De la problemă la algoritm

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

Aspecte ale construcției unui algoritm

Rutina profesională ne îndeamnă, uneori, să considerăm algoritmii ca punct de plecare pe drumul rezolvării unei probleme şi să uităm calea ce trebuie străbătută de la formularea problemei până la construcția unui algoritm de rezolvare corect şi performant. În cazurile nebanale, drumul este dificil şi impune utilizarea unei baze logistice impresionante din domenii cum ar fi calculabilitatea, complexitatea şi, nu în ultimul rând, specificarea problemelor. Dificultatea transpare şi dacă ne gândim că descrierea unei probleme este declarativă, în timp ce, în marea majoritate a cazurilor, algoritmul corespunzător are un caracter imperativ. Descrierea defineşte proprietățile soluției problemei, în timp ce algoritmul o construieşte (o calculează). Etapele drumului de la problemă la algoritm şi 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 şi 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 şi 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 şi datorită deciziei delicate a alegerii tipului de limbaj de programare adecvat rezolvării problemei şi 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 paşii transformării unei probleme, specificată formal, într-un algoritm. Ca preambul, detalierea acestor paşi impune precizarea sensului termenilor: problemă, procedură efectivă, algoritm, rezolvare mecanică şi complexitate.

Probleme

- 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 şi a căror corectitudine poate fi riguros demonstrată. Entitățile formează universul problemei. Acea parte a universului problemei cunoscută apriori reprezintă datele problemei, iar variantele de înlăturare a nedeterminărilor sunt 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., deşi acestea sunt problemele cu care ne confruntăm adesea[1]. Mai mult, deliberat, pentru a evita speculațiile, limităm problemele investigate la mulțimea celor reductibile la următoarele tipuri de abstractizări:

1. calculul efectiv al unei funcții şi
2. decizia unei mulțimi.

În primul caz, problema este formulată ca o relație funcțională între date şi soluții. A rezolva problema înseamnă a calcula efectiv funcția, unde termenul efectiv subliniază folosirea unei maşini de calcul. Chestiunea "problema este rezolvabilă mecanic?" este transformată în "funcția este efectiv calculabilă?". De exemplu problema simplă "fiind dat un număr natural x să se calculeze rădăcina pătrată întreagă" poate fi rezolvată folosind funcția $isqr: \mathbf{N} \to \mathbf{N}$, unde $\mathbf{N}$ 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:

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

În cel de al doilea caz, problema este formulată ca proces de decizie a apartenenței unui element la o mulțime dată. A rezolva problema înseamnă a decide

[1] În schimb acceptăm probleme ale căror soluții sunt incerte în limitele unor probabilități riguros controlate sau probleme cu soluții aproximative în limite impuse.

dacă valoare cu rol de presupusă soluție este într-adevăr o soluție a problemei. Întrebarea "problema este rezolvabilă?" este transformată în "se poate decide dacă o valoare face parte din mulțimea soluțiilor problemei?". De exemplu, fiind dat un număr natural **x** să se testeze dacă **x** este prim sau, altfel spus, fiind dat un număr natural **x** să se decidă dacă **x** aparține mulțimii numerelor prime, mulțime definită:

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

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 $x \in M$ şi `0` altfel.

- O funcție $f: I \rightarrow o$ este *efectiv calculabilă* dacă există o *procedură efectivă* capabilă să calculeze rezultatul funcției pentru fiecare valoare $x \in I$.

- O *procedură efectivă* este o mulțime finită de instrucțiuni, finite ca dimensiune a reprezentării, a căror execuție se termină în timp finit şi implică o cantitate finită de date, cu reprezentare finită, pentru a obține un rezultat, mereu acelaşi pentru date identice.

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

Este interesant de investigat posibilitatea de a dezvolta un formalism astfel încât orice procedură efectivă intuitiv să poată fi executată conform formalismului şi, 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, maşina teoretică construită fiind astăzi acceptată ca model al procedurii efective sau, echivalent, ca model de bază al calculabilității. Dar maşina Turing (care calculează funcții parțial recursive) 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 maşinii 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 şi că orice procedură efectivă corespunde unei funcții parțial recursive.

