Capitolul 1

Introducere

1.1 Rezolvarea algoritmică a problemelor

În termeni generali un *algoritm* este o metodă de rezolvare pas cu pas a *problemelor*. O problemă este caracterizată de *datele de intrare* şi un enunţ care specifică relaţia existentă între acestea şi *soluţie*. În cadrul algoritmului sunt descrise prelucrările necesare pentru a obţine soluţia problemei pornind de la datele de intrare.

În rezolvarea algoritmică a problemelor ideea fundamentală este de a urma schema de rezolvare a unei probleme propusă de către Polya cu peste 50 de ani în urmă [**?**]:

- *Înţelegerea problemei.* Presupune a identifica elementele problemei (date de intrare, date de ieşire şi relaţiile dintre ele) precum şi a răspunde la întrebări de genul: sunt posibil de satisfăcut condiţiile specificate în enunţ? sunt informaţiile despre problemă suficiente, redundante sau contradictorii?

- *Stabilirea unei strategii.* Presupune identificarea unor probleme similare pentru care se cunoaşte o metodă de rezolvare. Dacă nu este evidentă similaritatea cu probleme cunoscute se poate încerca reformularea problemei şi eventual rezolvarea unei instanţe mai simple a acesteia.

- *Punerea în aplicare a strategiei.* Se aplică tehnica de rezolvare identificată verificând corectitudinea fiecărei etape parcurse.

- *Analiza rezultatelor.* Se verifică dacă rezultatele satisfac toate cerinţele problemei.

Particularizând aceste etape în cazul rezolvării algoritmice a problemelor rezultă următoarele etape:

1. Formularea clară, completă şi neambiguă a problemei, eventual prin utilizarea unor specificaţii formale.

2. Identificarea clasei din care face parte problema.

3. Identificarea unui algoritm care permite construcţia soluţiei pornind de la specificaţiile problemei.

4. Analiza corectitudinii algoritmului (are ca scop să verifice dacă algoritmul corespunde specificaţiilor problemei).

5. Analiza eficienţei algoritmului (are ca scop să verifice dacă soluţia poate fi obţinută prin utilizarea unui volum rezonabil de resurse).

6. Implementarea algoritmului şi execuţia acestuia.

Noţiunea de algoritm este foarte frecvent folosită în informatică însă nu există o definiţie unanim acceptată. În continuare considerăm că:

> *Un algoritm este o succesiune bine precizată de prelucrări care aplicate asupra datelor de intrare ale unei probleme permit obţinerea soluţiei acesteia după un număr finit de operaţii.*

Termenul de algoritm provine de la numele unui matematician persan, al-Khowarizmi (al-Kwarizmi), ce a trăit în secolul al IX-lea şi care a scris o lucrare despre efectuarea calculelor numerice într-o manieră algebrică. Primul algoritm se consideră a fi *algoritmul lui Euclid* (utilizat pentru determinarea celui mai mare divizor comun a două numere naturale).

Noţiunea de algoritm poate fi înţeleasă în sens larg nefiind neapărat legată de rezolvarea unei probleme cu caracter ştiinţific, ci doar pentru a descrie într-o manieră ordonată activităţi care constau în parcurgerea unei succesiuni de paşi (cum este de exemplu utilizarea unui telefon public sau a unui bancomat). În matematică există o serie de metode binecunoscute care posedă caracteristicile unui algoritm: metoda rezolvării ecuaţiei de gradul doi, *metoda lui Euclid* pentru calculul celui mai mare divizor comun a două numere naturale, *metoda lui Eratostene* (pentru generarea numerelor prime mai mici decât o anumită valoare), *schema lui Horner* (pentru evaluarea unui polinom dar şi pentru determinarea câtului şi restului împărţirii unui polinom la un binom) etc.

Soluţia problemei se obţine prin *execuţia* algoritmului. Algoritmul poate fi executat pe o maşină formală (în faza de proiectare şi analiză) sau pe o maşină fizică (calculator) după ce a fost codificat într-un *limbaj de programare*. Spre deosebire de un *program*, care depinde de un limbaj de programare, un algoritm poate fi interpretat ca o entitate matematică, descrisă folosind un limbaj specific, şi care este independentă de maşina pe care va fi executat.

Elaborarea unui algoritm necesită cunoştinţe specifice domeniului de unde provine problema de rezolvat (necesare pentru o mai bună înţelegere a problemei), cunoaşterea unor tehnici generale de rezolvare a problemelor (utile pentru

a identifica algoritmul adecvat unui problemei de rezolvat) intuiţie şi gândire algoritmică.

Indiferent de complexitatea unei aplicaţii informatice, la bazele ei stau algoritmi destinaţi rezolvării problemelor fundamentale ale aplicaţiei. Oricât de sofisticată ar fi tehnologia software utilizată eficienţa aplicaţiei este în mod esenţial determinată de eficienţa algoritmilor implicaţi. Un algoritm prost conceput nu poate fi "reparat" prin artificii de programare.

1.1.1 Proprietăţi ale algoritmilor

Un algoritm trebuie să posede următoarele proprietăţi:

Generalitate. Un algoritm destinat rezolvării unei probleme trebuie să permită obţinerea rezultatului pentru orice date de intrare nu numai pentru valori particulare ale acestora.

Finitudine. Un algoritm trebuie să admită o descriere finită şi fiecare dintre prelucrările pe care le conţine trebuie să poate fi executată în timp finit. Prin intermediul algoritmilor nu pot fi prelucrate structuri infinite.

Rigurozitate. Prelucrările algoritmului trebuie specificate riguros, fără ambiguităţi. În orice etapă a execuţiei algoritmului trebuie să se ştie exact care este următoarea etapă şi cum poate fi executată aceasta.

Eficienţă. Algoritmii pot fi efectiv utilizaţi doar dacă folosesc *resurse de calcul* în volum acceptabil. Resursele de calcul se referă la spaţiul necesar stocării datelor şi timpul necesar execuţiei prelucrărilor.

În continuare sunt analizate câteva exemple care ilustrează proprietăţile enumerate mai sus.

Exemplul 1.1 *Nu orice problemă poate fi rezolvată algoritmic.* Considerăm un număr natural n şi următoarele două probleme: (i) să se construiască mulţimea divizorilor lui n; (ii) să se construiască mulţimea multiplilor lui n. Pentru rezolvarea primei probleme se poate elabora uşor un algoritm, în schimb pentru a doua problemă nu se poate scrie un algoritm întrucât mulţimea soluţiilor este infinită (ceea ce face să nu se cunoască un criteriu de oprire a prelucrărilor).

Exemplul 1.2 *Un algoritm trebuie să funcţioneze pentru orice dată de intrare.* Să considerăm problema ordonării crescătoare a şirului de valori: $(2, 1, 4, 3, 5)$. O modalitate de ordonare, care pare naturală la prima vedere, ar fi următoarea: se compară primul element cu al doilea iar dacă nu se află în ordinea bună se interschimbă (în felul acesta se obţine şirul $(1, 2, 4, 3, 5)$); pentru şirul astfel transformat se compară al doilea element cu al treilea şi dacă nu se află în ordinea dorită se interschimbă (la această etapă şirul rămâne neschimbat); se continuă procedeul până când penultimul element se compară cu ultimul. În

felul acesta se obţine $(1, 2, 3, 4, 5)$. Deşi metoda descrisă mai sus a permis ordonarea crescătoare a şirului $(2, 1, 4, 3, 5)$ ea nu poate fi considerată un algoritm general de ordonare întrucât dacă este aplicată şirului $(3, 2, 1, 4, 5)$ conduce la $(2, 1, 3, 4, 5)$, ceea ce evident nu este un şir ordonat crescător .

Exemplul 1.3 *Un algoritm trebuie să se oprească după un număr finit de prelucrări.* Se consideră următoarea secvenţă de prelucrări:

Pas 1. Atribuie variabilei x valoarea 1;

Pas 2. Măreşte valoarea lui x cu 2;

Pas 3. Dacă x este egal cu 100 atunci se opreşte prelucrarea altfel se reia de la Pas 2.

Este uşor de observat că x nu va lua niciodată valoarea 100, deci succesiunea de prelucrări nu se termină niciodată. Din acest motiv nu poate fi considerată un algoritm corect. Dacă scopul urmărit era generarea tuturor valorilor impare mai mici decât 100 atunci condiţia de la Pas 3 ar fi trebuit să fie "x este mai mare decât 100".

Exemplul 1.4 *Prelucrările dintr-un algoritm trebuie să fie neambigue.* Considerăm următoarea secvenţă de prelucrări:

Pas 1. Atribuie variabilei x valoarea 0;

Pas 2. *Fie* se măreşte x cu 1 *fie* se micşorează x cu 1;

Pas 3. Dacă $x \in [-10, 10]$ se reia de la Pasul 2, altfel se opreşte algoritmul.

Atât timp cât nu se stabileşte un criteriu după care se decide dacă x se măreşte sau se micşorează, secvenţa de mai sus nu poate fi considerată un algoritm. Ambiguitatea poate fi evitată prin utilizarea unui limbaj mai riguros de descriere a algoritmilor. Să considerăm că Pas 2 se înlocuieşte cu:

Pas 2. Se aruncă o monedă. Dacă se obţine cap se măreşte x cu 1 iar dacă se obţine pajură se micşorează x cu 1;

În acest caz specificarea prelucrărilor nu mai este ambiguă chiar dacă la execuţii diferite se obţin rezultate diferite. Ce se poate spune despre finitudinea acestui algoritm? Dacă s-ar obţine alternativ cap respectiv pajură, prelucrarea ar putea fi infinită. Există însă şi posibilitatea să se obţină de 11 ori la rând cap (sau pajură), caz în care procesul s-ar opri după 11 repetări ale pasului 2. Atât timp cât şansa ca algoritmul să se termine este nenulă algoritmul poate fi considerat corect (în cazul prezentat mai sus este vorba despre un *algoritm aleator*). Totuşi în implementarea unor astfel de algoritmi se adaugă, de regulă, o condiţie suplimentară referitoare la numărul de repetări ale prelucrărilor.

