**Daniela Zaharie**

---

# Introducere în proiectarea şi analiza algoritmilor

ISBN 978-973-673-121-1

DANIELA ZAHARIE

# INTRODUCERE ÎN PROIECTAREA

# ŞI

# ANALIZA ALGORITMILOR

eubeea,

2008

# Cuvânt înainte

Cartea prezintă o tematică clasică în informatică. Deşi tema a fost studiată şi prezentată în numeroase alte lucrări, de-a lungul ultimelor trei decenii, modalitatea în care este prezentată în această carte reflectă experienţa didactică îndelungată a autoarei în prezentarea problematicii la nivel academic. Remarcăm, în mod special, calitatea expunerii. Astfel, lucrarea este prezentată într-un stil riguros, coerent, îngrijit şi atent la detalii.

Utilizarea a numeroase exemple este binevenită. Structurarea volumului este inedită, firul roşu al cărţii fiind axat pe analiza şi verificarea algoritmilor, faţă de alte abordări care se bazează pe expunerea soluţiilor şi mai puţin pe formarea unei gândiri algoritmice corecte, precum în cazul acestei cărţi. Astfel, cititorilor cărora li se adresează, în principal, această carte, cei care fac primii paşi în domeniul informaticii, li se prezintă o imagine a tematicii, bazată pe logică, exactitate şi corectitudine.

Cititorul cărţii va găsi soluţii elegante ale unor probleme clasice precum cele legate de sortare sau operaţii matriciale, cât şi probleme mai dificile cum ar fi problema colorării hărţilor sau amplasarea damelor pe tabla de şah, sau probleme din domenii speciale, de exemplu, geometria computaţională sau teoria numerelor.

Nivelul solicitat cititorului creşte treptat de la un capitol la altul, astfel, încât atât cititorii nespecializşti în domeniu cât şi cei avizaţi pot găsi subiecte interesante pentru citire sau învăţare.

Numeroasele exemple de algoritmi sunt însoţite de comentarii consistente care ajută cititorii să înţeleagă soluţiile expuse. De asemenea, numeroasele probleme, ce stimulează cititorii să-şi îmbunătăţească cunoştinţele acumulate prin lectură, sunt în majoritate dublate de indicaţii şi pe alocuri de rezolvări complete.

Cartea se adresează, în principal, studenţilor din primii ani de studiu ai facultăţilor de ştiinţe exacte, în special în direcţia informatică. Mai mult, tematica abordată, fiind un subiect de interes constant în informatică, precum şi stilul plăcut al scrierii, fac ca această carte să prezinte un interes deosebit nu numai pentru studenţi, dar şi pentru întreaga comunitate educaţională şi ştiinţifică românească.

prof.dr. Dana Petcu

## Structura cărţii

Întrucât scopul acestei lucrări este familiarizarea cu selectarea, adaptarea, descrierea şi analiza algoritmilor destinaţi rezolvării problemelor specifice din informatică accentul este pus pe simplitate şi accesibilitate. Majoritatea algoritmilor prezentaţi se bazează pe utilizarea unor structuri de date simple, de tipul tablourilor, nefiind necesare cunoştinţe preliminare de algoritmică sau de programare.

În primul capitol sunt prezentate noţiunile generale şi limbajul de descriere a algoritmilor utilizat pe tot parcursul lucrării. In alegerea limbajului de descriere s-a optat pentru un limbaj simplu care să permită descrierea principalelor prelucrări implicate în algoritm fără a se intra însă în detaliile specifice limbajelor de programare.

În capitolul doi este prezentată problematica verificării corectitudinii algoritmilor punându-se accentul pe utilizarea conceptului de proprietate invariantă atât în demonstrarea corectitudinii cât şi în proiectarea algoritmilor.

Următorul capitol este destinat analizei eficienţei algoritmilor din perspectiva timpului de execuţie, interpretat ca număr de operaţii elementare efectuate. Tot în acest capitol sunt introduse notaţiile asimptotice utilizate pe tot parcursul lucrării pentru a exprima ordinul de complexitate al algoritmilor analizaţi.

În capitolul patru sunt prezentate şi analizate metode clasice de sortare bazate pe compararea dintre elemente, dar şi câteva metode bazate pe proprietăţi specifice elementelor tabloului de sortat.

Începând cu capitolul cinci sunt prezentate câteva tehnici generale de elaborare a algoritmilor: reducere şi divizare ("decrease and conquer" şi "divide and conquer"), căutare local optimală ("greedy"), programare dinamică, căutare sistematică în spaţiul soluţiilor ("backtracking" şi "branch and bound"). Ultimul capitol conţine câţiva algoritmi dedicaţi unor probleme specifice din domeniul aritmeticii, geometriei computaţionale şi căutării în şiruri.

Cartea se adresează celor ce doresc să păşească în lumea fascinantă a rezolvării algoritmice a problemelor. Pentru paşii următori cititorii interesaţi pot consulta una sau mai multe dintre lucrările din domeniu dintre care o mică parte sunt menţionate la bibliografie.

Un rol important în conturarea conţinutului acestei lucrări l-au avut cursurile şi seminariile prezentate la disciplina Algoritmică, astfel că la selecţia exemplelor prezentate au avut o contribuţie semnificativă studenţii din ultimele cinci promoţii ale secţiei de Informatică de la Universitatea de Vest din Timişoara.

Timişoara, octombrie 2008 Autoarea

# 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 [14]:

- *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.

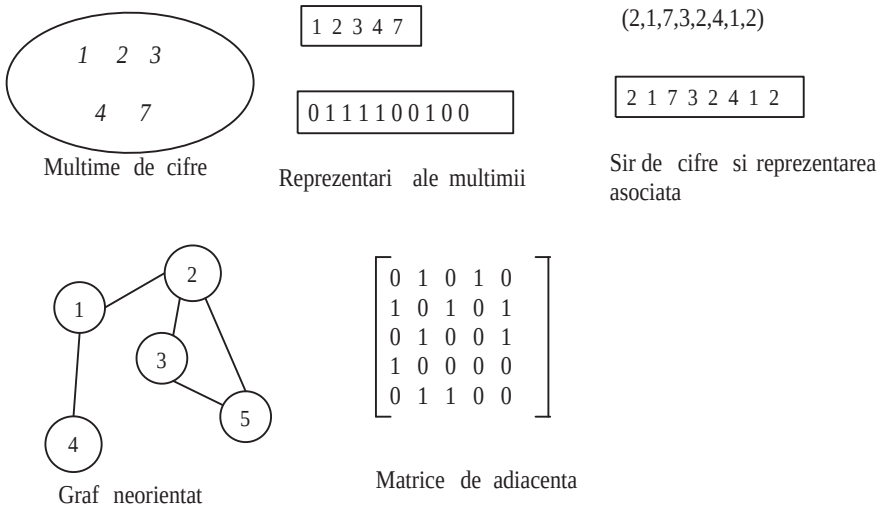


Figura 1.1: Modalităţi de reprezentare a datelor structurate

### 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), ^ (ridicare la putere), DIV sau / (câtul împărţirii întregi), MOD sau % (restul împărţirii întregi);
- *relaţionali:* = (egal), ≠ sau ! = (diferit), < (strict mai mic), ≤ sau <= (mai mic sau egal), > (strict mai mare), ≥ 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⟩

$\vdots$

⟨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:

**if** ⟨ conditie ⟩ **then** ⟨prelucrare⟩ **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:

**while** ⟨conditie⟩ **do** ⟨prelucrare⟩ **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

**for** $v \leftarrow v1, v2, pas$ **do** ⟨prelucrare⟩ **end for**

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

$v \leftarrow v1$
**while** $v \leq v2$ **do**
  ⟨prelucrare⟩
  $v \leftarrow v + pas$
**end while**

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:

**repeat** ⟨prelucrare⟩ **until** ⟨condiţie⟩

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 $\epsilon$ 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

```
1: ordonare(real a,real b,real c)
2: if a > b then
3:    a ↔ b
4: end if
5: if a > c then
6:    a ↔ c
7: end if
8: if b > c then
9:    b ↔ c
10: end if
11: 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 $\epsilon$ (şirul $T(i)$ fiind descrescător, toţi termenii $T(i)$ cu $i > n(\epsilon)$ sunt mai mici decât $\epsilon$). 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** $\epsilon$) |
|---|---|
| $S \leftarrow 0$ | $S \leftarrow 0$ |
| $T \leftarrow 1$ | $T \leftarrow 1$ |
| $i \leftarrow 1$ | $i \leftarrow 1$ |
| **repeat** | **repeat** |
| $T \leftarrow T * x/i$ | $T \leftarrow T * x/i$ |
| $S \leftarrow S + T$ | $S \leftarrow S + T$ |
| $i \leftarrow i + 1$ | $i \leftarrow i + 1$ |
| **until** $i > n$ | **until** $T < \epsilon$ |
| **return** $S$ | **return** $S$ |

$$
\begin{aligned}
&a = bq_1 + r_1, \quad 0 \le r_1 < b \\
&b = r_1q_2 + r_2, \quad 0 \le r_2 < r_1 \\
&r_1 = r_2q_3 + r_3, \quad 0 \le r_3 < r_2 \\
&\vdots \\
&r_{i-2} = r_{i-1}q_i + r_i, \quad 0 \le r_i < r_{i-1} \\
&\vdots \\
&r_{n-1} = r_nq_{n+1} + r_{n+1}, \quad r_{n+1} = 0
\end{aligned}
$$

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** | $r \leftarrow d$ MOD $i$ |
| $d \leftarrow i$ | $d \leftarrow i$ |
| $i \leftarrow r$ | $i \leftarrow r$ |
| $r \leftarrow d$ 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])                  maxim(real x[1..m])
real min                             real max
integer i                            integer i
min ← x[1]                           max ← x[1]
for i ← 2, n do                      for i ← 2, m do
  if min > x[i] then                   if max < x[i] then
    min ← x[i]                           max ← x[i]
  end if                               end if
end for                              end for
return min                           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: $4, 5, 3, 6, 1, 2 \to 4, 5, 3, 6, 2, 1 \to 1, 2, 6, 3, 5, 4 \to 1, 2, 6, 4, 5, 3 \to 1, 2, 3, 5, 4, 6 \to 1, 2, 3, 5, 6, 4 \to 1, 2, 3, 4, 6, 5 \to 1, 2, 3, 4, 5, 6$ 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

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

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
```

## 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 [2]: *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)
integer S
S ← 0
while n > 0 do
  S ← S + n MOD 10
  n ← n DIV 10
end while
return S
```

```
inversare_cifre(integer n)
integer m
m ← 0
while n > 0 do
  m ← m * 10 + n MOD 10
  n ← n DIV 10
end while
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 $\epsilon$.

*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 $\epsilon$) 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
```

# Capitolul 2

# Verificarea corectitudinii algoritmilor

Analiza unui algoritm presupune verificarea câtorva caracteristici ale acestuia: *corectitudine*, *eficienţă*, *simplitate*, *claritate*. Dintre acestea primele două pot fi evaluate în manieră obiectivă pe când ultimele au şi o componentă subiectivă care este mai dificil de evaluat. În acest capitol este prezentată o introducere în verificarea corectitudinii unui algoritm, iar în capitolul următor este prezentat modul în care poate fi evaluată eficienţa unui algoritm.

## 2.1 Etapele verificării corectitudinii

Un algoritm este considerat corect dacă permite obţinerea rezultatului problemei pornind de la datele iniţiale ale acesteia. Pentru a obţine informaţii privind abilitatea unui algoritm de a rezolva problema pentru care a fost proiectat se poate alege una dintre următoarele variante:

- *Experimentală.* Se testează algoritmul pentru diverse instanţe ale problemei. Principalul avantaj al acestei abordări îl reprezintă faptul că nu necesită tehnici speciale pentru a fi aplicată pe când principalul dezavantaj îl reprezintă faptul că testarea nu poate acoperi întotdeauna toate variantele posibile de date de intrare. Varianta experimentală permite uneori identificarea situaţiilor pentru care algoritmul nu funcţionează corect însă nu garantează întotdeauna corectitudinea.

- *Analitică.* Se demonstrează că algoritmul funcţionează corect pentru orice date de intrare. Principalul avantaj îl reprezintă faptul că este garantată corectitudinea pentru orice date de intrare. Dezavantajul este reprezentat de faptul că uneori este dificil de găsit o demonstraţie. Abordarea analitică poate însă conduce la o mai bună înţelegere a algoritmului

şi la identificarea porţiunilor ce conţin eventuale erori. Demonstrarea corectitudinii poate fi dificilă în cazul algoritmilor complecşi. În această situaţie algoritmul se descompune în subalgoritmi şi se demonstrează pentru fiecare în parte corectitudinea.

În verificarea analitică a corectitudinii se parcurg următoarele etape principale:

- Identificarea proprietăţilor datelor de intrare (*precondiţiile problemei*).
- Identificarea proprietăţilor pe care trebuie să le satisfacă datele de ieşire (*postcondiţiile problemei*).
- Demonstrarea faptului că pornind de la precondiţii şi aplicând prelucrările specificate în algoritm se ajunge la satisfacerea postcondiţiilor.

În analiza corectitudinii este utilă noţiunea de *stare* a unui algoritm înţeleasă ca ansamblul valorilor pe care le au variabilele prelucrate în cadrul algoritmului la un moment dat (pas al prelucrării). Ideea verificării este de a stabili care trebuie să fie starea la fiecare moment astfel încât în final să fie satisfăcute postcondiţiile. O dată stabilite aceste stări intermediare este suficient să se verifice că fiecare prelucrare asigură transformarea stării care o precede în cea care o succede. Starea algoritmului este modificată prin atribuiri de valori variabilelor. În cazul prelucrărilor secvenţiale (succesiuni de atribuiri), verificarea este în general simplă constând în a analiza succesiv efectul fiecărei atribuiri asupra stării algoritmului. Principala sursă de erori o reprezintă prelucrările repetitive pentru care există riscul de a specifica în mod greşit iniţializările, prelucrările din corpul ciclului sau condiţia de oprire (continuare). Demonstrarea corectitudinii unei prelucrări repetitive se bazează pe principiul inducţiei matematice.
**Exemplu.** Considerăm următorul algoritm, care determină minimul unui şir finit de numere reale:

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

Enunţul problemei nu impune nici o restricţie asupra tabloului $a[1..n]$ astfel că setul de precondiţii este constituit doar din $\{n \geq 1\}$ (şirul nu este vid). Postcondiţia este reprezentată de proprietatea valorii minime: $min \leq a[i]$ pentru orice $i = \overline{1, n}$. Rămâne să arătăm că postcondiţia rezultă în urma aplicării

algoritmului. Demonstrăm prin inducţie matematică după $i$ că $min \leq a[i]$, $i = \overline{1,n}$. Cum $min = a[1]$ şi valoarea lui $min$ este înlocuită doar cu una mai mică rezultă că $min \leq a[1]$. Presupunem că $min \leq a[k]$ pentru orice $k = \overline{1,i}$. Rămâne să arătăm că $min \leq a[i+1]$. La pasul $i$ al ciclului **for** valoarea lui $min$ se modifică astfel:

- Dacă $min \leq a[i+1]$ atunci $min$ rămâne nemodificat.
- Dacă $min > a[i+1]$ atunci $min$ se înlocuieşte cu $a[i+1]$.

Astfel în oricare dintre variante se obţine că $min \leq a[i+1]$. Conform metodei inducţiei matematice rezultă că postcondiţia este satisfăcută, deci, în ipoteza că se ajunge la un rezultat, acesta va fi corect. Să considerăm acum algoritmul:

```
minim(real a[1..n])
  for i ← 1, n − 1 do
    if a[i] < a[i + 1] then
      min ← a[i]
    else
      min ← a[i + 1]
    end if
  end for
  return min
```

În acest caz prelucrarea din corpul ciclului conduce la $min = \min\{a[i], a[i+1]\}$ astfel că nu se mai poate demonstra pornind de la $min = \min\{a[1..i]\}$ că $min = \min\{a[1..i+1]\}$. Dealtfel este uşor de găsit un contraexemplu (de exemplu şirul $(2, 1, 3, 5, 4)$) pentru care algoritmul de mai sus nu funcţionează corect. În schimb se poate demonstra că algoritmul returnează $\min\{a[n-1], a[n]\}$.

## 2.2 Elemente de analiză formală a corectitudinii

Pentru verificarea analitică a corectitudinii algoritmilor există metode formale bazate pe logica Floyd-Hoare. Aceste metode folosesc următoarea strategie:

- pe baza precondiţiilor şi postcondiţiilor problemei se stabilesc pentru fiecare prelucrare *aserţiuni* privind valorile datelor şi relaţiile dintre ele;
- pentru fiecare prelucrare se demonstrează că asigură satisfacerea aserţiunii care o urmează dacă este satisfăcută aserţiunea care o precede.

O noţiune importantă în acest context este cea de *triplet Hoare* definit ca fiind $\langle P, A, Q\rangle$, unde $P$ reprezintă precondiţiile problemei, $A$ algoritmul, iar $Q$ postcondiţiile. Se spune că tripletul $\langle P, A, Q\rangle$ reprezintă un algoritm corect, şi se notează acest lucru cu $P \xrightarrow{A} Q$, dacă următoarea afirmaţie este adevărată:

Dacă datele problemei satisfac precondiţiile $P$ atunci:
(i) rezultatul satisface postcondiţiile $Q$;
(ii) algoritmul $A$ se termină după un număr finit de paşi.

Dacă algoritmul satisface condiţia (i) însă nu s-a demonstrat că satisface şi condiţia (ii) (numită *condiţie de terminare*) atunci este denumit algoritm *parţial corect*. În cazul în care se demonstrează ambele proprietăţi atunci algoritmul este considerat *complet corect*.

Verificarea corectitudinii complete a unui algoritm este un demers dificil în principal datorită dificultăţii demonstrării faptului că algoritmul se termină. Adesea demonstrarea terminării unui algoritm este echivalentă cu rezolvarea unor probleme încă deschise în matematică. De exemplu este relativ uşor de proiectat şi de demonstrat corectitudinea parţială a unui algoritm care caută un număr perfect impar (un număr natural este denumit perfect dacă coincide cu suma divizorilor săi proprii). Faptul că algoritmul se termină nu poate fi însă demonstrat la ora actuală, atât timp cât încă nu se ştie dacă un astfel de număr există.

La rândul ei, verificarea corectitudinii parţiale poate fi dificilă în cazul algoritmilor care implică multe prelucrări. În astfel de situaţii se realizează verificarea pe porţiuni până se ajunge la nivelul structurilor fundamentale de prelucrare (secvenţială, alternativă, repetitivă). Pentru fiecare dintre aceste structuri de prelucrare există reguli care uşurează verificarea.

**Regula structurii secvenţiale.** Fie algoritmul $A$ constituit din succesiunea de prelucrări $(A_1, A_2, \ldots, A_n)$. Notăm cu $P_{i-1}$ şi $P_i$ starea algoritmului înainte şi respectiv după efectuarea prelucrării $A_i$. Regula structurii secvenţiale poate fi enunţată astfel:

Dacă $P \Rightarrow P_0$, $P_{i-1} \xrightarrow{A_i} P_i$ pentru $i = \overline{1,n}$ şi $P_n \Rightarrow Q$ atunci $P \xrightarrow{A} Q$

Această regulă afirmă că dacă precondiţia implică starea iniţială, fiecare prelucrare din secvenţă conduce de la aserţiunea care o precede la cea care o succede, iar ultima aserţiune implică postcondiţia atunci secvenţa este corectă.
*Exemplu.* Fie $a$ şi $b$ două valori, iar $x$ şi $y$ două variabile ce conţin valorile $a$ şi $b$. Să se interschimbe valorile celor două variabile.

Precondiţiile problemei sunt: $P = \{x = a, y = b\}$ iar postcondiţiile sunt $Q = \{x = b, y = a\}$. Considerăm următoarele prelucrări:

| $x \leftrightarrow y$ | |
|---|---|
| $A_1$ : | $aux \leftarrow x$ |
| $A_2$ : | $x \leftarrow y$ |
| $A_3$ : | $y \leftarrow aux$ |

Aserţiunile privind stările algoritmului sunt următoarele: $P_0 = \{x = a, y = b\}$ (înainte de $A_1$), $P_1 = \{aux = a, x = a, y = b\}$ (după $A_1$), $P_2 = \{aux = a, x = b, y = b\}$ (după $A_2$), $P_3 = \{aux = a, x = b, y = a\}$ (după $A_3$). Este uşor de remarcat că $P = P_0$, $P_3 \Rightarrow Q$ şi $P_{i-1} \xrightarrow{A_i} P_i$ pentru $i = \overline{1,3}$. Conform regulii structurii secvenţiale rezultă că secvenţa realizează corect interschimbarea valorilor celor două variabile. Dacă însă se consideră secvenţa de prelucrări:

| | |
|---|---|
| $A_1$ : | $x \leftarrow y$ |
| $A_2$ : | $y \leftarrow x$ |

aserţiunile corespunzătoare sunt: $\{x = b, y = b\}$ (atât după $A_1$ cât şi după $A_2$). În acest caz $A_1$ şi $A_2$ nu pot conduce la satisfacerea postcondiţiilor.

Aceeaşi problemă poate fi rezolvată, în cazul variabilelor de tip numeric, fără a folosi o variabilă auxiliară efectuând prelucrările:

| | |
|---|---|
| $A_1$ : | $x \leftarrow x - y$ |
| $A_2$ : | $y \leftarrow x + y$ |
| $A_3$ : | $x \leftarrow y - x$ |

Stările algoritmului sunt următoarele: $P_0 = \{x = a, y = b\}$, $P_1 = \{x = a - b, y = b\}$, $P_2 = \{x = a - b, y = a\}$, $P_3 = \{x = b, y = a\}$. Este uşor de remarcat că $P = P_0$, $P_3 = Q$ şi $P_{i-1} \xrightarrow{A_i} P_i$ pentru $i = \overline{1,3}$.

Uneori sunt utile şi următoarele reguli:

*Întărirea precondiţiei.* Dacă $R \Rightarrow P$ şi $P \xrightarrow{A} Q$ atunci $R \xrightarrow{A} Q$ (algoritmul rămâne corect dacă se particularizează datele problemei).

*Slăbirea postcondiţiei.* Dacă $P \xrightarrow{A} Q$ şi $Q \Rightarrow R$ atunci $P \xrightarrow{A} R$ (algoritmul rămâne corect dacă se restrâng cerinţele problemei).

*Proprietăţi însoţitoare.* Dacă $P_1 \xrightarrow{A} Q_1$ şi $P_2 \xrightarrow{A} Q_2$ atunci $P_1 \wedge P_2 \xrightarrow{A} Q_1 \wedge Q_2$.

**Regula structurii alternative.** Considerăm structura:

`S:` **if** $c$ **then** $A_1$ **else** $A_2$ **end if**

Dacă $P$ şi $Q$ sunt precondiţiile respectiv postcondiţiile, atunci regula poate fi enunţată în modul următor:

> Dacă $c$ este bine definită (poate fi evaluată), $P \wedge c \xrightarrow{A_1} Q$ şi $P \wedge \overline{c} \xrightarrow{A_2} Q$ atunci $P \xrightarrow{A} Q$.

Regula sugerează că trebuie verificată corectitudinea fiecărei ramuri (atât când $c$ este adevărată cât şi în cazul în care este falsă).

*Exemplu.* Considerăm problema determinării minimului dintre două valori reale, distincte:

---

**if** $a < b$ **then**
  $A_1$ : $m \leftarrow a$ $\quad \{ a < b, m = a\}$
**else**
  $A_2$ : $m \leftarrow b$ $\quad \{ a \geq b, m = b\}$
**end if**

---

Precondiţiile sunt $P = \{a \in \mathbb{R}, b \in \mathbb{R}, a \neq b\}$ iar postcondiţia este $Q = \{m = \min\{a, b\}\}$. Condiţia $c$ este $a < b$. Dacă $a < b$ atunci $m = a < b$ deci $m = \min\{a, b\}$. În caz contrar este adevărată relaţia $a > b$ iar prelucrarea $A_2$ conduce la $m = b < a$ deci din nou are loc $m = \min\{a, b\}$.

**Regula structurii repetitive.** Întrucât toate structurile repetitive pot fi reorganizate astfel încât să conţină o structură condiţionată anterior (de tip **while**) o vom analiza doar pe aceasta. Considerăm structura:

S: **while** $c$ **do** $A$ **end while**,

precondiţia $P$ şi postcondiţia $Q$. In cazul prelucrărilor repetitive intervine problema terminării algoritmului astfel că trebuie analizate două aspecte: faptul că algoritmul conduce la postcondiţie în cazul în care se termină şi faptul că algoritmul se termină după un număr finit de prelucrări. Pentru a demonstra că o structura repetitivă de tip **while** este corectă este suficient să se arate că există o aserţiune $I$ (numită *invariant* sau *proprietate invariantă* a prelucrării repetitive) asociată stării algoritmului şi o funcţie $t : \mathbb{N} \to \mathbb{N}$ (numită *funcţie de terminare* şi care depinde de numărul de ordine al iteraţiei, notat în continuare cu $p$) care satisfac proprietăţile:

(a) Aserţiunea $I$ este adevărată la începutul prelucrării repetitive ($P \Rightarrow I$).

(b) Aserţiunea $I$ este *invariantă*: dacă $I$ este adevărată înainte de efectuarea prelucrării $A$, iar $c$ este adevărată, atunci $I$ rămâne adevărată şi după efectuarea lui $A$ ($I \wedge c \xrightarrow{A} I$).

(c) La sfârşitul structurii repetitive (când $c$ devine falsă) postcondiţia $Q$ poate fi dedusă din $I$ ($I \wedge \overline{c} \Rightarrow Q$).

(d) După efectuarea prelucrării $A$ valoarea lui $t$ descreşte ($c \wedge (t(p) = k) \xrightarrow{A} t(p+1) < k$)

(e) Dacă $c$ este adevărată, atunci $t(p) \geq 1$ (mai există cel puţin o iteraţie), iar când valoarea lui $t$ este 0 condiţia $c$ devine falsă (algoritmul se termină după un număr finit de iteraţii).

Elementul esenţial în demonstrarea corectitudinii unei prelucrări repetitive îl reprezintă găsirea proprietăţii invariante. Aceasta este o aserţiune particulară privind relaţiile dintre variabilele algoritmului, care este adevărată înainte de intrarea in prelucrarea repetitivă, rămâne adevărată după fiecare iteraţie, iar când condiţia $c$ devine falsă, implică postcondiţiile. Găsirea unui invariant elimină necesitatea demonstraţiei explicite prin inducţie matematică. Din perspectiva demonstrării corectitudinii condiţia cea mai importantă pe care trebuie să o satisfacă un invariant este să implice postcondiţia la sfârşitul prelucrării repetitive.

Dacă se identifică invariantul unei prelucrări repetitive atunci aceasta este cel puţin parţial corectă. Identificarea unei funcţii de terminare este fie foarte simplă (în special în cazul prelucrărilor repetitive ce folosesc un contor) fie foarte dificilă.

**Exemplul 2.1** Reluăm problema determinării minimului dintr-un şir finit de valori, rescrisă de această dată folosind **while** şi analizăm algoritmul corespunzător.

| `minim`(**real** $a[1..n]$) | | |
|---|---|---|
| $A_1$ : | $min \leftarrow a[1]$ | $\{\ min = a[1]\ \}$ |
| $A_2$ : | $i \leftarrow 2$ | $\{\ min = a[1],\ i = 2\ \}$ |
| | **while** $i \leq n$ **do** | |
| $A_3$ : | **if** $min > a[i]$ | |
| | **then** $min \leftarrow a[i]$ | |
| | **end if** | $\{\ min \leq a[j],\ j = \overline{1,i}\}$ |
| $A_4$ : | $i \leftarrow i + 1$ | $\{\ min \leq a[j],\ j = \overline{1,i-1}\}$ |
| | **end while** | |
| | **return** $min$ | $\{\ i = n+1,\ min \leq a[j],\ j = \overline{1,i-1}\}$ |

Precondiţia este $P = \{n \geq 1\}$ iar postcondiţia este $P = \{min \leq x[i],\ i = \overline{1,n}\}$. Aserţiunile marcate după fiecare prelucrare sunt uşor de stabilit. Proprietatea invariantă este $I = \{min \leq a[j],\ j = \overline{1,i-1}\}$. Într-adevăr, după prelucrarea $A_2$ proprietatea este satisfăcută. Din aserţiunea specificată după $A_4$ rezultă că proprietatea rămâne adevărată după fiecare iteraţie. Condiţia de oprire este $i = n+1$ şi se observă că atunci când este satisfăcută, aserţiunea finală implică postcondiţia.

În acest caz poate fi identificată uşor o funcţie de terminare. Notând cu $p$ contorul iteraţiei aceasta poate fi descrisă prin $t(p) = n + 1 - i_p$, unde $i_p$ este valoarea indicelui $i$ corespunzător iteraţiei $p$. Cum $i_{p+1} = i_p + 1$ rezultă că $t(p+1) < t(p)$ şi descreşterea este cu o unitate la fiecare iteraţie. Aceasta înseamnă că după un anumit număr de iteraţii $t$ se anulează. Când $t(p) = 0$ variabila $i$ satisface $i = n+1$, deci condiţia de continuare devine falsă conducând la terminarea prelucrării repetitive.

**Exemplul 2.2** Analizăm algoritmul lui Euclid pentru determinarea celui mai mare divizor comun a două numere naturale nenule:

```
cmmdc (integer a, b)
  d ← a
  i ← b
  r ← d MOD i
  while r ≠ 0 do
    d ← i
    i ← r
    r ← d MOD i
  end while
  return i
```

Precondiţiile sunt $P = \{a, b \in \mathbb{N}^*\}$, iar postcondiţia este $Q = \{i = \text{cmmdc}(a, b)\}$. Considerăm proprietatea $I$ ca fiind $\text{cmmdc}(a, b) = \text{cmmdc}(d, i)$ şi funcţia $t$ definită prin $t(p) = r_p$ unde $r_p$ este restul obţinut la iteraţia $p$. Inainte de intrarea în prelucrarea repetitivă este adevărată aserţiunea $\{d = a, i = b\}$ deci $\text{cmmdc}(a, b) = \text{cmmdc}(d, i)$. Când condiţia de continuare devine falsă ($r = 0$) se obţine că $\text{cmmdc}(i, r) = i$, adică postcondiţia $\text{cmmdc}(a, b) = i$.

Pentru a arăta că $I$ este o proprietate invariantă este suficient să arătăm că dacă $r \neq 0$ atunci $\text{cmmdc}(d, i) = \text{cmmdc}(i, r)$. Notăm $d_1 = \text{cmmdc}(d, i)$ şi $d_2 = \text{cmmdc}(i, r)$. Pe de altă parte $d = i \cdot q + r$. Se pot verifica uşor următoarele implicaţii:

$$d_1|d \text{ şi } d_1|i \Rightarrow d_1|d - i \cdot q = r, d_1|i \Rightarrow d_1|i \text{ şi } d_1|r \Rightarrow d_1 \leq d_2,$$

$$d_2|i \text{ şi } d_2|r \Rightarrow d_2|i \cdot q + r, d_2|i \Rightarrow d_2|d \text{ şi } d_2|r \Rightarrow d_2 \leq d_1.$$

Rezultă că $d_1 = d_2$, deci proprietatea $I$ este într-adevăr invariantă. Cum funcţia de terminare este chiar valoarea restului, iar acesta este un număr natural care descreşte de la o iteraţie la alta, va ajunge să se anuleze după un anumit număr de iteraţii. Prin urmare este satisfăcută şi condiţia de terminare.

*Observaţie.* Invarianţii pot fi utilizaţi nu numai pentru a determina corectitudinea unui algoritm ci şi ca instrument de proiectare a algoritmilor, în sensul că identificarea unei proprietăţi invariante poate fi văzută ca o etapă preliminară elaborării unui ciclu. Invariantul poate fi considerat ca o specificaţie pentru ciclul respectiv şi poate fi utilizat în identificarea prelucrărilor necesare pentru iniţializare, pentru corpul ciclului şi în identificarea condiţiei de oprire (continuare) a ciclului.

**Exemplul 2.3** Să considerăm calculul sumei, $S$, a primelor $n$ numere naturale ($n \geq 1$).

Precondiţia este $P = \{n \geq 1\}$, iar postcondiţia este $Q = \{S = 1 + 2 + ... + n\}$. Întrucât suma va fi calculată adăugând succesiv termenul curent, $i$, un invariant adecvat ar fi $S = 1+2+\ldots+i$ (exprimă faptul ca la sfârşitul iteraţiei $i$ variabila $S$ conţine suma parţiala până la termenul $i$ inclusiv). Pentru a asigura validitatea acestui invariant termenul curent trebuie pregătit chiar înainte de a fi adăugat. Pe de altă parte pentru a ne asigura că la finalul ciclului postcondiţia este satisfăcută rezultă că la ieşirea din ciclu variabila $i$ trebuie să aibă valoarea $n$. Astfel condiţia de continuare a ciclului **while** ar trebui să fie $i < n$. Toate aceste observaţii conduc la algoritmul `suma1`.

| `suma1` (**integer** n) | `suma2` (**integer** n) |
|---|---|
| $i \leftarrow 1$ | $S \leftarrow 0$ |
| $S \leftarrow 1$ | $i \leftarrow 1$ |
| **while** $i < n$ **do** | **while** $i \leq n$ **do** |
| $i \leftarrow i + 1$ | $S \leftarrow S + i$ |
| $S \leftarrow S + i$ | $i \leftarrow i + 1$ |
| **end while** | **end while** |
| **return** $S$ | **return** $S$ |

Varianta cea mai frecvent utilizată în practică este cea descrisă în algoritmul `suma2`. Motivaţia provine din faptul că această variantă provine din descrierea clasică bazată pe utilizarea instrucţiunii **for**. Un invariant corespunzător acestei variante este $S = 1 + 2 + ... + (i - 1)$, iar o funcţie de terminare este $t(p) = (n + 1) - i_p$.

## 2.3 Probleme

**Problema 2.1** Să se demonstreze că secvenţa următoare asigură interschimbarea valorilor variabilelor $x$ şi $y$.

1: $x \leftarrow x + y$
2: $y \leftarrow x - y$
3: $x \leftarrow x - y$

*Indicaţie.* Se foloseşte regula structurii secvenţiale.

**Problema 2.2** Să se descrie un algoritm pentru calculul factorialului unui număr natural şi să se verifice corectitudinea lui.

*Rezolvare.* Pentru un număr natural $n$, valoarea factorialului $n! = 1 \cdot 2 \cdots n$ poate fi calculată prin oricare dintre variantele descrise în continuare (algoritmii `factorial1` şi `factorial2`).

| `factorial1`(**integer** $n$) | `factorial2`(**integer** $n$) |
|---|---|
| 1: $f \leftarrow 1$ | 1: $f \leftarrow 1$ |
| 2: $i \leftarrow 1$ | 2: $i \leftarrow 1$ |
| 3: **while** $i < n$ **do** | 3: **while** $i \leq n$ **do** |
| 4: $\quad i \leftarrow i + 1$ | 4: $\quad f \leftarrow f * i$ |
| 5: $\quad f \leftarrow f * i$ | 5: $\quad i \leftarrow i + 1$ |
| 6: **end while** | 6: **end while** |
| 7: **return** $f$ | 7: **return** $f$ |

Precondiţia problemei este $P = \{n \in \mathbb{N}\}$, iar postcondiţia este $Q = \{f = \prod_{i=1}^{n} i\}$.

După execuţia primelor două instrucţiuni, starea algoritmului este $\{f = 1, i = 1\}$ ceea ce implică $\{f = \prod_{j=1}^{i} j\}$. În continuare analizăm algoritmul `factorial1`.

Demonstrăm că proprietatea $f = \prod_{j=1}^{i} j$ este invariantă în raport cu ciclul **while**. Proprietatea este evident satisfăcută la intrarea în ciclu. Arătăm că este adevărată şi după execuţia corpului ciclului. După execuţia liniei 4 (incrementarea lui $i$) proprietatea poate fi scrisă $f = \prod_{j=1}^{i-1} j$ (pentru că $f$ conţine produsul numerelor până la valoarea variabilei $i$ de dinainte de incrementare). Prin execuţia liniei 5, $f$ se înmulţeşte cu valoarea variabilei $i$, devenind din nou $f = \prod_{j=1}^{i} j$, prin urmare este invariantă în raport cu ciclul. Rămâne să demonstrăm că la sfârşitul ciclului această proprietate implică postcondiţia. Într-adevăr, la ieşirea din ciclu variabila $i$ are valoarea $n$, iar proprietatea devine $f = \prod_{j=1}^{n} j$, adică postcondiţia. Prin urmare algoritmul de mai sus este unul parţial corect.

Rămâne de demonstrat finitudinea. În acest scop este suficient să găsim o funcţie de terminare $t : \mathbb{N} \to \mathbb{N}$ (în raport cu contorul implicit, $p$, al ciclului) care este strict descrescătoare şi care este nulă când condiţia de continuare este falsă. O astfel de funcţie este $t(p) = n - i_p$ unde $i_p$ este valoarea variabilei $i$ corespunzătoare celei de a $p$-a execuţii a ciclului.

În cazul algoritmului `factorial2` se poate demonstra similar corectitudinea folosind ca invariant $f = \prod_{j=1}^{i-1} j$ iar ca funcţie de terminare $t(p) = (n+1) - i_p$.

**Problema 2.3** Scrieţi un algoritm pentru calculul puterii întregi a unei valori reale nenule ($a^p$ cu $a \in \mathbb{R}^*$ şi $p \in \mathbb{Z}$) şi demonstraţi corectitudinea algoritmului.

*Indicaţie.* Se calculează $a^{|p|} = \prod_{i=1}^{|p|} a$ şi se analizează semnul lui $p$ pentru a decide dacă se returnează $a^{|p|}$ sau $1/a^{|p|}$. Pentru demonstrarea corectitudinii se poate folosi ideea de la algoritmul de calcul al factorialului.

**Problema 2.4** Să se analizeze corectitudinea algoritmilor `conversie_10_q` şi `conversie_q_10` care realizează conversia unui număr din baza 10 în baza $q$ ($q \geq 2$) respectiv din baza $q$ în baza 10.

| `conversie_10_q`(**integer** $n$, $q$) | `conversie_q_10`(**integer** $c[0..k], q$) |
|---|---|
| **integer** $c[0..k]$,$k, m$ | **integer** $i, n$ |
| 1: $m \leftarrow n$ | 1: $i \leftarrow k$ |
| 2: $k \leftarrow 0$ | 2: $n \leftarrow c[i]$ |
| 3: $c[k] \leftarrow m$ MOD $q$ | 3: **while** $i > 0$ **do** |
| 4: $m \leftarrow m$ DIV $q$ | 4: $\quad i \leftarrow i - 1$ |
| 5: **while** $m > 0$ **do** | 5: $\quad n \leftarrow n * q + c[i]$ |
| 6: $\quad k \leftarrow k + 1$ | 6: **end while** |
| 7: $\quad c[k] \leftarrow m$MOD$q$ | 7: **return** $n$ |
| 8: $\quad m \leftarrow m$DIV$q$ | |
| 9: **end while** | |
| 10: **return** $c[0..k]$ | |

*Rezolvare.* In cazul primului algoritm, $k$ reprezintă numărul de cifre în reprezentarea în baza $q$ a numărului $n$ iar tabloul $c[0..k]$ conţine cifrele reprezentării începând cu cea mai puţin semnificativă. Cu aceste notaţii, precondiţiile sunt $P = \{n \in \mathbb{N},\ q \in \mathbb{N},\ q \geq 2\}$, iar postcondiţia este $n = c[k]q^k + c[k-1]q^{k-1} + \ldots + c[1]q + c[0]$.

După execuţia primelor patru instrucţiuni, starea algoritmului este $\{n = m \cdot q + c[k], k = 0\}$ adică $\{n = m \cdot q^{k+1} + c[k] \cdot q^k\}$. Intuim că proprietatea invariantă este: $n = m \cdot q^{k+1} + \sum_{i=0}^{k} c[i] \cdot q^i$. Se observă imediat că pentru $k = 0$ este echivalentă cu starea algoritmului la intrarea în ciclul **while**.

După execuţia liniei 6 proprietatea devine $n = m \cdot q^k + \sum_{i=0}^{k-1} c[i] \cdot q^i$.

După execuţia liniilor 7 şi 8 vechea valoare a lui $m$ ($m_p$) poate fi exprimată prin noua valoare a lui $m$ ($m_{p+1}$) astfel: $m_p = m_{p+1} \cdot q + c[k]$ motiv pentru care proprietatea analizată devine iar $n = m \cdot q^{k+1} + \sum_{i=0}^{k} c[i] \cdot q^i$. Rămâne să arătăm că la finele ciclului **while** această proprietate implică postcondiţia. Intr-adevăr, la ieşirea din **while**, valoarea lui $m$ este 0 deci proprietatea devine: $n = \sum_{i=0}^{k} c[i] \cdot q^i$ adică tocmai postcondiţia.

In ceea ce priveşte finitudinea, se observă uşor că funcţia $t(p) = m_p$ ($m_p$ este valoarea variabilei $m$ la a $p$-a execuţie a ciclului) satisface proprietăţile unei funcţii de terminare.

Algoritmul `conversie_q_10` se bazează pe faptul că determinarea valorii în baza 10 corespunzătoare reprezentării în baza $q$, $c[0..k]$, este echivalentă cu calculul sumei $\sum_{i=0}^{k} c[i]q^i$. Precondiţia se rezumă la $P = \{k > 0\}$ iar postcondiţia este $Q = \{n = \sum_{i=0}^{k} c[i]q^i\}$. După execuţia primelor două linii, starea algoritmului este $n = c[k] = c[k] \cdot q^{k-i} = \sum_{j=i}^{k} c[j]q^{j-i}$. Intuim că $n = \sum_{j=i}^{k} c[j]q^{j-i}$ este proprietate invariantă. Intrucât la intrarea în ciclu este adevărată este suficient să arătăm că rămâne adevărată după execuţia corpului ciclului şi că la final implică postcondiţia.

După execuţia liniei 4 proprietatea devine (în raport cu noua valoare a variabilei $i$): $n = \sum_{j=i+1}^{k} c[j]q^{j-i-1}$. Prin execuţia liniei 5 valoarea variabilei $n$ se modifică astfel că devine din nou adevărată relaţia $n = \sum_{j=i}^{k} c[j]q^{j-i}$. La ieşirea din ciclu variabila $i$ are valoarea 0, astfel că se obţine $n = \sum_{j=0}^{k} c[j]q^j$ adică postcondiţia. Finitudinea rezultă imediat remarcând faptul că $t(p) = i_p$ are proprietăţile unei funcţii de terminare.

**Problema 2.5** Demonstraţi, punând în evidenţă un invariant şi o funcţie de terminare, că după execuţia algoritmului de mai jos variabila $n$ va conţine valoarea în baza 10 corespunzătoare şirului binar $b[0..k]$ ($b[i] \in \{0,1\}$, $i = \overline{0,k}$).

