problemei?". Este vorba de o funcție calculabilă al cărei codomeniu este mulțimea valorilor "adevărat" și "fals". De exemplu, fiind dat un număr natural  $x$  să se decidă dacă  $x$  aparține mulțimii numerelor prime:

$$Prime =_{\text{def}} \{x \in \mathbb{N} \mid x > 1 \bullet (\forall d \in 2..x-1 \bullet x \text{ modulo } d \neq 0)\}$$

## Calculabilitate

Ambele tipuri de probleme ilustrate mai sus se bazează pe posibilitatea de a *calcula efectiv* o funcție. În cazul (1) funcția este explicită, în cazul (2) deciderea apartenenței  $x \in M$  implică existența unei funcții, de exemplu  $test_M(x)$ , care reîntoarce 1 dacă  $x \in M$  și 0 altfel.

- O funcție  $f: \mathbb{I} \rightarrow \mathbb{O}$  este *efectiv calculabilă* dacă există o *procedură efectivă* capabilă să calculeze rezultatul funcției pentru fiecare valoare  $x$  din domeniul funcției.
- O *procedură efectivă* este o mulțime finită de operații, cu reprezentare finită, a căror execuție deterministă<sup>2</sup> implică o cantitate finită de date, cu reprezentare finită, și se termină în timp finit cu un rezultat finit ca reprezentare și mereu același pentru aceleași date.

O procedură efectivă nu corespunde unui anumit model teoretic al calculabilității și, deci, unui anumit tip de mașină de calcul. Conceptul este general, descriind intuitiv ce înseamnă a executa mecanic un proces de calcul, indiferent de natura mașinii folosite. În consecință, calculabilitatea efectivă a unei funcții se referă la existența unei mașini de calcul, oricare ar fi ea, și, implicit, a unui program care, prin execuție pe mașina aleasă, să calculeze valoarea funcției în orice punct din domeniul de definiție al acesteia.

<sup>2</sup> Execuția unei instrucțiuni este deterministă dacă exclude folosirea unor metode sau dispozitive aleatorii.

Există posibilitatea de a dezvolta un formalism astfel încât orice procedură efectivă intuitiv să poată fi executată conform formalismului și, reciproc, orice calcul reprezentabil în formalism să corespundă unei proceduri efective intuitive? Acest exercițiu a fost efectuat în 1936 de către matematicianul englez Alan Turing, mașina teoretică construită fiind astăzi acceptată ca model al procedurii efective sau, echivalent, ca model de bază al calculabilității. Mașina Turing nu este singurul model posibil al calculabilității. Alte modele sunt:

- Funcțiile recursive, definite de Kleene în 1936
- Calculul Lambda, dezvoltat de Church în 1936
- Algoritmii normali, dezvoltați de Markov în 1954

Modelele de mai sus sunt echivalente computațional mașinii Turing; orice calcul efectuat conform unui model poate fi efectuat în toate celelalte modele. De aici, derivă teza importantă a lui Church-Turing care susține, fără a demonstra, că orice funcție este efectiv calculabilă dacă este Turing calculabilă. Implicit, teza afirmă că funcțiile efectiv calculabile sunt exact cele parțial recursive și că orice procedură efectivă corespunde unei funcții parțial recursive.

Aspectele prezentate intuitiv sunt proprii teoriei calculabilității unde obiectivul central constă în investigarea rezolvabilității mecanice a problemelor indiferent de particularitățile mașinii de calcul folosite.

## Algoritmi

Specificarea unei proceduri efective, folosind o notație convențională, conform particularităților computaționale ale unui tip de mașină de calcul este un algoritm.

- Un *algoritm* acceptat de o mașină de calcul este o mulțime finită de instrucțiuni cu semnificație precisă, care pot fi executate de către mașina de calcul în timp finit pentru a calcula soluția unei probleme reprezentate în mod finit conform mașinii de calcul. Altfel spus, un algoritm este o particularizare a unei proceduri efective în raport un model dat al calculabilității.

Definiția algoritmului arată diferența esențială dintre o funcție și un algoritm, de fapt dintre un model funcțional al unei probleme și rezolvarea efectivă a problemei.

O funcție  $f: I \rightarrow O$  este o mulțime de perechi  $\{ (x, f(x)) \mid x \in \text{dom}(f) \wedge f(x) \in O \}$  care asociază un rezultat fiecărei valori valide (din domeniul funcției) a parametrului funcției, dar care nu arată în nici un fel modul în care rezultatul poate fi efectiv calculat. Un algoritm este un șir de simboluri, practic un text, care indică precis modul și, eventual, ordinea în care trebuie transformate datele de intrare pentru a obține rezultatul corespunzător. Cu alte cuvinte, putem construi algoritmi care să calculeze funcții. Dar, nu orice funcție este calculabilă, deci nu orice funcție acceptă algoritmi care să o calculeze. Prin urmare, nu este suficient ca o problemă să accepte un model funcțional pentru ca, automat, să fie rezolvabilă mecanic, deci să existe un algoritm de rezolvare. Este necesar să se demonstreze că un asemenea algoritm există.

Orice funcție calculabilă poate fi reprezentată indirect printr-un algoritm care o calculează. Astfel funcția  $\text{test\_prime}: \mathbb{N} \rightarrow \mathbb{N}$  de test al apartenenței unui număr  $x \in \mathbb{N}$  la mulțimea numerelor prime poate fi reprezentată prin următorul algoritm, destinat unei mașini de calcul imperative:

```
test_prime(x) {
    if(x ≤ 1) return 0;
    // x are divizori nebanali?
    for(d=2; d ≤ isqr(x); d++)
        if(x modulo d = 0) return 0; // x este divizibil prin d
    return 1; // x nu acceptă divizori diferiți de 1 și x
}
```

## Complexitate

Opus teoriei computabilității, teoria complexității algoritmilor se ocupă de problemele rezolvabile mecanic, chestiunea principală fiind legată de performanțele dinamice ale algoritmilor de rezolvare.