Pas 1. Atribuie variabilei x valoarea 0; Iniţializează contorul cu 0.

Pas 2. Se aruncă o monedă. Dacă se obţine cap se măreşte x cu 1 iar dacă se obţine pajură se micşorează x cu 1; Incrementează contorul.

Pas 3. Dacă $x \in [-10, 10]$ şi contorul este mai mic decât numărul maxim de iteraţii se reia de la Pasul 2, altfel se opreşte algoritmul.

Exemplul 1.5 *Un algoritm trebuie să se oprească după un interval rezonabil de timp.* Să considerăm că rezolvarea unei probleme implică prelucrarea a n date şi că numărul de prelucrări $T(n)$ depinde de n. Presupunem că timpul de execuţie a unei prelucrări este 10^{-3}s şi că problema are dimensiunea $n = 100$. Dacă se foloseşte un algoritm caracterizat prin $T(n) = n$ atunci timpul de execuţie va fi $100 \times 10^{-3} = 10^{-1}$ secunde. Dacă, însă se foloseşte un algoritm caracterizat prin $T(n) = 2^n$ atunci timpul de execuţie va fi de circa 10^{27} secunde adică aproximativ 10^{19} ani.

1.1.2 Tipuri de date

Prelucrările specificate în cadrul unui algoritm se efectuează asupra unor *date*. Acestea sunt entităţi purtătoare de informaţie considerată a fi relevantă pentru problema de rezolvat. Putem interpreta datele ca fiind "containere" ce conţin informaţie, *valoarea* curentă a unei date fiind informaţia pe care o conţine la un moment dat. În funcţie de rolul jucat în cadrul algoritmului datele pot fi *constante* (valoarea lor rămâne nemodificată pe parcursul algoritmului) sau *variabile* (valoarea lor poate fi modificată pe parcursul execuţiei algoritmului).

Din punctul de vedere al informaţiei pe care o poartă datele pot fi:

- *Simple:* conţin o singură valoare (aceasta poate fi un număr, o valoare de adevăr sau un caracter).
- *Structurate:* sunt colecţii constituite din mai multe date simple între care poate exista o relaţie de structură. Dacă toate datele componente au aceeaşi natură atunci structura este *omogenă*, altfel este o structură *heterogenă*.

Datele structurate cu care se va opera corespund unor structuri algebrice cunoscute şi sunt descrise succint în continuare.

Mulţime. Reprezintă o colecţie de valori distincte pentru care nu are importanţă ordinea în care sunt specificate.

Multiset. Reprezintă o colecţie de valori nu neapărat distincte pentru care nu are importanţă ordinea în care sunt specificate. Singura diferenţă dintre multiset şi mulţime este faptul că într-un multiset pot fi prezente mai multe instanţe ale aceluiaşi element. De exemplu, mulţimea cifrelor numărului 21423712 este $\{1, 2, 3, 4, 7\}$ pe când multisetul corespunzător este $\{1, 1, 2, 2, 2, 3, 4, 7\}$.

Şir. Este o succesiune de elemente specificate într-o ordine prestabilită. De exemplu, şirul cifrelor numărului 21423712 specificat începând cu cifra cea mai puţin semnificativă este: $(2, 1, 7, 3, 2, 4, 1, 2)$. Pentru un şir dat se poate defini noţiunea de *secvenţă* precum şi cea de *subşir*. O secvenţă este o succesiune de elemente consecutive din şir pe când un subşir este o succesiune de elemente din şir care nu sunt neapărat consecutive. De exemplu, $(2, 1, 4, 2, 3)$ reprezintă secvenţa celor mai semnificative cinci cifre ale numărului de mai sus, iar $(2, 7, 2, 1)$ reprezintă subşirul cifrelor aflate pe poziţii impare începând cu poziţia cea mai semnificativă.

Matrice. Reprezintă o colecţie de elemente distribuite pe liniile şi coloanele unui tabel bidimensional. În varianta clasică toate liniile (coloanele) au acelaşi număr de elemente şi fiecare element al matricii poate fi identificat prin specificarea unui indice de linie şi a unui indice de coloană.

Graf. Este o structură care permite specificarea unei relaţii arbitrare între elementele unei mulţimi. Din punct de vedere matematic un graf se defineşte ca o pereche (V, E) unde V este o mulţime finită (numită mulţimea *nodurilor*) iar $E \subset V \times V$ este o mulţime de perechi de noduri. E este numită mulţimea *muchiilor* (dacă perechile nu sunt ordonate) sau a *arcelor* (dacă perechile sunt ordonate). Două noduri v şi v' sunt considerate *adiacente* dacă $(v, v') \in E$ iar o succesiune de noduri $v_1 v_2 \ldots v_k$ se numeşte *cale* sau *drum* în graf dacă orice două noduri succesive sunt adiacente. Un graf este considerat *conex* dacă între oricare două noduri există o cale şi neconex în caz contrar. Un *ciclu* într-un graf este o cale în care primul şi ultimul nod coincid îar un graf care nu conţine nici un ciclu este numit *aciclic*.

Arbore. Un arbore este un graf conex care nu conţine cicluri. Într-un arbore există o unică cale între oricare două noduri. Cea mai cunoscută modalitate de a vizualiza un arbore este cea în care nodurile sunt distribuite pe nivele: pe primul nivel se află un singur nod, numit *rădăcină*, pe al doilea nivel se află nodurile adiacente nodului rădacină, pe următorul nodurile adiacente acestora ş.a.m.d. În continuare se va face referire la arbori pentru a ilustra structura de apel în cazul algoritmilor recursivi sau pentru a vizualiza spaţiul stărilor unei probleme.

Există diferite variante de a reprezenta aceste structuri. In continuare, în marea majoritate a situaţiilor vom considera că ele sunt reprezentate prin *tablouri* de elemente. Un tablou este o modalitate de reprezentare a datelor caracterizată prin faptul că fiecare valoare componentă poate fi specificată prin precizarea unuia sau mai multor indici. Cel mai frecvent sunt folosite tablourile unidimensionale (pentru reprezentarea şirurilor şi mulţimilor) şi cele bidimensionale (pentru reprezentarea matricilor sau a relaţiilor binare cum sunt cele specifice grafurilor). Figura 1.1 ilustrează câteva modalităţi de descriere şi reprezentare a structurilor menţionate mai sus.

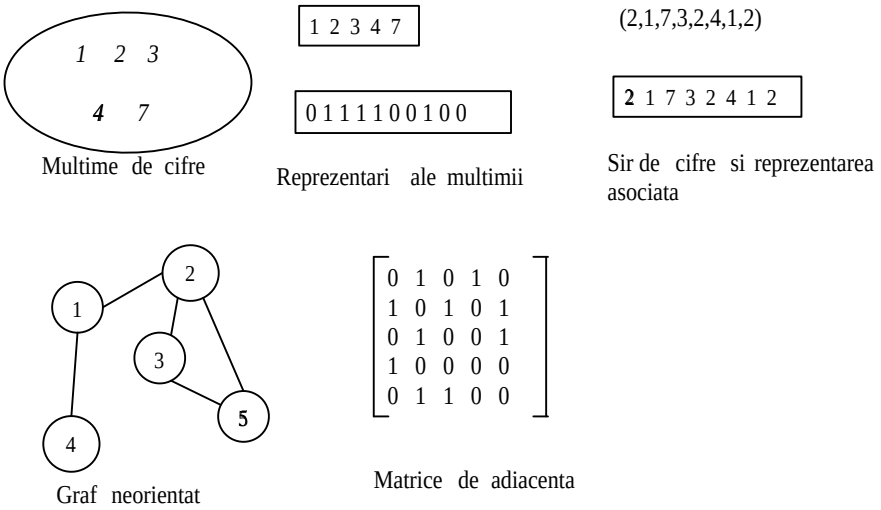

Figura 1.1: Modalităţi de reprezentare a datelor structurate

1.1.3 Tipuri de prelucrări

Asemenea datelor şi prelucrările ce intervin într-un algoritm pot fi clasificate în simple şi structurate. Prelucrările simple sunt:

Atribuire. Permite asignarea unei valori unei variabile. Valoarea atribuită poate fi rezultatul evaluării unei expresii. O *expresie* descrie un calcul efectuat asupra unor date şi conţine *operanzii* (datele asupra cărora se efectuează calculele) şi *operatori* (care permit specificarea operaţiilor ce se vor efectua asupra operanzilor).

Transfer. Permit preluarea datelor de intrare ale problemei şi furnizarea rezultatului/rezultatelor.

Control. În mod normal prelucrările din algoritm se efectuează în ordinea în care sunt specificate. În cazul în care se doreşte modificarea ordinii naturale se transferă controlul execuţiei la o anumită prelucrare. De exemplu, o prelucrare prin care se specifică trecerea la un anumit pas al algoritmului este o astfel de prelucrare de control.

Structurile principale de prelucrare sunt:

Secvenţială. Este o succesiune de prelucrări (simple sau structurate). Execuţia structurii secvenţiale constă în execuţia prelucrărilor componente în ordinea în care sunt specificate.

De decizie (condiţională). Permite specificarea situaţiilor în care în funcţie de realizarea sau nerealizarea unei *condiţii* se efectuează o prelucrare sau o altă prelucrare. Condiţia este de regulă o expresie a cărui rezultat este o *valoare logică* (adevărat sau fals). O astfel de prelucrare apare de exemplu în evaluarea unei funcţii definite prin:

$$f(x) = \begin{cases} -1 & \text{dacă } x < 0 \\ 0 & \text{dacă } x = 0 \\ 1 & \text{dacă } x > 0 \end{cases}$$

Dacă x este strict negativ prin evaluare se obţine -1, dacă este 0 se obţine 0 iar dacă este strict pozitiv se obţine 1.