```
alg(integer b[0..k])
integer n, i
 1: i ← 0
 2: n ← 0
 3: while i ≤ k do
 4:     n ← n * 2 + b[k − i]
 5:     i ← i + 1
 6: end while
 7: return n
```

*Indicaţie.* Se arată că $n = \sum_{j=0}^{i-1} b[k-j]2^{k-j}$ este proprietate invariantă iar $t(p) = (k+1) - i_p$ este funcţie de terminare.

**Problema 2.6** Să se demonstreze corectitudinea algoritmului de determinare a valorii obţinute prin inversarea ordinii cifrelor unui număr natural. Considerând că $n = c_k c_{k-1} \dots c_1 c_0$ este numărul iniţial, valoarea căutată este $m = c_0 c_1 \dots c_k$. Prin urmare precondiţia este: $n = \sum_{i=0} c_i 10^i$ iar postcondiţia este $m = \sum_{i=0} c_i 10^{k-i}$. Metoda de calcul a lui $m$ este descrisă în algoritmul `inversare` descris în continuare.

```
inversare(integer n)
integer m, p
 1: m ← 0
 2: p ← 0
 3: while n > 0 do
 4:     p ← p + 1
 5:     m ← m * 10 + n MOD 10
 6:     n ← n DIV 10
 7: end while
 8: return m
```

*Rezolvare.* Utilizarea variabilei $p$ nu este necesară pentru descrierea algoritmului însă introducerea ei uşurează demonstrarea corectitudinii. Să con-

siderăm relaţiile: $\{n = \sum_{i=p}^{k} c[i]10^{i-p},\ m = \sum_{i=0}^{p-1} c[i]10^{p-1-i}\}$. Se observă că precondiţia implică prima relaţie (pentru $p = 0$) iar a doua este evident adevărată pentru $m = 0$ şi $p = 0$ ($m$ este descrisă în acest caz ca o sumă vidă). După execuţia liniei 4 relaţiile devin: $\{n = \sum_{i=p-1}^{k} c[i]10^{i-p+1},\ m = \sum_{i=0}^{p-2} c[i]10^{p-2-i}\}$.
După execuţia liniei 5, a doua relaţie devine $m = \sum_{i=0}^{p-2} c[i]10^{p-1-i} + c[p-1] = \sum_{i=0}^{p-1} c[i]10^{p-1-i}\}$. După execuţia liniei 6 obţinem că $n = \sum_{i=p}^{k} c[i]10^{i-p}$. Când $n$ este 0 înseamnă că $p = k+1$ astfel că $m = \sum_{i=0}^{p-1} c[i]10^{p-1-i}$ implică postcondiţia.

Finitudinea este asigurată de faptul că oricare dintre funcţiile $t(p) = n_p$ sau $t(p) = k + 1 - p$ satisface proprietăţile unei funcţii de terminare.

**Problema 2.7** Să se demonstreze că algoritmul `produs` calculează corect produsul a două numere naturale nenule.

```
produs(integer a,b)
integer s
 1: x ← a
 2: y ← b
 3: p ← 0
 4: s ← 0
 5: while x > 0 do
 6:    p ← p + 1
 7:    if x MOD 2=1 then
 8:       s ← s + y
 9:    end if
10:    x ← x DIV 2
11:    y ← 2 * y
12: end while
13: return s
```

*Indicaţie.* Se observă că algoritmul corespunde metodei de înmulţire descrisă în Problema 1.1. Pornim de la ipoteza că valoarea $a$ poate fi reprezentată în baza 2 prin $k+1$ cifre binare $c_k c_{k-1} ... c_1 c_0$ specificate în tabloul $c[0..k]$.

Cu aceste notaţii precondiţia este $a = c_k 2^k + c_{k-1} 2^{k-1} \dots c_1 2 + c_0$, iar postcondiţia $s = ab$. Proprietatea invariantă este constituită din relaţiile: $\{s = (\sum_{i=0}^{p-1} c_i 2^i)b,\ x = \sum_{i=p}^{k} c_i 2^{i-p},\ y = b2^p\}$. Este uşor de verificat că aceste relaţii sunt satisfăcute înainte de intrarea în ciclu. Prin execuţia liniei 6 relaţiile devin: $\{s = (\sum_{i=0}^{p-2} c_i 2^i)b,\ x = \sum_{i=p-1}^{k} c_i 2^{i-p+1},\ y = b2^{(p-1)}\}$. Prin execuţia liniei 7 se modifică prima relaţie, devenind: $s = (\sum_{i=0}^{p-1} c_i 2^i)b$. Prin execuţia liniei 8, a doua relaţie devine $x = \sum_{i=p}^{k} c_i 2^{i-p}$ iar prin execuţia liniei 9 a treia relaţie devine $y = b2^p$. Prin urmare relaţiile sunt invariante în raport cu ciclul. La

părăsirea ciclului valoarea lui $x$ este 0, iar $p = k + 1$, astfel că prima relaţie implică postcondiţia.

Finitudinea se demonstrează folosind ca funcţie de terminare pe $t(p) = x_p$.

**Problema 2.8** Să se demonstreze că algoritmul `adunare` descris în continuare corespunde adunării în baza 2 a două numere reprezentate în binar pe $n + 1$ poziţii.

```
adunare(integer a[0..n], b[0..n])
integer c[0..n + 1], s
 1: c[0..n + 1] ← 0
 2: i ← 0
 3: while i ≤ n do
 4:     s ← a[i] + b[i] + c[i + 1]
 5:     c[i] ← s MOD 2
 6:     c[i + 1] ← s DIV 2
 7:     i ← i + 1
 8: end while
 9: return  c[0..n + 1]
```

*Indicaţie.* Notând cu $x_{0..i}$ valoarea corespunzătoare tabloului de cifre binare $x[0..i]$ se poate arăta că proprietatea $\{c_{0..i} = a_{0..i-1} + b_{0..i-1}\}$ este invariantă în raport cu ciclul, iar la ieşirea din ciclu implică $c_{0..(n+1)} = a_{0..n} + b_{0..n}$ adică tabloul $c[0..n+1]$ conţine suma valorilor corespunzătoare reprezentărilor binare din $a[0..n]$ şi $b[0..n]$.

**Problema 2.9** Se consideră algoritmul `alg` descris în continuare. Să se identifice o proprietate invariantă corespunzătoare ciclului şi să se stabilească care este efectul algoritmului.

```
alg(real a[1..n])
integer i
 1: i ← 1
 2: while i ≤ n − 1 do
 3:     if a[i] > a[i + 1] then
 4:         a[i] ↔ a[i + 1]
 5:     end if
 6:     i ← i + 1
 7: end while
 8: return  a[1..n]
```

*Indicaţie.* După prima execuţie a corpului ciclului va fi satisfăcută proprietatea $a[1] \leq a[2]$. După a două execuţie a ciclului va fi satisfăcută proprietatea $a[2] \leq a[3]$ (de remarcat faptul că valoarea elementului de pe poziţia 2 nu este neapărat

aceeaşi cu cea obţinută după prima etapă a algoritmului ceea ce înseamnă că nu e în mod necesar adevărată afirmaţia că $a[1] \leq a[2] \leq a[3]$, ci doar afirmaţia că $a[3] \geq a[2]$ şi $a[3] \geq a[1]$). Acestea sugerează că după execuţia pasului $i$ al ciclului este adevărată afirmaţia $a[i] = \max_{j=\overline{1,i}} a[j]$. Să demonstrăm că această afirmaţie este într-adevăr un invariant al ciclului. Pentru $i = 1$ este implicit satisfăcută. După execuţia liniei 3 devine $a[i+1] = \max_{j=\overline{1,i+1}} a[j]$. După execuţia liniei 4 devine iar adevărată afirmaţia $a[i] = \max_{j=\overline{1,i}} a[j]$ prin urmare aceasta este o proprietate invariantă a ciclului. La părăsirea ciclului, $i$ va fi $n$ deci se obţine că $a[n] = \max_{j=\overline{1,n}} a[j]$, adică efectul algoritmului este că aduce valoarea maximă a tabloului pe ultima poziţie.

**Problema 2.10** Propuneţi un algoritm care transformă un tablou $a[1..n]$ prin interschimbări de elemente vecine astfel încât valoarea minimă să ajungă pe prima poziţie a tabloului. Demonstraţi corectitudinea algoritmului identificând un invariant şi o funcţie de terminare.

*Indicaţie.* Se parcurge tabloul pornind de la ultimul element şi se compară fiecare element cu predecesorul său. Dacă elementul curent este mai mic decât predecesorul atunci se interschimbă elementele. Invariantul este similar cu cel de la problema anterioară.

**Problema 2.11** Să se demonstreze că algoritmul `inversare_sir` descris în continuare realizează inversarea ordinii elementelor din tabloul $x[1..n]$ primit ca parametru.

```
inversare_sir(a[1..n])
1: i ← 0
2: while i ≤ ⌊(n + 1)/2⌋ do
3:     i ← i + 1
4:     x[i] ↔ x[n + 1 − i]
5: end while
6: return x[1..n]
```

*Indicaţie.* Fie $x_0[1..n]$ conţinutul iniţial al tabloului. Cu această notaţie precondiţia poate fi specificată prin $P = \{x[i] = x_0[i], i = \overline{1,n}\}$ iar postcondiţia prin $Q = \{x[i] = x_0[n+1-i], i = \overline{1,n}\}$. Se poate demonstra că următoarele relaţii $\{i \leq n+1-i,\ x[j] = x_0[n+1-j],\ x[n+1-j] = x_0[j],\ j = \overline{1,i}\}$ sunt invariante în raport cu ciclul **while** iar la ieşirea din ciclu ($i = \lfloor (n+1)/2 \rfloor$) implică postcondiţia.

**Problema 2.12** Identificaţi proprietăţi invariante pentru prelucrările repetitive din algoritmii `alg1` şi `alg2` descrişi în continuare şi stabiliţi ce returnează fiecare dintre ei când este apelat pentru valori naturale nenule.

| `alg1`(**integer** $a, b$) | `alg2`(**integer** $a, b$) |
|---|---|
| **integer** $x, y, z$ | **integer** $x, y, z$ |
| 1: $x \leftarrow a$ | 1: $x \leftarrow a$ |
| 2: $y \leftarrow b$ | 2: $y \leftarrow b$ |
| 3: $z \leftarrow 0$ | 3: $z \leftarrow 0$ |
| 4: **while** $y > 0$ **do** | 4: **while** $x \geq y$ **do** |
| 5: $z \leftarrow z + x$ | 5: $x \leftarrow x - y$ |
| 6: $y \leftarrow y - 1$ | 6: $z \leftarrow z + 1$ |
| 7: **end while** | 7: **end while** |
| 8: **return** $z$ | 8: **return** $z$,$x$ |

*Indicaţie.* Notăm cu $p$ contorul prelucrării iterative ($p$ va avea valoarea 0 înainte de intrarea în ciclu şi este incrementat cu 1 la fiecare execuţie a ciclului). Folosind această notaţie (chiar dacă $p$ nu este variabilă explicită în cadrul algoritmului) se observă că pentru `alg1` relaţiile $\{y = b - p,\ z = px,\ x = a\}$ sunt invariante în raport cu prelucrarea repetitivă. La ieşirea din ciclul **while** variabila $y$ are valoarea 0, prin urmare $p = b$ iar $z = px = ba$. Deci algoritmul `alg1` returnează produsul $ab$. In cazul algoritmului `alg2` relaţiile $\{x = a - pb,\ y = b,\ z = p\}$ sunt invariante. La ieşirea din ciclu $a = z \cdot b + x$ şi $x < y = b$. Aceasta înseamnă că $z$ va conţine câtul împărţirii lui $a$ la $b$, iar $x$ restul aceleiaşi împărţiri.

**Problema 2.13** Să se demonstreze că algoritmii `cmmdc1` şi `cmmdc2` descrişi în continuare determină cel mai mare divizor comun al numerelor nenule $a$ şi $b$.

| `cmmdc1` (**integer** $a, b$) | `cmmdc2` (**integer** $a, b$) |
|---|---|
| 1: **while** $a \neq 0$ **and** $b \neq 0$ **do** | 1: **while** $a \neq b$ **do** |
| 2: $a \leftarrow a$ MOD $b$ | 2: **if** $a > b$ **then** |
| 3: **if** $a \neq 0$ **then** | 3: $a \leftarrow a - b$ |
| 4: $b \leftarrow b$ MOD $a$ | 4: **else** |
| 5: **end if** | 5: $b \leftarrow b - a$ |
| 6: **end while** | 6: **end if** |
| 7: **if** $a \neq 0$ **then** | 7: **end while** |
| 8: $d \leftarrow a$ | 8: $d \leftarrow a$ |
| 9: **else** | 9: **return** $d$ |
| 10: $d \leftarrow b$ | |
| 11: **end if** | |
| 12: **return** $d$ | |

*Indicaţie.* Dacă notăm cu $a_0$ şi $b_0$ valorile iniţiale ale variabilelor $a$ respectiv $b$ atunci precondiţiile sunt $P = \{a = a_0, b = b_0\}$ iar postcondiţia este $Q = \{d = \text{cmmdc}(a_0, b_0)\}$. Pentru ambele variante de algoritm proprietatea invariantă este cmmdc$(a, b)$=cmmdc$(a_0, b_0)$. Demonstrarea proprietăţii de invarianţă se

bazează pe proprietăţi cunoscute ale celui mai mare divizor comun şi anume, cmmdc$(a, b)$=cmmdc$(a$ MOD $b, b)$=cmmdc$(a, b$ MOD $a)$ respectiv cmmdc$(a, b)$ = cmmdc$(a - b, b)$ (dacă $a > b$) şi cmmdc$(a, b)$=cmmdc$(a, b - a)$ (dacă $a < b$). In ceea ce priveşte demonstrarea terminării aceasta se bazează pe două şiruri strict descrescătoare de numere naturale: $t(p) = \min(a_p, b_p)$ şi $t(p) = |a_p - b_p|$.

**Problema 2.14** Se consideră un tablou $x[1..n]$ care conţine valoarea $x_0$. Să se demontreze că: (a) algoritmul `cauta1` determină prima poziţie pe care se află valoarea $x_0$; (b) algoritmul `cauta2` determină toate poziţiile pe care se află $x_0$ în tabloul $x$.

| `cauta1` $(x[1..n],x0)$ | `cauta2` $(x[1..n],x0)$ |
|---|---|
| $i \leftarrow 1$ | $i \leftarrow 1$ |
| **while** $x[i] \neq x0$ **do** | $m \leftarrow 0$ |
| $i \leftarrow i + 1$ | **while** $i \leq n$ **do** |
| **end while** | **if** $x[i] = x0$ **then** |
| **return** $i$ | $m \leftarrow m + 1$ |
| | $poz[m] = i$ |
| | **end if** |
| | $i \leftarrow i + 1$ |
| | **end while** |
| | **return** $poz[1..m]$ |

*Indicaţie.* Pentru algoritmul `cauta1` se poate folosi ca invariant relaţia $\{x[j] \neq x_0,\ j = \overline{1, i-1}\}$ iar pentru algoritmul `cauta2` $\{poz[k] = x_{i_k},\ k = \overline{1, m}$ (unde $\{i_k,\ k = \overline{1, m}\}$ reprezintă poziţiile din tablou pe care se află valoarea căutată, $x_0$).

# Capitolul 3

# Analiza complexităţii algoritmilor

Analiza complexităţii unui algoritm are ca scop estimarea volumului de *resurse de calcul* necesare pentru execuţia algoritmului. Prin resurse se înţelege:

- *Spaţiul de memorie* necesar pentru stocarea datelor pe care le prelucrează algoritmul.
- *Timpul* necesar pentru execuţia tuturor prelucrărilor specificate în algoritm.

Această analiză este utilă pentru a stabili dacă un algoritm utilizează un volum acceptabil de resurse pentru rezolvarea unei probleme. În caz contrar algoritmul, chiar dacă este corect, nu este considerat eficient şi nu poate fi aplicat în practică. Analiza complexităţii, numită şi analiza eficienţei algoritmilor, este utilizată şi în compararea algoritmilor cu scopul de a-l alege pe cel mai eficient (cel care foloseşte cele mai puţine resurse de calcul).

În majoritatea algoritmilor volumul resurselor necesare depinde de *dimensiunea* problemei de rezolvat. Aceasta este determinată de regulă de volumul datelor de intrare. În cazul cel mai general acesta este dat de numărul biţilor necesari reprezentării datelor. Dacă se prelucrează o valoare numerică, $n$ (de exemplu, se verifică dacă $n$ este număr prim) atunci ca dimensiune a problemei se consideră numărul de biţi utilizaţi în reprezentarea lui $n$, adică $\lfloor \log_2 n \rfloor + 1$. Dacă datele de prelucrat sunt organizate sub forma unor tablouri atunci dimensiunea problemei poate fi considerată ca fiind numărul de componente ale tablourilor (de exemplu la determinarea minimului dintr-un tablou cu $n$ elemente sau în calculul valorii unui polinom de gradul $n$ se consideră că dimensiunea problemei este $n$). Sunt situaţii în care volumul datelor de intrare este specificat prin mai multe valori (de exemplu în prelucrarea unei matrici cu $m$

linii şi $n$ coloane). În aceste cazuri dimensiunea problemei este reprezentată de toate valorile respective (de exemplu, $(m, n)$).

Uneori la stabilirea dimensiunii unei probleme trebuie să se ţină cont şi de prelucrările ce vor fi efectuate asupra datelor. De exemplu, dacă prelucrarea unui text se efectuează la nivel de cuvinte atunci dimensiunea problemei va fi determinată de numărul cuvintelor, pe când dacă se efectuează la nivel de caractere atunci va fi determinată de numărul de caractere.

Spaţiul de memorare este influenţat de modul de reprezentare a datelor. De exemplu, o matrice cu elemente întregi având 100 de linii şi 100 de coloane din care doar 50 sunt nenule (matrice rară) poate fi reprezentată în una dintre variantele: (i) tablou bidimensional $100 \times 100$ (10000 de valori întregi); (ii) tablou unidimensional în care se reţin doar cele 50 de valori nenule şi indicii corespunzători (150 de valori întregi). Alegerea unui mod eficient de reprezentare a datelor poate influenţa complexitatea prelucrărilor. De exemplu algoritmul de adunare a două matrici rare devine mai complicat în cazul în care acestea sunt reprezentate prin tablouri unidimensionale. În general obţinerea unor algoritmi eficienţi din punct de vedere al timpului de execuţie necesită mărirea spaţiului de memorie alocat datelor şi reciproc.

Dintre cele două resurse de calcul, spaţiu şi timp, cea critică este timpul de execuţie. În continuare vom analiza doar dependenţa dintre timpul de execuţie (înţeles de cele mai multe ori ca număr de repetări ale unor operaţii) al unui algoritm şi dimensiunea problemei.

## 3.1 Timp de execuţie

În continuare vom nota cu $T(n)$ timpul de execuţie al unui algoritm destinat rezolvării unei probleme de dimensiune $n$. Pentru a estima timpul de execuţie trebuie stabilit un *model de calcul* şi o unitate de măsură. Vom considera un model de calcul (numit şi maşină de calcul cu acces aleator) caracterizat prin:

- Prelucrările se efectuează în mod secvenţial.
- Operaţiile *elementare* sunt efectuate în timp constant *indiferent* de valoarea operanzilor.
- Timpul de acces la informaţie nu depinde de poziţia acesteia (nu sunt diferenţe între prelucrarea primului element al unui tablou şi cea a oricărui alt element).

A stabili o unitate de măsură înseamnă a stabili care sunt operaţiile elementare şi a considera ca unitate de măsură timpul acestora de execuţie. În acest fel timpul de execuţie va fi exprimat prin numărul de operaţii elementare executate. Sunt considerate operaţii elementare cele aritmetice (adunare, scădere, înmulţire, împărţire), comparaţiile şi cele logice (negaţie, conjuncţie şi

| | | | |
|---|---|---|---|
| 1: $S \leftarrow 0$ | | | |
| 2: $i \leftarrow 0$ | Operaţie | Cost | Nr. repetări |
| 3: **while** $i < n$ **do** | 1 | $c_1$ | 1 |
| 4: $\quad i \leftarrow i + 1$ | 2 | $c_2$ | 1 |
| 5: $\quad S \leftarrow S + i$ | 3 | $c_3$ | $n + 1$ |
| 6: **end while** | 4 | $c_4$ | $n$ |
| 7: **return** $S$ | 5 | $c_5$ | $n$ |

Tabelul 3.1: Analiza costurilor în cazul algoritmului de calcul a sumei primelor $n$ numere naturale

disjuncţie). Cum scopul calculului timpului de execuţie este de a permite compararea algoritmilor, uneori este suficient să se contorizeze doar anumite tipuri de operaţii elementare, numite *operaţii de bază* sau *operaţii dominante* (de exemplu în cazul unui algoritm de căutare sau de sortare se pot contoriza doar operaţiile de comparare) şi/sau să se considere că timpul de execuţie a acestora este unitar (desi operaţiile de înmmulţire şi împărţire sunt mai costisitoare decât cele de adunare şi scădere în analiză se poate considera că ele au acelaşi cost).

Timpul de execuţie al întregului algoritm se obţine însumând timpii de execuţie ai prelucrărilor componente.

**Exemplul 3.1** Considerăm problema calculului sumei $\sum_{i=1}^{n} i$. Dimensiunea acestei probleme poate fi considerată $n$. Algoritmul şi tabelul cu costurile corespunzătoare prelucrărilor sunt prezentate în Tabelul 3.1. Însumând timpii de execuţie ai prelucrărilor elementare se obţine: $T(n) = n(c_3 + c_4 + c_5) + c_1 + c_2 + c_3 = k_1 n + k_2$, adică timpul de execuţie depinde liniar de dimensiunea problemei. Costurile operaţiilor elementare influenţează doar constantele ce intervin în funcţia $T(n)$.

Calculul sumei poate fi realizat şi utilizând o prelucrare de tip **for** $i \leftarrow 1, n$ **do** $S \leftarrow S + i$ **endfor**. Costul gestiunii contorului (iniţializare, testare şi incrementare) fiind $c_2 + (n + 1)c_3 + nc_4$. În cazul în care toate prelucrările au cost unitar se obţine $2(n + 1)$ pentru costul gestiunii contorului unui ciclu **for** cu $n$ iteraţii. Ipoteza costului unitar nu este în întregime corectă întrucât iniţializarea constă într-o simplă atribuire iar incrementarea într-o adunare şi o atribuire. Totuşi pentru a simplifica analiza o vom utiliza în continuare.

**Exemplul 3.2** Considerăm problema determinării produsului a două matrici: $A$ de dimensiune $m \times n$ şi $B$ de dimensiune $n \times p$. În acest caz dimensiunea problemei este determinată de trei valori: $(m, n, p)$. Algoritmul şi analiza costurilor este prezentată în Tabelul 3.2.

Costul prelucrărilor de pe liniile 1, 2 şi 4 reprezintă costul gestiunii contorului şi va fi tratat global. Presupunând că toate operaţiile aritmetice şi de comparare

```
1: for i ← 1, m do
2:    for j ← 1, p do
3:       c[i, j] ← 0
4:       for k ← 1, n do
5:          c[i, j] ← c[i, j] + a[i, k] ∗ b[k, j]
6:       end for
7:    end for
8: end for
9: return c[1..m, 1..p]
```

| Op. | Cost | Nr. repet. |
|---|---|---|
| 1 | $2(m+1)$ | 1 |
| 2 | $2(p+1)$ | $m$ |
| 3 | 1 | $m \cdot p$ |
| 4 | $2(n+1)$ | $m \cdot p$ |
| 5 | 2 | $m \cdot p \cdot n$ |

Tabelul 3.2: Analiza costurilor în cazul produsului a două matrici

```
1: m ← x[1]
2: for i ← 2, n do
3:    if m > x[i] then
4:       m ← x[i]
5:    end if
6: end for
7: return m
```

| Op. | Cost | Nr. repet. |
|---|---|---|
| 1 | 1 | 1 |
| 2 | $2n$ | 1 |
| 3 | 1 | $n-1$ |
| 4 | 1 | $\tau(n)$ |

Tabelul 3.3: Analiza costurilor în cazul algoritmului de determinare a minimului dintr-un tablou

au cost unitar, se obţine timpul de execuţie $T(m, n, p) = 4mnp+5mp+4m+2$.

În practică nu este necesară o analiză atât de detaliată ci este suficient să se identifice *operaţia dominantă* şi să se estimeze numărul de repetări ale acesteia. Prin operaţie dominantă se înţelege operaţia care contribuie cel mai mult la timpul de execuţie a algoritmului şi de regulă este operaţia ce apare în ciclul cel mai interior. În exemplul de mai sus ar putea fi considerată ca operaţie dominantă, operaţia de înmulţire. În acest caz costul execuţiei algoritmului ar fi $T(m, n, p) = mnp$.

**Exemplul 3.3** Considerăm problema determinării valorii minime dintr-un tablou $x[1..n]$. Dimensiunea problemei este dată de numărul $n$ de elemente ale tabloului. Prelucrările algoritmului şi costurile corespunzătoare sunt prezentate în Tabelul 3.3.

Spre deosebire de exemplele anterioare timpul de execuţie nu poate fi calculat explicit întrucât numărul de repetări ale prelucrării numerotate cu 4 depinde de valorile aflate în tablou. Dacă cea mai mică valoare din tablou se află chiar pe prima poziţie atunci prelucrarea 4 nu se efectuează nici o dată iar $\tau(n) = 0$. Acesta este considerat cazul cel mai *favorabil*.

Dacă, în schimb, elementele tabloului sunt în ordine strict descrescătoare atunci prelucrarea 4 se efectuează la fiecare iteraţie adică $\tau(n) = n - 1$. Acesta este

```
1:  gasit ← false
2:  i ← 1                                          Operaţie  Cost  Nr. repet.
3:  while (gasit =false) and i ≤ n do              1         1     1
4:    if v = x[i] then                             2         1     1
5:      gasit ← true                               3         3     τ1(n) + 1
6:    else                                         4         1     τ1(n)
7:      i ← i + 1                                  5         1     τ2(n)
8:    end if                                       6         1     τ3(n)
9:  end while
10: return gasit
```

Tabelul 3.4: Analiza costurilor în cazul căutării secvenţiale

cazul cel mai *defavorabil*.

Timpul de execuţie poate fi astfel încadrat între două limite: $3n \leq T(n) \leq 4n-1$. În acest caz este uşor de observat că se poate considera ca operaţie dominantă cea a comparării dintre valoarea variabilei $m$ şi elementele tabloului. În acest caz costul algoritmului ar fi $T(n) = n-1$. In ambele situaţii dependenţa timpului de execuţie de dimensiunea problemei este liniară.

**Exemplul 3.4** Considerăm problema căutării unei valori $v$ printre elementele unui tablou, $x[1..n]$. Dimensiunea problemei este din nou $n$ iar o primă variantă a algoritmului şi costurile corespunzătoare algoritmului sunt prezentate în Tabelul 3.4.

În cazul în care valoarea $v$ se află în tablou notăm cu $k$ prima poziţie pe care se află. Se obţine:

$$\tau_1(n) = \begin{cases} k & \text{valoarea se află în şir} \\ n & \text{valoarea nu se află în şir} \end{cases} \qquad \text{deci } 1 \leq \tau_1(n) \leq n.$$

$$\tau_2(n) = \begin{cases} 1 & \text{valoarea se află în şir} \\ 0 & \text{valoarea nu se află în şir} \end{cases}.$$

$$\tau_3(n) = \begin{cases} k-1 & \text{valoarea se află în şir} \\ n & \text{valoarea nu se află în şir} \end{cases} \qquad \text{deci } 0 \leq \tau_3(n) \leq n.$$

Cazul cel mai favorabil este cel în care valoarea se află pe prima poziţie în tablou, caz în care $T(n) = 3(\tau_1(n)+1) + \tau_1(n) + \tau_2(n) + \tau_3(n) + 2 = 6 + 1 + 1 + 0 + 2 = 10$.

Cazul cel mai defavorabil este cel în care valoarea nu se află în tablou: $T(n) = 3(n+1) + n + 0 + n + 2 = 5(n+1)$.

| | | | |
|---|---|---|---|
| 1: $i \leftarrow 1$ | | | |
| 2: **while** $(x[i] \neq v)$ **and** $(i < n)$ **do** | Operaţie | Cost | Nr. repet. |
| 3: $i \leftarrow i + 1$ | 1 | 1 | 1 |
| 4: **end while** | 3 | 3 | $\tau(n) + 1$ |
| 5: **if** $x[i] = v$ **then** | 4 | 1 | $\tau(n)$ |
| 6: $gasit \leftarrow$**true** | 5-8 | 2 | 1 |
| 7: **else** | | | |
| 8: $gasit \leftarrow$**false** | | | |
| 9: **end if** | | | |
| 10: **return** $gasit$ | | | |

Tabelul 3.5: Analiza costurilor în cazul căutării secvenţiale

O altă variantă a algoritmului de căutare este descrisă în Tabelul 3.5. Dacă există $k$ astfel încât $x[k] = v$ atunci $\tau(n) = k - 1$. In acest caz numărul total de operaţii este $4k + 2$. În cazul cel mai favorabil numărul de operaţii este 6, iar în cazul cel mai defavorabil numărul de operaţii este $4n + 2$.

### 3.1.1 Analiza cazurilor extreme

Aşa cum rezultă din analiza algoritmului de căutare prezentat în Exemplul 3.4, numărul de operaţii executate depinde uneori nu doar de dimensiunea problemei ci şi de proprietăţile datelor de intrare. Astfel pentru instanţe diferite ale problemei se execută un număr diferit de prelucrări. In astfel de situaţii numărul de operaţii executate nu poate fi calculat exact, astfel că se calculează margini ale acestuia analizând cele două cazuri extreme: cazul cel mai *favorabil* şi cazul cel mai *defavorabil*.

Cazul favorabil corespunde acelor instanţe ale problemei pentru care numărul de operaţii efectuate este cel mai mic. Analiza în cazul cel mai favorabil permite identificarea unei limite inferioare a timpului de execuţie. Această analiză este utilă pentru a identifica algoritmi ineficienţi (dacă un algoritm are un cost mare în cel mai favorabil caz, atunci el nu poate fi considerat o variantă acceptabilă). Sunt situaţii în care frecvenţa instanţelor corespunzătoare celui mai favorabil caz sau apropiate acestuia este mare. In astfel de situatii estimările obţinute în cel mai favorabil caz furnizează informaţii utile despre algoritm. De exemplu algoritmul de sortare prin inserţie, analizat în Capitolul 4, se comportă bine în cazul în care tabloul este deja sortat, iar această comportare rămâne valabilă şi pentru tablouri "aproape" sortate.

Cazul cel mai defavorabil corespunde instanţelor pentru care numărul de operaţii efectuate este maxim. Analiza acestui caz furnizează o limită superioară a timpului de execuţie. In aprecierea şi compararea algoritmilor interesează în special cel mai defavorabil caz deoarece furnizează cel mai mare timp de execuţie relativ la *orice* date de intrare de dimensiune dată. Pe de altă parte

pentru anumiţi algoritmi cazul cel mai defavorabil este relativ frecvent.

### 3.1.2 Analiza cazului mediu

Uneori, cazurile extreme (cel mai defavorabil şi cel mai favorabil) se întâlnesc rar, astfel că analiza acestor cazuri nu furnizează suficientă informaţie despre algoritm. In aceste situaţii este utilă o altă măsură a complexităţii algoritmilor şi anume *timpul mediu de execuţie*. Acesta reprezintă o valoare medie a timpilor de execuţie calculată în raport cu distribuţia de probabilitate corespunzătoare spaţiului datelor de intrare. Stabilirea acestei distribuţii de probabilitate presupune împărţirea mulţimii instanţelor posibile ale problemei în clase astfel încât pentru instanţele din aceeaşi clasă numărul de operaţii efectuate să fie acelaşi.

Dacă $\nu(n)$ reprezintă numărul cazurilor posibile (clase de instanţe pentru care algoritmul efectuează acelaşi număr de operaţii), $P_k$ este probabilitatea de apariţie a cazului $k$ iar $T_k(n)$ este timpul de execuţie corespunzător cazului $k$ atunci timpul mediu este dat de relaţia:

$$T_m(n) = \sum_{k=1}^{\nu(n)} T_k(n) P_k.$$

Dacă toate cazurile sunt echiprobabile ($P_k = 1/\nu(n)$) se obţine $T_m(n) = \sum_{k=1}^{\nu(n)} T_k(n)/\nu(n)$.

**Exemplu.** Considerăm din nou problema căutării unei valori $v$ într-un tablou $x[1..n]$ (exemplul 3.4). Pentru a simplifica analiza vom considera că elementele tabloului sunt distincte. Pentru a calcula timpul mediu de execuţie trebuie să facem ipoteze asupra distribuţiei datelor de intrare.

Dificultatea principală în stabilirea timpului mediu constă în stabilirea distribuţiei corespunzătoare spaţiului datelor de intrare. Pentru exemplul în discuţie s-ar putea lua în considerare şi ipoteza că probabilitatea ca valoarea $v$ să se afle în tablou este $P(v \text{ se află în tablou}) = p$, iar probabilitatea ca valoarea să nu se afle în tablou este $P(v \text{ nu se află în tablou}) = 1-p$. In plus considerăm că în cazul în care se află în tablou elementul căutat se găseşte cu aceeaşi probabilitate pe oricare dintre cele $n$ poziţii: $P(v \text{ se află pe poziţia k}) = 1/n$. În acest caz timpul mediu este:

$$T_m(n) = \frac{p}{n} \sum_{k=1}^{n} 5(k+1) + 5(1-p)(n+1) = \frac{(10-5p)n + (5p+10)}{2}$$

Dacă şansa ca elementul să se afle în şir coincide cu cea ca el să nu se afle atunci se obţine $T_m(n) = (15n+25)/4$.

O altă ipoteză asupra distribuţiei de probabilitate corespunzătoare diferitelor instanţe ale problemei este să considerăm că valoarea $v$ se poate afla pe oricare

dintre poziţiile din tablou sau în afara acestuia cu aceeaşi probabilitate. Cum numărul cazurilor posibile este $\nu(n) = n+1$ ($n$ cazuri în care valoarea se află în cadrul tabloului şi unul în care $v$ nu se află în tablou) rezultă că probabilitatea fiecărui caz este $1/(n+1)$. Cum timpul corespunzător cazului în care $v$ se află pe poziţia $k$ este $T_k = 5(k+1)$ iar cel corespunzător cazului în care valoarea nu se află în tablou este $T_{n+1} = 5(n+1)$ rezultă că timpul mediu de execuţie este:

$$T_m(n) = \frac{1}{n+1}\left(\sum_{k=1}^{n} 5(k+1) + 5(n+1)\right) = \frac{5(n^2+5n+2)}{2(n+1)}$$

Această ipoteză este naturală în cazul în care se analizează o variantă a algoritmului caracterizată prin faptul că se adaugă la tablou un element care coincide cu valoarea căutată. Aceasta variantă se bazează pe *tehnica fanionului* şi este descrisă în algoritmul 3.1. Algoritmul returnează poziţia pe care se află valoarea căutată. Dacă poziţia returnată este $n+1$ atunci înseamnă că valoarea căutată nu se află în tabloul iniţial.

**Algoritmul 3.1** Căutare secvenţială folosind tehnica fanionului