Performanțele statice, precum claritatea algoritmului evaluat, numărul de linii sau de operații folosite în textul algoritmului, nivelul de abstractizare al operațiilor, originalitatea, etc. sunt de mai mică importanță, cu atât mai mult cu cât unele dintre aceste caracteristici nu au o măsură clară.

Performanțele dinamice ale unui algoritm se referă la cantitatea de resurse de calcul (mai ales timp și spațiu de memorie) necesare execuției algoritmului și, teoretic, pot fi măsurate exact sau, cel puțin practic, pot fi approximate în limite acceptabile. Vom spune că un algoritm are o complexitate temporală (respectiv spațială) cu atât mai mare, cu cât timpul (respectiv spațiul de memorie) cerut de algoritm este mai mare.

Complexitatea este un criteriu esențial în selecția unui algoritm. Astfel, din perspectiva teoriei complexității, elaborarea unui algoritm aduce în prim plan chestiuni precum:

- Ce probleme pot fi rezolvate cu limite impuse de timp și spațiu de memorie și care sunt caracteristicile acestor probleme?
- Există limite ale resurselor de calcul pentru care o anumită problemă nu poate fi rezolvată?
- În ce limite de aproximare a soluției exacte poate fi rezolvată o problemă astfel încât resursele de calcul consumate să fie acceptabile?
- Există probleme sau algoritmi care cer, inherent, mai multe resurse de calcul?
- Care este complexitatea algoritmului dezvoltat pentru rezolvarea unei probleme? Ce fel de complexitate măsurăm: în cazul cel mai defavorabil sau în medie, pentru o secvență de operații care includ cazurile cele mai defavorabile (complexitate amortizată)?

Mai multe convenții stau la baza demersului teoretic implicat de întrebările de mai sus. Una dintre convenții se referă la metrica folosită pentru a măsura resursele de calcul consumate de un algoritm; alta precizează cum se măsoară complexitatea unui algoritm: exact sau aproximativ, folosind notații de complexitate care ajută la compararea algoritmilor chiar atunci când complexitatea lor exactă diferă. De asemenea, este important de precizat ce se înțelege prin complexitate acceptabilă sau, altfel spus, *calcul tractabil*.

Metrica măsurării resurselor de calcul folosite de un algoritm stabilește unitatea de măsură folosită pentru a evalua cantitatea resurselor consumate de operațiile elementare ale algoritmului și modul în care aceste valori sunt estimate și combinate pentru a calcula complexitatea de ansamblu a algoritmului. Metrica depinde în mare măsură de natura mașinii de calcul care execută algoritmul. De exemplu, pentru un algoritm destinat unei mașini imperative, gen C/C++ sau Java, se poate accepta modelul costului unitar pe operație, considerând că timpul sau spațiul consumat de fiecare operație elementară este o unitate convențională de timp sau spațiu, iar costul unui algoritm se calculează ca sumă a costurilor pașilor săi. Este important ca metrica aleasă să conducă la o complexitate cât mai apropiată de cea reală.

Complexitatea exactă a unui algoritm este caracterizată de o funcție (de complexitate)  $f: \mathbb{N} \rightarrow \mathbb{R}_+$ , unde  $\mathbb{N}$  este mulțimea numerelor naturale, iar  $\mathbb{R}_+$  este mulțimea realilor pozitivi. Valoarea  $f(n)$  desemnează cantitatea de resurse consumate de algoritm pentru *dimensiunea*  $n$  a datelor (a problemei rezolvate).

Dimensiunea unei probleme desemnează o măsură a datelor de intrare ale algoritmului și, intuitiv, influențează performanțele dinamice ale algoritmului. Teoretic, dimensiunea este definită ca lungime a șirului de simboluri ce reprezintă datele respective, simbolurile fiind considerate atomice din punctul de vedere al prelucrărilor efectuate de algoritm. Practic, în funcție de natura problemei, dimensiunea poate fi reprezentată indirect prin parametri specifici problemei, parametri de care depinde implicit lungimea șirului de simboluri ce reprezintă datele prelucrate.

De exemplu, dimensiunea  $\dim(G)$  a unei probleme de prelucrare a unui graf  $G=(V, E)$  este lungimea unui șir de simboluri care reprezintă nodurile  $v$  și arcele  $e$  ale grafului. Dacă graful are  $n$  noduri și  $m$  arce atunci lungimea șirului variază în raport cu  $n$  și  $m$ . În cazul în care simbolurile sunt considerate atomice, este puțin important cum sunt reprezentate. De exemplu, un nod poate fi reprezentat printr-un întreg ce identifică nodul, iar un arc poate fi reprezentat printr-o pereche de întregi ce corespund nodurilor de la capetele arcului. În acest caz, șirul care codifică graful are lungime  $n+2m$ , fiind format cu  $n$  simboluri (numere întregi) distincte. Aceste simboluri pot fi considerate atomice, deoarece algoritmul de prelucrare a grafului nu le modifică. Dimensiunea grafului variază liniar în raport cu  $n$  și  $m$ , iar complexitatea prelucrării se poate exprima în raport cu  $n$  și  $m$ .