De ciclare (repetitivă). Permite modelarea situaţiilor când o prelucrare trebuie repetată. Se caracterizează prin existenţa unei prelucrări care se repetă şi a unei *condiţii de oprire* (sau de *continuare*). În funcţie de momentul în care este analizată condiţia există prelucrări repetitive condiţionate *anterior* (condiţia este analizată înainte de a efectua prelucrarea) şi prelucrări condiţionate *posterior* (condiţia este analizată după efectuarea prelucrării). O astfel de prelucrare apare în calculul unei sume finite, de exemplu $\sum_{i=1}^{n} 1/i^2$. În acest caz prelucrarea care se repetă este adunarea iar condiţia de oprire o reprezintă faptul că au fost adunaţi toţi cei n termeni.

1.2 Descrierea algoritmilor

Metodele de rezolvare a problemelor sunt adesea descrise în limbaj matematic. Deşi riguros, acesta nu este întotdeauna adecvat pentru descrierea algoritmilor întrucât nu permite specificarea unor detalii importante in etapa codificării într-un limbaj de programare. În aceste condiţii, pentru descrierea algoritmilor se foloseşte un limbaj specific, numit limbaj algoritmic sau *pseudocod*. Pseudocodul este un limbaj artificial constituit dintr-un vocabular restrâns ce conţine *cuvinte cheie* asociate prelucrărilor. Ca orice limbaj se bazează pe un *alfabet*, un *vocabular* şi un set de *reguli de sintaxă*. Regulile de sintaxă sunt mai puţin stricte decât în cazul limbajelor de programare, fiind acceptate inclusiv expresii descrise în manieră matematică.

1.2.1 Elementele pseudocodului utilizat

Pseudocodul pe care îl vom folosi în continuare permite specificarea prelucrărilor simple şi structurate după cum urmează.

Atribuire. Pentru atribuirea valorii obţinute prin evaluarea unei expresii, variabilei cu numele v se specifică:

$$v \leftarrow \langle \text{expresie} \rangle$$

Expresiile se utilizează, de regulă, pentru a descrie calcule şi sunt constituite din *operanzi* şi *operatori*. Operanzii pot fi variabile şi valori constante. Operatorii utilizaţi sunt:

- *aritmetici:* $+$ (adunare), $-$ (scădere), $*$ (înmulţire), / (împărţire), ˆ (ridicare la putere), DIV sau / (câtul împărţirii întregi), MOD sau % (restul împărţirii întregi);
- *relaţionali:* $=$ (egal), $\neq$ sau $!=$ (diferit), $<$ (strict mai mic), $\leq$ sau $<=$ (mai mic sau egal), $>$ (strict mai mare), $\geq$ sau $>=$ (mai mare sau egal);
- *logici:* **or** (disjuncţie), **and** (conjuncţie), **not** (negaţie).

Citire. Pentru completarea variabilei cu numele v cu o valoare preluată de la dispozitivul de intrare se specifică:

read v

Scriere. Pentru transmiterea către dispozitivul de ieşire a valorii unei variabile sau a celei obţinute prin evaluarea unei expresii se specifică:

write ⟨expresie⟩

În specificarea oricărui algoritm este suficientă utilizarea următoarelor structuri de prelucrare: *secvenţială*, *condiţională* şi *repetitivă*.
Structura secvenţială. O succesiune de n prelucrări se specifică prin:

⟨prelucrare 1⟩

⟨prelucrare 2⟩

⋮

⟨prelucrare n⟩

Structuri condiţionale. Se utilizează pentru specificarea prelucrărilor în a căror execuţie se urmează o ramură sau alta în funcţie de satisfacerea sau nesatisfacerea unei condiţii. O structură condiţională cu două variante de prelucrare se descrie prin:

if⟨conditie⟩ **then** ⟨prelucrare 1⟩ **else** ⟨prelucrare 2⟩ **end if**

La execuţia acestei structuri, dacă prin evaluarea condiţiei se obţine valoarea *adevărat* atunci se execută prima prelucrare, altfel se efectuează cea de a doua. Cazul particular al unei singure variante se descrie prin:

$$\textbf{if } \langle \text{ conditie } \rangle \textbf{ then } \langle \text{prelucrare} \rangle \textbf{ end if}$$

În acest caz prelucrarea specificată se efectuează doar dacă condiţia este adevărată, altfel se trece direct la prelucrarea următoare din algoritm.

Structuri repetitive. Permit descrierea prelucrărilor ce trebuie efectuate în mod repetat. În funcţie de modul de plasare a condiţiei de continuare (sau de oprire) există două variante de structuri repetitive: condiţionată anterior şi condiţionată posterior. Varianta condiţionată anterior se specifică prin:

$$\textbf{while } \langle \text{conditie} \rangle \textbf{ do } \langle \text{prelucrare} \rangle \textbf{ end while}$$

La execuţie, prelucrarea se repetă atât timp cât condiţia este adevărată. Dacă condiţia este de la început falsă prelucrarea nu se efectuează nici o dată. Dacă în cadrul prelucrării nu se modifică componente ale condiţiei astfel încât aceasta să devină falsă atunci prelucrarea va fi repetată la nesfârşit.

Un caz particular de structură repetitivă condiţionată anterior este cel al *prelucrării repetitive cu contor* utilizată atunci când se cunoaşte de la început numărul de repetări ale prelucrării. Se foloseşte o variabilă cu rol de contor a cărei valoare variază între două limite: o valoare iniţială ($v1$) şi o valoare finală ($v2$) fiind modificată la fiecare ciclu cu o altă valoare (pas).

Descrierea

$$\textbf{for } v \leftarrow v1, v2, pas \textbf{ do } \langle \text{prelucrare} \rangle \textbf{ end for}$$

este echivalentă, în cazul în care valoarea pas este pozitivă, cu secvenţa:

$$\begin{array}{l} v \leftarrow v1 \\ \textbf{while } v \leq v2 \textbf{ do} \\ \quad \langle \text{prelucrare} \rangle \\ \quad v \leftarrow v + pas \\ \textbf{end while} \end{array}$$

În cazul în care valoarea pas este negativă se schimbă doar condiţia de la **while** (devine $v \geq v2$). În cazul în care $pas = 1$ acesta se poate omite din descriere. Varianta condiţionată posterior se specifică prin:

$$\textbf{repeat } \langle \text{prelucrare} \rangle \textbf{ until } \langle \text{condiţie} \rangle$$

La execuţie, prelucrarea se repetă până când condiţia devine adevărată. Prelucrarea se efectuează cel puţin o dată chiar dacă condiţia este de la început adevărată. Dacă în cadrul prelucrării nu se modifică componente ale condiţiei astfel încât aceasta să devină adevărată, atunci prelucrarea va fi repetată la nesfârşit. Indiferent de varianta de prelucrare repetitivă utilizată, la descrierea acesteia trebuie stabilite: (i) valorile iniţiale ale variabilelor ce intervin în prelucrare; (ii) prelucrarea/ prelucrările care se repetă; (iii) criteriul de oprire (sau de continuare).

Comentarii. În cadrul unui algoritm comentariile vor fi specificate prin:

// ⟨ text comentariu ⟩

1.2.2 Specificarea datelor

În cadrul algoritmilor uneori este util să se declare variabilele ce vor fi utilizate, specificând în acelaşi timp tipul lor. De exemplu, pentru specificarea datelor simple se pot folosi următoarele cuvinte cheie: **integer** (pentru valori întregi), **real** (pentru valori reale), **boolean** (pentru valori logice care pot lua doar valorile **true** - adevărat şi **false** - false) şi **char** (pentru variabile ce pot lua ca valori simboluri oarecare). Astfel, dacă a este o variabilă cu valori întregi, b una cu valori reale, c o variabilă de tip logic iar d una de tip caracter se va specifica:

integer a
real b
boolean c
char d

Structura de tip tablou se specifică indicând natura elementelor şi limitele între care variază indicii. De exemplu, în cazul unui tablou unidimensional cu elemente întregi se specifică **integer** $t1[n1..n2]$, iar în cazul unuia bidimensional cu elemente reale şi indici de linie variind între $m1$ şi $m2$, iar indici de coloană variind între $n1$ şi $n2$ se specifică **real** $t2[m1..m2, n1..n2]$. Tabloul $t1$ poate corespunde unui şir finit (sau unei mulţimi) cu $n2 - n1 + 1$ elemente, iar $t2$ unei matrici cu $m2 - m1 + 1$ linii şi $n2 - n1 + 1$ coloane. Elementele unui tablou se specifică prin intermediul indicilor lor. De exemplu elementul aflat pe poziţia i ($n1 \leq i \leq n2$) în $t1$ se specifică prin $t1[i]$, iar cel aflat pe linia i şi coloana j ($m1 \leq i \leq m2$, $n1 \leq j \leq n2$) în $t2$ se specifică prin $t2[i, j]$. Pot fi specificate şi subtablouri constând din elemente consecutive prin $t1[i1..i2]$ ($n1 \leq i1 < i2 \leq n2$). În cazul în care $n1 > n2$ se consideră că tabloul $t[n1..n2]$ este vid.