```
1: x[n+1] ← v
2: i ← 1
3: while x[i] ≠ v do
4:     i ← i + 1
5: end while
6: return i
```

Pentru cazul în care valoarea căutată se află pe poziţia $k$ ($k \in \{1, \ldots, n+1\}$) numărul de operaţii efectuate este $T_k(n) = 2k+1$. Astfel timpul mediu este $T_m(n) = \left(\sum_{k=1}^{n+1}(2k+1)\right)/(n+1) = n+3$.

Timpul mediu de execuţie depinde de ipotezele făcute asupra distribuţiei datelor de intrare şi în general nu este o simplă medie aritmetică a timpilor corespunzători cazurilor extreme (cel mai favorabil respectiv cel mai defavorabil).

Datorită dificultăţilor ce pot interveni în estimarea timpului mediu şi datorită faptului că în multe situaţii acesta diferă de timpul în cazul cel mai defavorabil doar prin valori ale constantelor implicate, de regulă analiza se limitează la estimarea timpului în cazul cel mai defavorabil. Timpul mediu are semnificaţie atunci când pentru problema în studiu cazul cel mai defavorabil apare rar.

## 3.2 Ordin de creştere

Pentru a aprecia eficienţa unui algoritm nu este necesară cunoaşterea expresiei detaliate a timpului de execuţie. Mai degrabă interesează modul în care timpul de execuţie creşte o dată cu creşterea dimensiunii problemei. O măsură utilă în acest sens este *ordinul de creştere*. Acesta este determinat de *termenul dominant* din expresia timpului de execuţie. Când dimensiunea problemei este mare valoarea termenului dominant depăşeşte semnificativ valorile celorlalţi termeni astfel că aceştia din urmă pot fi neglijaţi. Pentru a stabili ordinul de creştere al timpului de execuţie al unui algoritm este suficient să se contorizeze operaţia dominantă (cea mai costisitoare şi mai frecvent executată).

*Exemple.* Dacă $T(n) = an + b$ $(a > 0)$ când dimensiunea problemei creşte de $k$ ori şi termenul dominant din timpul de execuţie creşte de acelaşi număr de ori căci $T(kn) = k \cdot (an) + b$. În acest caz este vorba despre un ordin liniar de creştere. Dacă $T(n) = an^2+bn+c$ $(a > 0)$, atunci $T(kn) = k^2 \cdot (an^2)+k \cdot (bn)+c$, deci termenul dominant creşte de $k^2$ ori, motiv pentru care spunem că este un ordin pătratic de creştere.

Dacă $T(n) = a\lg n$ atunci $T(kn) = a \lg n + a \lg k$, adică termenul dominant nu se modifică, timpul de execuţie crescând cu o constanta (în raport cu $n$). În relaţiile anterioare şi in toate cele ce vor urma prin lg notăm logaritmul în baza 2 (întrucât trecerea de la o bază la alta este echivalentă cu înmulţirea cu o constantă ce depinde doar de bazele implicate, iar în stabilirea ordinului de creştere se ignoră constantele, în analiza eficienţei baza logaritmilor nu este relevantă). Un ordin logaritmic de creştere reprezintă o comportare bună. Dacă în schimb $T(n) = a2^n$ atunci $T(kn) = a(2^n)^k$ adică ordinul de creştere este exponenţial.

Întrucât problema eficienţei devine critică pentru probleme de dimensiuni mari analiza eficienţei se efectuează pentru valori mari ale lui $n$ (teoretic se consideră că $n \to \infty$), în felul acesta luându-se în considerare doar comportarea termenului dominant. Acest tip de analiză se numeşte *analiză asimptotică*. În cadrul analizei asimptotice se consideră că un algoritm este mai eficient decât altul dacă ordinul de creştere al timpului de execuţie al primului este mai mic decât cel al celui de-al doilea.

Relaţia între ordinele de creştere are semnificaţie doar pentru dimensiuni mari ale problemei. Dacă considerăm timpii $T_1(n) = 10n + 10$ şi $T_2(n) = n^2$ atunci se observă cu uşurinţă că $T_1(n) > T_2(n)$ pentru $n \leq 10$, deşi ordinul de creştere al lui $T_1$ este evident mai mic decât cel al lui $T_2$ (Figura 3.1).

Prin urmare un algoritm asimptotic mai eficient decât altul reprezintă varianta cea mai bună doar în cazul problemelor de dimensiuni mari.

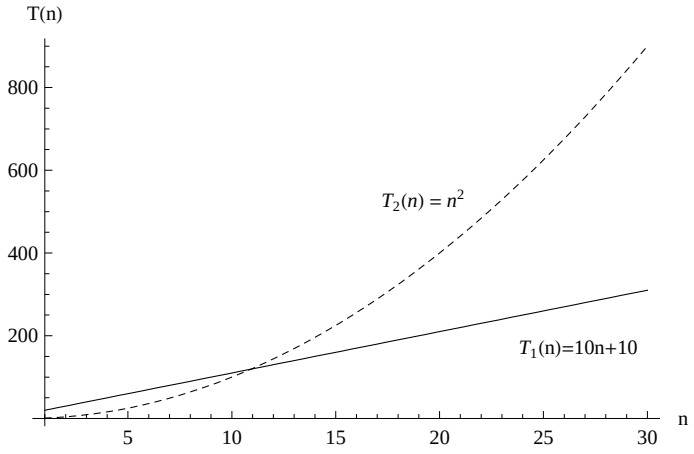


Figura 3.1: Ordin liniar şi ordin pătratic de creştere

## 3.3 Notaţii asimptotice

Pentru a uşura analiza asimptotică şi pentru a permite gruparea algoritmilor în clase în funcţie de ordinul de creştere a timpului de execuţie (făcând abstracţie de eventualele constante ce intervin în expresiile detaliate ale timpilor de execuţie) s-au introdus nişte clase de funcţii şi notaţii asociate.

### 3.3.1 Notaţia $\Theta$

**Definiţia 3.1** *Pentru o funcţie* $g : \mathbb{N} \to \mathbb{R}_+$, $\Theta(g(n))$ *reprezintă mulţimea de funcţii:*

$$\begin{aligned} \Theta(g(n)) &= \{f : \mathbb{N} \to \mathbb{R}_+; \exists c_1, c_2 \in \mathbb{R}_+^*, n_0 \in \mathbb{N} \text{ astfel încât} \\ &\quad c_1 g(n) \leq f(n) \leq c_2 g(n), \forall n \geq n_0\} \end{aligned} \tag{3.1}$$

Despre timpul de execuţie al unui algoritm, $T(n)$, se spune că este de ordinul $\Theta(g(n))$ dacă $T(n) \in \Theta(g(n))$. Prin abuz de notaţie în algoritmică se obişnuieşte să se scrie $T(n) = \Theta(g(n))$. Din punct de vedere intuitiv faptul că $f(n) \in \Theta(g(n))$ înseamnă că $f(n)$ şi $g(n)$ sunt asimptotic echivalente, adică au acelaşi ordin de creştere. Altfel spus $\lim_{n\to\infty} f(n)/g(n) = k$, $k$ fiind o valoare finită strict pozitivă.

În figura 3.2 este ilustrată această idee folosind reprezentările grafice ale funcţiilor $f(n) = n^2 + 5n\lg n + 10$ şi $g(n) = c_i n^2$, $c_i \in \{1, 2\}$, pentru diverse domenii de variaţie ale lui $n$ ($n \in \{1, \ldots, 5\}$, $n \in \{1, \ldots, 50\}$ respectiv $n \in \{1, \ldots, 500\}$) şi valorile $c_1 = 1$ şi $c_2 = 4$. Pentru aceste valori ale constantelor

$c_1$ şi $c_2$ se observă din grafic că este suficient să considerăm $n_0 = 3$ pentru a arăta că $f \in \Theta(g(n))$. Constanta $c_2 = 4$ conduce la o margine superioară relaxată. În realitate orice valoarea supraunitară poate fi considerată, însă cu cât $c_2$ este mai mică cu atât $n_0$ va fi mai mare.

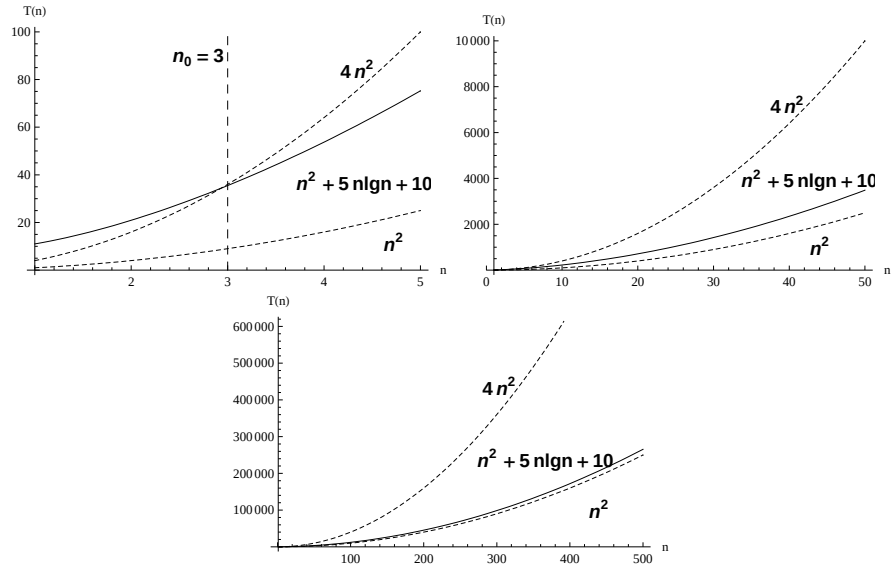


Figura 3.2: Graficele funcţiilor $f$ (linie continuă), $c_1g$,$c_2g$ (linie punctată)

**Exemplu.** Pentru Exemplul 3.1 (calcul sumă) s-a obţinut $T(n) = k_1n + k_2$ ($k_1 > 0$, $k_2 > 0$) prin urmare pentru $c_1 = k_1$, $c_2 = k_1 + 1$ şi $n_0 > k_2$ se obţine că $c_1n \leq T(n) \leq c_2n$ pentru $n \geq n_0$, adică $T(n) \in \Theta(n)$.

Pentru Exemplul 3.3 (determinare minim) s-a obţinut că $3n \leq T(n) \leq 4n-1$ prin urmare $T(n) \in \Theta(n)$ (este suficient să se considere $c_1 = 3$, $c_2 = 4$ şi $n_0 = 1$).

**Propoziţia 3.1** *Notaţia* $\Theta$ *are următoarele proprietăţi:*

*(i) Dacă* $T(n) = a_kn^k + a_{k-1}n^{k-1} + \ldots + a_1n + a_0$, $a_k > 0$ *atunci* $T(n) \in \Theta(n^k)$.

*(ii)* $\Theta(cg(n)) = \Theta(g(n))$ *pentru orice constantă strict pozitivă,* $c$. *Ca o consecinţă a acestei proprietăţi rezultă că* $\Theta(\log_a(n)) = \Theta(\log_b(n))$ *pentru orice valori reale pozitive* $a$ *şi* $b$ *(strict mai mari decât* 1*).*

*(iii)* $f(n) \in \Theta(f(n))$ *(reflexivitate).*

*(iv) Dacă* $f(n) \in \Theta(g(n))$ *atunci* $g(n) \in \Theta(f(n))$ *(simetrie).*

*(v) Dacă* $f(n) \in \Theta(g(n))$ *şi* $g(n) \in \Theta(h(n))$ *atunci* $f(n) \in \Theta(h(n))$ *(tranzitivitate).*

*(vi)* $\Theta(f(n) + g(n)) = \Theta(\max\{f(n), g(n)\})$.

*Demonstraţie.* (i) Într-adevăr, din $\lim_{n\to\infty} T(n)/n^k = a_k$ rezultă că pentru orice $\varepsilon > 0$ există $n_0(\varepsilon)$ cu proprietatea că $|T(n)/n^k - a_k| \leq \varepsilon$ pentru orice $n \geq n_0(\varepsilon)$. Deci

$$a_k - \varepsilon \leq \frac{T(n)}{n^k} \leq a_k + \varepsilon, \quad \forall n \geq n_0(\varepsilon)$$

adică pentru $c_1 = a_k - \varepsilon$, $c_2 = a_k + \varepsilon$ şi $n_0 = n_0(\varepsilon)$ se obţin inegalităţile din (3.1).
(ii) Proprietatea rezultă din definiţie, iar consecinţa din faptul că $\log_a(n) = \log_b(n)/\log_b(a)$.
(iii)-(vi) Toate proprietăţile rezultă simplu din definiţie.

□

Proprietatea (ii) sugerează faptul că în analiza eficienţei nu are importanţă baza logaritmului, motiv pentru care în continuare logaritmul va fi specificat generic prin lg fără a se face referire la baza lui (implicit se consideră baza 2). Proprietăţile (iii)-(v) sugerează că notaţia $\Theta$ permite definirea unei clase de echivalenţă ($f(n)$ şi $g(n)$ sunt echivalente dacă $f(n) \in \Theta(g(n))$). Clasele de echivalenţă corespunzătoare sunt numite *clase de complexitate*.

Notaţia $\Theta$ se foloseşte atunci când se poate determina explicit expresia timpului de execuţie sau timpii de execuţie corespunzători cazurilor extreme au acelaşi ordin de creştere.

În cazul Exemplului 3.4 (problema căutării) s-a obţinut că $10 \leq T(n) \leq 5(n+1)$ ceea ce sugerează că există cazuri (de exemplu, valoarea este găsită pe prima poziţie) în care numărul de operaţii efectuate nu depinde de dimensiunea problemei. În acest caz $T(n) \notin \Theta(n)$ deoarece nu poate fi găsit un $c_1$ şi un $n_0$ astfel încât $c_1 n \leq 10$ pentru orice $n \geq n_0$. În astfel de situaţii se analizează comportarea asimptotică a timpului în cazul cel mai defavorabil. În situaţia în care pentru toate datele de intrare timpul de execuţie nu depinde de volumul acestora (de exemplu în cazul algoritmului de determinare a valorii minime dintr-un şir ordonat crescător), atunci se notează $T(n) \in \Theta(1)$ (timp de execuţie constant).

### 3.3.2 Notaţia $\mathcal{O}$

**Definiţia 3.2** *Pentru o funcţie* $g : \mathbb{N} \to \mathbb{R}_+$, $\mathcal{O}(g(n))$ *reprezintă mulţimea de funcţii:*

$$\begin{aligned} \mathcal{O}(g(n)) \quad = \quad & \{f : \mathbb{N} \to \mathbb{R}_+; \exists c \in \mathbb{R}_+^*, n_0 \in \mathbb{N} \text{ astfel încât} \\ & 0 \leq f(n) \leq cg(n), \forall n \geq n_0\} \end{aligned} \tag{3.2}$$

Această clasă de funcţii permite descrierea comportării unui algoritm în cazul cel mai defavorabil fără a se face referire la celelalte situaţii. Întrucât de regulă ne interesează comportarea algoritmului pentru date arbitrare de intrare este suficient să specificăm o margine superioară pentru timpul de execuţie. Intuitiv, faptul că $f(n) \in \mathcal{O}(g(n))$ înseamnă că $f(n)$ creşte asimptotic cel mult la fel de repede ca $g(n)$. Altfel spus $\lim_{n\to\infty} f(n)/g(n) = k$, $k$ fiind o valoare pozitivă, dar nu neapărat nenulă.

Folosind definiţia se pot demonstra următoarele proprietăţi ale notatiei $\mathcal{O}$.

**Propoziţia 3.2** *Notaţia $\mathcal{O}$ are următoarele proprietăţi:*

*(i) Dacă $T(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0$, $a_k > 0$ atunci $T(n) \in \mathcal{O}(n^p)$ pentru orice $p \geq k$.*

*(ii) $f(n) \in \mathcal{O}(f(n))$ (reflexivitate).*

*(iii) Dacă $f(n) \in \mathcal{O}(g(n))$ şi $g(n) \in \mathcal{O}(h(n))$ atunci $f(n) \in \mathcal{O}(h(n))$ (tranzitivitate).*

*(iv) $\mathcal{O}(f(n) + g(n)) = \mathcal{O}(\max\{f(n), g(n)\})$.*

*(v) $\Theta(g(n)) \subset \mathcal{O}(g(n))$.*

După cum ilustrează Exemplul 3.4, incluziunea de la proprietatea (v), adică $\Theta(g(n)) \subset \mathcal{O}(g(n))$, este strictă.

Folosind definiţia lui $\mathcal{O}$ se verifică uşor că dacă $g_1(n) < g_2(n)$ pentru $n \geq n_0$ iar $f(n) \in \mathcal{O}(g_1(n))$ atunci $f(n) \in \mathcal{O}(g_2(n))$. Prin urmare dacă $T(n) \in \mathcal{O}(n)$ atunci $T(n) \in \mathcal{O}(n^d)$ pentru orice $d \geq 1$. Evident la analiza unui algoritm este util să se pună în evidenţa cea mai mică margine superioară. Astfel, pentru algoritmul din Exemplul 3.4 vom spune ca are ordinul de complexitate $\mathcal{O}(n)$ şi nu $\mathcal{O}(n^2)$ (chiar dacă afirmaţia ar fi corectă din punctul de vedere al definiţiei).

**Notaţia** *o.* Dacă în Definiţia 3.2 în locul inegalităţii $f(n) \leq cg(n)$ se specifică inegalitatea strictă $f(n) < cg(n)$ care are loc pentru orice constantă pozitivă $c$ atunci se obţine clasa $o(g(n))$. Aceasta este echivalent cu faptul că $\lim_{n\to\infty} f(n)/g(n) = 0$. Cu această notaţie putem scrie că $3n - 1 \in o(n^2)$ dar $n^2 + 3n - 1 \notin o(n^2)$ (deşi $n^2 + 3n - 1 \in \mathcal{O}(n^2)$). Această notaţie este mai puţin frecvent folosită în practică decât $\mathcal{O}$.

### 3.3.3 Notaţia $\Omega$

**Definiţia 3.3** *Pentru o funcţie $g : \mathbb{N} \to \mathbb{R}_+$, $\Omega(g(n))$ reprezintă mulţimea de funcţii:*

$$\begin{aligned} \Omega(g(n)) \quad = \quad & \{f : \mathbb{N} \to \mathbb{R}_+; \exists c \in \mathbb{R}_+^*, n_0 \in \mathbb{N} \text{ astfel încât} \\ & cg(n) \leq f(n), \forall n \geq n_0\} \end{aligned} \tag{3.3}$$

Notaţia $\Omega$ se foloseşte pentru a exprima eficienţa algoritmului pornind de la timpul de execuţie corespunzător celui mai favorabil caz. Intuitiv, faptul că $f(n) \in \Omega(g(n))$ înseamnă că $f(n)$ creşte asimptotic cel puţin la fel de repede ca $g(n)$, adică $\lim_{n\to\infty} f(n)/g(n) \geq k$, $k$ fiind o valoare strict pozitivă dar nu neapărat finită (limita poate fi chiar infinită).

**Propoziţia 3.3** *Notaţia $\Omega$ are următoarele proprietăţi:*

*(i) Dacă $T(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0$, $a_k > 0$ atunci $T(n) \in \mathcal{O}(n^p)$ pentru orice $p \leq k$.*

*(ii) $f(n) \in \Omega(f(n))$ (reflexivitate).*

*(iii) Dacă $f(n) \in \Omega(g(n))$ şi $g(n) \in \Omega(h(n))$ atunci $f(n) \in \Omega(h(n))$ (tranzitivitate).*

*(iv) $\Omega(f(n) + g(n)) = \Omega(\max\{f(n), g(n)\})$.*

*(v) $\Theta(g(n)) \subset \Omega(g(n))$.*

*(vi) Dacă $f(n) \in \mathcal{O}(g(n))$ atunci $g(n) \in \Omega(f(n))$ şi reciproc.*

Aşa cum rezultă din Exemplul 3.4 incluziunea $\Theta(g(n)) \subset \Omega(g(n))$ este strictă: $T(n) \in \Omega(1)$ - corespunde cazului în care valoarea se găseşte pe prima poziţie - însă $T(n) \notin \Theta(1)$ întrucât în cazul cel mai defavorabil timpul de execuţie depinde de $n$. Din proprietăţile (v) din Propoziţiile 3.2 şi 3.3 rezultă că $\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$.

**Notaţia $\omega$.** Dacă în Definiţia 3.3 în locul inegalităţii $cg(n) \leq f(n)$ se specifică inegalitatea strictă $cg(n) < f(n)$ care are loc pentru orice constantă pozitivă $c$ atunci se obţine clasa $\omega(g(n))$. Aceasta este echivalent cu faptul că $\lim_{n\to\infty} f(n)/g(n) = \infty$. Cu această notaţie putem scrie că $3n^2 - 1 \in \omega(n)$ dar $3n - 1 \notin \omega(n)$ (deşi $3n - 1 \in \Omega(n)$).

### 3.3.4 Analiza asimptotică a principalelor structuri de prelucrare

Considerăm problema determinării ordinului de complexitate în cazul cel mai defavorabil pentru structurile algoritmice: *secvenţială*, *alternativă* şi *repetitivă*. Presupunem că structura secvenţială este constituită din prelucrările $A_1$, ..., $A_k$ şi fiecare dintre acestea are ordinul de complexitate $\mathcal{O}(g_i(n))$. Atunci structura va avea ordinul de complexitate $\mathcal{O}(g_1(n) + \ldots + g_k(n)) = \mathcal{O}(\max\{g_1(n), \ldots, g_k(n)\})$.

Dacă evaluarea condiţiei unei structuri alternative are cost constant iar prelucrările corespunzătoare celor două variante au ordinele de complexitate $\mathcal{O}(g_1(n))$ respectiv $\mathcal{O}(g_2(n))$, atunci costul structurii alternative va fi $\mathcal{O}(\max\{g_1(n), g_2(n)\})$.

În cazul unei structuri repetitive, pentru a determina ordinul de complexitate în cazul cel mai defavorabil se consideră numărul maxim de iteraţii. Dacă acesta este $n$, iar dacă în corpul ciclului prelucrările sunt de cost constant, atunci se obţine ordinul $\mathcal{O}(n)$. În cazul unui ciclu dublu, dacă atât pentru ciclul interior cât şi pentru cel exterior limitele variază între 1 şi $n$ atunci se obţine de regulă o complexitate pătratică, $\mathcal{O}(n^2)$. Dacă însă limitele ciclului interior se modifică este posibil să se obţină un alt ordin. Să considerăm următoarea prelucrare:

```
m ← 1
for i ← 1, n do
  m ← 3 * m
  for j ← 1, m do
    prelucrare de ordin Θ(1)      //prelucrare de cost constant
  end for
end for
```

Cum pentru fiecare valoarea a lui $i$ se obţine $m = 3^i$ rezultă că timpul de execuţie este de forma $T(n) = 1 + \sum_{i=1}^{n}(3^i + 1) \in \Theta(3^n)$.

### 3.3.5 Clase de complexitate

Majoritatea algoritmilor întâlniţi în practică se încadrează în una dintre clasele menţionate în Tabelul 3.6. Ordinele de complexitate menţionate în tabel corespund celui mai defavorabil caz.

| Complexitate | Ordin | Exemplu |
|---|---|---|
| logaritmică | $\mathcal{O}(\lg n)$ | căutare binară |
| liniară | $\mathcal{O}(n)$ | căutare secvenţială |
| | $\mathcal{O}(n \lg n)$ | sortare prin interclasare |
| pătratică | $\mathcal{O}(n^2)$ | sortare prin inserţie |
| cubică | $\mathcal{O}(n^3)$ | produsul a două matrici pătratice de ordin $n$ |
| exponenţială | $\mathcal{O}(2^n)$ | prelucrarea tuturor submulţimilor unei mulţimi cu $n$ elemente |
| factorială | $\mathcal{O}(n!)$ | prelucrarea tuturor permutărilor unei mulţimi cu $n$ elemente |

Tabelul 3.6: Clase de complexitate şi exemple de algoritmi reprezentativi

În ierarhizarea algoritmilor după ordinul de complexitate sunt utile relaţiile (3.4) cunoscute din matematică.

$$\lim_{n\to\infty}\frac{(\lg n)^b}{n^k}=0,\quad \lim_{n\to\infty}\frac{n^k}{a^n}=0,\quad \lim_{n\to\infty}\frac{a^n}{n^n}=0,\quad \lim_{n\to\infty}\frac{a^n}{n!}=0\qquad (a>1) \tag{3.4}$$

Algoritmii aplicabili pentru probleme de dimensiune mare sunt doar cei din clasa $\mathcal{O}(n^k)$ ($k \ll n$ constantă) cunoscuţi sub numele de algoritmi *polinomiali*. Algoritmii de complexitate exponenţială sunt aplicabili doar pentru probleme de dimensiune mică.

Pentru stabilirea clasei (ordinului) de complexitate a unui algoritm se parcurg următoarele etape:

1. Se stabileşte dimensiunea problemei.
2. Se identifică operaţia de bază (operaţia dominantă).
3. Se verifică dacă numărul de execuţii ale operaţiei de bază depinde doar de dimensiunea problemei. Dacă da, se determină acest număr. Dacă nu, se analizează cazul cel mai favorabil, cazul cel mai defavorabil şi (dacă este posibil) cazul mediu.
4. Se stabileşte clasa de complexitate căruia îi aparţine algoritmul.

## 3.4 Analiza empirică

**Motivaţie.** Analiza teoretică a eficienţei algoritmilor poate fi dificilă în cazul unor algoritmi care nu sunt simpli, mai ales dacă este vorba de analiza cazului mediu. O alternativă la analiza teoretică a eficienţei o reprezintă *analiza empirică*.

Aceasta poate fi utilă pentru: (i) a obţine informaţii preliminare privind clasa de complexitate a unui algoritm; (ii) pentru a compara eficienţa a doi (sau mai mulţi) algoritmi destinaţi rezolvării aceleiaşi probleme; (iii) pentru a compara eficienţa mai multor implementări ale aceluiaşi algoritm; (iv) pentru a obţine informaţii privind eficienţa implementării unui algoritm pe o anumită maşină de calcul; (v) pentru a identifica porţiunile cele mai costisitoare din cadrul programului (*profilare*).

**Etapele analizei empirice.** În analiza empirică a unui algoritm se parcurg de regulă următoarele etape:

1. Se stabileşte scopul analizei.
2. Se alege metrica de eficienţă ce va fi utilizată (numărul de execuţii ale unei/unor operaţii sau timpul de execuţie a întregului algoritm sau a unei porţiuni din algoritm).

3. Se stabilesc proprietăţile datelor de intrare în raport cu care se face analiza (dimensiunea datelor sau proprietăţi specifice).

4. Se implementează algoritmul într-un limbaj de programare.

5. Se generează mai multe seturi de date de intrare.

6. Se execută programul pentru fiecare set de date de intrare.

7. Se analizează datele obţinute.

Alegerea măsurii de eficienţă depinde de scopul analizei. Dacă, de exemplu, se urmăreşte obţinerea unor informaţii privind clasa de complexitate sau chiar verificarea acurateţei unei estimări teoretice atunci este adecvată utilizarea numărului de operaţii efectuate. Dacă însă scopul este evaluarea comportării implementării unui algoritm atunci este potrivit timpul de execuţie.

Pentru a efectua o analiză empirică nu este suficient un singur set de date de intrare ci mai multe, care să pună în evidenţă diferitele caracteristici ale algoritmului. În general este bine să se aleagă date de diferite dimensiuni astfel încât să fie acoperită o plajă cât mai largă de dimensiuni. Pe de altă parte are importanţă şi analiza diferitelor valori sau configuraţii ale datelor de intrare. Dacă se analizează un algoritm care verifică dacă un număr este prim sau nu şi testarea se face doar pentru numere ce nu sunt prime sau doar pentru numere care sunt prime atunci nu se va obţine un rezultat relevant. Acelaşi lucru se poate întâmpla pentru un algoritm a cărui comportare depinde de gradul de sortare a unui tablou (dacă se aleg fie doar tablouri aproape sortate după criteriul dorit fie tablouri ordonate în sens invers analiza nu va fi relevantă).

În vederea analizei empirice la implementarea algoritmului într-un limbaj de programare vor trebui introduse secvenţe al căror scop este monitorizarea execuţiei. Dacă metrica de eficienţă este numărul de execuţii ale unei operaţii atunci se utilizează un contor care se incrementează după fiecare execuţie a operaţiei respective. Dacă metrica este timpul de execuţie atunci trebuie înregistrat momentul intrării în secvenţa analizată şi momentul ieşirii. Majoritatea limbajelor de programare oferă funcţii de măsurare a timpului scurs între două momente. Este important, în special în cazul în care pe calculator sunt active mai multe taskuri, să se contorizeze doar timpul afectat execuţiei programului analizat. În special dacă este vorba de măsurarea timpului este indicat să se ruleze programul de test de mai multe ori şi să se calculeze valoarea medie a timpilor.

La generarea seturilor de date de intrare scopul urmărit este să se obţină date tipice rulărilor uzuale (să nu fie doar excepţii). În acest scop adesea datele se generează în manieră aleatoare. În realitate este vorba de o pseudo-aleatoritate întrucât este simulată prin tehnici cu caracter determinist.

După execuţia programului pentru datele de test se înregistrează rezultatele, iar în scopul analizei fie se calculează mărimi sintetice (media, abaterea

standard etc.), fie se reprezintă grafic perechi de puncte de forma (dimensiune problema, măsură de eficienţă).

## 3.5 Analiza amortizată

Să considerăm problema numărării în baza 2 de la 0 până la $n = 2^k - 1$ considerând valoarea binară curentă stocată într-un tablou $b[0..k-1]$ ($b[0]$ reprezintă bitul cel mai puţin semnificativ iar $b[k-1]$ reprezintă bitul cel mai semnificativ). Algoritmul determinării valorii $n$ prin numărare constă în aplicarea repetată a unui algoritm de incrementare în baza 2 (Algoritmul 3.2).

**Algoritmul 3.2** Incrementare binară

```
increm(integer b[0..k-1])
integer i
1: i ← 0
2: while i < k and b[i] = 1 do
3:     b[i] ← 0
4:     i ← i + 1
5: end while
6: b[i] ← 1
7: return  b[0..k-1]
```

```
numărare(integer n, k)
integer b[0..k-1],j
1: b[0..k-1] ← 0; write b[0..k-1]
2: for j ← 1, n do
3:     b[0..k-1] ← increm(b[0..k-1])
4:     write b[0..k-1]
5: end for
```

Se pune problema determinării ordinului de complexitate în cazul cel mai defavorabil. Dimensiunea problemei este determinată de valorile $k$ şi $n$ iar operaţia dominantă este cea de schimbare a valorii unei cifre binare: $b[i]$ se transformă din 1 în 0 (linia 3 a algoritmului `increm`) sau din 0 în 1 (linia 6 a algoritmului `increm`). Dacă se analizează eficienţa algoritmului `increm` independent de contextul general al problemei atunci se observă uşor că în cazul cel mai defavorabil numărul de modificări ale unei cifre binare este $k$. Deci algoritmul `increm` aparţine lui $\mathcal{O}(k)$. Cum algoritmul `numărare` apelează algoritmul `increm` de exact $n$ ori rezultă că în cazul cel mai defavorabil se efectuează $kn$ operaţii pentru a se număra de la 1 la $n$.

Dacă însă se urmăreşte numărul de operaţii efectuate pentru $k = 4$ şi $n = 15$ (Tabelul 3.7) se observă că numărul de operaţii nu depăşeşte $2n$. Aceasta înseamnă că marginea superioară obţinută aplicând analiza clasică este prea largă. Se observă că cifra de pe poziţia 0 se modifică la fiecare iteraţie, cifra de pe poziţia 1 din în două iteraţii, cea de pe poziţia 2 din patru în patru iteraţii ş.a.m.d. Prin urmare numărul de operaţii efectuate este

$$\sum_{i=0}^{k-1} \lfloor \frac{n}{2^i} \rfloor \leq n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n$$

Deci la fiecare dintre cele $n$ iteraţii din algoritmul `numărare` se efectuează

| $j$ | $b[0..k-1]$ | | | | Nr. | $j$ | $b[0..k-1]$ | | | | Nr. |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | $b_3$ | $b_2$ | $b_1$ | $b_0$ | operaţii | | $b_3$ | $b_2$ | $b_1$ | $b_0$ | operaţii |
| 0 | 0 | 0 | 0 | 0 | 0 | 8 | **1** | **0** | **0** | **0** | 15 |
| 1 | 0 | 0 | 0 | **1** | 1 | 9 | 1 | 0 | 0 | **1** | 16 |
| 2 | 0 | 0 | **1** | **0** | 3 | 10 | 1 | 0 | **1** | **0** | 18 |
| 3 | 0 | 0 | 1 | **1** | 4 | 11 | 1 | 0 | 1 | **1** | 19 |
| 4 | 0 | **1** | **0** | **0** | 7 | 12 | 1 | **1** | **0** | **0** | 22 |
| 5 | 0 | 1 | 0 | **1** | 8 | 13 | 1 | 1 | 0 | **1** | 23 |
| 6 | 0 | 1 | **1** | **0** | 10 | 14 | 1 | 1 | **1** | **0** | 25 |
| 7 | 0 | 1 | 1 | **1** | 11 | 15 | 1 | 1 | 1 | **1** | 26 |

Tabelul 3.7: Numărul de operaţii de transformare a unei cifre binare în cazul numărării de la 0 la 15. Cifrele marcate sunt modificate la iteraţia $j$ în cadrul algoritmului `numărare`

în cazul cel mai defavorabil un număr mediu de 2 modificări ale cifrelor binare. Aceasta înseamna că în contextul utilizării în algoritmul `numărare`, algoritmul de incrementare (care luat independent este de complexitate $\mathcal{O}(k)$) poate fi considerat de complexitate $\mathcal{O}(n)/n$ ceea ce înseamnă că se poate considera ca aparţine lui $\mathcal{O}(1)$. Un cost astfel determinat este numit cost *amortizat*, iar analiza bazată pe astfel de costuri se numeşte analiză amortizată. Metoda folosită mai sus este cunoscută sub numele de analiză bazată pe *agregare*. In această metodă costul amortizat este considerat acelaşi pentru toate operaţiile (setarea unei cifre binare pe 1 respectiv setarea pe 0 au acelaşi cost). O altă variantă este cea care acordă costuri diferite pentru aceste operaţii. Intrucât numărul de operaţii este mai mare cu cât sunt mai multe cifre de 1 pe primele poziţii se asignează un cost mai mare setării unei cifre pe 1 decât setării unei cifre pe 0. De exemplu se consideră că setarea unei cifre pe 1 este cotată cu costul 2. La setarea efectivă a unei cifre pe 1 se contorizează costul efectiv scazând 1 din valoarea cotată, 2. Restul joacă rolul unui credit care este folosit la setarea unei cifre pe 0. In felul acesta costul unui apel al funcţiei `increm` poate fi considerat egal cu costul setării unei cifre pe 1 adică 2. Ideea este inspirată de amortizarea costurilor în sistemele economice, de unde provine şi denumirea.

Analiza amortizată se bazează, ca şi analiza cazului mediu, tot pe noţiunea de cost mediu, dar nu în sens statistic, nefiind astfel necesară stabilirea unei distribuţii de probabilitate pentru diferitele clase de instanţe ale problemei. Pentru detalii suplimentare privind analiza amortizată pot fi consultate [3], [6], [12].

## 3.6 Probleme

**Problema 3.1** Să se determine numărul de operaţii efectuate de către un algoritm care determină numărul de inversiuni ale unei permutări.

*Rezolvare.* Pentru o permutare $(a_1, \ldots, a_n)$, $a_i \in \{1, \ldots, n\}$, $i = \overline{1,n}$ o inversiune este o pereche de indici $(i, j)$ cu proprietatea că $i < j$ şi $a_i > a_j$. Considerând permutarea ca fiind reprezentată prin tabloul $a[1..n]$ numărul de inversiuni poate fi determinat prin algoritmul `inversiuni` descris în Tabelul 3.8.

```
inversiuni(integer a[1..n])
1: k ← 0
2: for i ← 1, n − 1 do
3:    for j ← i + 1, n do
4:       if a[i] > a[j] then
5:          k ← k + 1
6:       end if
7:    end for
8: end for
9: return k
```

| Linie | Cost | Nr. repet. |
|---|---|---|
| 1 | 1 | 1 |
| 2 | $2n$ | 1 |
| 3 | $2(n-i+1)$ | $i = \overline{1,n-1}$ |
| 4 | $n-i$ | $i = \overline{1,n-1}$ |
| 5 | $\tau(n,i)$ | $i = \overline{1,n-1}$ |

Tabelul 3.8: Algoritmul pentru determinarea numărului de inversiuni şi tabelul costurilor

Dimensiunea problemei este reprezentată de numărul de elemente ale tabloului $a$. Numărul (costul) operaţiilor elementare executate pentru fiecare linie a algoritmului este detaliat în tabelul de costuri. $\tau(n,i)$ reprezintă numărul de incrementări ale lui $k$. Acesta depinde de proprietăţile datelor de intrare şi este cuprins între 0 şi $n-i$. Însumând costurile operaţiilor elementare se obţine:

$$T(n) = 1 + 2n + 2\sum_{i=1}^{n-1}(n-i+1) + \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} 1 + \sum_{i=1}^{n-1}\tau(n,i) =$$

$$= \frac{3n(n-1)}{2} + 4n - 1 + \sum_{i=1}^{n-1}\tau(n,i)$$

deci

$$\frac{3n^2+5n-2}{2} \leq T(n) \leq 2n^2 + 2n - 1.$$

In acest caz putem afirma că $T(n) \in \Theta(n^2)$.

**Problema 3.2** Să se analizeze eficienţa unui algoritm care verifică dacă un tablou este ordonat crescător sau nu atât în cazurile extreme (cel mai favorabil, cel mai defavorabil) cât şi în cel mediu.

*Rezolvare.* Rezultatul algoritmului se colectează într-o variabilă booleană ce va conţine **true** dacă $a[i] \leq a[i+1]$ pentru $i = \overline{1, n-1}$ respectiv **false** dacă există $i$ cu $a[i] > a[i+1]$. Algoritmul şi tabelul costurilor sunt prezentate în Tabel 3.9.

`ordonat`$(a[1..n])$

| | Linie | Cost | Nr. repet. |
|---|---|---|---|
| 1: $r \leftarrow$ **true** | 1 | 1 | 1 |
| 2: $i \leftarrow 0$ | 2 | 1 | 1 |
| 3: **while** $(i < n-1)$ **and** $(r =$**true**$)$ **do** | 3 | 3 | $\tau_1(n)+1$ |
| 4: $\quad i \leftarrow i+1$ | 4 | 1 | $\tau_1(n)$ |
| 5: $\quad$ **if** $a[i] > a[i+1]$ **then** | 5 | 1 | $\tau_1(n)$ |
| 6: $\quad\quad r \leftarrow$ **false** | 6 | 1 | $\tau_2(n)$ |
| 7: $\quad$ **end if** | | | |
| 8: **end while** | | | |
| 9: **return** $r$ | | | |

Tabelul 3.9: Algoritm pentru a verifica dacă un tablou este ordonat crescător şi costurile operaţiilor

Cazul cel mai favorabil (din punctul de vedere al numărului de operaţii executate) corespunde situaţiei în care $a[1] > a[2]$ ceea ce conduce la $\tau_1(n) = 1$ iar $\tau_2(n) = 1$. Numărul de operaţii efectuate în acest caz este $2 + 3 \cdot 2 + 3 = 10$. Cazul cel mai defavorabil este cel în care tabloul este ordonat crescător. În acest caz $\tau_1(n) = n-1$ iar $\tau_2(n) = 0$, deci numărul de operaţii executate este: $2 + 3n + 2(n-1) = 5n$. Prin urmare $10 \leq T(n) \leq 5n$. Dependenţa timpului de execuţie de dimensiunea problemei în cazul cel mai favorabil este diferită de cea corespunzătoare cazului cel mai defavorabil. Nu putem spune despre $T(n)$ că aparţine lui $\Theta(n)$ ci doar că $T(n) \in \Omega(1)$ şi $T(n) \in \mathcal{O}(n)$.

In cazul în care considerăm comparaţia din interiorul ciclului ca fiind operaţia dominantă şi facem abstracţie de celelalte operaţii obţinem că $1 \leq T(n) \leq n-1$.

Să analizăm cazul mediu în ipoteza că există $n$ clase distincte de date de intrare. Clasa $i$ ($i \in \{1, \ldots, n-1\}$) corespunde vectorilor pentru care prima pereche de valori consecutive care încalcă condiţia de ordonare crescătoare este $(a[i], a[i+1])$. Când se ajunge la $i = n$ considerăm că tabloul este ordonat crescător. Numărul de comparaţii corespunzător clasei $i$ este $i$. Notând cu $p_i$ probabilitatea de apariţie a unei instanţe din clasa $C_i$ rezultă că timpul mediu

de execuţie este:

$$T_m(n) = \sum_{i=1}^{n-1} ip_i + (n-1)p_n.$$

Intrucât $\sum_{i=1}^{n} p_i = 1$ şi $i \leq n$ rezultă $T_m(n) \leq n-1$. In ipoteza că toate cele $n$ clase au aceeaşi probabilitate de apariţie se obţine următoarea estimare pentru timpul mediu:

$$T_m(n) = \sum_{i=1}^{n-1} \frac{1}{n} i + \frac{n-1}{n} = \frac{1}{n} \cdot \frac{n(n-1)+2(n-1)}{2} = \frac{(n-1)(n+2)}{2n}.$$

In realitate cele $n$ clase nu sunt echiprobabile însă probabilităţile corespunzătoare nu sunt uşor de determinat. Ceea ce se poate observa este faptul că $p_1 \geq p_2 \geq \ldots \geq p_n$ deci pe măsură ce probabilitatea corespunzătoare unei instanţe creşte numărul de operaţii scade.

**Problema 3.3** (a) Să se analizeze eficienţa unui algoritm care verifică dacă elementele unui tablou sunt distincte sau nu. (b) Să se analizeze complexitatea unui algoritm care verifică dacă elementele unui tablou sunt toate identice sau nu.

*Indicaţie.* Mai jos sunt descrise variante ale prelucrărilor cerute. În cazul cel mai defavorabil numărul de comparaţii efectuate de primul algoritm este $n(n-1)/2$ iar numărul de comparaţii efectuate de al doilea algoritm este $n-1$.

`distincte`($x[1..n]$)

```
1: for i ← 1, n do
2:    for j ← i + 1, n do
3:       if x[i] = x[j] then
4:          return false
5:       end if
6:    end for
7: end for
8: return true
```

`identice`($x[1..n]$)

```
1: i ← 2
2: while x[1] = x[i] and i < n do
3:    i ← i + 1
4: end while
5: if x[1] = x[i] then
6:    return true
7: else
8:    return false
9: end if
```

**Problema 3.4** (a) Să se analizeze eficienţa unui algoritm care determină reuniunea a două mulţimi reprezentate prin tablourile elementelor lor. (b) Să se analizeze eficienţa unui algoritm care determină intersecţia a două mulţimi reprezentate prin tablourile elementelor lor.

*Indicaţie.* Considerăm mulţimile reprezentate prin tablourile $a[1..m]$ respectiv $b[1..n]$. Atât pentru calculul reuniunii cât şi pentru calculul intersecţiei trebuie identificate elementele comune din cele două mulţimi. Astfel pentru fiecare element din $b[1..n]$ trebuie verificat dacă el este sau nu prezent în $a[1..m]$. În cel mai defavorabil caz numărul de comparaţii este $mn$ astfel că, dacă nu se

cunosc informaţii suplimentare privind ordinea elementelor din $a$ şi $b$, algoritmii pentru determinarea reuniunii şi intersecţiei aparţin lui $\mathcal{O}(m \cdot n)$.

**Problema 3.5** Se consideră două polinoame de grad $m$ respectiv $n$ ($m \geq n$) reprezentate prin tablourile coeficienţilor lor ($a[0..m]$ respectiv $b[0..n]$). (a) Să se analizeze eficienţa algoritmilor de adunare şi înmulţire a celor două polinoame. (b) Să se propună un algoritm de complexitate liniară pentru calculul valorii unui polinom.

*Indicaţie.* (a) Algoritmii clasici de adunare şi înmulţire a polinoamelor aparţin lui $\mathcal{O}(\max\{m, n\})$ respectiv lui $\mathcal{O}(mn)$. (b) Algoritmul bazat pe ideea de la schema lui Horner

$$a_m x^m + \ldots a_1 x + a_0 = (\ldots (a_m x + a_{m-1})x + \ldots a_1)x + a_0$$

adică

$v \leftarrow a[m]$
**for** $i \leftarrow m-1, 0, -1$ **do**
  $v \leftarrow v * x + a[i]$
**end for**

este de complexitate liniară.

**Problema 3.6** Se consideră un şir de valori întregi (pozitive şi negative - cel puţin un element este pozitiv). Să se determine suma maximă a elementelor unei secvenţe a şirului (succesiune de elemente consecutive).

*Rezolvare.* Pentru fiecare $i \in \{1, \ldots, n\}$ se calculează succesiv sumele elementelor subtablourilor $a[i..j]$ cu $j \in \{1, \ldots, n\}$ şi se reţine cea maximă. O primă variantă a algoritmului (`secventa1`) este descrisă în continuare.

---

`secventa1`(**integer** $a[1..n]$)

---

1: $max \leftarrow 0$; $imax \leftarrow 1$; $jmax \leftarrow 0$
2: **for** $i \leftarrow 1, n$ **do**
3:  $s \leftarrow 0$
4:  **for** $j \leftarrow i, n$ **do**
5:  $s \leftarrow s + a[j]$
6:  **if** $s > max$ **then**
7:  $max \leftarrow s$; $imax \leftarrow i$; $jmax \leftarrow j$
8:  **end if**
9:  **end for**
10: **end for**
11: **return** $max$, $imax$, $jmax$

---

Ca operaţie dominantă poate fi considerată adunarea ($s \leftarrow s + a[j]$) sau comparaţia ($s > max$). Numărul de repetări ale acestora este $T(n) = \sum_{i=1}^{n} \sum_{j=1}^{n} 1 = n(n-1)/2$.

Aceeaşi problemă poate fi rezolvată şi printr-un algoritm de complexitate liniară calculând succesiv suma elementelor din tablou şi reiniţiind calculul în momentul în care suma devine negativă (în acest caz reiniţializând cu 0 variabila $s$ valoarea obţinută este mai mare decât cea curentă, care este negativă). Algoritmul `secventa2` este descris în continuare.