Dacă problema este cea a verificării " $n$  este prim", unde  $n$  este un întreg pozitiv, atunci lungimea șirului de simboluri necesare reprezentării valorii  $n$  poate fi  $\dim(n) = \lfloor \lg(n) \rfloor + 1$ , dacă reprezentăm numărul în binar, sau  $\dim(n) = \lfloor \lg_{10}(n) \rfloor + 1$ , dacă numărul este reprezentat în zecimal. Cele două valori ale dimensiunii diferă doar

printr-o constantă și reprezintă numărul cifrelor binare sau zecimale necesare reprezentării numărului. Simbolul ce corespunde codului lui  $n$  nu este atomic, algoritmul aferent problemei lucrând cu părți ale simbolului. În schimb, cifrele reprezentării numărului pot fi considerate simboluri atomice. În acest caz, afirmația "dimensiunea problemei este  $n$ " devine discutabilă. Bunăoară, exceptând calculul  $\text{isqr}(n)$ , algoritmul  $\text{test\_prime}$  execută un număr de operații elementare proporțional cu valoarea  $\sqrt{n}$ . Dacă timpul necesar oricărei operații elementare cu numere pe  $k$  biți durează cel mult  $t(k)$ ,  $t: \mathbb{N} \rightarrow \mathbb{R}_+$ , atunci rezultă că timpul de calcul efectiv este proporțional cu  $t(k) \sqrt{2^k}$ . Pentru  $n$  relativ mic, putem accepta  $t(k) \leq c$ , cu  $c$  o constantă, iar timpul consumat de algoritm devine proporțional cu  $\sqrt{2^k}$ , adică cu  $\sqrt{n}$ , unde  $n$  desemnează valoarea numărului testat. În schimb, pentru  $n$  mare,  $\text{test\_prime}$  are complexitatea temporală exponențială  $t(k) \sqrt{2^k}$  în funcție de dimensiunea  $k$  a reprezentării lui  $n$ . Așa se explică de ce problema factorizării unui număr mare care nu este prim este dificilă și stă la baza unor algoritmi de criptare.

Cele două exemple de mai sus arată că dimensiunea unei probleme trebuie stabilită cu atenție atunci când algoritmul este numeric. Aici trebuie făcută diferența dintre valoarea datelor prelucrate și dimensiunea reprezentării lor. În schimb, pentru algoritmi care prelucrează structuri de date reprezentate prin simboluri atomice, numărul de elemente din structură poate desemna chiar dimensiunea problemei. În ceea ce privește cantitatea de resurse consumate, aceasta depinde de operațiile critice ale algoritmului, cele care costă cel mai mult sau sunt executate intensiv. Deseori, în analiza unui algoritm sunt contabilizate doar aceste operații critice, costul celorlalte fiind fie ignorat, fie absorbit în costul operațiilor critice. În carte, considerăm că toate operațiile elementare ale unui algoritm, deci și cele critice, au cost unitar. Implicit, considerăm că problemele analizate satisfac această convenție.

Determinarea funcției exacte de complexitate, notată  $f$ , este dificilă chiar și pentru probleme simple, așa cum sunt  $\text{isqr}$  și  $\text{test\_prime}$ . Soluția care permite evitarea acestor complicații, deseori inutile, constă în aproximarea funcției  $f$  printr-o altă funcție mai simplă, dar suficient de apropiată de complexitatea exactă pentru valori mari ale dimensiunii problemei. Într-adevăr, performanțele algoritmilor se degradează atunci când dimensiunea crește foarte mult. Așadar, este important ca aproximarea să fie suficient de bună mai ales în aceste cazuri, motiv pentru care complexitatea aproximată este numită *asimptotică*.

Aproximarea asimptotică se bazează pe notații de complexitate dintre care cele mai frecvent folosite sunt  $O$ ,  $\Theta$  și  $\Omega$ . Spunem că un algoritm are complexitatea  $O(g(n))$  dacă funcția exactă de complexitate este mărginită asimptotic superior de funcția  $g(n)$ , proporțional în raport cu o constantă, deci  $f(n) \leq c g(n)$  pentru valori mari ale lui  $n$ . Algoritmul are complexitatea  $\Omega(g(n))$  dacă funcția exactă de complexitate este mărginită asimptotic inferior de funcția  $g(n)$ , proporțional în raport cu o constantă, deci  $f(n) \geq c g(n)$  pentru valori mari ale lui  $n$ . Algoritmul are complexitatea  $\Theta(g(n))$  dacă funcția exactă de complexitate este mărginită asimptotic, inferior și superior, de funcția  $g(n)$ , proporțional în raport cu două constante, anume  $c g(n) \leq f(n) \leq c' g(n)$  pentru valori mari ale lui  $n$ . De exemplu, pentru numere

relativ mici, problema  $\text{isqr}(x)$  poate fi rezolvată cu complexitate temporală  $\Theta(\log_4(x))$ , deci timpul consumat de calculul funcției  $\text{isqr}(x)$  este proporțional cu  $\log_4(x)$ .

În privința complexității acceptabile, se consideră că un algoritm este *tractabil* dacă are complexitate polinomială  $O(n^k)$ , cu  $k$  o constantă. Altfel, algoritmul este catalogat drept *intractabil*. Se spune că o problemă rezolvabilă cu un algoritm tractabil este tractabilă, altfel problema este intractabilă. Convenția este discutabilă, deoarece pentru valori mari ale lui  $k$  complexitatea devine impracticabilă chiar și pentru mașinile de calcul foarte puternice.