Aspectele prezentate intuitiv mai sus sunt proprii teoriei calculabilității. Aici, obiectivul central constă în investigarea rezolvabilității mecanice a problemelor indiferent de particularitățile maşinii de calcul folosite, în particular maşini ce au resurse finite, dar oricât de mari de calcul. Câteva probleme elementare de calculabilitate sunt prezentate în capitolul 3.

Algoritmi

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

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

Definiția algoritmului arată diferența esențială ce există între o funcție şi un algoritm, de fapt între un model funcțional al unei probleme şi rezolvarea efectivă a acesteia.

O funcție matematică $f:I\rightarrow O$ este o mulțime de perechi $\{(x,f(x)) \mid x\in I \wedge f(x)\in O\}$ ce asociază un rezultat fiecărei valori a parametrului funcției, dar care nu arată în nici un fel modul în care rezultatul poate fi calculat.

Un algoritm este un şir de simboluri, practic un text, care indică precis modul şi ordinea în care trebuie transformate datele de intrare pentru a obține rezultatul corespunzător.

Astfel, putem avea algoritmi care primesc ca intrare reprezentarea unui număr şi produc drept rezultat reprezentarea altui număr, relația dintre cele două numere fiind descrisă de o funcție. 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 astfel de algoritm chiar există.

Orice funcție calculabilă poate fi reprezentată indirect printr-un algoritm care o calculează. Astfel funcția $test_{Prime}:\mathbb{N}\rightarrow\mathbb{N}$ de test al apartenenței unui număr $x\in\mathbb{N}$ la mulțimea numerelor prime poate fi reprezentată prin următorul algoritm, destinat unei maşini 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 şi 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 şi spațiu de memorie) necesare execuției algoritmului şi, teoretic, pot fi măsurate exact sau, cel puțin practic, pot fi aproximate 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 şi spațiu de memorie şi 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, inerent, 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 stabileşte cantitatea resurselor consumate de operațiile elementare ale algoritmului şi modul în care aceste cantități sunt combinate pentru a calcula complexitatea de ansamblu a algoritmului. Metrica depinde în mare măsură de natura maşinii de calcul care execută algoritmul. De exemplu, pentru un algoritm destinat unei maşini 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 de timp sau spațiu, iar costul unui algoritm se calculează ca sumă a costurilor paşilor 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) $f:N\rightarrow R_+$, unde N este mulțimea numerelor naturale, iar 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 compusă a acelor date care influențează major performanțele dinamice ale algoritmului. Dimensiunea este definită ca lungime a şirului de simboluri ce reprezintă datele respective, simbolurile fiind considerate atomice din punctul de vedere al prelucrărilor efectuate de algoritm.

De exemplu, dimensiunea $dim(G)$ a unei probleme de prelucrare a unui graf $G=(V,E)$ este lungimea unui şir de simboluri care reprezintă nodurile V şi arcele E ale grafului. Dacă graful are n noduri şi m arce atunci lungimea şirului este proporțională cu $n+m$. 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, şirul care codifică graful are lungime $n+2m$, fiind format cu n simboluri (numere întregi) distincte. Într-adevăr, aceste simboluri pot fi considerate atomice, deoarece algoritmul de prelucrare a grafului nu le modifică. Dimensiunea grafului este $dim(G) = n+2m$.

Dacă problema este cea a verificării "n este prim", unde n este un întreg pozitiv, atunci lungimea şirului de simboluri necesare reprezentării valorii n poate fi $dim(n)=\lfloor lg(n) \rfloor+1$, dacă reprezentăm numărul în binar, sau $dim(n)=\lfloor \log_{10}(n) \rfloor+1$, dacă numărul este reprezentat în zecimal. Cele două valori ale dimensiunii diferă doar printr-o constantă şi 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 $isqr(n)$, algoritmul $test_{Prime}$ 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 $t(k)$, $t:N\rightarrow R_+$, 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, $test_{Prime}$

are complexitatea temporală exponențială $t(k)\sqrt{2^k}$ în funcție de dimensiunea k a reprezentării lui n. Aşa se explică de ce problema factorizării unui număr mare care nu este prim este dificilă şi stă la baza unor algoritmi de criptare.