```
secventa2(integer a[1..n])
 1: max ← 0; icrt ← 1; imax ← 1
 2: jmax ← 0
 3: s ← 0
 4: for i ← 1, n do
 5:    s ← a[i]
 6:    if s < 0 then
 7:       s ← 0; icrt ← i + 1
 8:    else if s > max then
 9:       max ← s; imax ← icrt; jmax ← i
10:    end if
11: end for
12: return max, imax, jmax
```

Numărul de execuţii ale operaţiei dominante ($s \leftarrow s + a[i]$) este $T(n) = n$.

**Problema 3.7** Să se analizeze complexitatea unui algoritm care generează toate numerele prime mai mici decât o valoare dată $n$ ($n \geq 2$).

*Rezolvare.* Vom analiza două variante de rezolvare: (a) parcurgerea tuturor valorilor cuprinse între 2 şi $n$ şi reţinerea celor prime; (b) algoritmul lui Eratostene.

(a) Presupunem că algoritmul `prim(i)` returnează **true** dacă $n$ este prim şi **false** în caz contrar (analizând divizorii potenţiali dintre 2 şi $\lfloor\sqrt{i}\rfloor$).

(b) *Algoritmul lui Eratostene.* Se porneşte de la tabloul $a[2..n]$ iniţializat cu valorile naturale cuprinse între 2 şi $n$ şi se marchează succesiv toate valorile ce sunt multipli ai unor valori mai mici. Elementele rămase nemarcate sunt prime.

Ambele variante sunt descrise în Algoritmul 3.3.

In ambele cazuri considerăm dimensiunea problemei ca fiind $n$.

In prima variantă considerăm că operaţiile dominante sunt cele efectuate în cadrul subalgoritmului `prim`. Luăm în considerare doar operaţia de analiză a divizorilor potenţiali. Pentru fiecare $i$ numărul de comparaţii efectuate este $\lfloor\sqrt{i}\rfloor - 1$. Deci:

$$T_1(n) = \sum_{i=2}^{n}(\lfloor\sqrt{i}\rfloor - 1) \leq \sum_{i=2}^{n}\sqrt{i} - (n-1).$$

**Algoritmul 3.3** Algoritmi pentru generarea numerelor prime

```
generare(integer n)
1: k ← 0
2: for i ← 2, n do
3:    if prim(i) =true then
4:       k ← k + 1
5:       p[k] ← i
6:    end if
7: end for
8: return p[1..k]
```

```
eratostene(integer n)
1: for i ← 2, n do
2:    a[i] ← i
3: end for
4: for i ← 2, ⌊√n⌋ do
5:    if a[i] ≠ 0 then
6:       j ← i ∗ i
7:       while j ≤ n do
8:          a[j] ← 0; j ← j + i
9:       end while
10:   end if
11: end for
12: k ← 0
13: for i ← 2, n do
14:   if a[i] ≠ 0 then
15:      k ← k + 1; p[k] ← a[i]
16:   end if
17: end for
18: return p[1..k]
```

Pentru a estima suma de mai sus se aproximează cu integrala $\int_2^n \sqrt{x}dx = 2/3n\sqrt{n} - 4\sqrt{2}/3$ obţinându-se că:

$$T_1(n) \leq 2/3n\sqrt{n} - 4\sqrt{2}/3 - (n-1).$$

Pe de altă parte, întrucât $\sqrt{i} - 1 < \lfloor\sqrt{i}\rfloor \leq \sqrt{i}$, rezultă că $T_1(n) \geq 2/3n\sqrt{n} - 4\sqrt{2}/3 - 2(n-1)$. Se obţine astfel că $T_1(n) \in \Theta(n\sqrt{n})$.

In cazul algoritmului lui Eratostene operaţia dominantă poate fi considerată cea de marcare a elementelor tabloului $a$. Pentru fiecare $i$ se marchează cel mult $\lfloor(n - i^2)/i + 1\rfloor$ elemente astfel că numărul total de astfel de operaţii satisface:

$$T_2(n) \leq \sum_2^{\lfloor\sqrt{n}\rfloor} \lfloor\frac{n-i^2}{i} + 1\rfloor \leq \sum_2^{\sqrt{n}}(\frac{n-i^2}{i} + 1) = n\sum_2^{\lfloor\sqrt{n}\rfloor}\frac{1}{i} - \sum_2^{\lfloor\sqrt{n}\rfloor} i + \lfloor\sqrt{n}\rfloor - 1.$$

Se aplică din nou tehnica aproximării unei sume prin integrala corespunzătoare şi se obţine:

$$T_2(n) \leq n\ln\sqrt{n} - n\ln\sqrt{2} - \sqrt{n}(\sqrt{n}+1)/2 + \sqrt{n}$$

deci $T_2(n) \in \mathcal{O}(n\lg n)$.

Se observă că varianta bazată pe algoritmul lui Eratostene are un ordin de complexitate mai mic decât prima variantă însă necesită utilizarea unui tablou suplimentar de dimensiune $n-1$.

**Problema 3.8** Să se determine ordinul de creştere al valorii variabilei $x$ (în funcţie de $n$) după execuţia algoritmilor `alg1`, `alg2`, `alg3`, `alg4`, `alg5` şi `alg6`.

```
alg1(integer n)
1: x ← 0
2: for i ← 1, n do
3:    for j ← 1, i do
4:       for k ← 1, j do
5:          x ← x + 1
6:       end for
7:    end for
8: end for
9: return x
```

```
alg2(integer n)
1: x ← 0
2: i ← n
3: while i ≥ 1 do
4:    x ← x + 1
5:    i ← i DIV 2
6: end while
7: return x
```

```
alg3(integer n)
1: x ← 0
2: i ← n
3: while i ≥ 1 do
4:    for j ← 1, n do
5:       x ← x + 1
6:    end for
7:    i ← i DIV 2
8: end while
9: return x
```

```
alg4(integer n)
1: x ← 0
2: i ← n
3: while i ≥ 1 do
4:    for j ← 1, i do
5:       x ← x + 1
6:    end for
7:    i ← i DIV 2
8: end while
9: return x
```

*Indicaţie.* `alg1`:

$$x = \sum_{i=1}^{n}\sum_{j=1}^{i}\sum_{k=1}^{j} 1 = \sum_{i=1}^{n}\sum_{j=1}^{i} j = \sum_{i=1}^{n} \frac{i(i+1)}{2} = \frac{n(n+1)(n+2)}{6} \in \Theta(n^3)$$

`alg2`: $x$ va conţine numărul de cifre ale reprezentării în baza 2 a lui $n$, adică $\lfloor \log_2(n) \rfloor + 1 \in \Theta(\lg n)$.

`alg3`: Spre deosebire de exemplul anterior, pentru fiecare valoare a lui $i$ variabila $x$ este mărită cu $n$, deci valoarea finală va fi $n(\lfloor \log_2(n) \rfloor + 1) \in \Theta(n \lg n)$.

`alg4`: Este similar algoritmului `alg3` însă pe linia 4 limita superioară a ciclului **for** este $i$ în loc de $n$. Variabila $x$ va conţine suma $n + \lfloor \frac{n}{2} \rfloor + \ldots + \lfloor \frac{n}{2^k} \rfloor$ cu $k$ având proprietatea că $2^k \leq n < 2^{k+1}$. Ordinul de mărime al lui $x$ se poate stabili pornind de la observaţia că $2^{k+1} - 1 \leq x < 2(2^{k+1} - 1)$. Deci $x \in \Theta(2^k) = \Theta(n)$.

| `alg5`(**integer** $n$) | `alg6`(**integer** $n$) |
|---|---|
| 1: $x \leftarrow 0$ | 1: $x \leftarrow 0$ |
| 2: $i \leftarrow 1$ | 2: $i \leftarrow 2$ |
| 3: **while** $i < n$ **do** | 3: **while** $i < n$ **do** |
| 4: $\quad x \leftarrow x + 1$ | 4: $\quad x \leftarrow x + 1$ |
| 5: $\quad i \leftarrow 2 * i$ | 5: $\quad i \leftarrow i * i$ |
| 6: **end while** | 6: **end while** |
| 7: **return** $x$ | 7: **return** $x$ |

`alg5:` Variabila $x$ va conţine cel mai mare număr natural $k$ cu proprietatea că $2^k \leq n$, deci $x = \lfloor \log_2 n \rfloor \in \Theta(\lg n)$.

`alg6:` Variabila $x$ va conţine cel mai mare număr natural $k$ cu proprietatea că $2^{2^k} \leq n$ deci $x = \lfloor \log_2 \log_2 n \rfloor \in \Theta(\lg\lg n)$.

**Problema 3.9** Să se determine ordinul de mărime al variabilei $x$ după execuţia următoarelor prelucrări:

1: $x \leftarrow 0$
2: $j \leftarrow n$
3: **while** $j \geq 1$ **do**
4: $\quad$ **for** $i \leftarrow 1, j$ **do**
5: $\quad\quad x \leftarrow x + 1$
6: $\quad$ **end for**
7: $\quad j \leftarrow j$ DIV 3
8: **end while**
9: **return** $x$

*Indicaţie.* Algoritmul este similar cu algoritmul 4 de la exerciţiul anterior, deci $x \in \Theta(n)$.

**Problema 3.10** Să se determine termenul dominant şi ordinul de creştere pentru expresiile:

(a) $2\lg n + 4n + 3n \lg n$
(b) $2 + 4 + 6 + \ldots 2n$
(c) $\frac{(n+1)(n+3)}{n+2}$
(d) $2 + 4 + 8 + \ldots + 2^n$

*Indicaţie.* (a) Termenul dominant în $2\lg n + 4n + 3n \lg n$ este evident $3n \lg n$ iar ordinul de creştere este $n \lg n$.
(b) Termenul dominant al sumei $2 + 4 + 6 + \ldots 2n$ nu este $2n$ ci $n^2$ întrucât $2 + 4 + 6 + \ldots 2n = n(n + 1)$. Deci ordinul de creştere este chiar $n^2$.
(c) Termenul dominant este $n^2/(n + 2)$ iar ordinul de creştere este $n$.
(d) Întrucât $2+4+8+\ldots+2^n = 2(1+2+4+\ldots+2^{n-1}) = 2(2^n - 1)$, termenul dominant este $2 \cdot 2^n$ iar ordinul de creştere este $2^n$.

**Problema 3.11** Să se arate că:

(a) $n! \in \mathcal{O}(n^n)$, $n! \in \Omega(2^n)$

(b) $\lg n! \in \Theta(n \lg n)$

(c) $2^n \in \mathcal{O}(n!)$

(d) $\sum_{i=1}^{n} i \lg i \in \mathcal{O}(n^2 \lg n)$

(e) $\lg(n^k + c) \in \Theta(\lg n)$, $k > 0, c > 0$.

*Indicaţie.* (a) Întrucât $n! \leq n^n$ pentru orice $n$ natural, rezultă imediat că $n! \in \mathcal{O}(n^n)$. Pe de altă parte, $n! \geq 2^{n-1}$ pentru orice $n$, deci $n! \in \Omega(2^n)$.
(b) Se porneşte de la aproximaţia Stirling $n! \simeq \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$ care este adevărată pentru valori mari ale lui $n$. Mai exact, există $c_1, c_2 \in \mathbb{R}_+$ şi $n_0 \in \mathbb{N}$ astfel încât

$$\sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \frac{c_1}{n}\right) \leq n! \leq \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \frac{c_2}{n}\right)$$

pentru orice $n \geq n_0$.

Prin logaritmare se obţine:

$$\ln(2\pi n)/2 + \ln(1 + c_1/n) + n \ln n - n \leq \ln n \leq \ln(2\pi n)/2 + \ln(1 + c_1/n) + n \ln n - n$$

deci $\ln n! \in \Theta(n \ln n)$. Facând abstracţie de baza logaritmului se obţine $\lg n! \in \Theta(n \lg n)$
(c) Cum $2^n < n!$ pentru orice $n \geq 4$ rezultă că $2^n \in \mathcal{O}(n!)$.
(d) $\sum_{i=1}^{n} i \lg i \leq \lg n \sum_{i=1}^{n} i \leq n^2 \lg n$, deci $\sum_{i=1}^{n} i \lg i \in \mathcal{O}(n^2 \lg n)$.
(e) Pentru valori suficient de mari ale lui $n$ are loc $\lg n^k \leq \lg(n^k + c) \leq \lg(2n^k)$, adică $k \lg n \leq \lg(n^k + c) \leq k \lg n + \lg 2$, deci $\lg(n^k + c) \in \Theta(\lg n)$.

**Problema 3.12** Care dintre următoarele afirmaţii sunt adevărate ? Demonstraţi.

(a) $\sum_{i=1}^{n} i^2 \in \Theta(n^2)$, $\sum_{i=1}^{n} i^2 \in \mathcal{O}(n^2)$, $\sum_{i=1}^{n} i^2 \in \Omega(n^2)$

(b) $cf(n) \in \Theta(f(n))$, $f(cn) \in \Theta(f(n))$

(c) $2^{n+1} \in \mathcal{O}(2^n)$, $2^{2n} \in \mathcal{O}(2^n)$ ?

*Indicaţie.* (a) Întrucât $\sum_{i=1}^{n} i^2 = n(n+1)(2n+1)/6$ rezultă că $\sum_{i=1}^{n} i^2 \in \Theta(n^3)$, deci $\sum_{i=1}^{n} i^2 \in \Omega(n^2)$ însa celelalte două afirmaţii sunt false.
(b)Întrucât $c_1 f(n) \leq cf(n) \leq c_2 f(n)$ pentru constante $c_1$ şi $c_2$ satisfăcând $0 < c_1 \leq c \leq c_2$ rezultă că $cf(n) \in \Theta(f(n))$. In schimb $f(cn) \in \Theta(f(n))$ nu este adevărată pentru orice funcţie $f$ şi orice constantă $c$. De exemplu $f(n) = \exp(n)$ şi $c > 1$ nu satisfac această proprietate întrucât nu există $c'$ astfel încât $\exp(cn) \leq c' \exp(n)$ pentru valori mari ale lui $n$.
(c) Întrucât $2^{n+1} = 2 \cdot 2^n$ rezultă că $2^{n+1} \in \Theta(2^n)$ deci implicit şi $2^{n+1} \in \mathcal{O}(2^n)$. Pe de altă parte, $\lim_{n\to\infty} 2^{2n}/2^n = \infty$ prin urmare $2^{2n} \notin \mathcal{O}(2^n)$ dar $2^{2n} \in \Omega(2^n)$.

**Problema 3.13** Propuneţi un algoritm de complexitate liniară pentru a determina tabloul frecvenţelor cumulate pornind de la tabloul frecvenţelor simple. Pentru un tablou $(f_1, \ldots, f_n)$ de frecvenţe, tabloul frecvenţelor cumulate $(fc_1, \ldots, fc_n)$ se caracterizează prin $fc_i = \sum_{j=1}^{i} f_j$.

*Rezolvare.* Este uşor de văzut că un algoritm care calculează pentru fiecare $i$ valoarea sumei $fc_i = \sum_{j=1}^{i} f_j$ necesită efectuarea a $\sum_{i=2}^{n} \sum_{j=1}^{i} 1 = \sum_{i=2}^{n} i = n(n+1)/2 - 1 \in \Theta(n^2)$. Observând că $fc_i = fc_{i-1} + f_i$ pentru $i = \overline{2,n}$ şi $fc_1 = f_1$ rezultă că frecvenţele cumulate pot fi calculate prin algoritmul `frecvente_cumulate` descris în continuare.

```
frecvente_cumulate(integer f[1..n])
integer i,fc[1..n]
 1: fc[1] ← f[1]
 2: for i ← 2, n do
 3:     fc[i] = fc[i − 1] + f[i]
 4: end for
 5: return  fc[1..n]
```

**Problema 3.14** Propuneţi un algoritm pentru determinarea celor mai mici două valori dintr-un tablou. Determinaţi numărul de comparaţii efectuate asupra elementelor tabloului şi stabiliţi ordinul de complexitate al algoritmului.

*Indicaţie.* O variantă de algoritm este descrisă în `valori_minime`.

```
valori_minime(integer a[1..n])
 1: if a[1] < a[2] then
 2:     min1 ← a[1]; min2 ← a[2];
 3: else
 4:     min1 ← a[2]; min2 ← a[1];
 5: end if
 6: for i ← 3, n do
 7:     if a[i] < min1 then
 8:         min2 ← min1; min2 ← a[i]
 9:     else if a[i] < min2 then
10:         min2 ← a[i]
11:     end if
12: end for
13: return  min1, min2
```

În cazul cel mai nefavorabil numărul de comparaţii efectuate asupra elementelor tabloului este $T(n) = 1 + 2(n-2) = 2n - 3$ deci algoritmul aparţine lui $\mathcal{O}(n)$.

**Problema 3.15** Se consideră următorii doi algoritmi pentru calculul puterii unui număr natural, $x$ ($p = x^n$):

```
putere1(real x, integer n)
1: p ← 1
2: for i ← 1, n do
3:     np ← 0
4:     for j ← 1, x do
5:         np ← np + p
6:     end for
7:     p ← np
8: end for
9: return p
```

```
putere1(real x, integer n)
1: p ← 1
2: for i ← 1, n do
3:     p ← p * x
4: end for
5: return p
```

Să se stabilească ordinele de complexitate considerând că toate operaţiile aritmetice au acelaşi cost.

# Capitolul 4

# Analiza algoritmilor de sortare

## 4.1 Problematica sortării

Se consideră un set finit de obiecte, fiecare având asociată o caracteristică, numită *cheie*, ce ia valori într-o mulţime pe care este definită o relaţie de ordine. *Sortarea* este procesul prin care elementele setului sunt rearanjate astfel încât cheile lor să se afle într-o anumită ordine.

**Exemplul 4.1** Considerăm setul de valori întregi: (5,8,3,1,6). În acest caz cheia de sortare coincide cu valoarea elementului. Prin sortare *crescătoare* se obţine setul (1,3,5,6,8) iar prin sortare *descrescătoare* se obţine (8,6,5,3,1).

**Exemplul 4.2** Considerăm un tabel constând din nume ale studenţilor şi note: ((Popescu,9), (Ionescu,10), (Voinescu,8), (Adam,9)). În acest caz cheia de sortare poate fi numele sau nota. Prin ordonare crescătoare după nume se obţine ((Adam,9), (Ionescu,10), (Popescu,9), (Voinescu,8)), iar prin ordonare descrescătoare după notă se obţine ((Ionescu,10), (Popescu,9), (Adam,9), (Voinescu,8)).

Pentru a simplifica prezentarea, în continuare vom considera că setul prelucrat este constituit din valori scalare ce reprezintă chiar cheile de sortare şi că scopul urmărit este ordonarea crescătoare a acestora. Astfel, a ordona setul $(x_1, x_2, \ldots, x_n)$ este echivalent cu a găsi o permutare de ordin $n$, $(p(1), p(2), \ldots, p(n))$ astfel încât $x_{p(1)} \leq x_{p(2)} \leq \ldots \leq x_{p(n)}$.

De asemenea vom considera că elementele setului sunt stocate pe un suport de informaţie ce permite accesul aleator la date. În acest caz este vorba despre *sortare internă*. În cazul în care suportul de informaţie permite doar accesul secvenţial la date trebuie folosite metode specifice încadrate în categoria metodelor de *sortare externă*.

Pe de altă parte, în funcţie de spaţiul de manevră necesar pentru efectuarea sortării există:

- Sortare ce foloseşte o zonă de manevră de dimensiunea setului de date. Dacă setul iniţial de date este reprezentat de tabloul $x[1..n]$, cel sortat se va obţine într-un alt tablou $y[1..n]$.
- Sortare în aceeaşi zonă de memorie (sortare pe loc - *in situ*). Elementele tabloului $x[1..n]$ îşi schimbă poziţiile astfel încât după încheierea procesului să fie ordonate. Este posibil ca şi în acest caz să se folosească o zonă de memorie însă aceasta este de regulă de dimensiunea unui element şi nu de dimensiunea întregului tablou.

În continuare vom analiza doar metode de sortare internă în aceeaşi zonă de memorie. Metodele de sortare pot fi caracterizate prin:

*Stabilitate.* O metodă de sortare este considerată stabilă dacă ordinea relativă a elementelor ce au aceeaşi valoare a cheii nu se modifică în procesul de sortare. De exemplu dacă asupra tabelului cu note din Exemplul 4.2 se aplică o metodă stabilă de ordonare descrescătoare după notă se obţine ((Ionescu,10), (Popescu,9), (Adam,9), (Voinescu,8)) pe când dacă se aplică una care nu este stabilă se va putea obţine ((Ionescu,10), (Adam,9), (Popescu,9), (Voinescu,8)).

*Naturaleţe.* O metodă de sortare este considerată naturală dacă numărul de operaţii scade odată cu *distanţa* dintre tabloul iniţial şi cel sortat. O măsură a acestei distanţe poate fi numărul de inversiuni al permutării corespunzătoare tabloului iniţial.

*Eficienţă.* O metodă este considerată eficientă dacă nu necesită un volum mare de resurse. Din punctul de vedere al spaţiului de memorie o metodă de sortare pe loc este mai eficientă decât una bazată pe o zonă de manevră de dimensiunea tabloului. Din punct de vedere al timpului de execuţie este important să fie efectuate cât mai puţine operaţii. În general, în analiză se iau în considerare doar operaţiile efectuate asupra elementelor tabloului (comparaţii şi mutări). O metodă este considerată *optimală* dacă ordinul său de complexitate este cel mai mic din clasa de metode din care face parte.

*Simplitate.* O metodă este considerată simplă dacă este intuitivă şi uşor de înţeles.

Primele două proprietăţi sunt specifice algoritmilor de sortare pe când ultimele sunt cu caracter general. În continuare vom considera câteva metode *elementare* de sortare caracterizate prin faptul că sunt simple, nu sunt cele mai eficiente metode, dar reprezintă punct de pornire pentru metode avansate. Pentru fiecare dintre aceste metode se va prezenta: principiul, verificarea corectitudinii şi analiza complexităţii.

## 4.2 Sortare prin inserţie

### 4.2.1 Principiu

Ideea de bază a acestei metode este:

> *Începând cu al doilea element al tabloului $x[1..n]$, fiecare element este inserat pe poziţia adecvată în subtabloul care îl precede.*

La fiecare etapă a sortării, elementului $x[i]$ i se caută poziţia adecvată în subtabloul destinaţie $x[1..i-1]$ (care este deja ordonat) comparând pe $x[i]$ cu elementele din $x[1..i-1]$ începând cu $x[i-1]$ şi creând spaţiu prin deplasarea spre dreapta a elementelor mai mari decât $x[i]$. Astfel structura generală este descrisă în algoritmul `inserţie`.

`inserţie`(**real** $x[1..n]$)
**for** $i \leftarrow 2, n$ **do**
  $\langle$ inserează $x[i]$ în subşirul $x[1..i-1]$ astfel încât $x[1..i]$ să fie ordonat$\rangle$
**end for**

Sortarea prin inserţie este detaliată în Algoritmul 4.1.

**Algoritmul 4.1** Sortare prin inserţie

```
inserţie1(real x[1..n])
for i ← 2, n do
  j ← i − 1
  aux ← x[i]
  while j ≥ 1 and aux < x[j] do
    x[j + 1] ← x[j]
    j ← j − 1
  end while
  x[j + 1] ← aux
end for
return x[1..n]
```

Există şi alte metode de implementare a metodei. Astfel, pentru a evita efectuarea comparaţiei $j \geq 1$ la fiecare iteraţie interioară se plasează pe poziţia 0 în tabloul $x$ valoarea lui $x[i]$, această poziţie jucând rolul unui *fanion*. Astfel cel mai târziu când $j = 0$ se ajunge la $x[j] = x[i]$. Această variantă este descrisă în Algoritmul 4.2.

### 4.2.2 Verificarea corectitudinii

Precondiţia problemei este $P = \{n \geq 1\}$, iar postcondiţia este $Q = \{x[1..n]$ este ordonat crescător$\}$. Pentru ciclul exterior (după $i$) demonstrăm că proprietatea invariantă este $\{x[1..i-1]$ este ordonat crescător$\}$. La începutul ciclului $i = 2$,

deci $x[1..1]$ poate fi considerat crescător. La sfârşitul prelucrării ($i = n + 1$) invariantul implică evident postcondiţia. Rămâne să arătăm că proprietatea rămâne adevărată şi după efectuarea prelucrărilor din cadrul ciclului. Pentru aceasta este suficient să arătăm că pentru ciclul interior (după $j$) aserţiunea $\{x[1..j]$ este crescător şi $aux \leq x[j+1] = x[j+2] \leq \ldots \leq x[i]\}$ este invariantă. La sfârşitul ciclului **while** această proprietate ar implica una dintre relaţiile:

- $x[1] \leq \ldots \leq x[j] \leq aux \leq x[j+1] = x[j+2] \leq \ldots \leq x[i]$ în cazul în care condiţia de ieşire din ciclu este $aux \geq x[j]$;
- $aux \leq x[1] = x[2] \leq \ldots \leq x[i]$ în cazul în care condiţia de ieşire din ciclu este $j = 0$.

Oricare dintre aceste două relaţii ar conduce prin atribuirea $x[j+1] \leftarrow aux$ la $x[1] \leq \ldots \leq x[j] \leq x[j+1] \leq x[j+2] \leq \ldots \leq x[i]$ iar prin trecerea la următoarea valoare a contorului ($i \leftarrow i + 1$) la faptul că $x[1..i-1]$ este crescător.

Rămâne doar să justificăm invariantul ciclului **while**. La intrarea în ciclu au loc relaţiile: $j = i - 1$, $aux = x[i]$ deci $aux = x[j+1] = x[i]$ iar cum $x[1..i-1]$ este crescător rezultă că proprietatea invariantă propusă pentru **while** este satisfăcută. Arătăm că ea nu este alterată de prelucrările din cadrul ciclului: dacă $aux < x[j]$, prin atribuirea $x[j+1] \leftarrow x[j]$ se obţine: $aux < x[j] = x[j+1] \leq \ldots \leq x[i]$ iar după $j \leftarrow j - 1$ va fi adevărată aserţiunea: $\{$ $x[1..j]$ crescător şi $aux < x[j+1] = x[j+2] \leq \ldots \leq x[i]$. Cum $x[j+1] = x[j+2]$ prin atribuirea $x[j+1] \leftarrow x[j]$ nu se pierde informaţie din tablou.

Oprirea algoritmului este asigurată de utilizarea câte unui contor pentru fiecare dintre cele două cicluri.

**Algoritmul 4.2** Sortarea prin inserţie cu folosirea unui fanion

```
inserţie2(real x[1..n])
for i ← 2, n do
   x[0] ← x[i]
   j ← i − 1
   while x[0] < x[j] do
      x[j + 1] ← x[j]
      j ← j − 1
   end while
   x[j + 1] ← x[0]
end for
return x[1..n]
```

### 4.2.3 Analiza complexităţii

Vom lua în considerare doar operaţiile de comparare şi mutare (atribuire) efectuate asupra elementelor tabloului. Fie $T_C(i)$ şi $T_M(i)$ numărul comparaţiilor

respectiv al atribuirilor care implică elementele tabloului pentru fiecare $i = \overline{2,n}$. In funcţie de problema concretă de rezolvat poate fi mai mare costul comparaţiilor (dacă compararea presupune efectuarea mai multor prelucrări) sau costul atribuirilor (dacă elementele tabloului sunt structurate).

Cazul cel mai favorabil corespunde şirului ordonat crescător iar cel mai defavorabil celui ordonat descrescător.

În cazul cel mai favorabil $T_C(i) = 1$, iar $T_M(i) = 2$ (este vorba de operaţiile care implică variabila ajutătoare *aux* şi care dealtfel în acest caz nu au nici un efect) deci $T(n) \geq \sum_{i=2}^{n}(T_C(i)+T_M(i)) = 3(n-1)$. În cazul cel mai defavorabil $T_C(i) = i$ iar $T_M(i) = i-1+2 = i+1$, obţinându-se că $T(n) \leq \sum_{i=2}^{n}(2i+1) = n^2 + 2n - 3$.

Prin urmare $3(n-1) \leq T(n) \leq n^2 + 2n - 3$ adică algoritmul sortării prin inserţie se încadrează în clasele $\Omega(n)$ şi $\mathcal{O}(n^2)$.

### 4.2.4 Proprietăţi ale sortării prin inserţie

O proprietate importantă este faptul că în cazul şirurilor aproape sortate (număr mic de inversiuni) comportarea algoritmului este apropiată de comportarea din cazul cel mai favorabil. Aceasta înseamnă ca este satisfăcută proprietatea de naturaleţe.

Atât timp cât condiţia de continuare a ciclului interior este $aux < x[j]$ algoritmul de sortare prin inserţie este stabil. Dacă însă se foloseşte $aux \leq x[j]$ atunci stabilitatea nu mai este asigurată.

O altă caracteristică a sortării prin inserţie este faptul că pentru fiecare comparaţie efectuată se realizează deplasarea unui element cu o singură poziţie. Aceasta înseamnă eliminarea a cel mult unei inversiuni, adică aranjarea în ordinea dorită (dar nu neapărat pe poziţiile finale) ale unei perechi de elemente. Cum în cazul cel mai defavorabil, cel al unui şir ordonat descrescător, sunt $n(n-1)/2$ inversiuni rezultă că sunt necesare tot atâtea comparaţii. Aceasta înseamnă ca orice algoritm de sortare bazat pe transformări locale (interschimbări între elemente vecine) aparţine lui $\mathcal{O}(n^2)$. Pentru reducerea ordinului de complexitate ar trebui să se realizeze interschimbări între elemente care nu sunt neapărat vecine. Aceasta este ideea variantei de algoritm descrise în continuare.

### 4.2.5 Sortare prin inserţie cu pas variabil

Unul dintre dezavantajele sortării prin inserţie este faptul că la fiecare etapă un element al şirului se deplasează cu o singură poziţie. O variantă de reducere a numărului de operaţii efectuate este de a compara elemente aflate la o distanţă mai mare ($h \geq 1$) şi de a realiza deplasarea acestor elemente peste mai multe poziţii. De fapt tehnica se aplică în mod repetat pentru valori din ce în ce mai mici ale pasului $h$, asigurând *h-sortarea* şirului.

Un şir $x[1..n]$ este considerat $h$-sortat dacă orice subşir $x[i_0], x[i_0+h], x[i_0+2h]$... este sortat ($i_0 \in \{1, \ldots, h\}$). Aceasta este ideea algoritmului propus de Donald Shell în 1959, algoritm cunoscut sub numele "Shell sort".

Elementul cheie al algoritmului îl reprezintă alegerea valorilor pasului $h$. Pentru alegeri adecvate ale secvenţei $h_k$ se poate obţine un algoritm de complexitate mai mică (de exemplu pentru $h_k = 2^k - 1$ se obţine un algoritm din $\mathcal{O}(n^{3/2})$ [10]). O astfel de secvență de paşi este $h_k = 2^k - 1$ pentru $1 \leq k \leq \lfloor \lg n \rfloor$.
*Exemplu.* Exemplificăm ideea algoritmului în cazul unui şir cu 15 elemente pentru următoarele valori ale pasului: $h = 13$, $h = 4$, $h = 1$ (care corespund unui şir $h_k$ dat prin relaţia $h_k = 3h_{k-1} + 1$, $h_1 = 1$). Pentru acest şir s-a ilustrat experimental că este eficient, fără însă a fi determinat teoretic ordinul de complexitate. Procesul de sortare constă în următoarele etape:

*Etapa 1:* pentru $h = 13$ se aplică algoritmul sortării prin inserţie subşirurilor $(x[1], x[14])$ şi $(x[2], x[15])$, singurele subşiruri cu elemente aflate la distanţa $h$ care au mai mult de un element:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 14 | 8 | 10 | 7 | 9 | 12 | 5 | 15 | 2 | 13 | 6 | 1 | 4 | 3 | 11 |

şi se obţine

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 8 | 10 | 7 | 9 | 12 | 5 | 15 | 2 | 13 | 6 | 1 | 4 | 14 | 11 |

*Etapa 2:* pentru $h = 4$ se aplică algoritmul sortării prin inserţie succesiv subşirurilor: $(x[1], x[5], x[9], x[13])$, $(x[2], x[6], x[10], x[14])$, $(x[3], x[7], x[11], x[15])$, $(x[4], x[8], x[12])$. După prima subetapă (prelucrarea primului subşir) prin care se ordonează subşirul constituit din elementele marcate:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 8 | 10 | 7 | 9 | 12 | 5 | 15 | 2 | 13 | 6 | 1 | 4 | 14 | 11 |

se obţine:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 8 | 10 | 7 | 3 | 12 | 5 | 15 | 4 | 13 | 6 | 1 | 9 | 14 | 11 |

La a doua subetapă se aplică sortarea prin inserţie asupra subşirului constituit din elementele marcate:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 8 | 10 | 7 | 3 | 12 | 5 | 15 | 4 | 13 | 6 | 1 | 9 | 14 | 11 |

obţinându-se aceeaşi configuraţie (subşirul este deja ordonat crescător). Se prelucrează acum cel de al treilea subşir:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 8 | 10 | 7 | 3 | 12 | 5 | 15 | 4 | 13 | 6 | 1 | 9 | 14 | 11 |

obţinându-se

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 8 | 5 | 7 | 3 | 12 | 6 | 15 | 4 | 13 | 10 | 1 | 9 | 14 | 11 |

Se aplică acum sortarea prin inserţie asupra subşirului constituit din elementele marcate:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 8 | 5 | 7 | 3 | 12 | 6 | 15 | 4 | 13 | 10 | 1 | 9 | 14 | 11 |

obţinându-se

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 8 | 5 | 1 | 3 | 12 | 6 | 7 | 4 | 13 | 10 | 15 | 9 | 14 | 11 |

*Etapa 3:* Se aplică sortarea prin inserţie asupra întregului şir. Întrucât şirul la care s-a ajuns prin prelucrările anterioare este aproape sortat numărul de operaţii efectuate în ultima etapă (când se aplică sortarea clasică prin inserţie) este semnificativ mai mic decât dacă s-ar aplica algoritmul de sortare prin inserţie şirului iniţial. Pentru acest exemplu aplicarea directă a sortării prin inserţie conduce la execuţia a 68 de comparaţii pe când aplicarea sortarii cu pas variabil de inserţie conduce la execuţia a 34 de comparaţii. Atât prelucrarea specifică unei etape (pentru o valoare a pasului de inserţie) cât şi întreaga sortare sunt descrise în Algoritmul 4.3.

**Algoritmul 4.3** Sortarea prin inserţie cu pas variabil ("Shell sort")

```
inser_pas(real x[1..n], integer h)
integer i, j
real aux
for i ← h + 1, n do
   aux ← x[i]
   j ← i − h
   while j ≥ 1 and aux < x[j] do
      x[j + h] ← x[j]
      j ← j − h
   end while
   x[j + h] ← aux
end for
return x[1..n]
```

```
shellsort(real x[1..n])
integer h
h ← 1
while h ≤ n do
   h ← 3 ∗ h + 1
end while
repeat
   h ← hDIV3
   x[1..n] ← inser_pas(x[1..n], h)
until h = 1
return x[1..n]
```

## 4.3 Sortare prin selecţie

### 4.3.1 Principiu

Ideea de bază a acestei metode este:

> *Pentru fiecare poziţie i, începând cu prima, se selectează din subtabloul ce începe cu acea poziţie cel mai mic element şi se amplasează pe locul respectiv (prin interschimbare cu elementul curent de pe poziţia i)*

Structura generală este descrisă în algoritmul `selecţie`.