Măsurarea complexității asimptotice permite compararea algoritmilor și, implicit, a problemelor. Astfel, un algoritm cu complexitate  $O(n)$  este mai "bun" decât unul cu complexitate  $O(n^2)$ , iar o problemă rezolvabilă în  $O(n)$  este mai puțin dificilă decât una rezolvabilă în  $O(n^2)$ . Totuși, din perspectiva tractabilității ne interesează o împărțire mai netă a problemelor în raport cu dificultatea relativă a rezolvării. Se obțin astfel clase de complexitate, fiecare clasă fiind populată de probleme care sunt rezolvabile prin algoritmi a căror complexitate asimptotică corespunde unor tipuri de funcții. Considerând  $n$  dimensiunea problemei, există clasele:

- **LOGSPACE**: probleme rezolvabile determinist în spațiu de memorie  $O(\log(n))$
- **NLOGSPACE**: probleme rezolvabile nedeterminist în spațiu  $O(\log(n))$
- **PTIME** (sau **P**) - probleme rezolvabile determinist în timp  $O(n^k)$
- **NPTIME** (sau **NP**): probleme rezolvabile nedeterminist în timp  $O(n^k)$
- **PSPACE**: probleme rezolvabile determinist în spațiu  $O(n^k)$
- **NPSPACE**: probleme rezolvabile nedeterminist în spațiu  $O(n^k)$

Rezolvarea deterministă impune determinarea exactă a instrucțiunii (sau instrucțiunilor) algoritmului executate la un anumit pas al rezolvării, în raport cu instrucțiunile de la pasul anterior. Rezolvarea nedeterministă implică ghicirea simultană a tuturor soluțiilor potențiale, urmată de verificarea deterministă, dar simultană, a eligibilității fiecărei soluții potențiale ca soluție a problemei.

După cum se va arăta, există ierarhia de clase de complexitate:

$$\text{LOGSPACE} \subseteq \text{NLOGSPACE} \subseteq \text{PTIME} \subseteq \text{NPTIME} \subseteq \text{PSPACE} = \text{NPSPACE}$$

Formula arată că o problemă dintr-o clasă inferioară în ierarhie este rezolvabilă cu complexitatea caracteristică oricărei superclase. Este interesant de menționat că ierarhia de mai sus este *robustă* pentru majoritatea tipurilor de mașini de calcul secvențiale de interes practic. Robustetea este înțeleasă ca invarianță a complexității asimptotice a rezolvării problemei în raport cu mașina de calcul folosită, în limitele unor factori de proporționalitate polinomiali și cu unele decizii realiste privind metrica de complexitate folosită. Astfel, similar tezei Church-Turing, se presupune că o problemă tractabilă cu o anumită mașină de calcul este tractabilă pentru orice altă mașină de calcul sau, echivalent, clasa **PTIME** este aceeași pentru orice model al calculabilității (*teza invarianței*, atribuită lui Stephen Cook, un pionier al teoriei complexității).



Cele mai dure probleme din clasa  $NP$  formează o clasă de echivalență numită clasa problemelor  $NP-complete$ , fiind intractabile determinist. Aceste probleme sunt reductibile reciproc, în sensul că pentru oricare pereche de probleme există un algoritm determinist polinomial care transformă una dintre probleme în cealaltă problemă. Din acest motiv, dacă o problemă din clasa  $NP$  are un algoritm tractabil de rezolvare, atunci  $P=NP$ . Chestiunea  $P=NP?$  este deschisă. Totuși, ierarhia de mai sus este utilă din considerente practice. Clasificarea teoretică a unei probleme evită căutarea unui algoritm cu performanțe a căror realizare este cel puțin incertă. De exemplu, dacă o problemă este în clasa problemelor  $NP-complete$ , atunci nu are rost căutarea unui algoritm tractabil de rezolvare care ar exista doar dacă  $P=NP$ . Problema trebuie rezolvată folosind algoritmi tractabili euristici, probabilistici sau de aproximare.

### Corectitudine

O etapă esențială a procesului de dezvoltare a unui algoritm este verificarea corectitudinii acestuia: rezultatele calculate corespund soluțiilor problemei rezolvate de algoritm? Conceptual verificarea corectitudinii este un obiectiv precis și pare că se poate efectua lesne având la îndemână o specificare a proprietăților soluției în raport cu proprietățile datelor problemei. Dar simplitatea este doar aparentă, corectitudinea implicând două aspecte delicate: terminarea algoritmului și validitatea logică a acestuia. Terminarea algoritmilor este o problemă nerezolvabilă mecanic, deci nu ne putem aștepta la o tehnică generală de demonstrare, fiecare caz trebuind tratat aparte. Verificarea validității logice constă în a cerceta dacă soluția calculată de algoritm îndeplinește proprietățile cerute. În acest caz, există tehnici generale de demonstrare, dar aplicarea lor este, de asemenea, dependentă de problemă.

## 2. Algoritmi și paradigme de programare

Procesul de dezvoltare al unui algoritm este influențat de paradigma de programare folosită la implementarea acestuia. Într-adevăr, există mașini de calcul abstracte caracteristice diverselor modele de calculabilitate și care susțin teoretic diverse paradigme de programare, cum ar fi programarea logică, funcțională, imperativă, asociativă. Deși aceste mașini sunt echivalente din punct de vedere computațional, deci pot rezolva aceeași clasă de probleme, totuși modul de a gândi rezolvarea unei probleme poartă amprenta caracteristicilor mașinii utilizate.

Problema rezolvată în cele ce urmează susține această observație și merită atenție mai ales din punctul de vedere al caracterului declarativ al rezolvării.

Rezolvarea este bazată pe afirmații, care descriu o stare de fapt relativă la universul problemei, și pe comenzi, care descriu acțiuni ce au ca suport astfel de afirmații. Afirmațiile și comenzile formează *baza de cunoștințe* necesară rezolvării problemei. Condițiile în care o afirmație poate fi generată sau o comandă poate fi executată sunt descrise prin reguli, așa cum facem de multe ori noi înșine.