În carte, considerăm că operațiile elementare ale unui algoritm au cost unitar. Implicit, considerăm că problemele analizate satisfac această convenție.

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 şi dimensiunea reprezentării lor. În schimb, pentru algoritmii care prelucrează structuri de date reprezentate prin simboluri atomice, numărul de elemente din structură desemnează chiar dimensiunea problemei.

În ceea ce priveşte 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.

Determinarea funcției exacte de complexitate, notată f, este dificilă chiar şi pentru probleme simple, aşa cum sunt `isqr` şi $test_{Prime}$. 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 creşte foarte mult. Aşadar, 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, Θ şi Ω. 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 $f(n) \le c\ g(n)$ 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 $f(n) \ge c\ g(n)$ pentru valori mari ale lui n. Algoritmul are complexitatea $\Theta(g(n))$ dacă funcția exactă de complexitate este mărginită asimptotic, inferior şi superior, de funcția $g(n)$, proporțional în raport cu două constante, anume $c\ g(n) \le f(n) \le c'\ g(n)$ pentru valori mari ale lui n. De exemplu, pentru numere relativ mici, problema `isqr(x)` poate fi rezolvată cu complexitate temporală $\Theta(\log_4(x))$, deci timpul consumat de calculul funcției `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 şi pentru maşinile de calcul foarte puternice.

Măsurarea complexității asimptotice permite compararea algoritmilor şi, 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 $O(n^2)$. Totuşi, 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.

Există ierarhia de clase de complexitate:

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

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 maşini de calcul secvențiale de interes practic. Robustețea este înțeleasă ca invarianță a complexității asimptotice a rezolvării problemei în raport cu maşina de calcul folosită, în limitele unor factori de proporționalitate polinomiali şi cu unele decizii realiste privind metrica de complexitate folosită. Astfel, similar tezei Church-Turing, se presupune că o problemă tractabilă cu o anumită maşină de calcul este tractabilă pentru orice altă maşină de calcul sau, echivalent, clasa **PTIME** este aceeaşi 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.

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 şi 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 şi validitatea logică a acestuia. Terminarea algoritmilor este o problemă nerezolvabilă mecanic, deci nu ne putem aştepta la o tehnică generală de demonstrare, fiecare caz trebuind tratat aparte. Verificarea validității logice constă în a cerceta dacă soluția calculată de algoritm îndeplineşte proprietățile cerute. În acest caz, există tehnici generale de demonstrare, dar aplicarea lor este, de asemenea, dependentă de problemă.

Algoritmi şi paradigme de programare

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

Problema rezolvată în cele ce urmează susține această observație şi 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, şi pe comenzi, care descriu acțiuni ce au ca suport astfel de afirmații. Afirmațiile şi comenzile formează *baza de cunoştințe* necesară rezolvării problemei. Condițiile în care o afirmație poate fi generată sau o comandă poate fi executată sunt descrise prin reguli, aşa cum facem de multe ori noi înşine.

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 cunoştințe a problemei apar afirmații conforme precondițiilor regulii. Numim *şabloane de identificare* aceste precondiții, iar procesul de testare a conformității unei afirmații în raport cu un şablon îl numim proces de identificare. Interpretarea unei reguli CLIPS este:

regulă CLIPS	interpretare
(defrule r **fapt$_1$ ...**	**regula r** dacă există **fapt$_1$ ...**

faptₙ **=>** **acțiune₁ ...** **acțiuneₘ)**	și există **faptₙ** atunci execută **acțiune₁ ...** execută **acțiuneₘ**

Cunoştinţele faptice $\mathbf{fapt_k}$ şi acțiunile $\mathbf{acțiune_k}$ pot fi parametrizate, adică pot conține variabile ce vor fi particularizate în funcție de cunoştinţele faptice existente în baza de cunoştinţe a rezolvării problemei. De asemenea, o regulă se aplică o singură dată pentru un grup de afirmații care îi satisfac premisele.