```
selecţie(real x[1..n])
for i ← 1, n − 1 do
  ⟨ se determină valoarea minimă din x[i..n] şi se interschimbă cu x[i] ⟩
end for
```

Ciclul **for** continuă până la $n-1$ deoarece subtabloul $x[n..n]$ conţine un singur element care este plasat chiar pe poziţia potrivită, ca urmare a interschimbărilor efectuate anterior. O variantă de descriere a metodei selecţiei este Algoritmul 4.4.

**Algoritmul 4.4** Sortare prin selecţie

```
selecţie(real x[1..n])
for i ← 1, n − 1 do
  k ← i
  for j ← i + 1, n do
    if x[k] > x[j] then
      k ← j
    end if
  end for
  if k ≠ i then
    x[k] ↔ x[i]
  end if
end for
return (x[1..n])
```

### 4.3.2 Verificarea corectitudinii

Arătăm că un invariant al ciclului exterior (după $i$) este: $\{x[1..i-1]$ este ordonat crescător şi $x[i-1] \leq x[j]$ pentru $j = \overline{i,n}\}$. La început $i = 1$ deci $x[1..0]$ este un tablou vid. Ciclul **for** interior (după $j$) determină poziţia minimului din $x[i..n]$. Aceasta este plasată prin interschimbare pe poziţia $i$. Se obţine astfel că $x[1..i]$ este ordonat crescător şi că $x[i] \leq x[j]$ pentru $j = \overline{i+1,n}$. După incrementarea lui $i$ (la sfârşitul ciclului după $i$) se reobţine proprietatea invariantă. La final $i = n$, iar invariantul conduce la $x[1..n-1]$ crescător şi $x[n-1] \leq x[n]$ adică $x[1..n]$ este sortat crescător.

### 4.3.3 Analiza complexităţii

Indiferent de aranjarea iniţială a elementelor, numărul de comparaţii efectuate este:

$$T_C(n) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 1 = \sum_{i=1}^{n-1} (n-i) = n(n-1) - \frac{n(n-1)}{2} = \frac{n(n-1)}{2}$$

În cazul cel mai favorabil (şir ordonat crescător) numărul interschimbărilor este $T_M(n) = 0$. În cazul cel mai defavorabil (şir ordonat descrescător) pentru fiecare $i$ se efectuează o interschimbare (trei atribuiri), deci costul corespunzător modificării elementelor tabloului este $T_M(n) = 3\sum_{i=1}^{n-1} 1 = 3(n-1)$. Luând în considerare atât comparaţiile cât şi atribuirile se obţine că algoritmul sortării prin selecţie aparţine clasei $\Theta(n^2)$.

### 4.3.4 Proprietăţi ale sortării prin selecţie

Algoritmul este parţial natural (numărul de comparaţii nu depinde de gradul de sortare al şirului). În varianta prezentată (când minimul este interschimbat cu poziţia curentă) algoritmul nu este stabil. Dacă însă în locul unei singure interschimbări s-ar realiza deplasarea elementelor subtabloului $x[i..k-1]$ la dreapta cu o poziţie (ca în algoritmul inserţiei), iar $x[k]$ (salvat în prealabil într-o variabilă auxiliară) s-ar transfera in $x[i]$ algoritmul ar deveni stabil.

## 4.4 Sortare prin interschimbarea elementelor vecine

### 4.4.1 Principiu

Ideea de bază a acestei metode de sortare este:

> *Se parcurge tabloul de sortat şi se compară elementele vecine iar dacă acestea nu se află în ordinea corectă se interschimbă. Parcurgerea se reia până când nu mai este necesară nici o interschimbare.*

Structura generală este descrisă în algoritmul `interschimbări`.

```
interschimbări (real x[1..n])
repeat
   ⟨ se parcurge tabloul şi dacă două elemente vecine nu sunt în ordinea
   corectă sunt interschimbate ⟩
until ⟨ nu mai este necesară nici o interschimbare ⟩
```

Să considerăm secvenţa interschimbării elementelor vecine în cazul în care nu sunt în ordinea corectă:

```
for i ← 1, n − 1 do
  if x[i] > x[i + 1] then
    x[i] ↔ x[i + 1]
  end if
end for
```

Folosind ca invariant al prelucrării repetitive proprietatea $\{x[i] \geq x[j], j = \overline{1,i}\}$ se poate arăta că prelucrarea de mai sus conduce la satisfacerea postcondiţiei: $\{x[n] \geq x[i], i = \overline{1,n}\}$. Pentru $i = 1$ proprietatea invariantă este adevărată întrucât $x[1] \geq x[1]$. Presupunem că proprietatea este adevărată pentru $i$. Dacă $x[i] \leq x[i+1]$ atunci nu se efectuează nici o prelucrare şi rămâne adevărată şi pentru $i + 1$. Dacă în schimb $x[i] > x[i+1]$ atunci se efectuează interschimbarea astfel că $x[i] < x[i+1]$, deci proprietatea devine adevărată şi pentru $i + 1$.

Pe baza acestei proprietăţi a secvenţei de interschimbări se deduce că este suficient să aplicăm această prelucrare succesiv pentru $x[1..n]$, $x[1..n-1]$, ..., $x[1..2]$. Rezultă că o primă variantă a sortării prin interschimbarea elementelor vecine este cea descrisă în Algoritmul 4.5.

**Algoritmul 4.5** Sortare prin interschimbarea elementelor vecine

```
interschimbări1(real x[1..n])
for i ← n, 2, −1 do
  for j ← 1, i − 1 do
    if x[j] > x[j + 1] then
      x[j] ↔ x[j + 1]
    end if
  end for
end for
return (x[1..n])
```

### 4.4.2 Verificarea corectitudinii

Întrucât s-a demonstrat că efectul ciclului interior este că plasează valoarea maximă pe poziţia $i$ rezultă că pentru ciclul exterior poate fi considerată ca invariantă proprietatea $\{x[i+1..n]$ este crescător iar $x[i+1] \geq x[j]$ pentru $j = \overline{1,i}\}$. La ieşirea din ciclul exterior, valoarea contorului $i$ este 1 astfel că se obţine că $x[1..n]$ este ordonat crescător.

### 4.4.3 Analiza complexităţii

Numărul de comparaţii efectuate nu depinde de gradul de sortare al şirului iniţial fiind în orice situaţie:

$$T_C(n) = \sum_{i=2}^{n}\sum_{j=1}^{i-1} 1 = \sum_{i=2}^{n}(i-1) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}.$$

În schimb, numărul de interschimbări depinde de proprietăţile şirului astfel: în cazul cel mai favorabil (şir sortat crescător) se obţine $T_M(n) = 0$, iar în cazul cel mai defavorabil (şir sortat descrescător) se obţine costul $T_M(n) = 3n(n-1)/2$ (o interschimbare presupune efectuarea a 3 atribuiri). Astfel numărul de prelucrări analizate satisface: $n(n-1)/2 \leq T(n) \leq 2n(n-1)$ adică algoritmul prezentat mai sus aparţine clasei $\Theta(n^2)$.

### 4.4.4 Variante ale sortării prin interschimbarea elementelor vecine

Algoritmul `interschimbări1` poate fi îmbunătăţit prin reducerea numărului de comparaţii efectuate, în sensul că nu este necesar întotdeauna să se parcurgă tabloul pentru $i = \overline{2,n}$. De exemplu, dacă tabloul este de la început ordonat ar fi suficientă o singură parcurgere care să verifice că nu este necesară efectuarea nici unei interschimbări. Pornind de la această idee se ajunge la varianta descrisă în Algoritmul 4.6.

**Algoritmul 4.6** Variantă a algoritmului de sortare prin interschimbarea elementelor vecine

```
interschimbări2(real x[1..n])
repeat
  inter ← false
  for i ← 1, n − 1 do
    if x[i] > x[i + 1] then
      x[i] ↔ x[i + 1]
      inter ← true
    end if
  end for
until inter =false
return x[1..n]
```

Pentru acest algoritm numărul de comparaţii efectuate în cazul cel mai favorabil este $T_C(n) = n-1$ iar în cazul cel mai defavorabil este $T_C(n) = \sum_{j=1}^{n}(n-1) = n(n-1)$. În ceea ce priveşte costul interschimbărilor acesta este acelaşi ca pentru prima variantă a algoritmului şi anume $0 \leq T_M(n) \leq$

$3n(n-1)/2$. Prin urmare costul total al prelucrărilor analizate satisface:

$$n-1 \leq T(n) \leq \frac{5n(n-1)}{2}$$

adică algoritmul este din $\Omega(n)$ şi $\mathcal{O}(n^2)$.

Întrucât la sfârşitul şirului se formează un subşir crescător rezultă că nu mai este necesar să se facă comparaţii în acea porţiune. Această porţiune este limitată inferior de cel mai mare indice pentru care s-a efectuat interschimbare. Pornind de la această idee se ajunge la varianta descrisă în Algoritmul 4.7.

**Algoritmul 4.7** Variantă a algoritmului de sortare prin interschimbarea elementelor vecine

```
interschimbări3(real x[1..n])
m ← n
repeat
  t ← 0
  for i ← 1, m − 1 do
    if x[i] > x[i + 1] then
      x[i] ↔ x[i + 1]
      t ← i
    end if
  end for
  m ← t
until t ≤ 1
return x[1..n]
```

Sortarea prin interschimbarea elementelor vecine asigură, la fiecare pas al ciclului exterior, plasarea câte unui element (de exemplu maximul din subtabloul tratat la etapa respectivă) pe poziţia finală. O variantă ceva mai eficientă ar fi ca la fiecare etapă să se plaseze pe poziţiile finale câte două elemente (minimul respectiv maximul din subtabloul tratat la etapa respectivă). Pe de altă parte prin reţinerea indicelui ultimei interschimbări efectuate, atât la parcurgerea de la stânga la dreapta cât şi de la dreapta la stânga se poate limita regiunea analizată cu mai mult de o poziţie atât la stânga cât şi la dreapta (când tabloul conţine porţiuni deja sortate). Această variantă (cunoscută sub numele de "shaker sort") este descrisă în Algoritmul 4.8.

Atâta timp cât condiţia pentru interschimbare este specificată prin inegalitate strictă ($x[i] > x[i+1]$) oricare dintre variantele algoritmului este stabilă.

**Algoritmul 4.8** Varianta "shaker sort"

```
shakersort(real x[1..n])
integer s, d, i, t
s ← 1; d ← n
repeat
  t ← 0
  for i ← s, d − 1 do
    if x[i] > x[i + 1] then
      x[i] ↔ x[i + 1]; t ← i
    end if
  end for
  if t ≠ 0 then
    d ← t; t ← 0
    for i ← d, s + 1, −1 do
      if x[i] < x[i − 1] then
        x[i] ↔ x[i − 1]; t ← i
      end if
    end for
    s ← t
  end if
until t = 0 or s = d
return x[1..n]
```

## 4.5 Limite ale eficienţei algoritmilor de sortare bazaţi pe compararea între elemente

Valorile comparative ale numărului de prelucrări (comparaţii şi mutări asupra elementelor tabloului) pentru algoritmii de sortare descrişi sunt prezentate în Tabelul 4.1. În secţiunea referitoare la sortarea prin inserţie s-a menţionat faptul că algoritmii de sortare bazaţi pe transformări locale (de exemplu, interschimbări ale elementelor vecine) au o complexitate pătratică în cazul cel mai defavorabil.

În cazul general al algoritmilor de sortare ce folosesc compararea între elemente se poate arăta că ordinul de complexitate este cel puţin $n \lg n$. Acest lucru poate fi justificat prin faptul că procesul de sortare poate fi ilustrat printr-un arbore de decizie caracterizat prin faptul că fiecare nod intern corespunde unei comparaţii între două elemente iar frontiera conţine noduri corespunzătoare tuturor permutărilor posibile ale mulţimii elementelor ce trebuie sortate. Figura 4.1 ilustrează arborele de decizie corespunzător sortării prin inserţie în cazul unui tablou cu 3 elemente. Ordinul de complexitate al algoritmului este dat de adâncimea arborelui (numărul de nivele). În cazul unei mulţimi cu $n$ elemente, numărul permutărilor este $n!$ iar întrucât arborele de decizie este binar

| Algoritm | Caz favorabil | | Caz defavorabil | |
|---|---|---|---|---|
| | $T_C(n)$ | $T_M(n)$ | $T_C(n)$ | $T_M(n)$ |
| inserţie1 | $n-1$ | $2(n-1)$ | $\frac{n^2+n-2}{2}$ | $\frac{n^2+3n-4}{2}$ |
| | $3(n-1) \leq T(n) \leq n^2+2n-3$, $(\Omega(n), O(n^2))$ | | | |
| selecţie | $\frac{n(n-1)}{2}$ | 0 | $\frac{n(n-1)}{2}$ | $3(n-1)$ |
| | $\frac{n(n-1)}{2} \leq T(n) \leq \frac{n^2+5n-6}{2}$, $(\Theta(n^2))$ | | | |
| interschimbări1 | $\frac{n(n-1)}{2}$ | 0 | $\frac{n(n-1)}{2}$ | $3\frac{n(n-1)}{2}$ |
| | $\frac{n(n-1)}{2} \leq T(n) \leq 2n(n-1)$, $(\Theta(n^2))$ | | | |
| interschimbări2 | $n-1$ | 0 | $n(n-1)$ | $3\frac{n(n-1)}{2}$ |
| | $n-1 \leq T(n) \leq \frac{5n(n-1)}{2}$,$(\Omega(n), O(n^2))$ | | | |

Tabelul 4.1: Costurile algoritmilor elementari de sortare (număr de comparaţii si de transferuri de elemente) şi ordinele de complexitate corespunzătoare

rezultă că adâncimea lui, $k$, trebuie să satisfacă: $1 + 2 + \ldots + 2^k \geq n!$ adică $2^{k+1} - 1 \geq n!$ ceea ce implică faptul că $k \in \Omega(\lg n!) = \Omega(n \lg n)$. Prin urmare, numărul de comparaţii efectuate, în cel mai defavorabil caz, de către un algoritm de sortare bazat pe compararea elementelor este cel puţin $n \lg n$. Un exemplu de algoritm care atinge această limită inferioară este cel bazat pe utilizarea unei structuri de date speciale numită "heap". Pe lângă acest algoritm (descris în secţiunea următoare) există şi alţi algoritmi care au ordin de complexitate $n \lg n$ fie în cazul cel mai defavorabil (sortarea prin interclasare) sau în cazul mediu (sortarea rapidă). Aceşti algoritmi sunt prezentaţi în capitolul dedicat tehnicii divizării.

## 4.6 Sortarea folosind structura de tip "heap"

Un "heap", numit uneori *ansamblu* sau *movilă* este un tablou de elemente $H[1..n]$ care are proprietatea că pentru fiecare element $H[i]$ cu $1 \leq i \leq \lfloor n/2 \rfloor$ sunt adevărate inegalităţile: $H[i] \geq H[2i]$ şi $H[i] \geq H[2i+1]$ (în cazul în care $2i + 1 \leq n$. O astfel de structură poate fi vizualizată sub forma unui arbore binar aproape complet (arbore în care fiecare nod are doi fii, iar fiecare nivel cu excepţia ultimului este complet). Fii unui nod aflat pe poziţia $i$ în tablou se află pe poziţiile $2i$ (fiul stâng) respectiv $2i + 1$ (fiul drept). In baza aceleiaşi proprietăţi părintele nodului de pe poziţia $i$ în tablou se află pe poziţia $\lfloor i/2 \rfloor$. Câteva exemple de astfel de structură vizualizată arborescent sunt prezentate

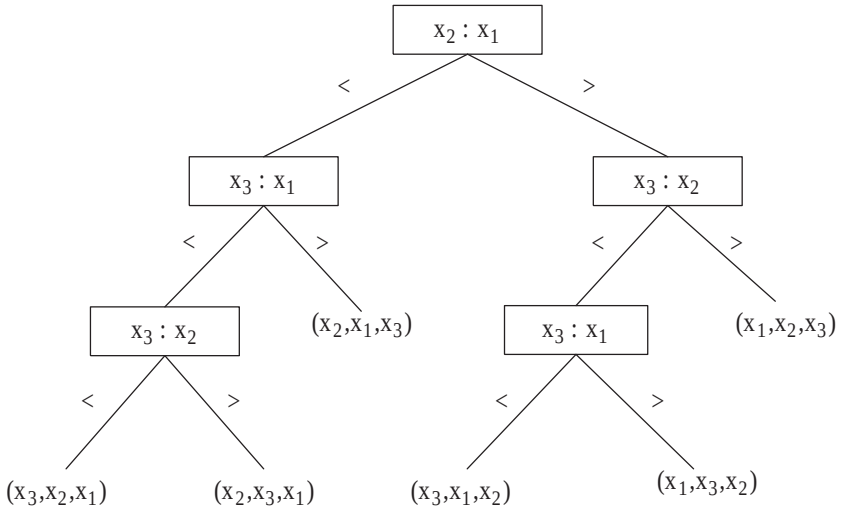


Figura 4.1: Arbore de decizie corespunzător sortării prin inserţie în cazul unui tablou cu trei elemente distincte

în Figura 4.2.

Aceste structuri de date sunt utile atât pentru sortarea eficientă a unui tablou cât şi pentru implementarea cozilor de priorităţi. În această secţiune structura de tip "heap" este folosită doar pentru implementarea unei metode de sortare.

Una dintre cele mai importante prelucrări referitoare la un heap este aceea de a asigura satisfacerea proprietăţii corespunzătoare fiecărui element (nod): valoarea corespunzătoare nodului să fie mai mare decât valorile corespunzătoare fiilor. Metoda de transformare a unui heap astfel încât nodul $i$ să satisfacă proprietatea specifică (în ipoteza că subarborii având rădăcinile în cei doi fii satisfac fiecare proprietatea de heap) este descrisă în Algoritmul 4.9 care primeşte ca date de intrare tabloul corespunzător heapului şi indicele nodului de unde începe procesul de "reparare". Trebuie precizat faptul că nu toate cele $n$ elemente ale tabloului sunt considerate active la un moment dat astfel ca dimensiunea heapului, specificată în algoritm prin $size(H)$ poate fi mai mică decât $n$.

Intrucât un subarbore are cel mult $2n/3$ noduri (cazul cel mai defavorabil este întâlnit când ultimul nivel al arborelului este completat pe jumătate) numărul de comparaţii efectuate, $T(n)$, satisface $T(n) \leq T(n/3) + 2$ ceea ce conduce la faptul că $T(n) \in \mathcal{O}(\lg\, n)$.

Procesul propriu-zis de sortare constă în două etape principale:

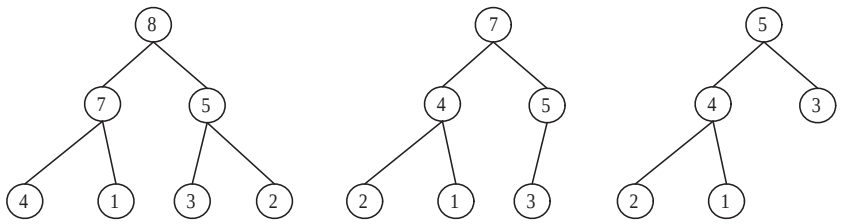


Figura 4.2: Exemple de heap-uri descrise arborescent

**Algoritmul 4.9** Restaurarea proprietăţii de heap pornind de la nodul $i$

```
reHeap(H[1..n],i)
if 2i ≤ size(H) and H[2i] > H[i] then
   imax ← 2i
else
   imax ← i
end if
if 2i + 1 ≤ size(H) and H[2i + 1] > H[imax] then
   imax ← 2i + 1
end if
if imax ≠ i then
   H[i] ↔ H[imax]
   H[1..n] ←reHeap(H,imax)
end if
return  H[1..n]
```

- Construirea, pornind de la tabloul iniţial, al unui heap. Procesul de construire se bazează pe utilizarea algoritmului `reHeap` pentru fiecare element din prima jumătate a tabloului începând cu elementul de pe poziţia $\lfloor n/2 \rfloor$
- Eliminarea succesivă din heap a nodului rădăcină şi plasarea acestuia la sfârşitul tabloului corespunzător heap-ului. La fiecare etapă dimensiunea heapului scade cu un element iar subtabloul ordonat de la sfârşitul zonei corespunzătoare tabloului iniţial creşte cu un element.

Cele două etape sunt descrise în Algoritmul 4.10.

Intrucât algoritmul `heapSort` apelează de $n-1$ ori algoritmul `reHeap` care are ordinul de complexitate $\mathcal{O}(\lg\ n)$ iar cum algoritmul de construire are ordinul $\mathcal{O}(n\ \lg\ n)$ rezultă că `heapSort` are de asemenea ordinul $\mathcal{O}(n\ \lg\ n)$. Merită menţionat faptul că marginea superioară $n\ \lg\ n$ pentru algoritmul de construire

**Algoritmul 4.10** Construirea unui heap şi sortarea pe baza acestuia

```
construireHeap(H[1..n])
size(H) ← n
for i ← ⌊n/2⌋, 1, −1 do
  H[1..n] ← reHeap(H,i)
end for
return H[1..n]
```

```
heapSort(H[1..n])
H[1..n] ←construireHeap(H)
for i ← n, 2, −1 do
  H[i] ↔ H[1]
  size(H) ← size(H) − 1
  H[1..n] ←reHeap(H,1)
end for
return H[1..n]
```

al unui heap este largă şi că se poate demonstra că acesta are de fapt complexitate liniară.

## 4.7 Algoritmi de sortare care nu folosesc compararea între elemente

Dacă se cunosc informaţii suplimentare despre elementele şirului de sortat (de exemplu că aparţin unei mulţimi de forma $\{1, \ldots, m\}$) atunci se poate evita compararea directă între elemente şi în felul acesta se pot obţine algoritmi de complexitate mai mică decât $n\lg n$.

### 4.7.1 Sortarea folosind tabel de frecvenţe

Să considerăm problema sortării unui tablou $x[1..n]$ având elemente din $\{1, 2, \ldots, m\}$. Pentru acest caz particular poate fi utilizat un algoritm de complexitate $\mathcal{O}(m + n)$ bazat pe următoarele idei:

- se construieşte tabelul $f[1..m]$ al frecvenţelor de apariţie ale valorilor elementelor tabloului $x[1..n]$;
- se calculează frecvenţele cumulate asociate;
- se foloseşte tabelul frecvenţelor cumulate pentru a construi tabloul ordonat $y$.

Ideea fundamentală a acestui algoritm este de a determina, pentru fiecare element al tabloului de sortat, câte dintre elemente sunt mai mici decât el. Din acest motiv metoda este cunoscută şi sub numele de sortare prin numărare ("counting sort"). Acest lucru este făcut însă calculând frecvenţe cumulate şi fără a compara direct elementele între ele. O variantă de implementare este descrisă în Algoritmul 4.11. Dacă $m \in \Theta(n)$ atunci algoritmul are complexitate liniară. Dacă însă $m$ este mult mai mare decât $n$ (de exemplu $m \in \Theta(n^2)$)

atunci ordinul de complexitate al algoritmului de sortare este determinat de $m$.

În ceea ce priveşte stabilitatea, atât timp cât ciclul de construire al tabloului $y$ este descrescător, algoritmul este stabil. Spre deosebire de ceilalţi algoritmi de sortare analizaţi până acum, acesta utilizează spaţiu de memorie adiţional de dimensiune cel puţin $n$.

**Algoritmul 4.11** Sortare pe baza unui tablou de frecvenţe

```
CountingSort(integer x[1..n],m)
integer i, f[1..m], y[1..n]
for i ← 1, m do
   f[i] ← 0
end for
for i ← 1, n do
   f[x[i]] ← f[x[i]] + 1
end for
for i ← 2, m do
   f[i] ← f[i − 1] + f[i]
end for
for i ← n, 1, −1 do
   y[f[x[i]]] ← x[i]
   f[x[i]] ← f[x[i]] − 1
end for
for i ← 1, n do
   x[i] ← y[i]
end for
return x[1..n]
```

### 4.7.2 Sortarea pe baza cifrelor

În cazul în care elementele tabloului de sortat sunt numere naturale cu cel mult $k$ cifre, iar $k$ este mic în raport cu $n$ atunci sortarea poate fi realizată în timp liniar în raport cu $n$. Ideea acestei metode este de a realiza succesiv sortarea la nivelul cifrelor numărului pornind de la cifra cea mai puţin semnificativă: se ordonează tabloul în raport cu cifra cea mai puţin semnificativă a fiecarui număr (folosind de exemplu sortarea pe baza tabelului de frecvenţe) după care se sortează în raport cu cifra de rang imediat superior ş.a.m.d. până se ajunge la cifra cea mai semnificativă. Algoritmul 4.12 descrie structura generală precum şi adaptarea algoritmului `CountingSort` în cazul sortării pe baza cifrelor (funcţia `cs`).

Algoritmul `cs` se apelează pentru $m = 9$ deci este de ordinul $\mathcal{O}(n)$. Pe de altă parte pentru ca algoritmul de sortare pe baza cifrelor să funcţioneze

**Algoritmul 4.12** Sortarea pe baza cifrelor

```
radixsort(integer x[1..n], k)
integer i
for i ← 0, k − 1 do
   x[1..n] ← cs(x[1..n], 9, i)
end for
return x[1..n]
```

```
cs(integer x[1..n],m,c )
integer i, j, f[0..m], y[1..n]
for i ← 0, m do
   f[i] ← 0
end for
for i ← 1, n do
   j ← (x[i]DIV putere(10, c))MOD 10
   f[j] ← f[j] + 1
end for
for i ← 1, m do
   f[i] ← f[i − 1] + f[i]
end for
for i ← n, 1, −1 do
   j ← (x[i]DIV putere(10, c))MOD 10
   y[f[j]] ← x[i]
   f[j] ← f[j] − 1
end for
for i ← 1, n do
   x[i] ← y[i]
end for
return x[1..n]
```

corect este necesar ca subalgoritmul de sortare prin numărare apelat să fie stabil. Dacă $m = 10^k$ este semnificativ mai mare decât $n$ atunci algoritmul nu este eficient. Dacă însă $k$ este mult mai mic decât $n$ atunci un algoritm de complexitate $\mathcal{O}(kn)$ ar putea fi acceptabil.

## 4.8 Probleme

**Problema 4.1** Se consideră un tablou ale cărui elemente conţin două tipuri de informaţii: nume şi nota. Să se ordoneze descrescător după notă, iar pentru aceeaşi notă crescător după nume.

*Indicaţie.* Se sortează tabloul crescător după nume, după care se aplică un algoritm stabil pentru sortarea descrescătoare după notă.

**Problema 4.2** Propuneţi un algoritm care sortează crescător elementele de pe poziţiile impare ale unui tablou şi descrescător cele de pe poziţii pare.

*Indicaţie.* Se aplică ideea de la $h$-sortarea prin inserţie pentru $h = 2$ însă în prima etapă se sortează crescător iar în a doua se sortează descrescător.

**Problema 4.3** Se consideră o matrice pătratică de dimensiune $n$ şi elemente

reale, $a_{ij}$. Să se reorganizeze matricea prin interschimbări de linii şi coloane astfel încât elementele diagonalei principale să fie ordonate crescător.

*Indicaţie.* Se aplică algoritmul de sortare prin selecţie asupra şirului ($a_{11}$, $a_{22}$, ..., $a_{nn}$) însă interschimbarea elementului $a_{ii}$ cu elementul $a_{jj}$ se realizează prin interschimbarea intregii linii $i$ cu linia $j$ şi a coloanei $i$ cu coloana $j$.

**Problema 4.4** Se consideră algoritmul:

```
alg(x[1..n])
for i ← 1, n − 1 do
  if i MOD 2 = 1 then
    for j ← 1, n − 1, 2 do
      if x[j] > x[j + 1] then
        x[j] ↔ x[j + 1]
      end if
    end for
  else
    for j ← 2, n − 1, 2 do
      if x[j] > x[j + 1] then
        x[j] ↔ x[j + 1]
      end if
    end for
  end if
end for
return x[1..n]
```

Să se stabilească care este efectul algoritmului asupra tabloului $x[1..n]$ şi să se stabilească ordinul de complexitate.

*Indicaţie.* Algoritmul asigură ordonarea crescătoare a lui $x[1..n]$ iar ordinul de complexitate este $\mathcal{O}(n^2)$.

**Problema 4.5** Să se adapteze algoritmul de sortare pe baza tabelului de frecvenţe (Algoritmul 4.11) astfel încât tabloul $y$ să conţine elementele lui $x$ în ordine descrescătoare.

*Indicaţie.* Atribuirea $y[f[x[i]]] \leftarrow x[i]$ se înlocuieşte cu $y[n+1-f[x[i]]] \leftarrow x[i]$, iar pentru a asigura stabilitatea parcurgerea lui $x[1..n]$ se va face începând cu primul element.

**Problema 4.6** Se consideră un set de valori din $\{1, \ldots, m\}$. Preprocesaţi acest set folosind un algoritm de complexitate $\mathcal{O}(\max\{m, n\})$ astfel încât răspunsul la întrebarea "câte elemente se află în intervalul $[a, b]$ ?", $a, b \in \{1, \ldots, m\}$ să poată fi obţinut în timp constant.

*Indicaţie.* În tabelul frecvenţelor cumulate (care poate fi construit in $\mathcal{O}(m+n)$) elementul $f[b]$ reprezintă numărul de elemente din tablou mai mici sau egale decât $b$, iar $f[a-1]$ reprezintă numărul elementelor strict mai mici decât $a$.

Astfel numărul de elemente cuprinse în $[a, b]$ se poate obţine printr-o singură operaţie: $f[b] - f[a-1]$.

**Problema 4.7** Propuneţi un algoritm de complexitate liniară pentru ordonarea crescătoare a unui tablou constituit din $n$ valori întregi aparţinând mulţimii $\{0, 1, \ldots, n^2 - 1\}$.

*Indicaţie.* Se vor interpreta valorile ca fiind reprezentate în baza $n$. Astfel fiecăreia dintre cele $n$ valori îi corespunde o pereche de "cifre", $c_0$,$c_1 \in \{0, \ldots, n-1\}$ obţinute ca rest respectiv cât al împărţirii la $n$. Se poate astfel aplica algoritmul sortării pe baza cifrelor în cazul în care valoarea maximă a "cifrelor" este $m = n - 1$ iar numărul de "cifre" este $k = 2$. Ordinul de complexitate este în acest caz $\mathcal{O}(k(m+n))$ adică $\mathcal{O}(n)$.

**Problema 4.8** Se consideră un tablou constituit din triplete de trei valori întregi corespunzătoare unei date calendaristice (zi,luna,an). Propuneţi un algoritm eficient de ordonare crescătoare după valoarea datei.

*Indicaţie.* Se aplică un algoritm stabil de complexitate liniară (de exemplu sortarea prin numărare) succesiv pentru zi, luna şi an.

# Capitolul 5

# Tehnica reducerii

Tehnica reducerii se bazează pe ideea de a reduce rezolvarea unei probleme de dimensiune dată la rezolvarea unei probleme similare de dimensiune mai mică. De cele mai multe ori dimensiunea este redusă prin scăderea unei constante (de la problema de dimensiune $n$ se ajunge la problema de dimensiune $n-1$, cum se întâmplă la calculul factorialului) însă poate fi redusă şi prin împărţirea la o constantă (ca în exemplul cu înmulţirea "à la russe").

## 5.1 Motivaţie

Considerăm problema calculului lui $x^n$ pentru $x > 0$ şi $n = 2^m$, $m \geq 1$ fiind un număr natural. Metoda clasică, bazată pe tehnica forţei brute, pentru calculul lui $x^n$ (pentru un $n$ natural arbitrar) este descrisă în algoritmul `putere1` şi se observă uşor că are complexitatea $\Theta(n)$. Dacă însă se analizează cazul particular $n = 2^m$ se observă că $x^n = x^{n/2} \cdot x^{n/2}$ căci $x^{2^m} = x^{2^{(m-1)}} \cdot x^{2^{(m-1)}}$. Prin urmare, pentru a calcula $x^n$ este suficient să se calculeze $x^{n/2}$ şi să se ridice la pătrat. La rândul său, calculul lui $x^{n/2}$ poate fi redus la calculul lui $x^{n/4}$ şi la o ridicare la pătrat ş.a.m.d. Descompunerea poate continua până se ajunge la $x^2$ al cărui calcul este simplu. O astfel de variantă ascendentă de calcul a lui $x^{2^m}$ este descrisă în algoritmul `putere2`.

Folosind ca invariant pentru prelucrarea repetitivă afirmaţia: $\{ p = x^{2^{(i-1)}} \}$ se poate demonstra cu uşurinţă că algoritmul `putere2` calculează $x^{2^m}$. Pe de altă parte se observă că prelucrarea este de complexitate $\Theta(m) = \Theta(\lg(n))$. Reducerea complexităţii derivă din faptul că la fiecare etapă se calculează valoarea unui singur factor din cei doi, întrucât sunt identici. Ideea de rezolvare a acestei probleme este comună tehnicilor de reducere care se bazează la rezolvarea unei probleme de dimensiune $n$ pe rezolvarea unei probleme similare dar de dimensiune mai mică. Reducerea dimensiunii continuă până când se ajunge la o problemă de dimensiune suficient de mică pentru a putea fi rezol-

**Algoritmul 5.1** Calculul puterii unui număr folosind tehnica forţei (`putere1`) brute respectiv tehnica reducerii (`putere2`)

```
putere1 (real x,integer n)
p ← 1
for i ← 1, n do
  p ← p * x
end for
return p
```

```
putere2 (real x,integer m)
p ← x
for i ← 1, m do
  p ← p * p
end for
return p
```

vată direct (de exemplu $n = 2$ sau $m = 1$). O descriere a acestei metode de rezolvare care ilustrează mai bine ideea reducerii dimensiunii este cea în care se foloseşte o abordare descendentă (algoritmul `putere3`). Dacă se transmit ca parametri valorile $x$ şi $m$ algoritmul pentru calculul lui $x^{2^m}$ poate fi descris ca în `putere4`.

**Algoritmul 5.2** Variante recursive ale algoritmilor de calcul a puterii unui număr

```
putere3 (real x,integer n)
if n = 2 then
  return x * x
else
  p ←putere3(x, n/2)
  return p * p
end if
```

```
putere4 (real x,integer m)
if m = 1 then
  return x * x
else
  p ← putere4(x, m − 1)
  return p * p
end if
```

Algoritmii `putere3` şi `putere4` prezintă o particularitate: în cadrul prelucrărilor pe care le efectuează există şi un auto-apel (în cadrul funcţiei se apelează aceeaşi funcţie). Astfel de algoritmi se numesc *recursivi*.

Exemplul de mai sus ilustrează faptul că folosind ideea reducerii rezolvării unei probleme la rezolvarea unei probleme similare, dar de dimensiune mai mică, *poate* conduce la reducerea complexităţii. În plus există probleme pentru care abordarea rezolvării în această manieră este mai uşoară conducând la algoritmi mai intuitivi.

Trebuie menţionat totodată că nu întotdeauna tehnicile din această categorie conduc la o reducere a complexităţii. Un exemplu în acest sens îl reprezintă calculul factorialului (după cum se va ilustra în Exemplul 5.1, aplicând tehnica reducerii se obţine un algoritm de complexitate $\Theta(n)$ la fel ca prin aplicarea tehnicii forţei brute).

## 5.2 Analiza algoritmilor recursivi

Ultimii algoritmi prezentaţi în secţiunea anterioară fac parte din categoria algoritmilor recursivi. Aceştia sunt utilizaţi pentru a descrie prelucrări ce se pot specifica prin ele însele. Un algoritm recursiv este caracterizat prin:

- *Auto-apel.* Se auto-apelează cel puţin o dată pentru alte valori ale parametrilor. Valorile parametrilor corespunzătoare succesiunii de apeluri trebuie să asigure apropierea de satisfacerea unei condiţii de oprire.
- *Condiţie de oprire.* Specifică situaţia în care rezultatul se poate obţine prin calcul direct fără a mai fi necesar auto-apelul.

Ca urmare a cascadei de auto-apeluri un algoritm recursiv realizează de fapt o prelucrare repetitivă, chiar dacă aceasta nu este explicită. Un exemplu simplu de prelucrare repetitivă descrisă recursiv este cea corespunzătoare determinării celui mai mare divizor comun a două numere naturale nenule, $a \geq b > 0$. Variantele recursive de determinare a celui mai mare divizor comun se bazează pe proprietăţile pe care le satisface acesta şi anume:

$$cmmdc(a,b) = \begin{cases} a & b = 0 \\ cmmdc(b, a \text{ MOD } b) & b \neq 0 \end{cases}$$

respectiv

$$cmmdc(a,b) = \begin{cases} a & a = b \\ cmmdc(b, a-b) & a > b \\ cmmdc(a, b-a) & a < b \end{cases}$$

Aceste relaţii pot fi uşor descrise în manieră recursivă (vezi Algoritmul 5.3).

**Algoritmul 5.3** Variante recursive ale algoritmului lui Euclid

```
cmmdcr1 (integer a, b)
if b = 0 then
  rez ← a
else
  rez ← cmmdcr1(b, a MOD b)
end if
return rez
```

```
cmmdcr2 (integer a, b)
if a = b then
  rez ← a
else if a > b then
  rez ← cmmdcr2(b, a − b)
else
  rez ← cmmdcr2(a, b − a)
end if
return rez
```

Exemplele prezentate până acum se caracterizează prin recursivitate *simplă* şi *directă*. Un algoritm este considerat simplu recursiv dacă se auto-apelează o

singură dată şi multiplu recursiv dacă conţine două sau mai multe auto-apeluri (de exemplu, algoritmul `hanoi` din secţiunea următoare).

Există şi posibilitatea ca algoritmul să nu se auto-apeleze direct ci indirect prin intermediul altui algoritm. De exemplu algoritmul *A1* apelează algoritmul *A2* iar acesta apelează algoritmul *A1*. În acest caz este vorba de *recursivitate indirectă*.

Conceptul de recursivitate poate fi utilizată şi în contextul definirii unor noţiuni. De exemplu noţiunea de expresie aritmetică poate fi definită astfel: "o expresie aritmetică este constituită din operanzi şi operatori; un operand poate fi o constantă, o variabilă sau o altă *expresie*". Descrierea recursivă permite inclusiv specificarea unor structuri infinite folosind un set finit de reguli.

Algoritmii recursivi sunt adecvaţi pentru rezolvarea problemelor sau prelucrarea datelor descrise în manieră recursivă.

### 5.2.1 Arbori de apel

Pentru a ilustra modul de lucru al unui algoritm recursiv poate fi util să se reprezinte grafic structura de apeluri şi reveniri cu returnarea rezultatului obţinut. În cazul unui algoritm recursiv arbitrar structura de apeluri este una ierarhică conducând la un *arbore de apel*. În cazul recursivităţii simple arborele de apel degenerează într-o structură liniară (Figura 5.1).

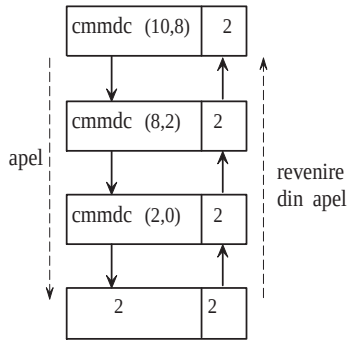


Figura 5.1: Structura de apel pentru algoritmul `cmmdcr1`

### 5.2.2 Verificarea corectitudinii algoritmilor recursivi

Dacă relaţia de recurenţă ce descrie legătura dintre soluţiile corespunzătoare instanţelor de diferite dimensiuni ale problemei este corectă atunci şi algoritmul care o implementează este corect. Pe de altă parte întrucât un algoritm recursiv

specifică o prelucrare repetitivă implicită pentru a demonstra corectitudinea acestuia este suficient să se arate că:

- Există o aserţiune referitoare la starea algoritmului care are proprietăţile: (i) este adevărată pentru cazul particular (când e satisfăcută condiţia de oprire); (ii) rămâne adevărată la revenirea din apelul recursiv şi după efectuarea eventualelor prelucrări locale; (iii) pentru valorile de apel ale parametrilor de intrare implică postcondiţia.
- Condiţia de oprire este satisfăcută după o succesiune finită de apeluri recursive.

Pentru algoritmul `cmmdcr1` o aserţiune invariantă este $\{rez = cmmdc(a, b)\}$. Pentru cazul particular $b = 0$ avem $rez = a = cmmdc(a, 0) = cmmdc(a, b)$. Dacă $rez = cmmdc(a, b)$ înainte de apelul recursiv întrucât $cmmdc(a, b) = cmmdc(b, a \text{ MOD } b)$ rezultă că $rez = cmmdc(a, b)$ este adevărată şi după apelul recursiv. Pentru algoritmul `cmmdcr2` proprietatea invariantă este de asemenea $\{rez = cmmdc(a, b)\}$. În ambele situaţii condiţia de oprire va fi satisfăcută după un număr finit de apeluri recursive, datorită proprietăţilor restului unei împărţiri întregi.

### 5.2.3 Analiza complexităţii algoritmilor recursivi

Considerăm o problemă de dimensiune $n$ şi algoritmul recursiv având forma generală descrisă în `algrec`.

`algrec`$(n)$
**if** $n = n_0$ **then**
  $P_1$
**else**
  $algrec(h(n))$
**end if**

În algoritmul `algrec`, $h(n)$ este o funcţie descrescătoare cu proprietatea că există $k$ cu $h^{(k)}(n) = (h \circ h \circ \ldots \circ h)(n) = n_0$ (aceasta înseamnă că după $k$ apeluri recursive este satisfăcută condiţia de oprire). Dacă prelucrarea $P_1$ are cost constant ($c_0$), iar determinarea lui $h(n)$ are costul $c$ atunci costul algoritmului `algrec` poate fi descris prin următoarea relaţie de recurenţă:

$$T(n) = \begin{cases} c_0 & \text{dacă } n = n_0 \\ T(h(n)) + c & \text{dacă } n > n_0 \end{cases}$$

Pentru determinarea expresiei lui $T(n)$ pornind de la relaţia de recurenţă se poate folosi una dintre metodele următoare:

- *Metoda substituţiei directe.* Pornind de la relaţia de recurenţă se intuieşte forma generală a lui $T(n)$ după care se demonstrează prin inducţie matematică validitatea expresiei lui $T(n)$.

- *Metoda iteraţiei.* Se scrie relaţia de recurenţă pentru $n$, $h(n)$, $h(h(n))$, ..., $n_0$ după care se substituie succesiv $T(h(n))$, $T(h(h(n)))$ ş.a.m.d. Din relaţia obţinută, pe baza unor calcule algebrice rezultă expresia lui $T(n)$. Metoda mai este cunoscută şi ca metoda *substituţiei inverse*.

**Exemplul 5.1** Să considerăm cazul $h(n) = n - 1$. Prin metoda iteraţiei se obţine:

$$\begin{array}{lcl} T(n) & = & T(n-1) + c \\ T(n-1) & = & T(n-2) + c \\ \vdots & & \vdots \\ T(n_0+1) & = & T(n_0) + c \\ T(n_0) & = & c_0 \end{array}$$

Prin însumarea tuturor relaţiilor şi reducerea termenilor corespunzători se obţine: $T(n) = c(n - n_0) + c_0$ pentru $n > n_0$.

**Exemplul 5.2** Considerăm algoritmul recursiv (`putere3`) pentru calculul puterii $x^n = x^{2^m}$. Dacă notăm numărul înmulţirilor efectuate cu $T(n)$ se obţine relaţia de recurenţă:

$$T(n) = \begin{cases} 1 & \text{dacă } n = 2 \\ T(n/2) + 1 & \text{dacă } n > 2 \end{cases}$$

Aplicând tehnica iteraţiei obţinem:

$$\begin{array}{lcl} T(n) & = & T(n/2) + 1 \\ T(n/2) & = & T(n/2^2) + 1 \\ \vdots & & \vdots \\ T(4) & = & T(2) + 1 \\ T(2) & = & 1 \end{array}$$

Cum numărul relaţiilor de mai sus este $m$, prin însumarea tuturor şi reducerea termenilor corespunzători se obţine $T(n) = m = \lg n$.

## 5.3 Principiul tehnicii reducerii

Tehnica reducerii se bazează pe relaţia ce există între soluţia unei probleme şi soluţia unei instanţe de dimensiune redusă a aceleiaşi probleme. De regulă reducerea dimensiunii se bazează pe scăderea unei constante (în majoritatea situaţiilor 1) din dimensiunea problemei, însă poate fi realizată şi prin împărţirea dimensiunii problemei la o altă valoare (în acest caz abordarea este similară cu cea de la tehnica divizării). Să considerăm câteva exemple.

**Exemplul 5.3** (*Calculul factorialului*) Cel mai simplu exemplu este cel al calculului factorialului. Relaţia de la care se porneşte este:

$$n! = \begin{cases} 1 & \text{dacă } n = 0 \\ (n-1)! \cdot n & \text{dacă } n > 0 \end{cases}$$

Varianta recursivă de calcul a factorialului este descrisă în algoritmul `factrec`.

**Algoritmul 5.4** Variantă recursivă pentru calculul factorialului