Programul de calcul rezultat este asemănător unui compendiu de reguli care arată ce trebuie făcut în anumite situații. Aplicarea regulilor este secvențială. O regulă este aplicată imediat ce în baza de cunoștințe a problemei apar afirmații conforme condițiilor regulii. Numim *șabloane de identificare* aceste precondiții, iar procesul de testare a conformității unei afirmații în raport cu un șablon îl numim proces de identificare. Interpretarea unei reguli CLIPS este:

regulă CLIPS	interpretare
<pre>(defrule r   fapt<sub>1</sub> ...   fapt<sub>n</sub> =&gt;   acțiune<sub>1</sub> ...   acțiune<sub>m</sub>)</pre>	<pre>regula r   dacă există fapt<sub>1</sub> ...   și există fapt<sub>n</sub>   atunci     execută acțiune<sub>1</sub> ...     execută acțiune<sub>m</sub></pre>

Cunoștințele factice  $fapt_k$  și acțiunile  $acțiune_k$  pot fi parametrizate, adică pot conține variabile ce vor fi particularizate în funcție de cunoștințele factice existente în baza de cunoștințe a rezolvării problemei. De asemenea, o regulă se aplică o singură dată pentru un grup de afirmații care îi satisfac premisele.

### Problema subsecvenței de sumă maximă

Fie  $s$  o secvență de întregi (elemente din  $\mathbb{Z}$ ). Să se determine suma maximă a subsecvențelor de întregi conținute în  $s$ . Pentru a construi o specificație a problemei, numim  $\tau \text{ seq}$  mulțimea secvențelor cu elemente de tip  $\tau$  și convenim asupra următoarelor notații:

- $\langle \rangle$  este secvența vidă;
- $\langle s_1 \ s_2 \ \dots \ s_n \rangle$  desemnează o secvență nevidă cu elementele  $s_i, i=1, n$ ;
- $\#s$  desemnează lungimea secvenței  $s$ ;

De asemenea, fie operatorul  $\in_{\text{seq}}: \tau \text{ Seq} \times \tau \text{ Seq} \rightarrow \{0,1\}$ , astfel încât  $a \in_{\text{seq}} s$  reîntoarce 1 dacă  $a$  este o subsecvență în  $s$  și 0 altfel. Secvența  $a$  este subsecvență în  $s$  dacă și numai dacă  $a$  este vidă sau există un indice  $i > 0$  astfel încât numărul elementelor de indice  $j \geq i$  din  $s$  să fie cel puțin egal cu lungimea secvenței  $a$  și  $a_j = s_{i+j-1}$ , pentru  $j=1, 2, \dots, \#a$ . Spus formal:

$$a \in_{\text{seq}} s \Leftrightarrow a = \langle \rangle \vee (\exists i \in \mathbb{N}_1 \mid i + \#a - 1 \leq \#s \bullet (\forall j \in 1.. \#a \bullet a_j = s_{i+j-1}))$$

Specificația problemei sumei maxime a subsecvențelor unei secvențe  $s \in \tau \text{ Seq}$  este:

$$\text{smax}(s) =_{\text{def}} \max \{a \in \mathbb{Z} \text{ Seq} \mid a \in_{\text{seq}} s \bullet \text{sum}(a)\},$$

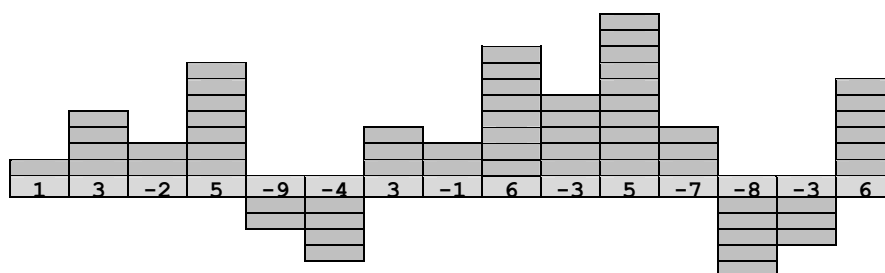
unde  $\text{sum}(a) = \sum_{i=1}^{\#a} a_i$ . Specificația arată că soluția problemei este maximul dintre sumele elementelor tuturor sub-secvențelor din  $s$ , inclusiv secvența vidă.

Funcția C corespunzătoare specificației  $\text{smax}(s)$  este ilustrată în figura 1(a). Corectitudinea este evidentă, din moment ce sunt parcurse toate subsecvențele posibile ale secvenței  $s$ , reprezentată ca un vector. Complexitatea rezultă  $\Theta(n^2)$ , unde  $n$  este lungimea secvenței<sup>3</sup>.

a) Complexitate $\Theta(n^2)$	b) Complexitate $\Theta(n)$
<pre> int summax(int s[], int n) {     int smax=0;     for(i=0; i&lt;n; i++) {         int sum=0;         for(j=i; j&lt;n; j++) {             sum+=s[j];             if(sum&gt;smax) smax=sum;         }         return smax;     } } </pre>	<pre> int summax(int s[], int n) {     int i,sum,smax;     if(n == 0) return 0;     smax = sum = s[0];     for(i=1; i&lt; n; i++) {         sum = sum &lt; 0? s[i]:sum+s[i];         if(sum &gt; smax) smax = sum;     }     return smax; } </pre>

**Figura 1.** Funcții C pentru calculul sumei maxime a subsecvențelor unei secvențe

O observație simplă conduce însă la o rezolvare în timp liniar. Doar o parte din subsecvențele din  $s$  contribuie la soluție. Într-adevăr, în momentul în care suma subsecvenței curent prelucrate devine negativă, prelucrarea subsecvenței poate fi abandonată pentru că suma ei nu va putea depăși suma oricărei alte subsecvențe ce începe cu un număr pozitiv. Figura 2 ilustrează acest fenomen. Dintr-un total de 120 de subsecvențe nevide, doar 5 trebuie prelucrate, anume:  $\langle 1 \ 3 \ -2 \ 5 \ -9 \rangle$ ,  $\langle -4 \rangle$ ,  $\langle 3 \ -1 \ 6 \ -3 \ 5 \ -7 \ -8 \ -3 \ 6 \rangle$ ,  $\langle -3 \rangle$  și  $\langle 6 \rangle$ .