1.2.3 Tehnica rafinării succesive şi subalgoritmi

Cea mai simplă metodă de rezolvare algoritmică a problemelor este de a le descompune în subprobleme şi de a le rezolva pe fiecare dintre acestea. Rezolvarea fiecărui tip de subproblemă se va realiza în cadrul unui (sub)algoritm, iar algoritmul de rezolvare a problemei iniţiale va utiliza subalgoritmii pentru a construi rezultatul final. Dacă problema conţine mai multe subprobleme de aceeaşi natură atunci acestea pot fi rezolvate de acelaşi subalgoritm. În acest scop prelucrările din subalgoritm vor fi efectuate asupra unor *date generice* ce vor fi înlocuite cu datele concrete specifice problemei doar în momentul execuţiei. Transferul controlului execuţiei de la algoritm la un subalgoritm se

numeşte *apel*, datele generice sunt numite de regulă *parametri formali*, iar valorile lor concrete sunt numite *parametri efectivi*. Efectul unui subalgoritm constă fie în *returnarea* unor valori către algoritm fie în modificarea, prin intermediul parametrilor, a unora dintre variabilele algoritmului. In marea majoritate a situaţiilor datele de intrare sunt parametrii specificaţi alături de numele algoritmului, iar datele de ieşire sunt cele specificate după **return**.

Structura generală a unui subalgoritm este:

⟨nume subalgoritm⟩ (⟨ date generice ⟩)
    ⟨ date ajutătoare ⟩
    ⟨ prelucrări specifice ⟩
    return ⟨ rezultate ⟩ //returnarea rezultatelor

Uneori un subalgoritm poate avea, pe lângă rezultatele pe care le returnează, şi *efecte laterale*. Acestea constau de regulă în modificarea valorilor parametrilor sau a altor date aparţinând algoritmului general. În continuare, majoritatea algoritmilor vor fi descrişi sub forma unor subalgoritmi, iar pentru simplificare natura datelor generice şi a celor *locale* (datele ajutătoare utilizate de către subalgoritm) nu va fi specificată atunci când se deduce uşor din enunţul problemei.

1.2.4 Exemple

Exemplul 1.6 Se consideră trei valori reale reţinute în variabilele a, b şi c. (i) Să se determine cea mai mică dintre cele trei valori. (ii) Să se interschimbe valorile variabilelor astfel încât variabila a să conţină cea mai mică valoare, variabila b să conţină valoarea intermediară, iar variabila c să conţină cea mai mare valoare.

Descrierea algoritmilor. (i) O primă variantă de rezolvare bazată pe compararea valorilor două câte două este descrisă in algoritmul 1.1 (`minim1`).

O altă variantă, mai compactă, este descrisă în algoritmul `minim2`. În această variantă se iniţializează variabila ce conţine valoarea minimă cu valoarea variabilei a, după care se compară cu următoarele valori iar în momentul identificării unei valori mai mici valoarea minimă este actualizată. Această variantă poate fi uşor extinsă pentru cazul unui şir de valori.

(ii) Ideea, descrisă în algoritmul 1.2 (`ordonare`) constă în a aduce valoarea minimă în variabila a şi de a interschimba variabilele b şi c, dacă este cazul. Operaţia de interschimbare a valorilor a două variabile, specificată prin operatorul $\leftrightarrow$ constă în 3 operaţii de atribuire care presupun utilizarea unei variabile auxiliare cu rol de zonă intermediară de stocare. Interschimbarea $a \leftrightarrow b$ este echivalentă cu secvenţa:

1: $aux \leftarrow a$
2: $a \leftarrow b$

Algoritmul 1.1 Minimul a trei valori

```
// varianta 1
minim1(real a,real b,real c)
real min
if a < b then
  if a < c then
    min ← a
  else
    min ← c
  end if
else
  if b < c then
    min ← b
  else
    min ← c
  end if
end if
return min
```

```
// varianta 2
minim2(real a,real b,real c)
real min
min ← a
if b < min then
  min ← b
end if
if c < min then
  min ← c
end if
return min
```

3: $b \leftarrow aux$

Exemplul 1.7 Să se calculeze sumele: (a) $\sum_{i=1}^{n} i^2$, $n \in N^*$; (b) $\sum_{i=1}^{n} x^i/i!$ pentru $x \in (0,1)$ şi $n \in N^*$; (c) $\sum_{i=1}^{n(\epsilon)} x^i/i!$, unde $x \in (0,1)$, $\epsilon > 0$ iar $n(\epsilon) \in N^*$ este cea mai mică valoare naturală pentru care $x^{n(\epsilon)}/n(\epsilon)! < \epsilon$ (suma specificată poate fi considerată o aproximare cu precizia ϵ a sumei infinite $\sum_{i=1}^{\infty} x^i/i!$).

Descrierea algoritmilor. (a) Prelucrarea care se repetă este cea de adunare a unui termen. Procesul repetitiv este finalizat în momentul în care au fost adunaţi toţi termenii sumei. Prelucrarea poate fi descrisă utilizând oricare dintre cele trei variante de structură repetitivă. Descrierile din algoritmul 1.3 sunt echivalente.

(b) Suma poate fi rescrisă ca $\sum_{i=1}^{n} T(i)$ unde $T(i) = x^i/i!$. Problema poate fi rezolvată într-o manieră similară celei de la pct. (a), calculând pentru fiecare $i = \overline{1,n}$ valoarea termenului $T(i)$. Se observă însă că între termenii succesivi există o relaţie care permite calculul lui $T(i)$ pornind de la $T(i-1)$:

$$T(i) = \frac{x^i}{i!} = \frac{x^{i-1}}{(i-1)!} \cdot \frac{x}{i} = T(i-1) \cdot \frac{x}{i}$$

Cum $n \in N^*$ rezultă că suma conţine cel puţin un termen, astel că prelucrarea poate fi descrisă printr-o structură de tip **repeat** ca în algoritmul `suma4` (Algoritm 1.4).

Algoritmul 1.2 Ordonarea a trei valori

```
ordonare(real a,real b,real c)
if a > b then
   a ↔ b
end if
if a > c then
   a ↔ c
end if
if b > c then
   b ↔ c
end if
return a,b,c
```

Algoritmul 1.3 Calculul unei sume finite

`suma1`(**integer** i)	`suma2`(**integer** i)	`suma3`(**integer** i)
$S \leftarrow 0$	$S \leftarrow 0$	$S \leftarrow 0$
$i \leftarrow 1$	**for** $i \leftarrow 1, n$ **do**	$i \leftarrow 1$
while $i \leq n$ **do**	$S \leftarrow S + i * i$	**repeat**
$S \leftarrow S + i * i$	**end for**	$S \leftarrow S + i * i$
$i \leftarrow i + 1$	**return** S	$i \leftarrow i + 1$
end while		**until** $i > n$
return S		**return** S

(c) Întrucât nu se cunoaşte numărul de termeni se va folosi un alt criteriu de oprire: prelucrarea se opreşte după ce a fost adăugat primul termen mai mic decât valoarea ϵ (şirul $T(i)$ fiind descrescător, toţi termenii $T(i)$ cu $i > n(\epsilon)$ sunt mai mici decât ϵ). Algoritmul corespunzător este descris în 1.4 (`suma5`).

Exemplul 1.8 Să se determine cel mai mare divizor comun a două numere naturale nenule, a şi b.

Metoda de rezolvare. Se împarte a la b şi se reţine restul r. Dacă r este nul atunci cel mai mare divizor comun este b. Altfel se împarte b la r şi se reţine din nou restul. Procesul continuă, folosind ca deîmpărţit vechiul împărţitor şi ca împărţitor ultimul rest obţinut, până când se ajunge la un rest nul. Ultimul rest nenul (ultimul împărţitor) reprezintă cel mai mare divizor comun al numerelor a şi b.

Metoda de mai sus, cunoscută sub numele de *algoritmul lui Euclid*, poate fi descrisă matematic după cum urmează:

Algoritmul 1.4 Calculul unei sume finite şi aproximarea sumei unei serii

`suma4`(**real** x, **integer** n)	`suma5`(**real** x, **real** ϵ)
$S \leftarrow 0$	$S \leftarrow 0$
$T \leftarrow 1$	$T \leftarrow 1$
$i \leftarrow 1$	$i \leftarrow 1$
repeat	**repeat**
$\quad T \leftarrow T * x/i$	$\quad T \leftarrow T * x/i$
$\quad S \leftarrow S + T$	$\quad S \leftarrow S + T$
$\quad i \leftarrow i + 1$	$\quad i \leftarrow i + 1$
until $i > n$	**until** $T < \epsilon$
return S	**return** S

$$
\begin{aligned}
&a = bq_1 + r_1, \quad 0 \le r_1 < b\\
&b = r_1 q_2 + r_2, \quad 0 \le r_2 < r_1\\
&r_1 = r_2 q_3 + r_3, \quad 0 \le r_3 < r_2\\
&\vdots\\
&r_{i-2} = r_{i-1} q_i + r_i, \quad 0 \le r_i < r_{i-1}\\
&\vdots\\
&r_{n-1} = r_n q_{n+1} + r_{n+1}, \quad r_{n+1} = 0
\end{aligned}
$$

Metoda are caracter algoritmic deoarece succesiunea de împărţiri este finită (şirul resturilor este un şir strict descrescător de valori naturale). Metoda poate fi descrisă în pseudocod în oricare dintre variantele din Algoritmul 1.5 unde sunt folosite ca variabile ajutătoare: d (pentru deîmpărţit), i (pentru împărţitor) şi r (pentru rest). Algoritmul poate fi descris şi fără aceste variabile însă utilizarea lor îl face mai lizibil.

Algoritmul 1.5 Variante ale algoritmului lui Euclid

`euclid1`(**integer** a, b)	`euclid2`(**integer** a, b)
integer d,i,r	**integer** d,i,r
$d \leftarrow a$	$d \leftarrow a$
$i \leftarrow b$	$i \leftarrow b$
$r \leftarrow d$ MOD i	**repeat**
while $r \neq 0$ **do**	$\quad r \leftarrow d$ MOD i
$\quad d \leftarrow i$	$\quad d \leftarrow i$
$\quad i \leftarrow r$	$\quad i \leftarrow r$
$\quad r \leftarrow d$ MOD i	**until** $r = 0$
end while	**return** d
return i	

Exemplul 1.9 Să se determine cel mai mare dintre minimele valorilor de pe fiecare linie a unei matrici $(a_{ij})_{i=\overline{1,m}, j=\overline{1,n}}$, adică $\max_{i=\overline{1,m}} \min_{j=\overline{1,n}} a_{ij}$.