```
factrec (integer n)
if n = 0 then
   f ← 1
else
   f ← factrec(n − 1) ∗ n
end if
return f
```

Notând cu $T(n)$ numărul de operaţii de înmulţire efectuate se obţine relaţia de recurenţă:

$$T(n) = \begin{cases} 0 & \text{dacă } n = 0 \\ T(n-1) + 1 & \text{dacă } n > 0 \end{cases}$$

Din această relaţie se poate intui că $T(n) = n$. Demonstrăm acest lucru prin inducţie matematică după $n$. Pentru $n = 1$ se obţine $T(1) = T(0) + 1 = 1$. Presupunem că $T(n-1) = n-1$. Din relaţia de recurenţă vom obţine $T(n) = T(n-1)+1 = n$. Deci într-adevăr $T(n) = n$. Acelaşi rezultat se obţine aplicând tehnica iteraţiei:

$$\begin{array}{lcl} T(n) & = & T(n-1) + 1 \\ T(n-1) & = & T(n-2) + 1 \\ \vdots & & \vdots \\ T(1) & = & T(0) + 1 \\ T(0) & = & 0 \end{array}$$

Prin însumarea tuturor relaţiilor şi efectuarea reducerilor se obţine: $T(n) = n$.

**Exemplul 5.4** (*Înmulţirea "à la russe"*) Reducerea dimensiunii se poate realiza nu doar prin scăderea unei constante ci şi prin împărţirea la o constantă. Cazul cel mai frecvent este acela al împărţirii dimensiunii problemei la 2. Un exemplu simplu în acest sens este cel al înmulţirii "à la russe". Regula de calcul în acest caz este:

$$a \cdot b = \begin{cases} 0 & \text{dacă } a = 0 \\ \frac{a}{2} \cdot 2b & \text{dacă a este par} \\ \frac{a-1}{2} \cdot 2b + b & \text{dacă a este impar} \end{cases}$$

Metoda este descrisă în variantă recursivă în Algoritmul 5.5.

**Algoritmul 5.5** Variantă recursivă pentru înmulţirea "à la russe"

```
prodrec(integer a,b)
if a = 0 then
  return 0
else if a MOD 2 = 0 then
  return prodrec(a DIV 2, 2 * b)
else
  return prodrec((a − 1) DIV 2, 2 * b) + b
end if
```

Pentru a analiza complexitatea algoritmului considerăm că dimensiunea problemei este reprezentată de perechea $(a, b)$ iar operaţia dominantă este împărţirea lui $a$ la 2 (celelalte operaţii, adică înmulţirea lui $b$ cu 2 şi adunarea se efectuează de acelaşi număr de ori). Astfel, pentru timpul de execuţie $T(a, b)$ se poate scrie relaţia de recurenţă:

$$T(a,b) = \begin{cases} 0 & \text{dacă } a = 0 \\ T(a/2, 2b) + 1 & \text{dacă } a > 0 \end{cases}$$

Aplicăm metoda iteraţiei pentru cazul particular $a = 2^k$:

$$\begin{array}{lcl} T(2^k, b) & = & T(2^{k-1}, 2b) + 1 \\ T(2^{k-1}, 2b) & = & T(2^{k-2}, 2^2 b) + 1 \\ \vdots & & \vdots \\ T(2, 2^{k-1}b) & = & T(1, 2^k b) + 1 \\ T(1, 2^k b) & = & T(0, 2^{k+1} b) + 1 \\ T(0, 2^{k+1} b) & = & 0 \end{array}$$

Însumând relaţiile, rezultă $T(a, b) = k + 1 = \lg a + 1$. Se observă că timpul de execuţie depinde doar de valoarea lui $a$ şi pentru $a = 2^k$ avem $T(a) \in \Theta(\lg a)$. Extinderea acestui rezultat pentru valori arbitrare ale lui $n$ se bazează pe următorul rezultat cunoscut sub numele de regula funcţiilor netede ("smoothness rule").

**Propoziţia 5.1** *Dacă $T(n)$ satisface următoarele proprietăţi:*

*(i) $T(n)$ este crescătoare pentru valori ale lui $n$ suficient de mari;*

*(ii) $T(n) \in \Theta(f(n))$ pentru $n = 2^m$;*

*iar $f(n)$ satisface condiţia de netezime ($f(cn) \in \Theta(f(n))$ pentru orice constantă pozitivă, c) atunci $T(n) \in \Theta(f(n))$ pentru orice valoare a lui $n$.*

Rezultatul de mai sus este adevărat şi pentru celelalte două clase de complexitate ($\mathcal{O}$ şi $\Omega$). Condiţiile din Propoziţia 5.1 ($T(n)$ crescătoare pentru valori mari ale lui $n$ şi $f(cn) \in \Theta(f(n))$) sunt satisfăcute în toate cazurile cand $f$ este polinomială. Condiţia $f(cn) \in \Theta(f(n))$ nu este însă satisfăcută pentru funcţii $f$ cu creştere rapidă, de exemplu $f(n) = a^n$ cu $a > 1$.

Revenind la exemplul 5.4, întrucât $T(a)$ este crescătoare pentru valori mari ale lui $a$ iar $\lg ca = \lg a + \lg c \in \Theta(\lg a)$ rezultă că rezultatul obţinut pentru cazul $a = 2^k$ este valabil şi pentru cazul general.

## 5.4 Aplicaţii

### 5.4.1 Generarea permutărilor

Să considerăm problema generării permutărilor de ordin $n$. Există mai multe moduri de a aplica tehnica reducerii. O variantă porneşte de la ideea că o permutare de ordin $n$ se poate obţine dintr-o permutare de ordin $n-1$ prin plasarea succesivă a valorii $n$ pe toate cele $n$ poziţii posibile. Astfel dacă $n = 3$ există două permutări de ordin $n - 1 = 2$: $(1, 2)$ şi $(2, 1)$. Pentru fiecare dintre acestea valoarea 3 poate fi inserată în fiecare dintre cele trei poziţii posibile: prima, a doua şi a treia conducând la $(3, 1, 2)$, $(1, 3, 2)$, $(1, 2, 3)$ respectiv la $(3, 2, 1)$, $(2, 3, 1)$, $(2, 1, 3)$. Această idee ar corespunde unei abordări ascendente ("bottom-up" - pornind de la o permutare de ordin dat se construieşte o permutare de ordin imediat superior).

Pentru o abordare descendentă ("top-down") a problemei (o permutare de ordin $n$ se specifică prin permutări de ordin $n-1$) să observăm că pentru a genera toate permutările de ordin $n$ este suficient să plasăm pe poziţia $n$ succesiv toate valorile posibile (din $\{1, 2, \ldots, n\}$) şi pentru fiecare valoare astfel plasată să generăm toate permutările corespunzătoare valorilor aflate pe primele $n-1$ poziţii. Pentru a le genera pe acestea se folosesc permutările de ordin $n-2$ până se ajunge la permutări de ordin 1 (o singură permutare care constă chiar din valoarea aflată pe poziţia 1).

In descrierea algoritmului 5.6 presupunem că permutările se vor obţine într-un tablou $x[1..n]$ accesat în comun de către toate (auto)apelurile algoritmului şi iniţializat astfel încât $x[i] = i$ pentru fiecare poziţie $i$. În momentul în care $x$ conţine o permutare, aceasta este afişată. Algoritmul va fi descris pentru un parametru de intrare generic, $k$, şi va fi apelat pentru $k = n$.

**Algoritmul 5.6** Generarea tuturor permutărilor de ordin $n$

```
permutari(integer k)
if k = 1 then
  write x[1..n]
else
  for i ← 1, k do
    x[i] ↔ x[k]
    permutari(k − 1)
    x[i] ↔ x[k]
  end for
end if
```

Aplicând acest algoritm, permutările de ordin 3 se obţin în ordinea următoare: $(2,3,1)$, $(3,2,1)$, $(3,1,2)$, $(1,3,2)$, $(2,1,3)$, $(1,2,3)$. Pentru a analiza complexitatea algoritmului contorizăm numărul de interschimbări efectuate şi observăm că acesta satisface:

$$T(k) = \begin{cases} 0 & \text{dacă } k = 1 \\ k(T(k-1)+2) & \text{dacă } k > 1 \end{cases}$$

Aplicăm tehnica iteraţiei şi obţinem:

$$\begin{array}{lcl|l} T(k) & = & kT(k-1)+2k & \\ T(k-1) & = & (k-1)T(k-2)+2(k-1) & \cdot k \\ T(k-2) & = & (k-2)T(k-3)+2(k-2) & \cdot k(k-1) \\ \vdots & & \vdots & \\ T(2) & = & 2T(1)+2 & \cdot k(k-1)\cdots 3 \\ T(1) & = & 0 & \cdot k(k-1)\cdots 3\cdot 2 \end{array}$$

Înmulţind fiecare relaţie cu factorii specificaţi în ultima coloană şi însumând toate relaţiile se obţine: $T(k) = k! + 2(k(k-1)\cdots 4 + \ldots + k(k-1) + k)$. Prin urmare pentru $k = n$ se obţine $T(n) \in \Omega(n!)$ şi $T(n) \in \mathcal{O}(n \cdot n!)$ . Ordinul mare de complexitate este de aşteptat având în vedere că se generează $n!$ permutări.

### 5.4.2 Problema turnurilor din Hanoi

O problemă clasică ce permite ilustrarea ideii de la tehnica reducerii este problema turnurilor din Hanoi. Se consideră trei vergele plasate vertical şi identificate prin $s$ (sursă), $d$ (destinaţie) şi $i$ (intermediar). Pe vergeaua $s$ se află dispuse $n$ discuri în ordinea descrescătoare a razelor (primul disc are raza maximă). Se cere să se transfere toate discurile pe vergeaua $d$ astfel încât să fie plasate în aceeaşi ordine. Se poate folosi vergeaua $i$ ca intermediar cu restricţia că în orice moment peste un disc se află doar discuri de rază mai mică. Ideea rezolvării

este: "se transferă $n-1$ discuri de pe $s$ pe $i$ folosind $d$ ca intermediar; se transferă discul rămas pe $s$ direct pe $d$; se transferă cele $n-1$ discuri de pe $i$ pe $d$ folosind $s$ ca intermediar". Această idee poate fi descrisă simplu în manieră recursivă (Algoritmul 5.7).

**Algoritmul 5.7** Problema turnurilor din Hanoi

```
hanoi(integer n,s,d,i)
if n = 1 then
  s → d (transferă de la s la d)
else
  hanoi(n − 1, s, i, d)
  s → d (transferă de la s la d)
  hanoi(n − 1, i, d, s)
end if
```

În algoritmul `hanoi` primul argument specifică numărul de discuri ce vor fi transferate, al doilea indică vergeaua sursă, al treilea vergeaua destinaţie iar ultimul pe cea folosită ca intermediar. Prelucrarea $s \to d$ specifică faptul că va fi transferat discul de pe vergeaua $s$ pe vergeaua $d$. Pentru cazul a trei discuri procesul de transfer este ilustrat în fig 5.2 iar structura apelurilor recursive în fig. 5.3. Succesiunea mutărilor este obţinută parcurgând blocurile haşurate de la stânga la dreapta: $s \to d$, $s \to i$, $d \to i$, $s \to d$, $i \to s$, $i \to d$, $s \to d$.

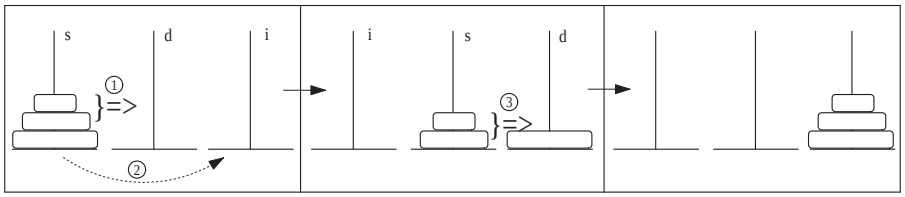


Figura 5.2: Ideea de rezolvare a problemei turnurilor din Hanoi ($n = 3$)

Pentru a analiza complexitatea, contorizăm numărul de mutări de discuri. Se observă că:

$$T(n) = \begin{cases} 1 & \text{dacă } n = 1 \\ 2T(n-1) + 1 & \text{dacă } n > 1 \end{cases}$$

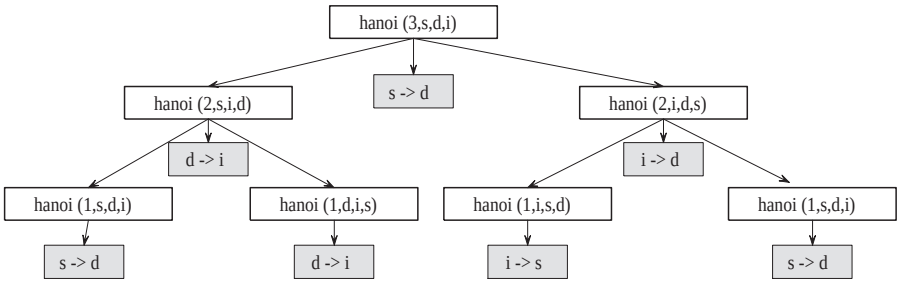


Figura 5.3: Arborele de apel pentru algoritmul `hanoi` când $n = 3$

Prin metoda iteraţiei se obţine:

$$\begin{array}{lcl|l} T(n) & = & 2T(n-1)+1 & \\ T(n-1) & = & 2T(n-2)+1 & \cdot 2^1 \\ T(n-2) & = & 2T(n-3)+1 & \cdot 2^2 \\ \vdots & & \vdots & \\ T(2) & = & 2T(1)+1 & \cdot 2^{n-2} \\ T(1) & = & 1 & \cdot 2^{n-1} \end{array}$$

Însumând relaţiile rezultă $T(n) = 1 + 2^1 + 2^2 + \ldots + 2^{n-1} = 2^n - 1$ adică algoritmul are ordinul de complexitate $\Theta(2^n)$.

### 5.4.3 Căutare binară

Să considerăm problema căutării unei valori $v$ într-un tablou $a[1..n]$ ordonat crescător. Tehnica reducerii se poate aplica aici astfel:

- se alege un element $a[j]$ din $a[1..n]$ care se compară cu $v$;
- dacă $v = a[j]$ atunci a fost găsit elementul căutat;
- dacă $v < a[j]$ atunci căutarea continuă în subtabloul $a[1..j-1]$ altfel se continuă în subtabloul $a[j+1..n]$.

Alegerea cea mai naturală pentru $j$ este să fie cât mai aproape de mijlocul tabloului, astfel dimensiunea problemei este redusă prin împărţire la 2. Un astfel de proces de căutare este cunoscut sub numele de *căutare binară* şi o primă variantă de implementare este descrisă în Algoritmul 5.8 cu $s$ şi $d$ delimitând zona din tablou unde se concentrează la un moment dat căutarea. La început se caută în întreg tabloul astfel că la primul apel avem $s = 1$ şi $d = n$.

Variante iterative ale tehnicii căutării binare sunt descrise în Algoritmul 5.9. Varianta `cautbin3` evită specificarea compararea de două ori în aceeaşi

**Algoritmul 5.8** Variantă recursivă a algoritmului de căutare binară

```
cautbin1(a[s..d],v)
if s > d then
  return false
else
  m ← ⌊(s + d)/2⌋
  if a[m] = v then
    return true
  else if v < a[m] then
    return cautbin1(a[s..m − 1], v)
  else
    return cautbin1(a[m + 1..d], v)
  end if
end if
```

iteraţie a valorii căutate cu elemente ale tabloului ($a[m] = v$ şi $v < a[m]$) şi este mai uşor de analizat însă nu conduce neapărat la un număr mai mic de comparaţii.

**Algoritmul 5.9** Variante iterative ale algoritmului de căutare binară

```
cautbin2(a[1..n],v)
s ← 1
d ← n
gasit ← false
while (s ≤ d) and (gasit =false) do
  m ← ⌊(s + d)/2⌋
  if a[m] = v then
    gasit ← true
  else if v < a[m] then
    d ← m − 1
  else
    s ← m + 1
  end if
end while
return gasit
```

```
cautbin3(a[1..n],v)
s ← 1
d ← n
while s < d do
  m ← ⌊(s + d)/2⌋
  if v ≤ a[m] then
    d ← m
  else
    s ← m + 1
  end if
end while
if v = a[s] then
  return true
else
  return false
end if
```

Pentru a analiza algoritmul căutării binare să considerăm că $T(n)$ reprezintă numărul maxim de comparaţii (la fiecare etapă se contorizează o singură comparaţie) efectuate asupra elementelor tabloului (se atinge în cazul cel mai defavorabil, când $v$ nu se află în tablou). Relaţia de recurenţă corespunzătoare

este:

$$T(n) = \begin{cases} 1 & \text{dacă } n = 1 \\ T(\lfloor n/2 \rfloor) + 1 & \text{dacă } n > 1 \end{cases}$$

Pentru stabili valoarea lui $T(n)$ să analizăm pentru început cazul particular $n = 2^m$. Aplicând metoda iteraţiei obţinem:

$$\begin{array}{lcllcl} T(2^m) & = & T(n) & = & T(n/2) + 1 \\ T(2^{m-1}) & = & T(n/2) & = & T(n/4) + 1 \\ \vdots & & \vdots & & \vdots \\ T(2^1) & = & T(2) & = & T(1) + 1 \\ T(2^0) & = & T(1) & = & 1 \end{array}$$

Însumând cele $m + 1$ relaţii se obţine $T(n) = m + 1 = \lg\ n + 1$, pentru $n = 2^m$. Pentru $n$ arbitrar relaţia se demonstrează prin inducţie matematică. Presupunem că $T(k) = \lfloor \lg\ k \rfloor + 1$ pentru orice $k < n$ şi arătăm că $T(n) = \lfloor \lg\ n \rfloor + 1$. Tratăm separat cazurile: (i) $n = 2k$; (ii) $n = 2k + 1$. În primul caz se obţine:

$$T(n) = T(k) + 1 = \lfloor \lg\ k \rfloor + 2 = \lfloor \lg k + 1 \rfloor + 1 = \lfloor lg(2k) \rfloor + 1 = \lfloor \lg\ n \rfloor + 1.$$

În al doilea caz obţinem:

$$T(n) = T(k) + 1 = \lfloor \lg\ k \rfloor + 2 = \lfloor \lg\ k + 1 \rfloor + 1 = \lfloor \lg\ (2k) \rfloor + 1 = \lceil \lg(2k + 1) \rceil =$$

$$= \lfloor \lg\ n \rfloor + 1.$$

folosind relaţiile $\lfloor \lg\ n \rfloor + 1 = \lceil lg(n + 1) \rceil$ pentru $n \in \mathbb{N}$ şi $\lfloor x \rfloor + 1 = \lceil x \rceil$ pentru $x \notin \mathbb{N}$. Prin urmare căutarea binară este de complexitate $\mathcal{O}(\lg\ n)$. Sigur că rezultatul pentru $n$ arbitrar poate fi obţinut aplicând direct Propoziţia 5.1.

## 5.5 Probleme

**Problema 5.1** (*Permutări în ordine lexicografică*) Să se genereze toate permutările de ordin $n$ în ordine lexicografică (în ordinea crescătoare a valorii asociate permutării). Pentru $n = 3$ aceasta înseamnă generarea valorilor în ordinea: $(1, 2, 3)$, $(1, 3, 2)$, $(2, 1, 3)$, $(2, 3, 1)$, $(3, 1, 2)$, $(3, 2, 1)$.

*Rezolvare.* Se porneşte de la permutarea identică $p[1..n]$, $p[i] = i, i = \overline{1, n}$ (care are asociată valoarea $123 \ldots n$) şi se generează succesiv valoarea imediat următoare constituită din aceleaşi cifre. Pentru fiecare nouă permutare generată se parcurg etapele:

- identifică cel mai mare indice $i \in \{2, 3, \ldots, n\}$ cu proprietatea $p[i] > p[i - 1]$;

- determină poziţia, $k$, a celei mai mici valori din $p[i..n]$ cu proprietatea că $p[k] > p[i-1]$;
- interschimbă $p[k]$ cu $p[i-1]$;
- inversează ordinea elementelor din subtabloul $p[i..n]$.

Prelucrarea este descrisă în Algoritmul 5.10 unde `identific`($p[1..n]$) determină cel mai mare indice $i$ cu proprietatea că $x[i] > x[i-1]$, `minim`($p[i..n]$,$i-1$) determină indicele celui mai mic element din $p[i..n]$ care este mai mare decât $p[i-1]$, iar `inversare`($p[i..n]$) returnează elementele $p[i..n]$ în ordine inversă.

**Algoritmul 5.10** Generarea permutărilor în ordine lexicografică

```
permLex(integer n)
integer i,kmin
for i ← 1,n do
  p[i] ← i
end for
write p[1..n]
i ← identific(p[1..n])
while i > 1 do
  kmin ←minim(p[i..n],i − 1)
  p[i − 1] ↔ p[kmin]
  p[i..n] ← inversare(p[i..n])
  write p[1..n]
  i ← identific(p[1..n])
end while
```

Pentru fiecare dintre cele $n! - 1$ permutări diferite de permutarea identică se efectuează un număr de operaţii de ordinul $\mathcal{O}(n)$. Ordinul de complexitate al algoritmului este $\mathcal{O}(n \cdot n!)$.

**Problema 5.2** (*Algoritmul Johnson-Trotter.*) Să se genereze permutările de ordin $n$ astfel încât fiecare permutare să fie obţinută din cea imediat anterioară prin interschimbarea a exact două elemente.

*Rezolvare.* Acest lucru se poate realiza prin algoritmul Johnson-Trotter ale cărui etape principale sunt:

- Se porneşte de la permutarea identică; asociază fiecărui element un sens de parcurgere (iniţial toate elementele au asociat un sens de parcurgere înspre stânga).
- Determină elementul *mobil* de valoare maximă (un element este considerat mobil dacă sensul ataşat lui indică către o valoare adiacentă mai mică).
- Interschimbă elementul mobil cu elementul mai mic către care indică.
- Modifică direcţia tuturor elementelor mai mari decât elementul mobil.

Ultimele trei etape de mai sus se repetă până nu mai poate fi găsit un element mobil. În cazul permutărilor de ordin 3 algoritmul conduce la următoarea secvenţă de rezultate:

$\overleftarrow{1}\ \overleftarrow{2}\ \overleftarrow{3}$
$\overleftarrow{1}\ \overleftarrow{3}\ \overleftarrow{2}$
$\overleftarrow{3}\ \overleftarrow{1}\ \overleftarrow{2}$
$\overrightarrow{3}\ \overleftarrow{2}\ \overleftarrow{1}$
$\overleftarrow{2}\ \overrightarrow{3}\ \overleftarrow{1}$
$\overleftarrow{2}\ \overleftarrow{1}\ \overrightarrow{3}$

Sensurile de parcurgere pot fi uşor modelate printr-un tablou $d[1..n]$ în care valoarea $-1$ corespunde orientării înspre stânga iar valoarea $+1$ corespunde orientării înspre dreapta. In aceste ipoteze metoda poate fi descrisă prin Algoritmul 5.11.

**Algoritmul 5.11** Algoritmul Johnson-Trotter pentru generarea permutărilor

```
JohnsonTrotter(integer n)
integer p[1..n], d[1..n], k, delta
for i ← 1, n do
   p[i] = i; d[i] = −1;
end for
k ← mobil(p[1..n], d[1..n])
while k > 0 do
   delta ← d[k]
   p[k] ↔ p[k + delta]
   d[k] ↔ d[k + delta]
   d[1..n] ←modific(p[1..n],d[1..n],k + delta)
   write p[1..n]
   k ← mobil(p[1..n], d[1..n])
end while
```

Subalgoritmul de determinare a elementului mobil este:

```
mobil(integer p[1..n], d[1..n])
integer i, k, j
k ← 0; i ← 1;
while k = 0 and i ≤ n do
   if (i + d[i] ≥ 1) and (i + d[i] ≤ n) and (p[i] > p[i + d[i]]) then
      k ← i
   else
      i ← i + 1
   end if
end while
```

```
for j ← i + 1, n do
  if (j + d[j] ≤ n) and (p[j] > p[j + d[j]]) and (p[j] > p[k]) then
    k ← j
  end if
end for
return k
```

Subalgoritmul de modificare a sensurilor asociate elementelor mai mari decât $p[k]$:

```
modific(integerp[1..n], d[1..n],k)
integer i
for i ← 1, n do
  if p[i] > p[k] then
    d[i] ← −d[i]
  end if
end for
return d[1..n]
```

**Problema 5.3** Să se genereze toate şirurile binare cu $n$ elemente (problema este echivalentă cu generarea tuturor submulţimilor unei mulţimi cu $n$ elemente).

*Rezolvare.* O primă variantă de rezolvare constă în numărarea în baza doi pornind de la şirul valorilor egale cu 0 până la şirul binar corespunzător valorii $m = 2^n - 1$. Algoritmul descris în secţiunea corespunzătoare analizei amortizate realizează această prelucrare iar ordinul lui de complexitate este $\mathcal{O}(m) = \mathcal{O}(2^n)$. O variantă bazată pe tehnica reducerii este prezentată în Algoritmul 5.12.

**Algoritmul 5.12** Generarea tuturor şirurilor binare

```
generare(integer k)
if k = 1 then
  p[1] ← 0
  write (a[1..n])
  p[1] ← 1
  write (a[1..n])
else
  p[k] ← 0
  generare(k − 1)
  p[k] ← 1
  generare(k − 1)
end if
```

Pentru a genera toate submulţimile mulţimii reprezentate prin tabloul $a[1..n]$

algoritmul se apelează prin `generare`$(n)$ iar $a[1..n]$ este considerată o variabilă globală ce poate fi accesată la fiecare apel al algoritmului. Pentru a determina ordinul de complexitate al algoritmului notăm cu $T(n)$ numărul de atribuiri efectuate. Acesta va satisface relaţia de recurenţă:

$$T(n) = \begin{cases} 2 & n = 1 \\ 2T(n-1) + 2 & n > 1 \end{cases}$$

Pentru rezolvarea relaţiei de recurenţă se aplică metoda substituţiei inverse:

$$\begin{array}{l|l} T(n) = 2T(n-1) + 2 & \cdot 1 \\ T(n-1) = 2T(n-2) + 2 & \cdot 2 \\ \vdots & \\ T(2) = 2T(1) + 2 & \cdot 2^{n-1} \\ T(1) = 2 & \cdot 2^n \end{array}$$

Prin însumarea relaţiilor şi reducerea termenilor asemenea se obţine $T(n) = 2(1 + 2 + \ldots + 2^{n-1} + 2^n) = 2(2^{n+1} - 1) \in \Theta(2^n)$.

**Problema 5.4** (*Codul Gray*) Să se genereze secvenţa tuturor şirurile binare pe $k$ poziţii pornind de la şirul constituit din cifre nule şi până la cel în care toate cifrele sunt egale cu 1 astfel încât trecerea de la un şir la altul să se realizeaze prin modificarea unei singure poziţii binare. Codul obţinut asociind aceste şiruri valorilor cuprinse între 0 şi $2^k - 1$ se numeşte codul Gray.

*Rezolvare* Pentru cazul în care $n = 3$ succesiunea şirurilor binare ar trebui să fie: $(0,0,0)$, $(0,0,1)$, $(0,1,1)$, $(0,1,0)$, $(1,1,0)$,$(1,1,1)$,$(1,0,1)$, $(1,0,0)$. Algoritmul este oarecum similar cu cel pentru generarea şirurilor binare (Algoritmul 5.12), însă la completarea unei anumite poziţii cu 0 sau 1 trebuie să se ţină cont de ordinea în care se plasează valorile, urmându-se o strategie de "oglindire": dacă pe poziţia $k$ prima dată s-a completat 0 iar apoi 1, pentru poziţia $k+1$ se procedează invers: se completează prima dată 1 şi apoi 0. Arborii de generare pentru cele două coduri (binar şi Gray) sunt ilustraţi pentru $n = 3$ în figurile 5.4 şi respectiv 5.5.

Aceasta idee este descrisă în Algoritmul 5.13 care se apelează cu `gray`$(0)$ şi în care se presupune că tabloul $g[1..n]$ este global şi este iniţializat cu 0. Tot variabilă globală se consideră a fi şi $n$.

**Problema 5.5** Să se calculeze $A^p$ unde $A$ este o matrice $n \times n$ şi $p$ este o valoare naturală mai mare decât 1. Să se analizeze eficienţa algoritmului propus.

*Rezolvare.* Presupunem că `produs`$(A[1..n, 1..n], B[1..n, 1..n])$ este un algoritm care returnează produsul matricilor $A$ şi $B$ specificate ca parametri de intrare.

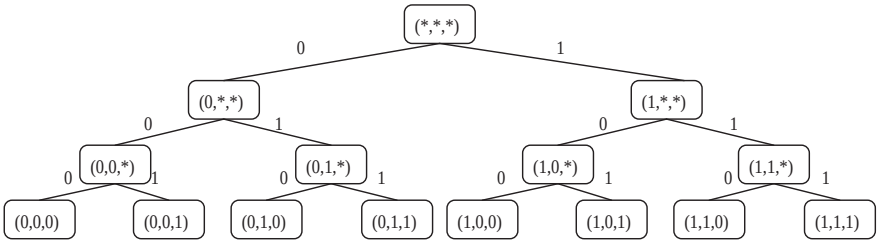


Figura 5.4: Arborele corespunzător construirii codului binar ($n = 3$)

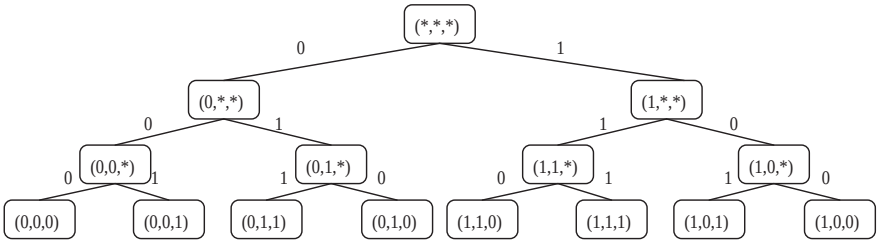


Figura 5.5: Arborele corespunzător construirii codului Gray ($n = 3$)

**Algoritmul 5.13** Generarea codului Gray pe $n$ poziţii

```
gray(integer k)
integer v
if k = n then
   v ← g[k]; write g[1..n]
   v ← 1 − v; g[k] ← v; write g[1..n]
else
   v ← g[k]; gray(k + 1)
   v ← 1 − v; g[k] ← v; gray(k + 1)
end if
```

O primă variantă de calcul a lui $A^p$ se bazează pe metoda forţei brute şi conduce la un algoritm de forma:

```
putere1(real A[1..n, 1..n], integer p)
real P[1..n.1..n]
integer i
P[1..n, 1..n] ← A[1..n, 1..n]
for i ← 2, p do
   P ←produs(P, A)
end for
return  P[1..n, 1..n]
```

Pentru a analiza complexitatea considerăm că dimensiunea problemei este determinată de perechea $(n, p)$ şi că operaţia dominantă este cea de înmulţire efectuată în cadrul algoritmului `produs`. Este uşor de stabilit că la fiecare apel se efectuează $n^3$ operaţii de înmulţire astfel, că numărul total de înmulţiri efectuate de către algoritmul `putere1` este $T(n, p) = (p-1)n^3 \in \Theta(pn^3)$.

Varianta bazată pe tehnica reducerii (Algoritmul 5.14) porneşte de la relaţia:

$$A^p = \begin{cases} A & p = 1 \\ A^{p/2} \cdot A^{p/2} & p > 1, p \text{ par} \\ A^{(p-1)/2} \cdot A^{(p-1)/2} \cdot A & p > 1, p \text{ impar} \end{cases} \tag{5.1}$$

**Algoritmul 5.14** Calculul puterii $p$ a unei matrici pătratice folosind metoda reducerii

```
putere2(real A[1..n, 1..n], integer p)
if p = 1 then
   return  A[1..n, 1..n]
else if p MOD 2 = 0 then
   B ←putere2(A, p/2)
   return  produs(B, B)
else
   B ←putere2(A, (p − 1)/2)
   B ← produs(B, B)
   B ← produs(B, A)
   return  B[1..n, 1..n]
end if
```

.

Numărul de înmulţiri efectuate în cadrul algoritmului satisface relaţia de recurenţă:

$$T(n,p) = \begin{cases} 0 & p = 1 \\ T(n,p/2) + n^3 & p > 1, p \text{ par} \\ T(n,(p-1)/2) + 2n^3 & p > 1, p \text{ impar} \end{cases}$$

Considerăm cazul particular $p = 2^k$ şi aplicăm metoda substituţiei inverse:

$$\begin{aligned} &T(n,p) = T(n,p/2) + n^3 \\ &T(n,p/2) = T(n,p/4) + n^3 \\ &\vdots \\ &T(n,2) = T(1) + n^3 \\ &T(n,1) = 0 \end{aligned}$$

Prin însumarea relaţiilor de mai sus se obţine $T(n,p) = n^3 \lg p$ pentru $p = 2^k$. Întrucât $T(n,p)$ este crescătoare şi $n^3 \lg p$ este o funcţie netedă în raport cu $p$ rezultatul poate fi extins (cf. Propoziţiei 5.1) şi pentru $p$ arbitrar. Prin urmare, în varianta bazată pe metoda reducerii, algoritmul de ridicare la putere a unei matrice este din $\Theta(n^3 \lg p)$.

**Problema 5.6** Stabiliţi ce afişează algoritmul de mai jos atunci când este apelat pentru $k = n$ (în ipoteza ca lucrează asupra unui tablou global $a[1..n]$ iniţializat astfel încât $a[i] = i$ pentru $i = \overline{1,n}$) şi stabiliţi ordinul de complexitate.