**Figura 2.** Variația sumei maxime a subsecvențelor

Rezolvarea imperativă, în timp liniar, a problemei corespunde funcției C din figura 1(b) și are un avantaj suplimentar: este eficientă pentru secvențe foarte lungi. Deoarece fiecare număr din secvența  $s$  este parcurs o singură dată, nu este necesară memorarea întregii secvențe. Secvența poate fi citită incremental, doar numărul curent fiind păstrat pe durata adăugării lui la suma curent calculată. Deși, intuitiv, funcțiile - specificate imperativ - din figura 1 sunt ușor de înțeles, drumul de la specificația formală  $\text{smax}(s)$  la codul respectiv nu este de loc ușor. Evităm aceste

<sup>3</sup> Există și o variantă naivă cu complexitate  $\Theta(n^3)$ , care are un ciclu suplimentar pentru lungimea subsecvențelor parcurse.

complicații și ne concentrăm atenția asupra unei rezolvări care să fie cât mai apropiată de specificația problemei.

Rezolvarea declarativă de mai jos, în CLIPS, urmărește ad litteram specificația problemei, descriind de fapt ce se înțelege prin secvență de sumă maximă. . Notățiile și construcțiile CLIPS folosite în codificare sunt:

- `?nume` reprezintă o variabilă care poate lua ca valoare un simbol (în particular un număr), iar `$?nume` este o variabilă care poate avea ca valoare o secvență, eventual vidă, formată cu simboluri; `?` și `$?` sunt variabile anonime.
- `?nume&:predicat` restricționează legarea variabilei `?nume` la o valoare care satisface predicatul dat.
- `(assert fapt)` stabilește existența afirmației *fapt*.

Varianta declarativă are două particularități notabile: (a) în afara sumei maxime, sunt determinate, fără cod suplimentar, toate subsecvențele de sumă maximă; (b) o singură regulă esențială de rezolvare, anume `summax`, poate prelucra "simultan" mai multe secvențe pentru a le determina suma maximă.

```
(deffunction sum (?subsecventa)
  (bind ?s 0)
  (progn$ (?n ?subsecventa) (bind ?s (+ ?n ?s)))
  ?s)

(defrule summax
  (secventa ?id $? $?x $?)
  (not (summax ?id ?))
  (not (secventa ?id $? $?y&:(> (sum $?y) (sum $?x)) $?))
=> (assert (summax ?id (sum $?x)))
    (printout t ?id ": sum" $?x "=" (sum $?x) crlf))

(defrule date
=> (printout t "file: ")
    (load-facts (readline)))
```

O secvență este reprezentată de afirmația (`secvență nume numere`). Numele este necesar pentru a putea calcula "simultan", cu aceeași regulă, suma maximă a fiecărei secvențe date. Rezultatul este tipărit și înregistrat în baza de cunoștințe a programului sub forma afirmației (`summax nume valoare`). Un exemplu de rulare a programului este în tabelul 1

**Tabelul 1.** Suma maximă a elementelor subsecvențelor unei secvențe

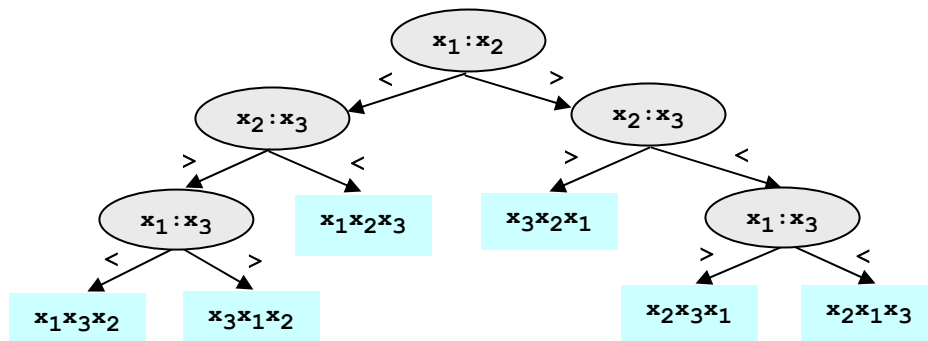
fișier <code>sumclp.dat</code>	rezultate
<pre>(secventa a 4 -1 8 -1) (secventa b 2 3 4 -1 2 -3 5) (secventa c 3 -5 -1 2 3 -6 1 15 -1) (secventa d -1 8 -1 8 -1) (secventa e)</pre>	<pre>CLIPS&gt; (reset) CLIPS&gt; (run) file: sumclp.dat e: sum()=0 d: sum(8 -1 8)=15 c: sum(1 15)=16 b: sum(2 3 4 -1 2 -3 5)=12 a: sum(4 -1 8)=11</pre>

Programul de mai sus este concis și ușor de construit. În contrapartidă, prețul plătit este complexitatea semnificativă a rezolvării. Ignorând timpul necesar identificării șabloanelor, efortul de calcul pentru determinarea  $\text{smax}(s)$  este  $\Theta(n^5)$  față de complexitatea  $\Theta(n)$  a variantei 1(b), scrise în C.