Metoda de rezolvare. Se construieşte un vector ce conţine valorile minime de pe linii: $b_i = \min_{j=\overline{1,n}} a_{ij}$, $i = \overline{1,m}$ după care se determină valoarea maximă din acest vector. Problema presupune astfel rezolvarea a două subprobleme: determinarea valorii minime dintr-un şir finit cu n elemente şi determinarea valorii maxime dintr-un şir finit cu m elemente.

Descrierea subalgoritmilor. Determinarea minimului unui şir finit reprezentat printr-un tablou cu n elemente reale se bazează pe compararea valorii unei variabile iniţializate cu primul element al şirului cu fiecare dintre următoarele elemente. În cazul în care este întâlnită o valoare mai mică atunci aceasta este reţinută în variabilă. Subalgoritmul ce permite determinarea maximului unui şir cu m elemente se bazează pe o metodă similară. Ambele metode sunt descrise în 1.6.

Algoritmul 1.6 Determinarea valorii minime (maxime) dintr-un tablou

```
minim(real x[1..n])
real min
integer i
min ← x[1]
for i ← 2, n do
   if min > x[i] then
      min ← x[i]
   end if
end for
return min
```

```
maxim(real x[1..m])
real max
integer i
max ← x[1]
for i ← 2, m do
   if max < x[i] then
      max ← x[i]
   end if
end for
return max
```

Descrierea algoritmului. Algoritmul descris în 1.7 constă în construirea elementelor unui tablou b (folosind subalgoritmul `minim`) şi în determinarea valorii maxime din acest tablou (folosind subalgoritmul `maxim`). Prin specificarea parametrilor de forma $x[1..n]$ sau $x[1..m]$ se subînţelege că numărul de elemente din tablou este n respectiv m. Parametrul $a[i, 1..n]$ utilizat la apelul subalgoritmului `minim` reprezintă linia i a matricii şi corespunde unui tablou unidimensional cu n elemente.

Algoritmul 1.7 Determinarea celei mai mari valori minime din liniile unei matrici

```
MaxMinMatrice(real a[1..m, 1..n])
real b[1..m],c
integer i
for i ← 1, m do
   b[i] ← minim(a[i, 1..n])
end for
c ← maxim(b[1..m])
return c
```

Exemplul 1.10 Se consideră un şir de valori şi se doreşte ordonarea crescătoare a acestora. Problema constă în găsirea unei metode de ordonare bazată doar pe inversarea ordinii elementelor unor secvenţe de elemente de la sfârşitul şirului (*sufixe* ale şirului sau eventual întregul şir). De exemplu, pornind de la şirul $4, 5, 3, 6, 1, 2$ şi aplicând transformările: $4, 5, 3, 6, 1, 2 \rightarrow 4, 5, 3, 6, 2, 1 \rightarrow 1, 2, 6, 3, 5, 4 \rightarrow 1, 2, 6, 4, 5, 3 \rightarrow 1, 2, 3, 5, 4, 6 \rightarrow 1, 2, 3, 5, 6, 4 \rightarrow 1, 2, 3, 4, 6, 5 \rightarrow 1, 2, 3, 4, 5, 6$ se ajunge la şirul ordonat crescător.

Metoda de rezolvare. Ideea rezolvării este următoarea: se identifică valoarea minimă şi se inversează secvenţa de valori ce începe cu aceasta; această operaţie va aduce valoarea minimă pe ultima poziţie în şir; se inversează întreg şirul şi astfel valoarea minimă ajunge pe prima poziţie a şirului; se aplică aceeaşi tehnică pentru secvenţa ce începe cu al doilea element al şirului ş.a.m.d. Metoda este descrisă în Algoritmul 1.8, împreună cu subalgoritmii `minim` (pentru determinarea indicelui valorii minime) şi `inversare` (pentru inversarea ordinii elementelor dintr-un subtablou).

Algoritmul 1.8 Sortarea prin inversarea secvenţelor sufix

```
ordonareSufix(real x[1..n])
integer imin, i
for i ← 1, n − 1 do
   imin ← minim(x[i..n])
   if imin ≠ i then
      x[imin..n] ← inversare(x[imin..n])
      x[i..n] ← inversare(x[i..n])
   end if
end for
return x[1..n]

minim(real x[s..d])
integer imin, i
imin ← s
for i ← s + 1, d do
   if x[imin] > x[i] then
      imin ← i
   end if
end for
return imin

inversare(real x[s..d])
integer mij, i
mij = ⌊(s + d)/2⌋
for i ← s, mij do
   x[i] ↔ x[s + d − i]
end for
return x[s..d]
```

Exemplul 1.11 Se consideră un număr natural constituit din 10 cifre distincte (de exemplu, 6709385421). Să se determine numărul imediat următor (în ordine crescătoare) din şirul tuturor numerelor naturale constituite din 10 cifre distincte.

Metoda de rezolvare. Considerăm numărul reprezentat prin tabloul cifrelor sale,

$x[1..10]$, în care cea mai semnificativă se află pe prima poziţie. Atâta timp cât cifrele numărului nu sunt în ordine descrescătoare (adică 9876543210) numărul cerut poate fi construit parcurgând următoarele etape:

Pas 1. Se parcurge şirul de la dreapta la stânga şi se identifică prima pereche (x_{i-1}, x_i) cu proprietatea $x_{i-1} < x_i$. Dacă numărul iniţial nu este constituit din secvenţa descrescătoare de cifre 9876543210, atunci există o astfel de pereche.

Pas 2. Se determină

$$x_k = \min\{x_j | j = \overline{i, n} \text{ cu } x_j > x_{i-1}\}$$

adică cel mai mic element al subtabloului $x[i..n]$ care îl depăşeşte pe x_{i-1} (existenţa unui astfel de element este asigurată de proprietatea $x_i > x_{i-1}$).

Pas 3. Se interschimbă x_{i-1} cu x_k. În felul acesta se obţine un număr mai mare decât cel iniţial.

Pas 4. Se ordonează crescător subtabloul $x[i..n]$. Datorită proprietăţilor lui $x[i..n]$ ordonarea este asigurată prin inversarea ordinii elementelor. Scopul ordonării crescătoare este acela de a obţine numărul imediat următor celui iniţial care satisface cerinţa de a fi constituit din cifre distincte.

Descrierea algoritmului. Algoritmul poate fi descompus în următorii subalgoritmi, corespunzători principalelor prelucrări specificate în descrierea de mai sus:

- `Identificare`: pentru găsirea perechii (x_{i-1}, x_i) cu proprietatea $x_{i-1} < x_i$. Subalgoritmul primeşte ca parametru întreg tabloul x şi returnează valoarea i. Dacă $i = 1$ rezultă că nu există succesorul cerut pentru numărul iniţial.
- `Minim`: pentru determinarea valorii minime din subtabloul $x[i..n]$ care are proprietatea că este mai mare decât x_{i-1}. Subalgoritmul va primi ca parametri: tabloul $x[1..n]$ şi indicele i şi va returna indicele valorii minime.
- `Sortare`: pentru ordonarea crescătoare a subtabloului $x[i..n]$ (de fapt este suficientă inversarea elementelor subtabloului, aşa cum este realizată în algoritmul `inversare` descris la exemplul anterior).

Structura generală precum şi subalgoritmii destinaţi identificării perechii (x_{i-1}, x_i) şi a indicelui minimului sunt descrişi în Agoritmul 1.9. Trebuie remarcat faptul că dacă $x[1..n]$ este ordonat descrescător atunci în ciclul **while** din subalgoritmul `identificare` variabila i ajunge la valoarea 1. Când i este 1 la evaluarea condiţiei "$i > 1$ **and** $x[i] < x[i-1]$" pot să apară probleme datorită inexistenţei lui $x[0]$. In continuare vom considera că la evaluarea unei astfel de conjuncţii evaluarea termenilor este stopată la întâlnirea primului termen ce are valoarea **false** (cum se întâmplă în cazul când $i = 1$).

Algoritmul 1.9 Determinarea următorului număr constituit din 10 cifre distincte

```
succesor(integer x[1..n])
i ←Identificare(x[1..n])
if i = 1 then
   return −1
else
   k ←Minim(x[1..n], i)
   x[i − 1] ↔ x[k]
   Inversare(x[i..n])
   return x[1..n]
end if

Minim(integer x[1..n], i)
k ← i
for j ← i + 1, n do
   if x[j] < x[k]and x[j] > x[i − 1] then
      k ← j
   end if
end for
return k

identificare(integer x[1..n])
i ← n
while (i > 1) and (x[i] < x[i − 1]) do
   i ← i − 1
end while
return i
```

1.3 Clase de probleme si tehnici de rezolvare

De multe ori în aplicaţiile reale apar probleme similare cu probleme cunoscute şi pentru care se cunosc metode de rezolvare. In funcţie de specificul lor şi de metodele de rezolvare corespunzătoare, problemele pot fi grupate în câteva clase importante.

1.3.1 Clase de probleme

Marea majoritate a problemelor întâlnite în practică se încadrează în câteva clase principale de probleme, printre care şi cele enumerate în continuare.

Căutare si sortare. Problema căutării se referă la verificarea prezenţei unui element intr-o colecţie de date. In cazul în care există o relaţie de ordine definită pe colecţia de date, căutarea se poate referi şi la identificarea uneia sau a tuturor poziţiilor pe care apare un anumit element. Verificarea prezenţei unui element se bazează adeseori pe compararea valorii unei componente a elementului numită *cheie de căutare*. Cea mai simplă tehnică de căutare constă în compararea elementului căutat cu toate elementele colecţiei. Simplitatea tehnicii este contrabalansată de faptul că nu este întotdeauna eficientă. Pentru a eficientiza procesul de căutare colecţia de date este de regulă reorganizată. Cea mai simplă metodă de reorganizare este ordonarea elementelor după cheia de căutare şi exploatarea acestui fapt prin reducerea numărului de elemente analizate.