Problemă

Fie **s** o secvență de întregi (elemente din **Z**). Să se determine suma maximă a subsecvențelor de întregi conținute în **s**. Pentru a construi o specificație a problemei, numim **T Seq** mulțimea secvențelor cu elemente de tip **T** şi convenim asupra următoarelor notații:

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

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

$$a \in_{seq} s \Leftrightarrow a = \langle\rangle \vee (\exists i \in N_1 \mid i+\#a-1 \le \#s \bullet (\forall j \in 1..\#a \bullet a_j = s_{i+j-1}))$$

Specificația problemei sumei maxime a subsecvențelor unei secvențe $s \in$ **T Seq** este:

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

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

Funcția C corespunzătoare specificației **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[2].

a) Complexitate $\Theta(n^2)$	b) Complexitate $\Theta(n)$
`int summax(int s[], int n) {`	`int summax(int s[], int n) {`

[2] Există şi o variantă naivă cu complexitate $\Theta(n^3)$, care are un ciclu suplimentar pentru lungimea subsecvețelor parcurse.

```
    int smax=0;
    for(i=0; i<n; i++) {
       int sum=0;
       for(j=i; j<n; j++) {
          sum+=s[j];
          if(sum>smax) smax=sum;
       }
    return smax;
}
```

```
    int i,sum,smax;
    if(n == 0) return 0;
    smax = sum = s[0];
    for(i=1; i< n; i++) {
       sum = sum < 0? s[i]:sum+s[i];
       if(sum > smax) smax = sum;
    }
    return smax;
}
```

Figura 1 Funcții C pentru **summax**

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 depăşi 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: ⟨**1 3 -2 5 -9**⟩, ⟨**-4**⟩, ⟨**3 -1 6 -3 5 -7 -8**⟩, ⟨**-3**⟩ şi ⟨**6**⟩.

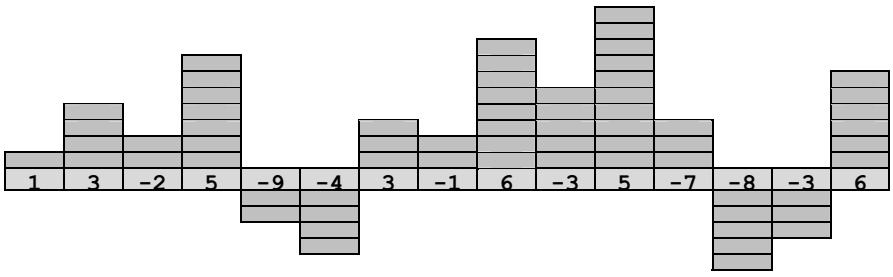

Figura 2 Variația sumei maxime a subsecvențelor

Rezolvarea imperativă, în timp liniar, a problemei corespunde funcției C din figura 1(b) şi 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.

Deşi, intuitiv, funcțiile - specificate imperativ - din figura 1(b) sunt uşor de înțeles, drumul de la specificația formală **smax(s)** la codul respectiv nu este de loc uşoară. Evităm aceste complicații şi ne concentrăm atenția asupra unei rezolvări care să fie cât mai apropiată de specificația problemei.

Rezolvarea declarativă de mai jos, în CLIPS, urmăreşte ad litteram specificația problemei, descriind de fapt ce se înțelege prin secvență de sumă maximă. 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 aceeaşi regulă, suma maximă a fiecărei secvențe date. Rezultatul este tipărit şi înregistrat în baza de cunoştințe 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

fişier `sumclp.dat`	rezultate

`(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)`
	`CLIPS> (reset)`
	`CLIPS> (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`

Programul de mai sus este concis şi uşor de construit. În contrapartidă, prețul plătit este complexitatea semnificativă a rezolvării. Ignorând timpul necesar identificării şabloanelor, complexitatea programului declarativ rezultă $\Theta(n^5)$ față de complexitatea $\Theta(n)$ a variantei scrise în C.