### 3. Probleme cu limită de complexitate

Uneori, procesul de dezvoltare a unui algoritm merită să fie dublat de analiza limitelor de complexitate ale rezolvării problemei, chiar și atunci când clasa de complexitate a problemei este cunoscută. Se poate evita astfel efortul inutil de căuta un algoritm cu o complexitate imposibil de atins. Un exemplu tipic îl constituie problema sortării unui vector prin comparare de chei. Problema este, după cum se știe, tractabilă, unul din cei mai performanți algoritmi de sortare fiind cel de sortare rapidă (quick-sort), cu complexitatea  $O(n \lg(n))$ , unde  $n$  este dimensiunea vectorului sortat. Se poate mai bine? Ce răspuns are întrebarea: există algoritmi de sortare prin comparare de chei cu complexitate inferioară limitei  $O(n \lg(n))$ ?

**Propoziția 1.** Complexitatea oricărui algoritm de sortare a vectorilor (în general a secvențelor), prin comparare de chei, este  $\Omega(n \lg(n))$ , unde  $n$  este numărul de elemente din vectorul sortat.



**Figura 3.** Arborele sortării vectorului  $x_1 \ x_2 \ x_3$

Fără a pierde generalitatea, acceptăm că fiecare element din vector este înlocuit prin cheia lui, iar cheile sunt numerice și distincte. Prin urmare, există atâtea chei distincte câte elemente sunt în vector, adică  $n$ . Astfel, reducem problema sortării unui vector cu  $n$  elemente la problema sortării unui vector cu  $n$  chei numerice distincte.

Să ne imaginăm un algoritm de sortare abstract care stabilește ordinea cheilor din vector în urma unor comparații. Procesul de sortare realizat de algoritm este descris de un arbore binar. Nodurile interne corespund comparațiilor cheilor, iar frunzele reprezintă soluții posibile ale sortării în raport cu poziția inițială a cheilor în

vector. Există  $n!$  permutări posibile ale cheilor din vector, deci sunt  $n!$  frunze în arbore. De exemplu, arborele din figura 3 descrie sortarea unui vector cu 3 chei, pentru orice configurație inițială a cheilor.

Algoritmul de sortare trebuie să fie rapid pentru orice configurație inițială a cheilor din vector. Cu alte cuvinte, algoritmul trebuie să se termine rapid pentru orice permutare finală din arborele virtual care descrie procesul sortării. Algoritmul este optim dacă procesul de sortare pe care îl realizează este cât mai aproape de forma unui arbore binar perfect echilibrat. Înălțimea arborelui echilibrat măsoară chiar numărul minim de comparații de chei efectuate de algoritm, deci indică limita inferioară a complexității algoritmului în raport cu dimensiunea  $n$  a vectorului sortat. Dacă notăm  $T(n)$  complexitatea algoritmului de sortare și cu  $h(n)$  înălțimea arborelui cu  $n!$  frunze, avem:

$$h(n) = \lg(n!) + 1 \geq \lg(n!)$$

$$T(n) \geq h(n) \geq \lg(n!), \text{ dar conform lui Stirling:}$$

$$n! = \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \left(1 + O\left(\frac{1}{n}\right)\right) \geq \left(\frac{n}{e}\right)^n$$

$$T(n) \geq \lg\left(\left(\frac{n}{e}\right)^n\right) = n \lg(n) - n \lg(e)$$

$$n \lg(n) - n \lg(e) \geq c n \lg(n), \text{ pentru } n \geq 3 \text{ și } c = 1 - \frac{\lg(e)}{\lg(3)}$$

$$T(n) = \Omega(n \lg(n)) \blacksquare$$

#### 4. Probleme nerezolvabile mecanic

Se poate argumenta că o problemă nerezolvabilă mecanic, folosind orice fel de mașină de calcul, este ambiguu formulată sau are date incomplete, incerte sau inconsistente, prin nerezolvabilitate înțelegându-se imposibilitatea determinării unei soluții credibile și acceptabile. Dar, chiar și intuiția ar putea să nu ajute în cazul unei asemenea probleme, iar arbitrariul joacă un rol preponderent, fiind discutabil ce se înțelege prin soluție credibilă și acceptabilă. Excludem o asemenea perspectivă.

Din punctul de vedere al teoriei calculului, prin nerezolvabilitate mecanică înțelegem imposibilitatea de a construi un algoritm, orice fel de algoritm, care să poată rezolva o problemă precis formulată și care, intuitiv, acceptă o soluție cu proprietăți cunoscute. Deci nerezolvabilitatea se referă la proprietăți intrinseci ale calculului necesar rezolvării problemei și nicidecum la proprietăți externe care ar putea face dintr-o problemă rezolvabilă una nerezolvabilă din simplul motiv că problema nu este bine înțeleasă, deci nu este o problemă în accepțiunea discutată în secțiunea 1.

Clasa problemelor ale căror soluții nu pot fi calculate mecanic acoperă multe aplicații cu valoare practică, în particular aplicații ce se reduc la probarea unor proprietăți ale programelor. Astfel, selecția unui modul de program dintr-o bibliotecă pe baza codului modului și a unei specificații referitoare la prelucrarea ce trebuie

îndeplinită de modul este o problemă din această categorie. Selecția impune decidera apartenenței modului la mulțimea programelor care satisfac specificația. Exemplul de mai jos este similar, dar mai simplu de formulat.

Să notăm  $P_{1,1}$  mulțimea programelor cu o singură intrare și cu o singură ieșire (rezultat). Programele din  $P_{1,1}$  calculează funcții  $N \rightarrow N$ , dar nu toate aceste funcții pot fi calculate efectiv. Printre cele necalculabile efectiv este și o funcția totală peste  $N$ :

$$BB(n) = \max \{P \in P_{1,1} \mid P(0) \text{ se termină} \wedge |P| \leq n \bullet P(0)\}$$

unde, prin convenție,  $\max \emptyset = 0$ , iar  $|P|$  reprezintă lungimea programului  $P$  (numărul de simboluri din textul programului). Calculul acestei funcții este cunoscut sub denumirea "problema vidrei ocupate" (busy-beaver).