Ordonarea elementelor unei colecţii de date, numită şi *sortare*, este utilă nu doar în contextul căutării cât şi pentru a facilita efectuarea altor prelucrări asupra datelor. Sortarea este una dintre cele mai frecvente prelucrări din informatică şi joacă un rol important atât în analiza datelor cât şi în pregătirea acestora în vederea rezolvării eficiente a unor probleme.

Satisfacerea restricţiilor şi optimizare combinatorială. Problemele din această categorie se referă la identificarea unor structuri discrete: (sub)mulţimi, secvenţe ordonate (permutări), grafuri , funcţii pe mulţimi finite etc. care satisfac anumite restricţii şi/sau optimizează un anumit criteriu. Astfel de probleme intervin în numeroase domenii din inginerie şi economie. Există o serie de probleme care au numeroase aplicaţii practice, cum ar fi: *problema rucsacului*, *problema planificării activităţilor*, *problema comis voiajorului*, *problema rutării vehiculelor* etc.

Prelucrarea şirurilor de caractere. Şirurile de caractere sunt secvenţe de elemente dintr-un anumit alfabet şi intervin în diferite domenii aplicative (analiza automată a documentelor, analiza secvenţelor ADN etc.). Una dintre prelucrările cele mai frecvente este cea a identificării tuturor apariţiilor unui subşir sau a unui şablon într-un şir de caractere de dimensiuni mari. Principala dificultate în cazul unei astfel de probleme o reprezintă numărul mare de comparaţii necesare.

Geometrie computaţională. Problemele din această categorie sunt întâlnite frecvent în domeniile graficii, prelucrării imaginilor, sistemelor de informare geografică şi roboticii. Specificul acestor probleme este faptul că operează cu obiecte geometrice (puncte, segmente de dreaptă, poligoane, curbe etc.). Exemple de probleme din această clasă sunt [**?**]: *problema înfăşurătorii convexe* (determinarea celui mai mic poligon convex care conţine toate elementele unei mulţimi de puncte în plan), *problema triangularizării unui poligon* (descompunerea unui poligon în triunghiuri cu interioare disjuncte astfel încât suma perimetrelor acestora să fie cât mai mică), *problema construirii diagramelor Voronoi* (partiţionarea unei regiuni din plan care conţine o mulţime finită de puncte de referinţă în aşa fel încât fiecare subregiune să conţină un singur punct de referinţă şi distanţele dintre acesta şi punctele subregiunii să fie mai mici decât distanţele dintre punctele subregiunii şi celelalte puncte de referinţă).

1.3.2 Tehnici de rezolvare

Tehnicile de elaborare a algoritmilor sunt strategii generale de proiectare a algoritmilor şi pot fi aplicate pentru clase largi de probleme ce intervin in diverse domenii ale informaticii. Câteva dintre cele mai utilizate metode generale de elaborare a algoritmilor sunt:

Metoda "forţei brute". Este cea mai simplă metodă de elaborare a algoritmilor fiind bazată pe abordarea directă a problemei. Este intuitivă şi uşor de implementat însă nu este în toate cazurile eficientă.

Metoda reducerii ("Decrease and conquer"). Ideea acestei metode este de a exploata relaţia care există între soluţia unei probleme şi soluţia unei instanţe de dimensiune mai mică a aceleiaşi probleme.

Metoda divizării ("Divide and conquer"). Se bazează pe ideea divizării problemei în subprobleme independente şi în rezolvarea fiecărei subprobleme aplicând aceeaşi tehnică.

Metoda căutării local optimale ("Greedy"). Este utilizată în principal pentru rezolvarea problemelor care necesită optimizarea unui criteriu şi se bazează pe ideea de a construi soluţia în manieră incrementală, alegând la fiecare etapă o componentă a soluţiei. Componenta aleasă este cea care pare a fi cea mai promiţătoare din punctul de vedere al apropierii de soluţie. Această alegere nu garantează faptul că soluţia finală este optimă.

Metoda programării dinamice ("Dynamic Programming"). Se bazează pe rezolvarea unei probleme prin descompunerea ei in subprobleme care se suprapun (legătura dintre soluţia unei probleme şi soluţiile subproblemelor poate fi descrisă printr-o relaţie de recurenţă). Elementul cheie al metodei este faptul că subproblemele care se repetă se rezolvă o singură dată şi nu de fiecare dată când sunt întâlnite.

Metoda căutării cu revenire ("Backtracking"). Este o tehnică de căutare sistematică a spaţiului soluţiilor bazată pe reţinerea traseului parcurs şi pe revenirea, dacă este cazul la configuraţii anterioare. În cazul problemelor de optimizare o tehnică care permite limitarea căutării este este cea de tip *ramifică şi mărgineşte* ("branch and bound").

Toate aceste metode vor fi descrise în capitolele următoare.

1.4 Probleme

Problema 1.1 (*Inmulţirea à la russe.*) Se consideră următoarea metodă de înmulţire a două numere naturale nenule x şi y. Se scrie x alături de y (pe aceeaşi linie). Se împarte x la 2 şi câtul împărţirii se scrie sub x (restul se ignoră deocamdată). Se înmulţeşte y cu 2 iar produsul se scrie sub y. Procedeul continuă construindu-se astfel două coloane de numere. Calculele se opresc în momentul în care pe prima coloană se obţine valoarea 1. Se adună toate valorile de pe coloana a doua care corespund unor valori impare aflate pe prima coloană.

Exemplu. Fie $x = 13$ şi $y = 25$. Succesiunea de rezultate obţinute prin aplicarea operaţiilor de mai sus este:

x	y	rest	factor multiplicare y
13	$\boxed{25}$	1	2^0
6	50	0	2^1
3	$\boxed{100}$	1	2^2
1	$\boxed{200}$	1	2^3
	325		

a) Prelucrarea descrisă prin metoda de mai sus se termină întotdeauna după un număr finit de paşi? Ce se întâmplă în cazul în care una dintre valori este egală cu 0?

b) Metoda de mai sus conduce întotdeauna la produsul celor două numere (cu alte cuvinte este o metodă corectă de înmulţire)?

c) Câte operaţii de înmulţire cu 2 sunt necesare? Depinde acest număr de ordinea factorilor?

Indicaţie.

a) Ca urmare a împărţirilor succesive la 2, valorile de pe prima coloană descresc până se ajunge la un cât egal cu 1 (în ipoteza că x este nenul). Prin urmare prelucrarea este finită. Dacă x este egal cu 0 metoda nu poate fi aplicată ca atare însă, dacă y este 0, ea conduce la un rezultat corect.

b) Corectitudinea prelucrării derivă din faptul că metoda realizează de fapt conversia în baza 2 a primului număr (cifrele reprezentării în baza doi sunt resturile obţinute prin împărţirile succesive la 2) iar produsul se obţine prin înmulţirea succesivă a celui de al doilea număr cu puteri ale lui 2 şi prin însumarea acelor produse care corespund unor cifre nenule în reprezentarea binară a primului număr.

c) Numărul înmulţirilor cu 2 este cu 1 mai mic decât numărul de cifre ale reprezentării binare a lui x, adică $\lfloor \log_2 x \rfloor$. Evident numărul înmulţirilor depinde de ordinea factorilor fiind mai mic dacă împărţirile la 2 se efectuează asupra celui mai mic număr.

Problema 1.2 Propuneţi o metodă bazată pe operaţii de adunare, scădere şi comparare pentru determinarea părţii întregi a unui număr real (cea mai mare valoare întreagă mai mică decât numărul dat).

Indicaţie. Se va ţine cont de semnul numărului. În cazul în care este pozitiv se scade succesiv valoarea 1 până când se ajunge la o valoare mai mică strict decât 1. Numărul scăderilor efectuate indică valoarea părţii întregi. Pentru numere negative se adună succesiv 1 până se obţine o valoare mai mare sau egală cu 0. Opusul numărului de adunări reprezintă valoarea părţii întregi. În continuare este prezentat câte un exemplu pentru valori pozitive, respectiv negative.

Exemplu.

$x > 0$		$x < 0$	
x	contor	x	contor
3.25	0	-3.25	0
2.25	1	-2.25	-1
1.25	2	-1.25	-2
0.25	$\boxed{3}$	-0.25	-3
		0.75	$\boxed{-4}$

Problema 1.3 Se consideră un dreptunghi având lungimile laturilor a respectiv b (ambele valori sunt numere naturale). Să se propună un algoritm care să determine latura pătratului care permit acoperirea (pavarea) completă a dreptunghiului astfel încât numărul de pătrate folosite să fie cât mai mic. În descrierea algoritmului este permisă folosirea doar a operaţiei de scădere şi a comparaţiilor.

Indicaţie. Pentru a se asigura acoperirea completă a dreptunghiului trebuie ca latura pătratelor să fie divizor atât pentru a cât şi pentru b. Pentru a se realiza acoperirea cu un număr cât mai mic de pătrate latura trebuie să fie cel mai mare divizor comun al celor două valori. S-a ajuns astfel la problema determinării celui mai mare divizor comun a două numere naturale folosind doar operaţii de scădere şi comparare. Algoritmul este descris in 1.10.

Algoritmul 1.10 Algoritmul lui Euclid bazat pe operaţii de scădere

```
euclid3 (integer a, b)
while a ≠ b do
  if a > b then
    a ← a − b
  else
    b ← b − a
  end if
end while
return a
```

Problema 1.4 (*Problema identificării monedei mai uşoare.*) Se consideră un set de n monede identice, cu excepţia uneia care are greutatea mai mică decât celelalte. Folosind o balanţă simplă să se identifice moneda cu greutatea mai mică folosind cât mai puţine comparaţii.