```
alg(integer k)
if k = 1 then
  write a[1..n]
else
  for i ← 1, k do
    alg(k − 1)
    if kMOD2 = 1 then
      a[1] ↔ a[k]
    else
      a[i] ↔ a[k]
    end if
  end for
end if
```

*Indicaţie.* Se determină numărul, $T(n)$, de apeluri ale funcţiei de afişare (**write**) corespunzătoare apelului `alg`$(n)$. Acesta satisface relaţia de recurenţă:

$$T(n) = \begin{cases} n \cdot T(n-1) & \text{dacă } n > 1 \\ 1 & \text{dacă } n = 1 \end{cases}$$

Aplicând metoda substituţiei inverse pentru rezolvarea relaţiei de recurenţă se obţine că $T(n) = n!$. Întrucât se porneşte de la tabloul conţinând primele $n$ numere naturale şi cum pe parcurs se efectuează doar interschimbări între elemente rezultă că elementele tabloului sunt tot timpul distincte. Rămâne de analizat dacă la fiecare afişare tabloul $a$ conţine elementele mulţimii $\{1, 2, \ldots, n\}$ într-o altă ordine. In caz afirmativ, rezultă că sunt generate permutările de ordin $n$. Pentru determinarea ordinului de complexitate se estimează numărul de interschimbări, care satisface relaţia de recurenţă:

$$T(n) = \begin{cases} n \cdot (T(n-1) + n) & \text{dacă } n > 1 \\ 1 & \text{dacă } n = 1 \end{cases}$$

**Problema 5.7** (*Inserţie binară*) Se consideră un tablou $a[1..n]$ ordonat crescător şi $v$ o valoare. Să se determine, folosind un algoritm din $\mathcal{O}(\lg n)$, poziţia unde poate fi inserată valoarea $v$ în tabloul $a$ astfel încât acesta să rămână ordonat crescător.

*Rezolvare.* Se foloseşte ideea de la căutarea binară obţinându-se variantele descrise în Algoritmul 5.15. În cazul primei variante corectitudinea poate fi demonstrată folosind ca proprietate invariantă faptul că $a[li-1] \leq v \leq a[ls+1]$ (în ipoteza că se presupune formal că $a[0] = -\infty$ şi $a[k+1] = \infty$). Intrucât structura algoritmului este similară algoritmului de căutare binară ordinul de complexitate este $\mathcal{O}(\lg n)$.

Şi în cazul celei de a doua variante $a[li-1] \leq v \leq a[ls+1]$ este proprietate invariantă astfel că la ieşirea din ciclu (când $li = ls+1$) are loc $a[li-1] \leq v \leq a[li]$. Prin urmare la ieşirea din ciclu trebuie returnată valoarea lui $li$.

Folosind acest algoritm de căutare binară a poziţiei de inserţie algoritmul de sortare prin inserţie poate fi transformat în Algoritmul 5.16.

Din punctul de vedere al numărului de comparaţii efectuate algoritmul de sortare prin inserţie binară are complexitatea $\mathcal{O}(n \lg n)$. Din punctul de vedere al numărului de deplasări de elemente efectuate, algoritmul de sortare prin inserţie binară aparţine însă tot lui $\mathcal{O}(n^2)$, la fel ca şi algoritmul bazat pe inserţie secvenţială.

**Problema 5.8** (*Căutare ternară.*) Să se extindă ideea de la căutarea binară în cazul în care tabloul este divizat în trei părţi.

*Rezolvare.* Tehnica căutării binare poate fi extinsă prin divizarea unui subşir $a[li..ls]$ în trei subşiruri $a[li..m1]$, $a[m1+1..m2]$, $a[m2+1..ls]$, unde $m1 = li + \lfloor (ls-li)/3 \rfloor$ iar $m1 = li + 2\lfloor (ls-li)/3 \rfloor$. Căutarea poate fi astfel realizată prin Algoritmul 5.17, care returnează fie indicele unui element egal cu $v$ fie $-1$, dacă $v$ nu este în tablou.

Notând cu $T(n)$ numărul maxim de comparaţii efectuate are loc relaţia:

**Algoritmul 5.15** Variante de căutare a unui poziţii de inserţie

```
cautare_pozitie1(a[1..k], v)
li ← 1; ls ← k
if (v ≤ a[li]) then
  return li
end if
if (v ≥ a[ls]) then
  return ls + 1
end if
while ls > li do
  m ← ⌊(li + ls)/2⌋
  if a[m] = v then
    return (m + 1)
  end if
  if v < a[m] then
    ls ← m − 1
  else
    li ← m + 1
  end if
end while
if v ≤ a[li] then
  return li
end if
if v > a[ls] then
  return ls + 1
end if
```

```
cautare_pozitie2(a[1..k], v)
li ← 1; ls ← k
while li ≤ ls do
  if v ≤ a[li] then
    return li
  end if
  if v ≥ a[ls] then
    return ls + 1
  end if
  m ← ⌊(li + ls)/2⌋
  if a[m] = v then
    return (m + 1)
  end if
  if v < a[m] then
    ls ← m − 1
  else
    li ← m + 1
  end if
end while
return li
```

**Algoritmul 5.16** Sortare prin inserţie binară

```
insertie_binara(a[1..n])
for i ← 2, n do
  aux ← a[i]
  poz ←cautare_pozitie1(a[1..i − 1],v)
  for j ← i − 1, poz, −1 do
    a[j + 1] ← a[j]
  end for
  a[poz] ← aux
end for
return a[1..n]
```

**Algoritmul 5.17** Căutare ternară

```
cautare_ternara (a[1..n], v)
li ← 1; ls ← n;
while li <= ls do
  m1 ← li + (ls − li) DIV 3; m2 ← li + 2 ∗ ((ls − li) DIV 3)
  if x[m1] = v then
    return m1
  end if
  if x[m2] = v then
    return m2
  end if
  if v < x[m1] then
    ls ← m1 − 1
  else if v < x[m2] then
    li ← m1 + 1;ls ← m2 − 1;
  else
    li ← m2 + 1
  end if
end while
return -1
```

$$T(n) = \begin{cases} 0 & n = 0 \\ T(n/3) + c & n > 0 \end{cases}$$

unde $c \in \{1, 2, 3, 4\}$ este numărul de comparaţii efectuate în cadrul unei iteraţii. Aplicând metoda substituţiei inverse pentru cazul particular $n = 3^k$ se obţine că $T(n) = c \log_3 n$ deci $T(n) \in \mathcal{O}(\lg n)$, adică acelaşi ordin de complexitate ca în cazul căutării binare (diferă doar constantele multiplicative).

**Problema 5.9** (*Metoda bisecţiei.*) Fie $f : [a, b] \to R$ o funcţie continuă având proprietăţile: (i) $f(a)f(b) < 0$; (ii) există un unic $x^*$ cu proprietatea că $f(x^*) = 0$. Să se aproximeze $x^*$ cu precizia $\epsilon > 0$.

*Rezolvare.* A determina pe $x^*$ cu precizia $\epsilon$ înseamnă a identifica un interval de lungime $\epsilon$ care conţine pe $x^*$ sau chiar un interval de lungime $2\epsilon$ dacă se consideră ca aproximare a lui $x^*$ mijlocul intervalului. Se poate aplica exact aceeaşi strategie ca la căutarea binară ţinându-se cont că $x^*$ se află în intervalul pentru care funcţia $f$ are valori de semne opuse în extremităţi. Astfel se obţine Algoritmul 5.18.

Complexitatea algoritmului `bisectie` este determinată de dimensiunea intervalului $[a, b]$ şi de precizia dorită a aproximării, $\epsilon$. Notând $n = (b - a)/\epsilon$ se obţine că algoritmul este de ordinul $\mathcal{O}(\lg n)$.

**Algoritmul 5.18** Metoda bisecţiei pentru aproximarea soluţiei unei ecuaţii

```
bisectie(real a,b,ε)
li ← a; ls ← b
repeat
  m ← (li + ls)/2
  if f(m) = 0 then
    return m
  end if
  if f(m) * f(li) < 0 then
    ls ← m
  else
    li ← m
  end if
until |ls − li| < 2ε
return (li + ls)/2
```

**Problema 5.10** Fie $x[1..n]$ un tablou ordonat strict crescător. Să se propună un algoritm de complexitate $\mathbf{O}(\lg\ n)$ care determină (dacă există) indicele $i$ pentru care $x[i] = i$.

*Indicaţie.* Se aplică ideea de la căutarea binară comparând elementul din mijlocul tabloului cu indicele corespunzător mijlocului.

# Capitolul 6

# Tehnica divizării

Este o tehnică similară tehnicii reducerii însă rezolvarea unei probleme de dimensiune $n$ nu se reduce la rezolvarea unei singure subprobleme de dimensiune mai mică ci la rezolvarea câtorva subprobleme de dimensiuni cât mai apropiate.

## 6.1 Principiul tehnicii divizării

Tehnica divizării (denumită şi "divide et impera" sau "divide and conquer") se bazează pe descompunerea problemei de rezolvat în două sau mai multe subprobleme *independente*, rezolvarea acestora şi combinarea rezultatelor obţinute. Etapele care se parcurg în aplicarea metodei sunt:

- *Descompunerea în subprobleme.* O problemă de dimensiune $n$ este descompusă în două sau mai multe subprobleme de dimensiune mai mică. Cazul clasic este acela când subproblemele au aceeaşi natură ca şi problema iniţială. Ideal este ca dimensiunile subproblemelor să fie cât mai apropiate (dacă problema de dimensiune $n$ se descompune în $k$ subprobleme e de preferat ca acestea să aibă dimensiuni apropiate de $n/k$). Pentru ca această tehnică să fie efectivă (să conducă de exemplu la reducerea costului calculului) trebuie ca subproblemele care se rezolvă să fie independente (o aceeaşi subproblemă să nu fie rezolvată de mai multe ori).
- *Rezolvarea subproblemelor.* Fiecare dintre subproblemele independente obţinute prin divizare se rezolvă. Dacă ele sunt similare problemei iniţiale atunci se aplică din nou tehnica divizării. Procesul de divizare continuă până când se ajunge la subprobleme de dimensiune suficient de mică (*dimensiunea critică*, $n_c$) pentru a fi rezolvate direct.

- *Combinarea rezultatelor.* Pentru a obţine răspunsul la problema iniţială uneori trebuie combinate rezultatele obţinute prin rezolvarea subproblemelor.

Structura generală a unui algoritm elaborat folosind tehnica divizării este descrisă în Algoritmul 6.1. În practică cel mai adesea se utilizează $k = 2$ şi $n_1 = \lfloor n/2 \rfloor$, $n_2 = n - \lfloor n/2 \rfloor$.

**Algoritmul 6.1** Structura generală a unui algoritm bazat pe tehnica divizării

```
divizare(P(n))
if n ≤ n_c then
   ⟨ rezolvare directă ⟩
else
   ⟨ descompune P(n) în k subprobleme P(n_1), ... P(n_k) ⟩
   for i ← 1, k do
      divizare(P(n_i))
   end for
   ⟨ compunerea rezultatelor ⟩
end if
```

Pentru a ilustra tehnica considerăm problema determinării maximului dintr-o secvenţă finită de valori reale stocate în tabloul $x[1..n]$. Aplicând ideea divizării rezultă că este suficient să determinăm maximul din subtabloul $x[1..\lfloor n/2 \rfloor]$ şi maximul din subtabloul $x[\lfloor n/2 \rfloor + 1..n]$. Rezultatul va fi cea mai mare dintre valorile obţinute. Varianta recursivă a algoritmului este descrisă în Algoritmul 6.2.

**Algoritmul 6.2** Determinarea maximului unui tablou folosind tehnica divizării

```
1:  maxim (real x[s..d])
2:  if s = d then
3:     max ← x[s]
4:  else
5:     m ← ⌊(s + d)/2⌋
6:     max1 ← maxim(x[s..m])
7:     max2 ← maxim(x[m + 1..d])
8:     if max1 < max2 then
9:        max ← max2
10:    else
11:       max ← max1
12:    end if
13: end if
14: return max
```

In acest caz se aplică tehnica divizării pentru $k = 2$ şi dimensiunea critică $n_c = 1$. Subproblemele sunt independente, iar compunerea rezultatelor constă în prelucrarea "**if** $max1 < max2$ **then** $max \leftarrow max2$ **else** $max \leftarrow max1$".

Notând cu $T(n)$ numărul de comparaţii efectuate se obţine relaţia de recurenţă:

$$T(n) = \begin{cases} 0 & \text{dacă } n = 1 \\ T(\lfloor n/2 \rfloor) + T(n - \lfloor n/2 \rfloor) + 1 & \text{dacă } n \geq 2 \end{cases}$$

În cazul în care $n$ nu este o putere a lui 2 metoda iteraţiei este mai dificil de aplicat. Pornim de la cazul particular $n = 2^m$ pentru care relaţia de recurenţă conduce la $T(n) = 2T(n/2) + 1$ pentru $n \geq 2$ obţinându-se succesiunea:

$$\begin{array}{lcl|l} T(n) & = & 2T(n/2) + 1 & \\ T(n/2) & = & 2T(n/4) + 1 & \cdot 2^1 \\ T(n/4) & = & 2T(n/8) + 1 & \cdot 2^2 \\ \vdots & & \vdots & \\ T(2) & = & 2T(1) + 1 & \cdot 2^{m-1} \\ T(1) & = & 0 & \cdot 2^m \end{array}$$

Înmulţind relaţiile cu factorii din ultima coloană, însumând relaţiile şi efectuând reducerile se obţine: $T(n) = 1 + 2 + 2^2 + \ldots 2^{m-1} = 2^m - 1 = n - 1$. Acest rezultat este valabil doar pentru $n = 2^m$. Pentru a arăta că este adevărat pentru orice $n$ aplicăm inducţia matematică. Evident $T(1) = 1 - 1 = 0$. Presupunem că $T(k) = k - 1$ pentru orice $k < n$. Rezultă că $T(n) = \lfloor n/2 \rfloor - 1 + n - \lfloor n/2 \rfloor - 1 + 1 = n - 1$, adică algoritmul are complexitate liniară, la fel ca cel obţinut aplicând metoda clasică. Ultimul rezultat poate fi obţinut direct folosind Propoziţia 5.1.

## 6.2 Analiza complexităţii algoritmilor bazaţi pe tehnica divizării

Analiza complexităţii algoritmilor elaboraţi prin tehnica divizării sau a reducerii se bazează pe un rezultat teoretic important cunoscut sub numele de *Teorema master*.

Presupunem că o problemă de dimensiune $n$ este descompusă în $m$ subprobleme de dimensiuni $n/m$ dintre care este necesar să fie rezolvate $k \leq m$. Considerăm că divizarea şi compunerea rezultatelor au împreună costul $T_{DC}(n)$ iar costul rezolvării în cazul particular este $T_0$. Cu aceste ipoteze costul corespunzător algoritmului verifică relaţia de recurenţă:

$$T(n) = \begin{cases} T_0 & \text{dacă } n \leq n_c \\ kT(n/m) + T_{DC}(n) & \text{dacă } n > n_c \end{cases}$$

Determinarea lui $T(n)$ pornind de la această recurenţă este uşurată de teorema următoare.

**Teorema 6.1** *(Teorema master) Dacă* $T_{DC}(n) \in \Theta(n^d)$, $d \geq 0$ *atunci:*

$$T(n) \in \begin{cases} \Theta(n^d) & dacă\ k < m^d \\ \Theta(n^d lg\ n) & dacă\ k = m^d \\ \Theta(n^{\log_m k}) & dacă\ k > m^d \end{cases}$$

Teorema rămâne valabilă şi în cazul notaţiei $\mathcal{O}$. Pentru demonstraţie pot fi consultate [3], [11], [12].

Pentru algoritmul `maxim` avem $d = 0$, $m = 2$, $k = 2$, deci se aplică cazul al treilea al teoremei obţinându-se $T(n) = n$.

Teorema poate fi aplicată şi în cazul algoritmilor bazaţi pe tehnica reducerii. Astfel pentru algoritmul căutării binare avem: $T_{DC}(n) = \Theta(1)$ deci $d = 0$, $m = 2$ şi $k = 1$. Se aplică cazul al doilea al teoremei master ($1 = 2^0$) şi se obţine $T(n) = \Theta(\lg\ n)$.

## 6.3 Algoritmi de sortare bazaţi pe tehnica divizării

Considerăm din nou problema ordonării crescătoare a unui şir de valori; ($x_1$,$x_2$, ..., $x_n$). Algoritmii bazaţi pe compararea elementelor prezentaţi în Cap. 4 au ordinul de complexitate $\mathcal{O}(n^2)$. Ideea de start o reprezintă încercarea de a reduce această complexitate folosind principiul divizării. Aceasta presupune: (i) descompunerea şirului iniţial în două subsecvenţe; (ii) ordonarea fiecăreia dintre acestea folosind aceeaşi tehnică; (iii) combinarea subsecvenţelor ordonate pentru a obţine varianta ordonată a şirului total.

Dimensiunea critică, sub care problema poate fi rezolvată direct este $n_c = 1$. În acest caz subsecvenţa se reduce la un singur element, implicit ordonat. Este posibil să se folosească şi $1 < n_c \leq 10$ aplicând pentru sortarea acestor subsecvenţe una dintre metodele elementare de sortare (de exemplu metoda inserţiei).

În continuare sunt prezentate două metode de sortare bazate pe principiul divizării: sortarea prin *interclasare* ("mergesort") şi sortarea *rapidă* ("quicksort"). Acestea diferă atât în etapa descompunerii şirului în subşiruri cât şi în recombinarea rezultatelor.

### 6.3.1 Sortare prin interclasare

**Idee.**

Şirul $(x_1, x_2, \ldots, x_n)$ se descompune în două subsecvenţe de lungimi cât mai apropiate: $(x_1, \ldots, x_{\lfloor n/2 \rfloor})$ şi $(x_{\lfloor n/2 \rfloor + 1}, \ldots, x_n)$; se ordonează fiecare dintre subsecvenţe aplicând aceeaşi tehnică; se construieşte şirul final ordonat parcurgând cele două subsecvenţe şi preluând elemente din ele astfel încât să fie

respectată relaţia de ordine (această prelucrare se numeşte *interclasare*). Modul de lucru al sortării prin interclasare este ilustrat în figura 6.1.

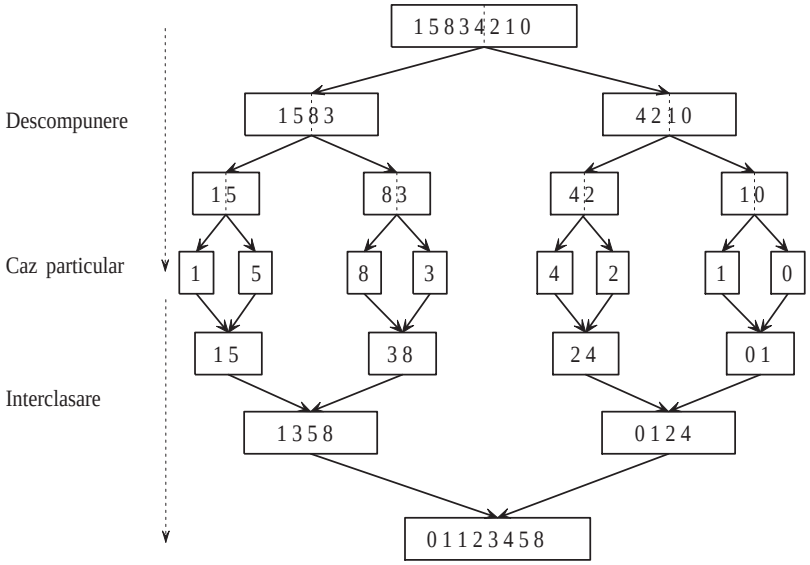


Figura 6.1: Sortare prin interclasare

**Structura generală a algoritmului.**

Etapele principale ale algoritmului de sortare prin interclasare sunt descrise în Algoritmul 6.3 iar pentru sortarea unui tablou $x[1..n]$ algoritmul trebuie apelat pentru $li = 1$ şi $ls = n$ (`mergesort`$(x[1..n])$).

**Algoritmul 6.3** Structura generală a algoritmului de sortare prin interclasare

`mergesort`$(x[li..ls])$
**if** $li < ls$ **then**
  $m = \lfloor (li + ls)/2 \rfloor$
  $x[li..m] \leftarrow$ `mergesort`$(x[li..m])$
  $x[m+1..ls] \leftarrow$ `mergesort`$(x[m+1..ls])$
  $x[li..ls] \leftarrow$ `merge`$(x[li..m], x[m+1..ls])$
**end if**
**return** $x[li..ls]$

**Interclasare**

Etapa cea mai importantă a prelucrării este reprezentată de interclasare. Scopul acestei prelucrări este construirea unui şir ordonat pornind de la două şiruri ordonate. Ideea prelucrării constă în a parcurge în paralel, folosind două contoare, cele două şiruri şi a compara elementele curente. In şirul final se transferă elementul mai mic dintre cele două iar contorul utilizat pentru parcurgerea şirului din care s-a transferat un element este incrementat. Procesul continuă până când unul dintre şiruri a fost transferat în întregime. Elementele celuilalt şir sunt transferate direct în şirul final.

Există diverse modalităţi de a descrie algoritmic această prelucrare. Una dintre acestea este prezentată în Algoritmul 6.4 în care $c$ reprezintă o zonă suplimentară de memorie.

**Algoritmul 6.4** Interclasarea a două şiruri ordonate crescător

```
merge(x[li..m],x[m+1..ls])
i ← li; j ← m+1; k ← 0 // iniţializarea contoarelor
// parcurgerea până la sfârşitul unuia dintre tablouri
while i ≤ m and j ≤ ls do
  if x[i] ≤ x[j] then
    // transfer din tabloul a
    k ← k+1; c[k] ← x[i]; i ← i+1
  else
    // transfer din tabloul b
    k ← k+1; c[k] ← x[j]; j ← j+1
  end if
end while
while i ≤ m do
  // transferul eventualelor elemente rămase în a
  k ← k+1; c[k] ← x[i]; i ← i+1
end while
while j ≤ ls do
  // transferul eventualelor elemente rămase în b
  k ← k+1; c[k] ← x[j]; j ← j+1;
end while
return c[1..k]
```

Precondiţiile Algoritmului 6.4: $\{x[li..m]$ crescător şi $x[m+1..ls]$ crescător$\}$, iar postcondiţia este: $\{c[1..p+q]$ este crescător$\}$ (notăm cu $p$ numărul de elemente din primul tablou şi cu $q$ numărul de elemente din al doilea). Pentru a demonstra că algoritmul asigură satisfacerea postcondiţiei este suficient să se arate că $\{c[1..k]$ este crescător, $k = i+j-2\}$ este proprietate invariantă pentru fiecare dintre cele trei prelucrări repetitive.

Contorizând numărul de comparaţii ($T_C(p,q)$) şi cel de transferuri ale ele-

mentelor ($T_M(p,q)$) se obţine: $T_C(p,q) \in \Omega(\min(p,q))$ (în cazul cel mai favorabil), $T_C(p,q) \in O(p+q-1)$ (în cazul cel mai defavorabil) iar $T_M(p,q) \in \Theta(p+q)$.

Să analizăm cazul interclasării a două tablouri $a[1..p]$ şi $b[1..q]$ (independent de algoritmul de sortare prin interclasare). O variantă de algoritm este cea care foloseşte câte o valoare "santinelă" pentru fiecare dintre tablourile $a[1..p]$ şi $b[1..q]$. Santinelele constau în valori mai mari decât toate valorile prezente în $a$ şi $b$ şi sunt plasate pe poziţiile $p+1$ respectiv $q+1$. În acest caz algoritmul poate fi descris într-o formă mai condensată (Algoritm 6.5) şi atât numărul de comparaţii cât şi numărul de transferuri este $T_C(p,q) = T_M(p,q) = p+q$. Prin urmare în varianta cu valori santinelă se efectuează ceva mai multe comparaţii decât în cea fără astfel de valori însă ordinul de complexitate rămâne liniar în raport cu dimensiunile tablourilor.

**Algoritmul 6.5** Varianta cu valori santinelă a algoritmului de interclasare

```
merge2(a[1..p+1],b[1..q+1])
a[p+1] ← ∞; b[q+1] ← ∞; // fixarea santinelelor
i ← 1; j ← 1; //iniţializarea indicilor de parcurgere
for k ← 1, p+q do
  // parcurgerea poziţiilor în tabloul final
  if a[i] ≤ b[j] then
    // preluare din a
    c[k] ← a[i]; i ← i+1
  else
    // preluare din b
    c[k] ← b[j]; j ← j+1
  end if
end for
return c[1..p+q]
```

**Analiza complexităţii.**

Notând cu $T(n)$ numărul de prelucrări (comparaţii şi transferuri) efectuate de către sortarea prin interclasare şi cu $T_{inter}(n)$ numărul celor efectuate pe parcursul interclasării se obţine relaţia de recurenţă:

$$T(n) = \begin{cases} 0 & n = 1 \\ T(\lfloor n/2 \rfloor) + T(n - \lfloor n/2 \rfloor) + T_{inter}(n) & n > 1 \end{cases}$$

Cum numărul de subprobleme este 2 ($m = 2$) şi ambele trebuie rezolvate ($k = 2$) iar $T_{inter}(n) \in \Theta(n)$ rezultă că se poate aplica cazul doi al Teoremei master ($d = 1$ astfel că $k = m^d$) obţinându-se că sortarea prin interclasare are complexitate $\Theta(n \lg n)$.

**Observaţie.** Costul redus al prelucrărilor din sortarea prin interclasare este însă contrabalansat de faptul că se utilizează (în etapa de interclasare) o zonă

de manevră de dimensiune proporţională cu cea a tabloului iniţial. In [12] (pag. 256) este descrisă o variantă nerecursivă a sortării prin interclasare caracterizată printr-un număr redus de transferuri de elemente.

### 6.3.2 Sortare rapidă

#### Idee

La sortarea prin interclasare descompunerea şirului iniţial în două subsecvenţe se realizează pe baza poziţiei elementelor. Asta face ca elemente cu valori mici (sau mari) să se poată afla în fiecare dintre subşiruri. Din această cauză este necesară combinarea rezultatelor prin interclasare. O altă modalitate de descompunere ar fi aceea în care se ţine cont şi de valoarea elementelor. Scopul urmărit este de a simplifica combinarea rezultatelor. Ideal ar fi ca prin concatenarea şirurilor ordonate să se obţină şirul ordonat în întregime.

**Exemplul 6.1** Considerăm şirul $x = (3, 1, 2, 4, 7, 5, 8)$. Se observă că elementul 4 se află pe o poziţie privilegiată ($q = 4$) întrucât toate elementele care îl preced sunt mai mici decât el şi toate cele care îl succed sunt mai mari. Aceasta înseamnă că se află deja pe poziţia finală. Un element $x[q]$ având proprietăţile: $x[i] \leq x[q]$ pentru $i = \overline{1, q-1}$ şi $x[i] \geq x[q]$ pentru $i = \overline{q+1, n}$ se numeşte *pivot*. Existenţa unui pivot permite reducerea sortării şirului iniţial la sortarea subsecvenţelor $x[1..q-1]$ şi $x[q+1..n]$. Nu întotdeauna există un astfel de pivot după cum se observă din subşirul $(3, 1, 2)$. În aceste situaţii trebuie "creat" unul prin modificarea poziţiilor unor elemente. Alteori pivotul, dacă există, se află pe o poziţie care nu permite descompunerea problemei iniţiale în subprobleme de dimensiuni apropiate (de exemplu în subşirul $(7, 5, 8)$ valoarea de pe ultima poziţie poate fi considerată valoare pivot).

**Exemplul 6.2** Să considerăm acum şirul $(3, 1, 2, 7, 5, 4, 8)$. Se observă că poziţia $q = 3$ are următoarea proprietate: $x[i] \leq x[j]$ pentru orice $i \in \{1, \ldots, q\}$ şi orice $j \in \{q+1, \ldots, n\}$. În acest caz sortarea şirului iniţial se reduce la sortarea subşirurilor $x[1..q]$ şi $x[q+1..n]$. Poziţia $q$ este numită *poziţie de partiţionare*.

#### Structura generală

Pornind de la exemplele de mai sus se pot dezvolta două variante de sortare bazate pe descompunerea şirului iniţial în două subsecvenţe: una care foloseşte un element pivot iar alta care foloseşte o poziţie de partiţionare. Acest tip de sortare este cunoscută sub numele de sortare rapidă ("quicksort") şi a fost dezvoltată de Hoare. Structura generală a acestor două variante este descrisă în Algoritmii 6.6 şi 6.7.

Diferenţa dintre cele două variante este mică: în prima variantă în prelucrarea de partiţionare se asigură şi plasarea pe poziţia $q$ a valorii finale pe când în a doua doar se determină poziţia de partiţionare.

**Algoritmul 6.6** Sortare rapidă bazată pe element pivot

```
quicksort1 (x[li..ls])
if li < ls then
   q ← partitie1(x[li..ls]
   x[li..q − 1] ← quicksort1(x[li..q − 1])
   x[q + 1..ls] ← quicksort1(x[q + 1..ls])
end if
return x[li..ls]
```

**Algoritmul 6.7** Sortare rapidă bazată pe poziţie de partiţionare

```
quicksort2 (x[li..ls])
if li < ls then
   q ← partitie2(x[li..ls]
   x[li..q] ← quicksort1(x[li..q])
   x[q + 1..ls] ← quicksort1(x[q + 1..ls])
end if
return x[li..ls]
```

### Partiţionare

Este prelucrarea cea mai importantă a algoritmului. Discutăm separat cele două variante deşi după cum se va vedea diferenţele dintre ele sunt puţine.

Considerăm problema identificării în $x[1..n]$ a unui pivot, $x[q]$ cu proprietatea că $x[i] \leq x[q]$ pentru $i < q$ şi $x[i] \geq x[q]$ pentru $i > q$. După cum s-a văzut în exemplul anterior nu întotdeauna există un pivot. În aceste situaţii se creează unul. Ideea este următoarea: se alege o valoare arbitrară dintre cele prezente în şir şi se rearanjează elementele şirului (prin interschimbări) astfel încât elementele mai mici decât această valoare să fie în prima parte a şirului iar cele mai mari în a doua parte a şirului. În felul acesta se determină şi poziţia $q$ pe care trebuie plasată valoarea. Prelucrarea este descrisă în Algoritmul 6.8.

**Exemplul 6.4** Considerăm şirul $(1, 7, 5, 3, 8, 2, 4)$. Iniţial $i = 0$, $j = 7$ iar $v = 4$. Succesiunea prelucrărilor este:
*Etapa 1.* După execuţia primului ciclu **repeat** se obţine $i = 2$ (căci $7 > 4$) iar după execuţia celui de al doilea se obţine $j = 6$. Cum $i < j$ se interschimbă $x[2]$ cu $x[6]$ obţinându-se $(1, 2, 5, 3, 8, 7, 4)$.
*Etapa 2.* Parcurgerea în continuare de la stânga la dreapta conduce la $i = 3$ iar de la dreapta la stânga la $j = 4$. Cum $i < j$ se interschimbă $x[3]$ cu $x[4]$ şi se obţine $(1, 2, 3, 5, 8, 7, 4)$.
*Etapa 3.* Continuând parcurgerile în cele două direcţii se ajunge la $i = 4$ şi $j = 3$.
Cum $i > j$ se iese din prelucrarea repetitivă exterioară, iar elementele $x[4]$ şi $x[7]$ se interschimbă obţinându-se $(1, 2, 3, 4, 8, 7, 5)$, poziţia pivotului fiind 4.

**Algoritmul 6.8** Determinarea poziţiei pivotului

```
partitie1(x[li..ls])
v ← x[ls] // se alege valoarea pivotului
i ← li − 1 // contor pentru parcurgere de la stânga la dreapta
j ← ls // contor pentru parcurgere de la dreapta la stânga
while i < j do
  repeat
    i ← i + 1
  until x[i] ≥ v
  repeat
    j ← j − 1
  until x[j] ≤ v
  if i < j then
    x[i] ↔ x[j]
  end if
end while
x[i] ↔ x[ls]
return i
```

Această poziţie asigură o partiţionare echilibrată a şirului. Această situaţie nu este însă obţinută întotdeauna.

**Exemplul 6.5** Să considerăm şirul $(4, 7, 5, 3, 8, 2, 1)$. Aplicând acelaşi algoritm, după prima etapă se obţine: $i = 1$, $j = 1$ (a se observa că în acest caz fără a folosi o poziţie suplimentară cu rol de fanion, $x[0] = v$ condiţia de oprire a celui de-al doilea **repeat** nu este niciodată satisfăcută), şirul devine $(1, 7, 5, 3, 8, 2, 4)$ iar poziţia pivotului este $q = 1$. In acest caz s-a obţinut o partiţionare dezechilibrată (şirul se descompune într-un subşir vid, pivotul aflat pe prima poziţie şi un subşir constituit din celelalte elemente). Se poate remarca că pentru primul ciclu **repeat** $x[ls]$ joacă rolul unui fanion.

*Observaţii.*

(i) Inegalităţile de tip $\geq$ şi $\leq$ din ciclurile **repeat** fac ca, în cazul unor valori egale, să se obţină o partiţionare echilibrată şi nu una dezechilibrată cum s-ar întâmpla dacă s-ar folosi inegalităţi stricte. În acelaşi timp dacă s-ar utiliza inegalităţi stricte valoarea $v$ nu ar putea fi folosită ca fanion.

(ii) Problema nesatisfacerii condiţiei de oprire pentru ciclul după $j$ poate să apară doar în cazul în care $li = 1$ şi atunci când $v$ este cea mai mică valoare din şir. Este de preferat să se folosească valoare fanion decât să se modifice condiţia de oprire de la ciclu (de exemplu $x[j] \leq v$ **and** $j < i$).

(iii) La ieşirea din prelucrarea repetitivă exterioară (**while**) indicii $i$ şi $j$ satisfac una dintre relaţiile: $i = j$ sau $i = j + 1$. Ultima interschimbare

asigură plasarea valorii fanion pe poziţia sa finală.

Pentru a justifica corectitudinea algoritmului `partiţie1` considerăm aserţiunea: {dacă $i < j$ atunci $x[k] \leq v$ pentru $k = \overline{li, i}$ iar $x[k] \geq v$ pentru $k = \overline{j, ls}$ iar dacă $i \geq j$ atunci $x[k] \leq v$ pentru $k = \overline{li, i}$ iar $x[k] \geq v$ pentru $k = \overline{j+1, ls}$}. Cum precondiţia este $li < ls$, deci $i < j$ şi postcondiţiile sunt $\{x[k] \leq x[i]$ pentru $k = \overline{1, i-1}$, $x[k] \geq x[i]$ pentru $k = \overline{i+1, ls}\}$, se poate arăta că aserţiunea de mai sus este invariantă în raport cu prelucrarea repetitivă **while** iar după efectuarea ultimei interschimbări implică postcondiţiile.

Considerăm acum cealaltă variantă de partiţionare. Aceasta diferă de prima în special prin faptul că permite ca valoarea pivotului să participe la interschimbări. În plus el nu este plasat la sfârşit pe poziţia finală fiind necesară astfel sortarea subtablourilor $x[li..q]$ şi $x[q+1..ls]$. Aceasta face să nu mai fie necesară plasarea pe poziţia 0 a unui fanion întrucât se creează în mod natural fanioane pentru fiecare dintre cele două cicluri **repeat**. Această variantă este descrisă în Algoritmul 6.9.

**Algoritmul 6.9** Determinarea poziţiei de partiţionare

```
partitie2(x[li..ls])
v ← x[li] // se alege valoarea pivotului
i ← li − 1 // contor pentru parcurgere de la stânga la dreapta
j ← ls + 1 // contor pentru parcurgere de la dreapta la stânga
while i < j do
  repeat
    i ← i + 1
  until x[i] ≥ v
  repeat
    j ← j − 1
  until x[j] ≤ v
  if i < j then
    x[i] ↔ x[j]
  end if
end while
return j
```

În acest caz se poate arăta că aserţiunea {dacă $i < j$ atunci $x[k] \leq v$ pentru $k = \overline{li, i}$ iar $x[k] \geq v$ pentru $k = \overline{j, ls}$ iar dacă $i \geq j$ atunci $x[k] \leq v$ pentru $k = \overline{li, i-1}$ iar $x[k] \geq v$ pentru $k = \overline{j+1, ls}\}$ este invariantă în raport cu ciclul **while** iar la ieşirea din ciclu (când $i = j$ sau $i = j+1$) implică $x[k] \leq v$ pentru $k = \overline{li, j}$ şi $x[k] \geq v$ pentru $k = \overline{j+1, ls}$ adică postcondiţia $x[k1] \leq x[k2]$ pentru orice $k1 \in \{li, \ldots, j\}$, $k2 \in \{j+1, \ldots, ls\}$.

De remarcat că dacă se alege ca pivot $x[ls]$, varianta `quicksort2` nu va funcţiona corect putând conduce la situaţia în care unul dintre subtablouri este vid (ceea ce provoacă o succesiune nesfârşită de apeluri recursive, întrucât

nu se mai reduce dimensiunea problemei). În cazul în care se doreşte ca $x[ls]$ să fie valoarea pivot este necesar ca apelurile recursive să se facă pentru: `quicksort2`$(x[li..q-1])$ şi `quicksort2`$(x[q..ls])$.

Ca urmare a reorganizării şirului în etapa determinării pivotului sau a poziţiei de partiţionare, poziţia relativă a elementelor având aceeaşi valoare a cheii de sortare nu este neapărat conservată. Prin urmare algoritmul de sortare rapidă nu este stabil.

**Analiza complexităţii**

Analizăm complexitatea variantei `quicksort1`. Pe parcursul partiţionării, numărul de comparaţii depăşeşte în general numărul de interschimbări, motiv pentru care vom analiza doar numărul de comparaţii efectuate. Pentru un şir de lungime $n$ numărul de comparaţii efectuate în cele două cicluri **repeat** din `partiţionare1` este $n+i-j$ ($i$ şi $j$ fiind valorile finale ale contoarelor). Astfel, dacă $i = j$ se vor efectua $n$ comparaţii iar dacă $i = j+1$ se vor efectua $n+1$ comparaţii.

Influenţa majoră asupra numărului de comparaţii efectuate de către algoritmul `quicksort1` o are raportul dintre dimensiunile celor două subşiruri obţinute după partiţionare. Cu cât dimensiunile celor două subşiruri sunt mai apropiate cu atât se reduce numărul comparaţiilor efectuate. Astfel cazurile cele mai favorabile corespund partiţionării echilibrate iar cele mai puţin favorabile partiţionării dezechilibrate.

*Analiza în cazul cel mai favorabil.* Presupunem că la fiecare partiţionare a unui şir de lungime $n$ se obţin două subsecvenţe de lungimi $\lfloor n/2 \rfloor$ respectiv $n - \lfloor n/2 \rfloor - 1$ iar numărul de comparaţii din partiţionare este $n$. În aceste condiţii putem considera că marginea superioară a numărului de comparaţii satisface o relaţie de recurenţă de forma $T(n) = 2T(n/2) + n$ ceea ce conduce prin teorema master la o complexitate de ordin $n\lg n$. Prin urmare în cazul cel mai favorabil ordinul de complexitate a algoritmului `quicksort1` este $n\lg n$, adică algoritmul aparţine lui $\Omega(n\lg n)$.

*Analiza în cazul cel mai defavorabil.* Presupunem că la fiecare partiţionare se efectuează $n+1$ comparaţii şi că se generează un subşir vid şi unul de lungime $n-1$. Atunci relaţia de recurenţă corespunzătoare este $T(n) = T(n-1)+n+1$. Aplicând metoda iteraţiei se obţine $T(n) = (n+1)(n+2)/2 - 3$. Se obţine astfel că în cazul cel mai defavorabil complexitatea este pătratică, adică sortarea rapidă aparţine clasei $\mathcal{O}(n^2)$. Spre deosebire de metodele elementare de sortare pentru care un şir iniţial sortat conducea la un număr minim de prelucrări în acest caz tocmai aceste situaţii conduc la costul maxim (pentru un şir deja ordonat la fiecare partiţionare se obţine pivotul pe ultima poziţie).

Pentru a evita partiţionarea dezechilibrată au fost propuse diverse variante de alegere a valorii pivotului, una dintre acestea constând în selecţia a trei valori

din şir şi alegerea ca valoare a pivotului a medianei acestor valori (valoarea aflată pe a doua poziţie în tripletul ordonat).

*Analiza în cazul mediu.* Se bazează pe ipoteza că toate cele $n$ poziţii ale şirului au aceeaşi probabilitate de a fi selectate ca poziţie pivot (probabilitatea este în acest caz $1/n$). Presupunând că la fiecare partiţionare a unui şir de lungime $n$ se efectuează $n+1$ comparaţii, numărul de comparaţii efectuate în cazul în care poziţia pivotului este $q$ satisface: $T_q(n) = T(q-1) + T(n-q) + n + 1$. Prin urmare numărul mediu de comparaţii satisface:

$$\begin{aligned} T_m(n) &= \tfrac{1}{n}\sum_{q=1}^{n} T_q(n) = (n+1) + \tfrac{1}{n}\sum_{q=1}^{n}(T_m(q-1) + T_m(n-q)) \\ &= (n+1) + \tfrac{2}{n}\sum_{q=1}^{n} T_m(q-1). \end{aligned}$$

Scăzând între ele relaţiile:

$$nT_m(n) = 2\sum_{q=1}^{n} T_m(q-1) + n(n+1)$$

$$(n-1)T_m(n-1) = 2\sum_{q=1}^{n-1} T_m(q-1) + (n-1)n$$

se obţine relaţia de recurenţă pentru $T_m$:

$$nT_m(n) = (n+1)T_m(n-1) + 2n$$

Aplicând metoda iteraţiei rezultă:

$$\begin{array}{lcll} T_m(n) & = & \dfrac{n+1}{n}T_m(n-1) + 2 & \\ T_m(n-1) & = & \dfrac{n}{n-1}T_m(n-2) + 2 & \cdot\frac{n+1}{n} \\ T_m(n-2) & = & \dfrac{n-1}{n-2}T_m(n-3) + 2 & \cdot\frac{n+1}{n-1} \\ \vdots & & \vdots & \\ T_m(2) & = & \dfrac{3}{2}T_m(1) + 2 & \cdot\frac{n+1}{3} \\ T_m(1) & = & 0 & \end{array}$$

iar prin însumare se obţine:

$$\begin{aligned} T_m(n) &= 2(n+1)(1/n + 1/(n-1) + \ldots + 1/3) + 2 \\ &= 2(n+1)\textstyle\sum_{i=3}^{n}\frac{1}{i} + 2 \simeq 2(n+1)(\ln n - \ln 3) + 2. \end{aligned}$$

Prin urmare numărul mediu de comparaţii este de ordinul $2n\ln n \simeq 1.38n\lg n$. Aceasta înseamnă că în cazul mediu sortarea rapidă este doar cu 38% mai costisitoare decât în cazul cel mai favorabil.

## 6.4 Alte aplicaţii ale tehnicii divizării

### 6.4.1 Problema selecţiei

Fie $a[1..n]$ un tablou, nu neapărat ordonat. Să se determine al $k$-lea element al tabloului, selectat în ordine crescătoare (pentru $k = 1$ se obţine minimul, pentru $k = n$ se obţine maximul, pentru $k = \lfloor n/2 \rfloor$ se obţine mediana etc.).

O primă variantă de rezolvare a problemei o reprezintă ordonarea crescătoare a tabloului (prin metoda selecţiei) până ajung ordonate primele $k$ elemente ale tabloului. Numărul de operaţii efectuate în acest caz este proporţional cu $kn$. Pentru $k$ apropiat de $n/2$ aceasta conduce la un algoritm de complexitate pătratică. Un algoritm de complexitate mai mică se obţine aplicând strategia de partiţionare de la sortarea rapidă: folosind o valoare de referinţă (de exemplu cea aflată pe prima poziţie în tablou) se partiţionează tabloul în două subtablouri astfel încât elementele aflate în primul subtablou să fie toate mai mici decât valoarea de referinţă iar elementele din al doilea subtablou să fie toate mai mari decât valoarea de referinţă.

Algoritmul de partiţionare poate fi cel folosit în cadrul sortării rapide (Algoritmul 6.9). Dacă poziţia de partiţionare este $q$ iar $k \leq q - li + 1$ atunci problema se reduce la selecţia celui de al $k$-lea element din subtabloul $a[li..q]$. Dacă însă $k > q - li + 1$ atunci problema se reduce la selecţia celui de al $k-(q-li+1)$ element din subtabloul $a[q+1..ls]$. Valoarea parametrului $k$ este întotdeauna cuprinsă între 1 şi numărul de elemente din subtabloul prelucrat (în cazul în care $li = ls$ valoarea lui $k$ va fi 1). Determinarea celui de al $k$-lea element în ordine crescătoare este descrisă în Algoritmul 6.10.

**Algoritmul 6.10** Selecţia celui de al $k$-lea element în ordine crescătoare