Demonstrarea imposibilității calculului mecanic al funcției  $BB$  folosește câteva convenții de bun simț în ceea ce privește structura programelor din  $P_{1,1}$ . Astfel, admitem că un program este un șir de simboluri de lungime finită, iar mulțimea simbolurilor cu care pot fi construite programele este, de asemenea, finită. În acest caz, numărul programelor care au cel mult o lungime fixată este finit. Aceste convenții sunt respectate întocmai de limbajele de programare.

**Propoziția 2.** Funcția  $BB$  nu este calculabilă efectiv.

Demonstrația este prin contradicție. Să presupunem că există un program  $Q \in P_{1,1}$  care pentru orice  $n \in N$  calculează  $BB(n)$ . Să încercăm să construim un program  $P \in P_{1,1}$  de dimensiune  $|P| \leq k$ , pentru o constantă  $k$  fixată, astfel încât  $P(0) = Q(k) + 1$ . Dacă reușim, înseamnă că:

$$Q(k) = BB(k) \geq P(0) = Q(k) + 1$$

ceea ce este imposibil. Folosind o notație apropiată de limbajul C, să o numim  $L$ , construim programul  $P$  de mai jos. Programul are două module, unul ce conține codul lui  $Q$ , altul corespunzător funcției  $P(x)$ .

```
program P = cod_Q + P(x) {return Q(k)+1;}
```

De asemenea considerăm că orice cuvânt rezervat, precum `return`, orice cifră a unei constante numerice, orice nume de variabilă sau de funcție, precum și orice delimitator, ca `(,)`, `{,}` și `;`, este un simbol al programului. Atunci, lungimea programului  $P$  este:

$$\begin{aligned} |P| &= |Q| + \text{număr\_simboluri}[P(x) \{return Q(k)+1;\}] + \text{lungime\_cod}[k] \\ |P| &= |Q| + 13 + \lfloor \log_{10}(k) \rfloor + 1 \end{aligned}$$

Să observăm că programul cel mai scurt<sup>4</sup>, codificat în limbajul  $L$ , este de forma  $R(x) \{return x;\}$  cu lungimea  $|R|=9$ . Prin urmare, lungimea oricărui program

<sup>4</sup> Programul calculează o funcție din  $N \rightarrow N$  și, deci trebuie să aibă un parametru și trebuie să reîntoarcă un rezultat. Astfel, programul  $R() \{ \}$  nu este valid.

codificat în  $L$  este cel puțin 9. Atunci, lungimea programului  $p$  îndeplinește inegalitatea:

$$|p| \leq |Q| + 2|Q| + \lfloor \log_{10}(k) \rfloor \leq 3|Q| + \log_{10}(k)$$

Pentru că lungimea lui  $p$  trebuie să fie cel mult  $k$ , impunem restricția:

$$3|Q| + \log_{10}(k) \leq k$$

Pentru alegerea constantei  $k$  să considerăm  $k = m \cdot |Q|$  și să rezolvăm inegalitatea:

$$(m-1)|Q| + \log_{10}(m \cdot |Q|) \leq m \cdot |Q|$$

$$\log_{10}(m) \leq |Q| - \log_{10}|Q|$$

$$m \leq 10^{|Q| - \log_{10}|Q|}$$

$$m \leq 10^{9 - \log_{10}(9)}$$

Astfel, constanta  $k = 4|Q|$  satisface inegalitatea  $3|Q| + \log_{10}(k) \leq k$  pentru orice lungime  $|Q| \geq 9$ . Construcția programului  $p$  s-a realizat pe baza programului  $Q$ , care calculează funcția  $BB$ . Dar, prin construcția lui  $p$  se ajunge la contradicția  $Q(k) = Q(k) + 1$ . Așadar, programul  $Q$  nu poate exista. ■

Problema  $BB$  folosește indirect o altă proprietate esențială a programelor: terminarea. Într-adevăr, să observăm că dacă programul  $Q$  de mai sus ar fi capabil să testeze terminarea calculului  $p(0)$  pentru orice program  $p$  atunci, într-adevăr, calculul valorii  $BB(n)$  ar putea fi efectuat mecanic. Imposibilitatea calculului valorii  $BB(n)$  înseamnă, implicit, imposibilitatea deciderii terminării unui program. Avem, deci, următorul corolar.

**Corolar 1.** Problema deciderii mecanice a terminării unui program este nerezolvabilă mecanic (terminarea programelor este nedecidabilă)

Din punctul de vedere al posibilității de rezolvare, structurarea teritoriului problemelor lasă loc cazurilor extreme, când problema dată fie nu poate fi rezolvată mecanic (folosind un program), așa cum sunt exemplele de mai sus, sau nu se știe dacă problema are soluție, așa cum este cazul conjecturii lui Goldbach.

Conform conjecturii lui Goldbach, fiecare număr par  $n \geq 4$  este suma dintre două numere prime. Este ușor să se scrie un program  $p$ , fără intrări și cu singură ieșire, care caută să găsească un contraexemplu. Programul  $p$  caută să genereze un număr par  $n \geq 4$  care verifică propoziția:

$$\forall x, y \in \mathbb{N} \bullet \text{prim}(x) \wedge \text{prim}(y) \Rightarrow n \neq x + y$$

Dacă propoziția de mai sus este satisfăcută, programul  $p$  reîntoarce 1 altfel 0. Evident,  $p$  reîntoarce 1 doar dacă se termină. Deci, terminarea lui  $p$  este echivalentă cu infirmarea conjecturii. Dar conjectura a rămas nedemonstrată timp de mai bine de 250 de ani, astfel încât nu se știe dacă  $p$  se termină sau nu.