Indicaţie. O primă variantă este aceea prin care se selectează la întâmplare două monede şi se pun pe balanţă. Dacă una dintre ele are greutatea mai mică atunci este chiar moneda căutată. Daca ambele au aceeaşi greutate se păstrează una dintre ele pe un taler al balanţei iar pe celălalt taler se pun succesiv monedele

rămase. Prelucrarea se opreşte în momentul în care se găseşte o monedă cu greutatea mai mică. Numărul de comparaţii este cel mult $n-1$.

O altă variantă este cea în care se împarte setul de monede în două subseturi (fiecare va avea $n/2$ elemente, dacă n e par, respectiv $(n-1)/2$ daca n este impar) care se pun pe cele două talere ale balanţei. Dacă subseturile au aceeaşi greutate (ceea ce se poate întâmpla doar dacă n este impar) atunci moneda pusă deoparte este cea căutată. În caz contrar se aplică acelaşi procedeu subsetului de greutate mai mică şi se continuă până se ajunge la un set de două sau trei monede. În acest moment este suficient să se mai efectueze o singură cântărire. Numărul de cântăriri este cel mult $\lfloor \log_2 n \rfloor$. Este adevărat acest rezultat pentru cazul în care se ştie doar că una dintre monede este diferită dar nu se ştie dacă este mai uşoară sau mai grea decât celelalte?

Problema 1.5 (*Problema clătitelor.*) La un restaurant bucătarul a pregătit clătite pe care le-a aşezat pe un platou sub forma unei stive. Din păcate nu toate clătitele au acelaşi diametru astfel că stiva nu arată prea frumos. Chelnerul ia platoul şi având la dispoziţie o spatulă reuşeşte să aranjeze cu o singură mână clătitele astfel încât să fie în ordinea descrescătoare a diametrelor. Cum a procedat? Descrieţi problema într-o manieră abstractă şi propuneţi un algoritm de rezolvare.

Indicaţie. Rearanjarea se face efectuând doar mişcări de răsturnare a unui "set" de clătite dintre cele aflate in partea de sus a stivei. Problema este identică cu cea a ordonării descrescătoare a unui şir de valori prin inversarea ordinii elementelor unor subsecvenţe de la sfârşitul şirului.

Exemplu. Pentru a ordona descrescător şirul $(5, 3, 4, 1, 6, 2)$ se aplică următoarea secvenţă de prelucrări în care subsecvenţa care se "răstoarnă" este încadrată:

```
5 3 4 1 [6 2]
[5 3 4 1 2 6]
6 [2 1 4 3 5]
6 5 3 [4 1 2]
6 5 [3 2 1 4]
6 5 4 [1 2 3]
6 5 4 3 2 1
```

Metoda constă în determinarea valorii maxime şi inversarea ordinii subsecvenţei care începe cu ea pentru a fi adusă pe ultima poziţie în şir. Se răstoarnă întregul şir, astfel că valoarea maximă ajunge pe prima poziţie. Se aplică aceeaşi metodă pentru subsecvenţa care începe cu al doilea element şi se continuă până se ajunge la o subsecvenţă constituită dintr-un singur element. Algoritmul este similar celui corespunzător ordonării crescătoare descris în secţiunea 1.2.4.

Problema 1.6 Considerăm următoarele două probleme:
(a) O gospodină a făcut o serie de cumpărături şi are la dispoziţie două sacoşe.

Problema constă în a distribui cumpărăturile în cele două sacoşe astfel încât diferenţa între greutăţile celor două sacoşe să fie cât mai mică.
(b) Se consideră două dispozitive de stocare a informaţiei (de exemplu discuri) şi o mulţime de fişiere a căror dimensiuni însumate nu depăşeşte capacitatea globală a celor două dispozitive. Se încearcă repartizarea fişierelor în cele două discuri astfel încât diferenţa dintre spaţiile rămase libere să fie cât mai mică. Este vreo legătură între cele două probleme? Propuneţi metode de rezolvare.

Indicaţie. Ambele probleme pot fi formalizate astfel: se consideră o mulţime de numere pozitive, $A = \{a_1, a_2, \ldots, a_n\}$ şi se cere să se determine două submulţimi disjuncte B şi C astfel încât $B \cup C = A$ şi modulul diferenţei dintre sumele elementelor din cele două submulţimi ($|\sum_{a \in B} a - \sum_{a \in C} a|$) este minimă. Cea mai simplă metodă este cea a "forţei brute" prin care se generează toate perechile de submulţimi (B, C) şi se alege cea care minimizează diferenţa specificată. Numărul de partiţii distincte este $2^{n-1}-1$. Pentru n mare, numărul de partiţii ce trebuie testate devine mare (pentru $n = 10$ este 511, iar pentru $n = 100$ este de ordinul 10^{29}). E evident că în astfel de situaţii metoda forţei brute nu este eficientă, astfel că trebuie căutate metode mai puţin costisitoare.

Problema 1.7 (*Problema identificării anagramelor din dicţionar.*) Se consideră un dicţionar cu $n = 50000$ de cuvinte şi un cuvânt de lungime m ($m \ll n$). Propuneţi o metodă, bazată pe un număr cât mai mic de comparaţii, care să determine toate anagramele cuvântului prezente în dicţionar.

Indicaţie. Se pot analiza cel puţin două variante: (a) se generează toate anagramele cuvântului iniţial (sunt cel mult $m!$ variante) după care se caută fiecare în dicţionar (în total aproximativ $n \cdot m \cdot m!$ comparaţii la nivel de caracter); (b) se ordonează crescător literele cuvântului (necesită circa m^2 operaţii), se parcurge dicţionarul şi fiecare cuvânt având aceeaşi lungime cu cuvântul analizat se ordonează crescător şi se compară cu varianta ordonată a cuvântului iniţial (în total $m^2 + n(m^2 + m)$ operaţii). Pentru $n = 50000$ iar $m = 12$ numărul de operaţii efectuate în primul caz este de circa $8 \cdot 10^{10}$, iar în al doilea caz este $8 \cdot 10^6$. În ipoteza că o operaţie necesită $10^{-3}s$ în primul caz sunt necesare aproximativ 24000 ore, iar în al doilea circa 2 ore.

Problema 1.8 Fie n un număr natural nenul. Descrieţi în pseudocod algoritmi pentru:

(a) Determinarea sumei tuturor cifrelor lui n. De exemplu, pentru $n = 26326$ se obţine valoarea 19.

(b) Determinarea valorii obţinute prin inversarea cifrelor numărului n. De exemplu, pentru valoarea 26326 se obţine valoarea 62362.

(c) Determinarea mulţimii tuturor cifrelor ce intervin în număr. De exemplu, pentru valoarea 26326 se obţine mulţimea $\{2, 3, 6\}$.

(d) Determinarea tuturor cifrelor binare ale lui n.

(e) Determinarea tuturor divizorilor proprii ai lui n.

(f) A verifica dacă numărul n este prim sau nu (algoritmul returnează *true* dacă numărul este prim şi *false* în caz contrar).

(g) Determinarea descompunerii în factori primi a lui n. De exemplu pentru $490 = 2^1 \cdot 5^1 \cdot 7^2$ se obţine mulţimea factorilor primi: $\{2, 5, 7\}$ şi puterile corespunzătoare: $\{1, 1, 2\}$.

Rezolvare.
(a) Cifrele numărului se extrag prin împărţiri succesive la 10. La fiecare etapă restul va reprezenta ultima cifră a valorii curente, iar câtul împărţirii întregi va reprezenta următoarea valoare ce se va împărţi la 10 (vezi `suma_cifre` in Algoritmul 1.11).

(b) Dacă $n = c_k 10^k + \ldots c_1 10 + c_0$ atunci numărul căutat este $m = c_0 10^k + \ldots c_{k-1} 10 + c_k = (\ldots(c_0 10 + c_1)10 + c_2 + \ldots c_{k-1})10 + c_k$. Cifrele lui n se extrag, începând de la ultima, prin împărţiri succesive la 10. Pentru a-l construi pe m este suficient să se pornească de la valoarea 0 şi pentru fiecare cifră extrasă din n să se înmulţească m cu 10 şi să se adune cifra obţinută. Algoritmul corespunzător este `inversare_cifre` (Algoritmul 1.11).

Algoritmul 1.11 Suma cifrelor şi valoarea obţinută prin inversarea ordinii cifrelor unui număr natural

`suma_cifre`(**integer** n)	`inversare_cifre`(**integer** n)
integer S	**integer** m
$S \leftarrow 0$	$m \leftarrow 0$
while $n > 0$ **do**	**while** $n > 0$ **do**
$S \leftarrow S + n$ MOD 10	$m \leftarrow m * 10 + n$ MOD 10
$n \leftarrow n$ DIV 10	$n \leftarrow n$ DIV 10
end while	**end while**
return S	**return** m

(c) Cifrele se determină prin împărţiri succesive la 10. Mulţimea cifrelor poate fi reprezentată fie printr-un tablou, $c[0..9]$, cu 10 elemente conţinând indicatori de prezenţă (elementul de pe poziţia i, este 1 dacă cifra i este prezentă în număr şi 0 în caz contrar) fie printr-un tablou care conţine cifrele distincte ce apar în număr. În al doilea caz, pentru fiecare cifră extrasă se verifică dacă nu se află deja în tabloul cu cifre. Cele două variante sunt descrise în Algoritmul 1.12.

(d) Cifrele binare ale lui n se obţin prin împărţiri succesive la 2 până se ajunge la valoarea 0. Restul primei împărţiri reprezintă cifra cea mai puţin semnificativă, iar restul ultimei împărţiri reprezintă cifra cea mai semnificativă. Presupunând că $2^{k-1} \leq n < 2^k$ sunt suficiente k poziţii binare pentru reprezentare, iar algoritmul poate fi descris prin:

`cifre_binare`(**integer** n)

Algoritmul 1.12 Determinarea mulţimii cifrelor unui număr natural

```
cifre1(integer n)
integer c[0..9], i
for i ← 0,9 do
   c[i] ← 0
end for
while n > 0 do
   c[n MOD 10] ← 1
   n ← n DIV 10
end while
return  c[0..9]
```

```
cifre2(integer n)
integer c[1..10], i,j
boolean prezent
i ← 0
while n > 0 do
   r ← n MOD 10; j ← 1; prezent ← false
   while j <= i and prezent =false do
      if c[j] = r then
         prezent ← true
      else
         j ← j + 1
      end if
   end while
   if  prezent =false then
      i ← i + 1; c[i] ← r
   end if
   n ← n DIV 10
end while
return  c[1..i]
```

```
integer b[0..k − 1],i
i ← 0
while n > 0 do
   b[i] ← n MOD 2
   i ← i + 1
   n ← n DIV 2
end while
return  b[0..i − 1]
```

Pentru a avea în tabloul b cifrele binare în ordinea naturală (începând cu cifra cea mai semnificativă) este suficient să se inverseze ordinea elementelor tabloului. Secvenţa de prelucrări care realizează acest lucru este:

```
for j ← 0, ⌊(i − 1)/2⌋ do
   b[j] ↔ b[i − 1 − j]
end for
```

În secvenţa de mai sus operatorul de interschimbare $\leftrightarrow$ presupune efectuarea a trei atribuiri şi utilizarea unei variabile auxiliare (aux) pentru reţinerea temporară a valorii uneia dintre variabilele implicate în interschimbare:

```
aux ← a
```

$a \leftarrow b$
$b \leftarrow aux$

(e) Pentru determinarea divizorilor proprii ai lui n (divizori diferiţi de 1 şi n) este suficient să se analizeze toate valorile cuprinse între 2 şi $\lfloor n/2 \rfloor$. Algoritmul care afişează valorile divizorilor proprii poate fi descris prin:

```
divizori(integer n)
integer i
for i ← 2, ⌊n/2⌋ do
  if n MOD i = 0 then
    write i
  end if
end for
```

(f) Se porneşte de la premiza că numărul este prim (iniţializând variabila ce va conţine rezultatul cu **true**) şi se verifică dacă are sau nu divizori proprii (este suficient să se parcurgă domeniul valorilor cuprinse între 2 şi $\lfloor \sqrt{n} \rfloor$). La detectarea primului divizor se poate decide că numărul nu este prim. Aceste prelucrări sunt descrise în algoritmul `prim` din 1.13.

Algoritmul 1.13 Verificarea proprietăţii de număr prim şi descompunerea în factori primi

```
prim(integer n)
integer i
boolean p
p ← true
i ← 2
while i ≤ ⌊√n⌋ and p =true do
  if n MOD i = 0 then
    p ← false
  else
    i ← i + 1
  end if
end while
return p
```

```
factori_primi(integer n)
integer f[1..k], p[1..k],i,d
i ← 0
d ← 2
repeat
  if n MOD d = 0 then
    i ← i + 1
    f[i] ← d
    p[i] ← 1
    n ← n DIV d
    while n MOD d = 0 do
      p[i] ← p[i] + 1
      n ← n DIV d
    end while
  end if
  d ← d + 1
until n < d
return f[1..i], p[1..i]
```

(g) Descompunerea în factori primi a numărului n se va reţine în două tablouri: tabloul f conţine valorile factorilor primi, iar tabloul p conţine valorile puterilor

corespunzătoare. Algoritmul este similar celui de determinare a divizorilor, însă când este descoperit un divizor se contorizează de câte ori se divide n prin acea valoare. În acelaşi timp factorul descoperit este "eliminat" din n prin împărţiri succesive. Aceasta asigură faptul că divizorii ulteriori nu-i vor conţine ca factori pe divizorii mai mici, adică vor fi valori prime. O variantă a acestei metode este descrisă în algoritmul `factori_primi` din 1.13.

Problema 1.9 Fie n un număr natural, x o valoare reală din $(0,1)$ şi $\epsilon > 0$ o valoare reală pozitivă. Descrieţi în pseudocod un algoritm pentru:

(a) Calculul sumei $\sum_{i=1}^{n}(-1)^i x^{2i}/(2i)!$.

(b) Calculul aproximativ al sumei $\sum_{i=1}^{\infty}(-1)^i x^{2i}/(2i)!$ cu precizia ϵ.

Indicaţie.
(a) Suma poate fi rescrisă ca $S = T(1) + \ldots + T(n)$ cu $T(i) = (-1)^i x^{2i}/(2i)!$. Pentru a reduce numărul calculelor implicate de evaluarea fiecărui termen se poate folosi relaţia $T(i) = -x^2 T(i-1)/((2i-1)2i)$ cu $T(1) = -x^2/2$.
(b) Calculul aproximativ al unei sume infinite presupune însumarea unui număr finit de termeni până când valoarea absolută a ultimului termen adăugat este suficient de mică ($|T(i)| < \epsilon$).

Problema 1.10 Să se afişeze primele N elemente şi să se aproximeze (cu precizia ϵ) limitele şirurilor (cu excepţia şirului de la punctul (d) care nu este neapărat convergent):

(a) $x_n = (1 + 1/n)^n$;

(b) $x_1 = a > 0$, $x_n = (x_{n-1} + a/x_{n-1})/2$;

(c) $x_n = f_{n+1}/f_n$, $f_1 = f_2 = 1$, $f_n = f_{n-1} + f_{n-2}$;

(d) $x_1 = s$, $x_n = (ax_{n-1} + b)$ MOD c, $n \geq 1$, cu $s, a, b, c \in N^*$.

Rezolvare.
(a) Afişarea primelor N elemente ale şirului x_n este descrisă în algoritmul `elemente_sir`. În cazul unui şir convergent, limita poate fi aproximată prin elementul x_L care satisface proprietatea $|x_L - x_{L-1}| < \epsilon$. În acest caz trebuie cunoscută la fiecare etapă atât valoarea curentă (xc) cât şi valoarea precedentă a şirului (xp). O variantă de estimare a limitei este ilustrată în algoritmul `limita_sir`.

```
elemente_sir(integer N)
integer n
for n ← 1, N do
   write putere(1 + 1/n,n)
end for
putere(real x,integer n)
real p
integer i
p ← 1
for i ← 1, n do
   p ← p * x
end for
return p
```

```
limita_sir(real eps)
integer n
real xc, xp
n ← 1
xc ← 2
repeat
   xp ← xc
   n ← n + 1
   xc ← putere((n + 1/n),n)
until |xc − xp| ≤ eps
return xc
```

(b) Pentru determinarea elementelor unui şir dat printr-o relaţie de recurenţă de ordin r, $x_n = f(x_{n-1}, \ldots, x_{n-r})$ pentru care se cunosc primele r valori, se pot utiliza $r+1$ variabile: r care conţin valorile anterioare din şir $(p_1, \ldots p_r)$ şi una care conţine valoarea curentă (p_0). Structura generală a acestei prelucrări este descrisă în algoritmul `sir_recurent`.

```
sir_recurent(integer N, real x[1..r])
integer i
real p[0..r]
p[1..r] ← x[1..r]
for i ← r + 1, N do
   for k ← r, 1, −1 do
      p[k] ← p[k − 1]
   end for
   p[0] ← f(p[1], ..., p[r])
   write p[0]
end for
```

Atribuirea $p[1..r] \leftarrow x[1..r]$ semnifică faptul că elementelor cu indicii cuprinşi între 1 şi r din tabloul p li se asignează valorile elementelor corespunzătoare din tabloul x. În cazul în care $r = 1$ algoritmul devine mai simplu. Generarea elementelor cât şi estimarea limitei (şirul converge către $\sqrt{a}$) sunt descrise în Algoritmul 1.14.

(c) Şirul f_n este dat printr-o relaţie de recurenţă de ordin 2 şi este cunoscut ca fiind *şirul lui Fibonacci*. Se poate demonstra că şirul x_n converge către $\theta = (1 + \sqrt{5})/2$. Generarea elementelor şi estimarea limitei sunt descrise în Algoritmul 1.15.

(d) Relaţiile de recurenţă de acest tip sunt folosite pentru generarea de valori (pseudo)aleatoare şi sunt utilizate pentru implementarea algoritmilor aleatori. Valorile generate prin relaţia dată sunt între 0 şi $c - 1$. Metoda de generare este descrisă în algoritmul 1.16.

Algoritmul 1.14 Generarea elementelor şi estimarea limitei unui şi dat printr-o relaţie de recurenţă simplă

```
sir_recurent2(integer N, real a)
integer n
real x
x ← a
for n ← 2, N do
   x ← (x + a/x)/2
   write x
end for
```

```
limita_sir2(real a,eps)
real xp,xc
xc ← a
repeat
   xp ← xc
   xc ← (xp + a/xp)/2
until |xp − xc| < eps
return xc
```

Algoritmul 1.15 Şirul rapoartelor elementelor din şirul Fibonacci

```
sir3(integer N)
integer n, f0,f1
real x
f0 ← 1
f1 ← 1
for 1 ← 3, N do
   write f0/f1
   f2 ← f1
   f1 ← f0
   f0 ← f1 + f2
end for
```

```
limita_sir3(real eps)
integer f0, f1
real xc, xp
f0 ← 1
f1 ← 1
xc ← f0/f1
repeat
   xp ← xc
   f2 ← f1
   f1 ← f0
   f0 ← f1 + f2
   xc ← f0/f1
until |xc − xp| ≤ eps
return xc
```

Algoritmul 1.16 Generarea unei secvenţe de valori pseudoaleatoare

```
sir_pseudoaleator(integer N, s, a, b, c)
integer x,n
x ← s
for n ← 2, N do
   x ← (a ∗ x + b) MOD c
   write x
end for
```