```
selectie(integer a[li..ls], k)
if li = ls then
  return a[li]
else
  q ←partitie2(a[li..ls])
  r ← q − li + 1
  if k ≤ r then
    return selectie(a[li..q], k)
  else
    return selectie(a[q + 1..ls], k − r)
  end if
end if
```

În cazul cel mai favorabil, partiţionarea este echilibrată la fiecare etapă (cele două subtablouri au aproximativ acelaşi număr de elemente). Întrucât algoritmul de partiţionare are complexitate liniară putem presupune că numărul de comparaţii efectuate asupra elementelor tabloului satisface următoarea relaţie

de recurenţă:

$$T(n) = \begin{cases} 0 & n = 1 \\ T(n/2) + n & n > 1 \end{cases}$$

Aplicând teorema master pentru estimarea ordinului de complexitate (pentru $m = 2$, $d = 1$, $k = 1$) se obţine că, în cazul cel mai favorabil, algoritmul are complexitate liniară. În cazul cel mai defavorabil partiţionarea ar conduce la fiecare etapă la descompunerea într-un subtablou constituit dintr-un singur element şi la un subtablou constituit din celelalte elemente. Relaţia de recurenţă ar fi în acest caz:

$$T(n) = \begin{cases} 0 & n = 1 \\ T(n-1) + n & n > 1 \end{cases}$$

ceea ce conduce la o complexitate pătratică. Similar algoritmului de sortare rapidă, si algoritmul selecţiei se comportă în medie similar celui mai favorabil caz având o complexitate liniară.

### 6.4.2 Înmulţirea matricilor

Considerăm problema înmulţirii a două matrici pătratice de ordin $n$, notate cu $A[1..n, 1..n]$ şi $B[1..n, 1..n]$. Aplicând metoda clasică (vezi Exemplul 3.2 ) se obţine un algoritm de complexitate $\Theta(n^3)$ în raport cu numărul de înmulţiri scalare efectuate. Dacă se analizează calculele efectuate pentru înmulţirea a două matrici de ordin 2 se observă că matricea produs $C = A \cdot B$ are următoarele elemente:

$$\begin{array}{ll} c_{11} = m_1 + m_4 - m_5 + m_7 & c_{12} = m_3 + m_5 \\ c_{21} = m_2 + m_4 & c_{22} = m_1 + m_3 - m_2 + m_6 \end{array} \tag{6.1}$$

unde

$$\begin{array}{ll} m_1 = (a_{11} + a_{22}) \cdot (b_{11} + b_{22}) & m_2 = (a_{21} + a_{22}) \cdot b_{11} \\ m_3 = a_{11} \cdot (b_{12} - b_{22}) & m_4 = a_{22} \cdot (b_{21} - b_{11}) \\ m_5 = (a_{11} + a_{12}) \cdot b_{22} & m_6 = (a_{21} - a_{11}) \cdot (b_{11} + b_{12}) \\ m_7 = (a_{12} - a_{22}) \cdot (b_{21} + b_{22}) & \end{array} \tag{6.2}$$

În felul acesta numărul înmulţirilor efectuate este 7 iar numărul adunărilor efectuate este 18. În varianta clasică numărul de înmulţiri scalare este 8 iar numărul de adunări este 4. Se observă astfel că dacă se aplică tehnica de mai sus, numărul înmulţirilor se reduce de la 8 la 7 cu preţul creşterii numărului de adunări de la 4 la 18. Pentru $n = 2$ nu se obţine un câştig prin aplicarea acestei tehnici însă pe măsură ce $n$ creşte diferenţa devine mai semnificativă.

Extinderea tehnicii de la cazul $n = 2$ la cazul general se bazează pe tehnica divizării. Se consideră că $n = 2^p$ (în cazul în care $n$ nu este o putere a lui doi atunci se extind dimensiunile matricii prin bordarea cu linii şi coloane nule

până se ajunge la cea mai mică putere a lui 2 care îl depăşeşte pe $n$) şi se împart matricile în patru submatrici de ordin $2^{p-1}$:

$$\left[\begin{array}{c|c} C_{11} & C_{12} \\ \hline C_{21} & C_{22} \end{array}\right] = \left[\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right] \cdot \left[\begin{array}{c|c} B_{11} & B_{12} \\ \hline B_{21} & B_{22} \end{array}\right] \tag{6.3}$$

iar în felul acesta $C_{ij}$ pot fi calculate folosind relaţiile (6.1) şi (6.2) extinse la cazul matricilor. Tehnica aceasta de înmulţire a matricilor este cunoscută ca metoda lui Strassen după numele celui care a propus-o. Numărul de operaţii de înmulţire efectuate, $T(n)$ satisface următoarea relaţie de recurenţă:

$$T(n) = \begin{cases} 1, & n = 1 \\ 7T(n/2), & n > 1 \end{cases}$$

Cum $n = 2^p$ se obţine că $T(n) = 7^p = 7^{\log_2 n} = n^{\log_2 7} \simeq n^{2.807}$. Se pune însă întrebarea dacă numărul de adunări nu devine în schimb foarte mare. Notând cu $S(n)$ numărul de adunări efectuate şi ţinând cont că la adunarea a două matrici de dimensiune $n$ se efectuează $n^2$ adunări, se obţine relaţia de recurenţă:

$$S(n) = \begin{cases} 0, & n = 1 \\ 7S(n/2) + 18(n/2)^2, & n > 1 \end{cases}$$

Aplicând Teorema master pentru $k = 7$, $m = 2$ şi $d = 2$ (care se încadrează în cazul 3 al teoremei) se obţine că $S(n) \in \Theta(n^{\log_2 7})$ adică acelaşi ordin de complexitate ca şi în cazul înmulţirilor. Prin urmare metoda lui Strassen are un ordin de complexitate mai mic decât metoda clasică însă câştigul obţinut nu este suficient de semnificativ pentru a compensa creşterea valorii constantei multiplicative şi pierderea simplităţii calculelor.

### 6.4.3 Înmulţirea polinoamelor

Din punct de vedere algebric un polinom de gradul $n$ este o expresie de forma $A(x) = a_n x^n + \ldots a_1 x + a_0$ unde $a_0, \ldots, a_n$ reprezintă coeficienţii iar simbolul $x$ este denumit nedeterminată. Un polinom poate fi reprezentat fie prin şirul coeficienţilor $(a_0, a_1, \ldots, a_n)$ fie printr-o mulţime de $n$ perechi de valori $\{(x_1, y_1), ..., (x_n, y_n)\}$ unde $y_i$ reprezintă valoarea polinomului în cazul în care nedeterminata este egală cu $x_i$. Trecerea de la reprezentarea bazată pe coeficienţi la cea bazată pe valori se realizează prin *evaluarea* polinomului pentru argumentele $x_1$,...,$x_n$ iar cea inversă se bazează pe procesul de *interpolare*.

Să considerăm două polinoame de grad $n$ respectiv $m$: $A(x) = a_n x^n + \ldots + a_1 x + a_0$ şi $B(x) = b_m x^m + \ldots + b_1 x + b_0$. Putem presupune că $n \geq m$. Suma celor două polinoame ($S(x) = A(x) + B(x)$) va fi un polinom de grad $n$ având coeficienţii: $s_i = a_i + b_i$ pentru $i = \overline{0, m}$ şi $s_i = a_i$ pentru $i = \overline{m+1, n}$. Produsul, $C(x) = A(x) \cdot B(x)$ va fi un polinom de gradul $m + n$ ai cărui

coeficienţi satisfac

$$c_k = \sum_{i \in M_k} a_i b_{k-i}, \qquad M_k = \{j \in \{0, \ldots, n\} | 0 \leq k - j \leq m\}. \tag{6.4}$$

Determinarea coeficienţilor produsului poate fi făcută folosind Algoritmul 6.11.

**Algoritmul 6.11** Produsul a două polinoame reprezentate prin tablourile coeficienţilor

```
1: produs(real a[0..n],b[0..m])
2: real c[0..m + n]
3: c[0..m + n] ← 0
4: for i ← 0, n do
5:    for j ← 0, m do
6:       c[i + j] ← c[i + j] + a[i] ∗ b[j]
7:    end for
8: end for
9: return c[0..m + n]
```

Atribuiri ca cea din linia 3 a Algoritmului 6.11 indică faptul că toate elementele tabloului vor fi iniţializate cu 0. Este uşor de văzut că algoritmul `produs` aparţine lui $\Theta(mn)$. Să analizăm dacă ordinul de complexitate poate fi redus prin aplicarea tehnicii divizării. Pentru a simplifica notaţiile să considerăm că ambele polinoame au gradul $n-1$ (deci $n$ coeficienţi). Ideea de divizare este de a rescrie polinoamele astfel:

$$\begin{aligned} A(x) &= (a_{n-1}x^{n/2-1} + \ldots + a_{n/2})x^{n/2} + (a_{n/2-1}x^{n/2-1} + \ldots + a_0) \\ &= A_1(x)x^{n/2} + A_0(x) \\ B(x) &= (b_{n-1}x^{n/2-1} + \ldots + b_{n/2})x^{n/2} + (b_{n/2-1}x^{n/2-1} + \ldots + b_0) \\ &= B_1(x)x^{n/2} + B_0(x) \end{aligned} \tag{6.5}$$

Produsul a două polinoame de grad $n-1$ se poate reduce la a calcula mai multe produse între polinoame de grad $n/2-1$:

$$A(x) \cdot B(x) = A_1(x) \cdot B_1(x) \cdot x^n + (A_1(x) \cdot B_0(x) + A_0(x) \cdot B_1(x)) \cdot x^{n/2} + A_0(x) \cdot B_0(x) \tag{6.6}$$

Dacă notăm cu $T(n)$ numărul de înmulţiri scalare între coeficienţii celor două polinoame se obţine relaţia de recurenţă:

$$T(n) = \begin{cases} 4T(n/2), & n > 1 \\ 1, & n = 1 \end{cases} \tag{6.7}$$

Aplicând Teorema Master pentru valorile $m = 2$, $k = 4$ şi $d = 1$ se obţine că $T(n) \in \Theta(n^{\log_2 4}) = \Theta(n^2)$ ceea ce corespunde şi variantei clasice de înmulţire. Pentru a reduce ordinul de complexitate ar trebui ca în loc de 4 înmulţiri ale unor polinoame de grad $n/2-1$ să se efectueaze mai puţine, eventual cu preţul efectuării mai multor adunări sau scăderi. Se observă uşor că:

$$A_1(x) \cdot B_0(x) + A_0(x) \cdot B_1(x) = \quad (A_1(x) + A_0(x)) \cdot (B_1(x) + B_0(x))$$
$$-A_1(x) \cdot B_1(x) - A_0(x) \cdot B_0(x)$$

Prin urmare numărul de înmulţiri poate fi redus de la 4 la 3. Relaţia de recurenţă devine $T(n) = 3T(n/2)$ iar ordinul de complexitate $\Theta(n^{\log_2 3}) = \Theta(n^{1.584})$. Această reducere a ordinului de complexitate este însă plătită prin faptul că algoritmul devine mai dificil de implementat şi necesită utilizarea unor tablouri de manevră suplimentare (Algoritm 6.12).

**Algoritmul 6.12** Produsul a două polinoame reprezentate prin tablourile coeficienţilor folosind tehnica divizării

`produs2`(**real** $a[0..n-1]$,$b[0..n-1]$)
**real** $c[0..2n-2]$, $a1[0..n/2-1]$,$a0[0..n/2-1]$
**real** $b1[0..n/2-1]$,$b0[0..n/2-1]$,$c[0..n/2]$
**real** $sa[0..n/2-1]$,$sb[0..n/2-1]$,$p[0..n-2]$
$a0[0..n/2-1] \leftarrow a[0..n/2-1]$
$a1[0..n/2-1] \leftarrow a[n/2..n-1]$
$b0[0..n/2-1] \leftarrow b[0..n/2-1]$
$b1[0..n/2-1] \leftarrow b[n/2..n-1]$
$sa[0..n/2-1] \leftarrow a0[0..n/2-1] + a1[0..n/2-1]$
$sb[0..n/2-1] \leftarrow b0[0..n/2-1] + b1[0..n/2-1]$
$p[0..n-2] \leftarrow$ `produs2`$(sa[0..n/2-1], sb[0..n/2-1])$
$c[n..2n-2] \leftarrow$ `produs2`$(a1[0..n/2-1], b1[0..n/2-1])$
$c[0..n-2] \leftarrow$ `produs2`$(a0[0..n/2-1], b0[0..n/2-1])$
$c[n/2..n-1] \leftarrow p[0..n-2] - c[n..2n-2] - c[0..n-2]$
**return** $c[0..2n-2]$

Există algoritmi mai eficienţi (de ordin $\Theta(n \lg n)$) pentru calculul produsului a două polinoame însă mai sofisticaţi (bazaţi pe utilizarea unor noţiuni şi tehnici din matematică: rădăcinile complexe ale unităţii şi transformarea Fourier discretă). Pentru detalii poate fi consultată [3]. Diferenţa între ordinele de complexitate $\Theta(n \lg n)$ şi $\Theta(n^{1.58})$ este semnificativă pentru valori mari ale lui $n$ (vezi Figura 6.2).

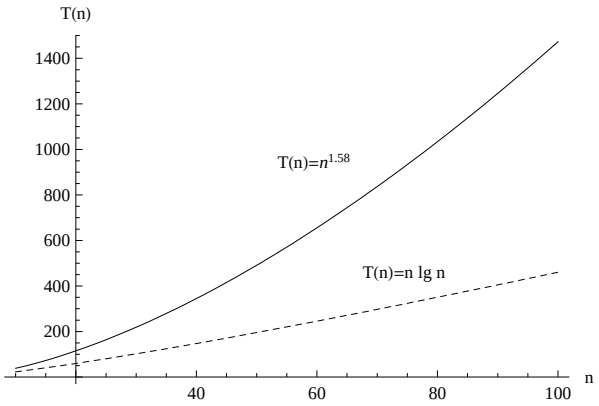


Figura 6.2: Diferenţe între $n\lg n$ şi $n^{1.58}$.

## 6.5 Probleme

**Problema 6.1** Să se aplice tehnica divizării pentru a determina valoarea minimă şi valoarea maximă a unui tablou $x[1..n]$. Să se determine numărul de comparaţii efectuate asupra elementelor tabloului.

*Rezolvare.* Se foloseşte ideea de la tehnica divizării: se împarte succesiv tabloul în două subtablouri de dimensiuni cât mai apropiate (prin înjumătăţire) până se ajunge la dimensiunea critică ($n_c = 2$ sau $n_c = 1$). Pentru fiecare dintre cele două subtablouri se determină valoarea minimă şi cea maximă. Rezultatele subproblemelor se combină comparând valoarea minimă din primul subtablou cu valoarea minimă din al doilea subtablou respectiv valoarea maximă din primul subtablou cu valoarea maximă din al doilea. O astfel de abordare este descrisă în Algoritmul 6.13 unde $li$ şi $ls$ reprezintă limita inferioară respectiv cea superioară a indicilor subtabloului, iar apelul se face pentru $li = 1$ şi $ls = n$.

Notând cu $T(n)$ numărul de comparaţii efectuate asupra elementelor tabloului şi considerând că $n = 2^k$ se obţine următoarea relaţie de recurenţă:

$$T(n) = \begin{cases} 1, & n = 2 \\ 2T(n/2) + 2, & n > 2 \end{cases}$$

Aplicând metoda substituţiei inverse pentru rezolvarea relaţiei de recurenţă se obţine că dacă $n = 2^k$ atunci $T(n) = 2^{k-1} + 2(2^{k-1} - 1) = 3/2n - 2$. Ordinul de complexitate este $\mathcal{O}(n)$ la fel ca prin aplicarea metodei clasice (parcurgerea secvenţială a tabloului) însă constanta multiplicativă este mai mică (în cazul abordării clasice se obţine $T(n) = 2(n-1)$).

**Problema 6.2** (*Determinarea medianei.*) Mediana unui tablou cu $n$ elemente este elementul aflat pe poziţia $\lfloor (n+1)/2 \rfloor$ în cadrul tabloului ordonat crescător

**Algoritmul 6.13** Determinarea minimului şi maximului dintr-un tablou folosind tehnica divizării

```
maxmin(real x[li..ls])
integer m
real min, max, min1, max1, min2, max2
if li = ls then
   min = x[li]; max = x[ls];
else
   if ls = li + 1 then
      if x[li] ≤ x[ls] then
         min ← x[li]; max ← x[ls]
      else
         min ← x[ls]; max ← x[li]
      end if
   else
      m ← ⌊(li + ls)/2⌋
      (min1, max1) ←maxmin(x[li..m])
      (min2, max2) ←maxmin(x[m + 1..ls])
      if min1 ≤ min2 then
         min ← min1
      else
         min ← min2
      end if
      if max1 ≤ max2 then
         max ← max1
      else
         max ← max2
      end if
   end if
end if
return (min, max)
```

(în cazul în care $n$ este par se consideră uneori ca mediană media aritmetică a celor două valori aflate în mijlocul tabloului). Fie $x[1..n]$ şi $y[1..n]$ două tablouri ordonate crescător. Să se determine mediana tabloului $z[1..2n]$ care conţine toate elementele din $x$ şi $y$.

*Rezolvare.* O primă variantă ar fi să se construiască tabloul $z$ prin interclasare şi să se returneze elementul de pe poziţia $n$. Este însă suficient să se realizeze o interclasare parţială până când se obţine elementul de poziţia $n$. În acest scop se poate folosi interclasarea bazată pe valori santinelă mai mari decât elementele din ambele tablouri. Ideea este ilustrată în Algoritmul 6.14. Este evident că algoritmul are complexitate liniară.

**Algoritmul 6.14** Determinarea medianei prin interclasare parţială

```
mediana (integer x[1..n], y[1..n])
x[n+1] ← |x[n]| + |y[n]|; y[n+1] ← |x[n]| + |y[n]|;
i ← 1; j ← 1;
for k ← 1, n do
  if x[i] < y[j] then
    z[k] ← x[i]; i ← i + 1
  else
    z[k] ← y[j]; j ← j + 1
  end if
end for
return z[n]
```

**Problema 6.3** Fie $a[1..m]$ şi $b[1..n]$ două tablouri ordonate crescător având elemente nu neapărat distincte. Să se construiască un tablou strict crescător ce conţine elementele distincte din $a$ şi $b$.

*Indicaţie.* Se aplică tehnica interclasării verificând de fiecare dată la completarea în tabloul destinaţie dacă elementul ce ar trebui adăugat este diferit de ultimul element din tablou.

**Problema 6.4** Se consideră două numere întregi date prin tablourile corespunzătoare descompunerilor lor în factori primi. Să se construiască tablourile similare corespunzătoare celui mai mic multiplu comun al celor două valori.

*Rezolvare.* Să considerăm numerele: $3415 = 3 \cdot 5^3 \cdot 7 \cdot 13$ şi $966280 = 2^3 \cdot 5 \cdot 7^2 \cdot 17 \cdot 29$. Primului număr îi corespund tablourile: $(3, 5, 7, 13)$ respectiv $(1, 3, 1, 1)$ iar celui de al doilea număr îi corespund tablourile $(2, 5, 7, 17, 29)$ respectiv $(3, 1, 2, 1, 1)$.

Tablourile corespunzătoare celui mai mic multiplu comun vor fi:

- *factori primi:* tabloul care conţine toţi factorii primi corespunzători celor două numere şi care poate fi obţinut prin interclasare ţinând cont că aceştia trebuie să fie distincţi.
- *exponenţi:* tabloul ce conţine valorile maxime ale exponenţilor.

În cazul exemplului cele două tablouri sunt: $(2, 3, 5, 7, 13, 17, 29)$ respectiv $(3, 1, 3, 2, 1, 1, 1)$. Algoritmul 6.15 descrie metoda de mai sus.

**Problema 6.5** Fie $a[1..m]$ şi $b[1..n]$ două tablouri ordonate strict crescător. Propuneţi un algoritm de complexitate liniară pentru determinarea mulţimii elementelor comune celor două tablouri.

*Indicaţie.* Se foloseşte ideea de la interclasare însă se transferă în tabloul rezultat doar elementele comune celor două tablouri.

**Algoritmul 6.15** Determinarea celui mai mic multiplu comun a două numere descrise prin descompunerea lor în factori primi

```
cmmc(integer fa[1..m],pa[1..m],fb[1..n],pb[1..n] )
integer i,j, k, fc[1..k],pc[1..k]
i ← 1; j ← 1; k ← 0
while i ≤ m and j ≤ n do
  if fa[i] < fb[j] then
    k ← k + 1; fc[k] ← fa[i]; pc[k] ← pa[i]; i ← i + 1
  else
    if fa[i] > fb[j] then
      k ← k + 1; fc[k] ← fb[j]; pc[k] ← pb[j]; j ← j + 1
    else
      k ← k+1; fc[k] ← fa[i]; pc[k] ← max(pa[i], pb[j]); i ← i+1;j ← j+1
    end if
  end if
end while
while i ≤ m do
  k ← k + 1; fc[k] ← fa[i]; pc[k] ← pa[i]; i ← i + 1
end while
while j ≤ n do
  k ← k + 1; fc[k] ← fb[j]; pc[k] ← pb[j]; j ← j + 1
end while
return fc[1..k], pc[1..k]
```

**Problema 6.6** Se consideră două numere întregi date prin tablourile corespunzătoare descompunerilor lor în factori primi. Să se construiască tablourile similare corespunzătoare celui mai mare divizor comun al celor două valori.

*Indicaţie.* Se determină factorii primi comuni iar pentru aceştia se ia puterea minimă.

**Problema 6.7** Propuneţi un algoritm de complexitate medie $\mathcal{O}(n \log n)$ pentru a verifica dacă elementele unui tablou sunt distincte sau nu.

*Indicaţie.* Se ordonează tabloul folosind sortarea rapidă după care se parcurge şi se compară elementele vecine. Dacă există cel puţin două elemente vecine identice atunci tabloul iniţial nu are elementele distincte.

**Problema 6.8** Se consideră un caroiaj de dimensiuni $2^k \times 2^k$ (pentru $k = 3$ se obţine o tablă de şah clasică) în care unul dintre pătrăţele este marcat. Propuneţi o strategie bazată pe tehnica divizării care acoperă întreaga tablă (cu excepţia pătrăţelului marcat) cu piese constând din 3 pătrăţele aranjate în forma de L (indiferent de orientare).

# Capitolul 7

# Tehnica alegerii local optimale

O clasă de probleme frecvent întâlnită în practică este cea a problemelor de *optimizare* de tipul:

> *Să se determine $x \in X$ astfel încât: (i) $x$ satisface anumite condiţii (restricţii); (ii) $x$ optimizează (minimizează sau maximizează) un criteriu.*

O subclasă importantă în informatică o reprezintă cea în care $X$ este o mulţime finită, problema de optimizare fiind numită în acest caz *discretă* sau *combinatorială*. La o primă vedere problema pare foarte simplă întrucât este suficient să se parcurgă $X$ şi să se aleagă elementul care satisface restricţiile şi optimizează criteriul. O astfel de abordare bazată pe metoda forţei brute devine ineficientă (sau chiar impracticabilă) atunci când numărul de elemente din $X$ este foarte mare.

**Exemplul 7.1** (*problema submulţimii de sumă dată şi cardinal minim*) Se consideră mulţimea $A = \{a_1, a_2, \ldots, a_n\} \subset \mathbb{R}$ şi valoarea $C \leq \sum_{i=1}^{n} a_i$. Se cere să se determine (dacă există) o submulţime având cât mai puţine elemente şi a căror sumă să fie egală cu $C$.

În acest caz $X$ reprezintă ansamblul submulţimilor lui $A$. O abordare prin metoda forţei brute a unei astfel de probleme necesită generarea tuturor submulţimilor lui $A$, reţinerea celor care satisfac restricţia (suma elementelor egală cu $C$) şi alegerea celei (celor) ce au cele mai puţine elemente. Un astfel de algoritm ar avea ordinul de complexitate $\mathcal{O}(2^n)$.

Pentru a rezolva mai eficient probleme de acest tip au fost dezvoltate tehnici specifice. Două dintre acestea sunt tehnica alegerii local optimale ("greedy" sau alegere lacomă) şi cea a programării dinamice.

## 7.1 Principiul tehnicii

Să considerăm problema de optimizare formulată într-o variantă uşor diferită:

> *Fie $A = (a_1, a_2, \ldots, a_n)$ un set finit de elemente (nu neapărat distincte). Să se determine $S = (s_1, s_2, \ldots s_k) \subset A$ astfel încât $S$ să satisfacă anumite restricţii şi să optimizeze un criteriu.*

În continuare prin set vom înţelege o mulţime în sens general (elementele sale nu sunt neapărat distincte). O astfel de entitate este uneori numită *multiset*. Principiul metodei constă în a construi succesiv soluţia $S$ pornind de la setul vid şi adăugând la fiecare etapă un nou element, şi anume acela care pare "optim" la momentul respectiv. Selecţia elementului care va fi eventual adăugat urmăreşte satisfacerea restricţiilor şi apropierea de optim. Insă alegerea care pare optimă la etapa curentă s-ar putea să nu fie cea mai bună şi din punct de vedere global astfel că e posibil ca o astfel de strategie "lacomă" să nu conducă la o soluţie finală optimă. Totuşi, o dată efectuată o alegere nu se mai revine asupra ei, aceasta fiind irevocabilă. Datorită modului în care se alege următoarea componentă a soluţiei această tehnică este cunoscută şi ca tehnica *alegerii lacome* sau pur şi simplu *greedy*. Trebuie menţionat şi faptul că prin strategia greedy se obţine o singură soluţie chiar dacă problema are mai multe.

Structura generală a algoritmului este descrisă în continuare.

`greedy1`($A$)
$S \leftarrow \emptyset$
**while** "$S$ nu este soluţie" **and** "există elemente neselectate în $A$" **do**
  selectează $a$ din $A$
  **if** $S \cup \{a\}$ satisface restricţiile **then**
    se adaugă $a$ la $S$
  **end if**
**end while**
**return** $S$

Etapa cea mai importantă este cea a selectării unui nou element. Pentru anumite probleme este posibil ca setul iniţial, $A$, să fie ordonat după un criteriu determinat de criteriul de optim astfel că elementele vor fi alese chiar în ordinea în care apar în $A$. Criteriul de sortare depinde de problema concretă care se rezolvă. În acest caz algoritmul poate fi descris ceva mai detaliat astfel:

`greedy2`($A[1..n]$)
$A[1..n] \leftarrow \text{sort}(A[1..n])$ // sortare după un criteriu corelat cu cel de optimizat
$i \leftarrow 1$ // contor de parcurgere a lui $A$
$k \leftarrow 0$ // contor utilizat în construirea lui $S$
**while** "$S[1..k]$ nu este soluţie" **and** $i \leq n$ **do**
  **if** "$(S[1], \ldots, S[k], A[i])$ satisface restricţiile" **then**
    $k \leftarrow k + 1$; $S[k] \leftarrow A[i]$ // se transferă elementul curent din $A$ în $S$
  **end if**

$i \leftarrow i + 1$ // se trece la următorul element din $A$
**end while**
**return** $S[1..k]$

În aplicaţii contează principiul metodei, algoritmii construiţi pe baza lui putând fi destul de diferiţi de structurile generale `greedy1` respectiv `greedy2`.

**Exemplul 7.2** (*Problema submulţimii de cardinal dat şi sumă maximă.*) Se cere determinarea unui subset cu $m$ elemente, $S$, dintr-un set $A$ având $n > m$ elemente astfel încât suma elementelor selectate să fie cât mai mare.

În acest caz este suficient să ordonăm descrescător elementele din $A$ şi să reţinem primele $m$ elemente:

`submulţime`($A[1..n]$,$m$)
$A[1..n] \leftarrow$ sortare_descrescătoare($A[1..n]$)
**for** $i \leftarrow 1, m$ **do**
  $S[i] \leftarrow A[i]$
**end for**
**return** $S[1..m]$

Pentru această problemă se va demonstra în secţiunea următoare că strategia greedy produce întotdeauna o soluţie optimă.

**Exemplul 7.3** (*Problema monedelor.*) Presupunem că o sumă de bani $C$ trebuie acoperită folosind cât mai puţine monede. Dacă valorile monedelor sunt $\{v_1, v_2, \ldots, v_n\}$ atunci a rezolva problema înseamnă a determina $(k_1, k_2, \ldots, k_n)$ ($k_i$ reprezintă numărul de monede de valoare $v_i$ şi poate fi egal cu 0) astfel încât să fie satisfăcută restricţia $\sum_{i=1}^{n} k_i v_i = C$ iar $\sum_{i=1}^{n} k_i$ să fie cât mai mică. Se observă că această problemă este similară cu problema submulţimii de sumă dată şi cardinal minim cu observaţia că setul $A$ este infinit: $A = \{v_1, v_1, \ldots, v_2, v_2, \ldots, \ldots, v_n, v_n, \ldots)$ (se presupune că pentru fiecare valoare $v_i$ există un număr nelimitat de elemente).

O abordare de tip greedy ar conduce (în cazul în care numărul de monede de fiecare tip este nelimitat) la un algoritm de forma:

`monede`($v[1..n]$,$C$)
$v[1..n] \leftarrow$ sortare_descrescătoare($v[1..n]$)
**for** $i \leftarrow 1, n$ **do**
  $S[i] \leftarrow 0$
**end for**
$i \leftarrow 1$
**while** $C > 0$ **and** $i \leq n$ **do**
  $S[i] \leftarrow C$ DIV $v[i]$
  $C \leftarrow C$ MOD $v[i]$
  $i \leftarrow i + 1$
**end while**
**if** $C = 0$ **then**

```
    return S[1..n]
  else
    "nu s-a gasit solutie"
  end if
```

*Observaţie.* Tehnica greedy nu conduce pentru orice instanţă a problemei monedelor la soluţia optimă. De exemplu pentru cazul monedelor cu valorile $(25, 20, 10, 5, 1)$ şi a valorii $C = 40$ strategia greedy ar conduce la soluţia $(1, 0, 1, 1, 0)$ în timp ce varianta $(0, 2, 0, 0, 0)$ este optimă. Pe de altă parte există situaţii în care problema nu are soluţii. De exemplu pentru monede având valorile $(20, 10, 5)$ şi $C = 17$ problema nu are soluţie. Se observă uşor că dacă există monedă cu valoarea 1 atunci pentru orice sumă $C$ se poate determina o soluţie. Aceasta nu este însă neapărat optimală. Tehnica greedy conduce la o soluţie optimă pentru problema monedelor doar dacă valorile acestora satisfac anumite restricţii. Un exemplu de valori pentru care se poate demonstra optimalitatea soluţiei este: $v_{i-1}$ este multiplu al lui $v_i$ pentru fiecare $i \geq 2$ (în condiţiile în care $v[1..n]$ este ordonat descrescător).

## 7.2 Verificarea corectitudinii şi analiza complexităţii

Tehnica alegerii local optimale este o tehnică intuitivă şi eficientă, dar nu garantează obţinerea soluţiei optime decât pentru anumite cazuri particulare de probleme.

### 7.2.1 Verificarea corectitudinii

Intrucât tehnica greedy nu asigură implicit optimalitatea soluţiei, aceasta trebuie demonstrată pentru cazul particular al problemei tratate. In general, problemele pentru care tehnica greedy poate fi aplicată cu succes au următoarele proprietăţi:

- *Proprietatea de alegere lacomă.* Inseamnă că se poate ajunge la soluţia optimă global efectuând alegeri local optimale. Pentru a demonstra că o problemă are această proprietate se porneşte de la o soluţie optimă global şi se arată că poate fi modificată astfel încât prima componentă să fie obţinută efectuând o alegere local optimală. Se analizează în continuare restul soluţiei (problema se reduce la una similară, dar de dimensiune mai mică). Aplicând principiul inducţiei matematice se arată că la fiecare pas poate fi efectuată o alegere de tip greedy. Această etapă se reduce de fapt la a arăta că problema are proprietatea de *substructură optimă.*

- *Proprietatea de substructură optimă.* Se consideră că o problemă are proprietatea de substructură optimă dacă o soluţie optimă a problemei

conţine soluţii optime ale subproblemelor. Pentru a demonstra acest lucru se poate folosi tehnica reducerii la absurd. Pornind de la o soluţie optimă $S(n) = (s_1, s_2, \ldots, s_n)$ se arată că $S(n-1) = (s_2, \ldots, s_n)$ este soluţie optimă pentru subproblema de dimensiune $n-1$. Se presupune că $S(n-1)$ nu este optimă. Atunci ar exista $S'(n-1) = (s'_2, \ldots, s'_n)$ o soluţie mai bună. Aceasta conduce de regulă la faptul că $S'(n) = (s_1, s'_2, \ldots, s'_n)$ este mai bună decât $S(n)$ ceea ce contrazice ipoteza că $S(n)$ este optimă.

**Exemplul 7.2** Demonstrăm că algoritmul `submulţime` generează soluţia optimă.
*Proprietatea de alegere lacomă.* Fie $O = (o_1, o_2, \ldots, o_m)$ o soluţie optimă a problemei. Cum nu contează ordinea elementelor în $O$ putem presupune că $o_1 > o_2 > \ldots > o_m$. Cum elementele lui $A$ sunt ordonate descrescător $(a_1 > a_2 > \ldots > a_n)$ prin alegere de tip greedy prima componentă ar trebui să fie $a_1$. Vom arăta că $o_1 = a_1$ prin metoda reducerii la absurd. Presupunem că $o_1 \neq a_1$. Rezultă că $a_1 > o_1$ iar soluţia $O' = (a_1, o_2, \ldots, o_m)$ are proprietatea că $\sum_{o \in O'} o - \sum_{o \in O} o = a_1 - o_1 > 0$ adică este mai bună decât $O$. Aceasta însă contrazice ipoteza că $O$ este soluţie optimă. Prin urmare $o_1 = a_1$ adică primul element al soluţiei este ales conform strategiei greedy. Este suficient acum să demonstrăm că problema posedă proprietatea de substructură optimă.
*Proprietatea de substructură optimă.* Fie $O = (o_1, o_2, \ldots, o_m)$ o soluţie optimă a problemei iniţiale. Presupunem că elementele lui $A$ sunt ordonate descrescător $(a_1 > a_2 > \ldots > a_n)$. Arătăm că $O_{(1)} = (o_2, \ldots, o_m)$ este soluţie optimă a problemei similare pentru $A_{(2)} = (a_2, \ldots, a_n)$ prin reducere la absurd. Dacă nu ar fi soluţie optimă înseamnă că ar exista o altă soluţie mai bună: $(o'_2, \ldots, o'_m)$ care completată pe prima poziţie cu $o_1$ ar conduce la o soluţie globală mai bună decât $O$. S-a ajuns la o contradicţie, deci problema posedă proprietatea substructurii optime.

**Exemplul 7.3** Presupunem că valorile din *problema monedelor* satisfac proprietăţile: $v_1 > v_2 > \ldots > v_n = 1$ şi $v_{i-1} = d_{i-1} v_i$ ($v_{i-1}$ este multiplu al lui $v_i$) pentru $i = \overline{2, n}$ şi demonstrăm că în acest caz tehnica greedy conduce la o soluţie optimă.

*Proprietatea alegerii greedy.* Fie $O = (o_1, \ldots, o_n)$ o soluţie optimă ($\sum_{i=1}^{n} o_i v_i = C$ şi $\sum_{i=1}^{n} o_i = K$ este minimă). Arătăm că prima componentă a lui $O$ satisface proprietatea alegerii greedy, adică $o_1 = \lfloor C/v_1 \rfloor$. Fie $C_2 = C - o_1 v_1$ suma ce rămâne după ce s-au dat $o_1$ monede de valoare $v_1$. Analizăm două situaţii:

*(i)* $C_2 < v_1$. În acest caz $C_2 = C \bmod v_1 = C - \lfloor C/v_1 \rfloor v_1$ deci $o_1 = \lfloor C/v_1 \rfloor$.

*(ii)* $C_2 \geq v_1$. Pornind de la $O$ construim o nouă soluţie $O'$ caracterizată prin $o'_1 = o_1 + \lfloor C_2/v_1 \rfloor$. Suma ce rămâne de acoperit după ce s-au dat monedele de valoare $v_1$ este de această dată $C'_2 = C - o'_1 v_1 = C_2 - \lfloor C_2/v_1 \rfloor v_1 < C_2$. Diferenţa $C_2 - C'_2$ era acoperită în soluţia $O$ din monedele de valoare mai

mică. Să considerăm cazul particular în care $\lfloor C_2/v_1 \rfloor v_1 \leq o_2 v_2$ (diferenţa era acoperită doar cu monede de valoare $v_2$). Noua soluţie va fi atunci: $O' = (o'_1, o'_2, o_3, \ldots, o_n)$ cu $o'_1 = o_1 + \lfloor C_2/v_1 \rfloor$ şi $o'_2 = o_2 - \lfloor C_2/v_1 \rfloor d_1$. Aceasta acoperă valoarea:

$$o'_1 v_1 + o'_2 v_2 + o_3 v_3 + \ldots + o_n v_n = (o_1 + \lfloor C_2/v_1 \rfloor) v_1 + (o_2 - \lfloor C_2/v_1 \rfloor d_1) v_2 + \sum_{i=3}^{n} o_i v_i.$$

Cum $v_1 = d_1 v_2$ rezultă că suma este chiar $C$. În ceea ce priveşte numărul de monede folosite, acesta este:

$$o'_1 + o'_2 + o_3 + \ldots + o_n = o_1 + \lfloor C_2/v_1 \rfloor + o_2 - \lfloor C_2/v_1 \rfloor d_1 + \sum_{i=3}^{n} o_i =$$

$$= \sum_{i=1}^{n} o_i + \lfloor C_2/v_1 \rfloor (1 - d_1) < K$$

adică alegerea de tip greedy conduce la utilizarea a mai puţine monede. Aceasta contrazice ipoteza că $O$ este soluţie optimă. Prin urmare primul element se alege după tehnica greedy. Acelaşi raţionament se aplică şi dacă $\lfloor C_2/v_1 \rfloor v_1 > o_2 v_2$ doar că suma se acoperă din contribuţia mai multor monede de valoare mai mică, fapt posibil datorită relaţiei de divizibilitate existente între valori).

*Proprietatea de substructură optimă.* Fie $O = (o_1, \ldots, o_n)$ o soluţie optimă. Arătăm că $O_{(2)} = (o_2, \ldots, o_n)$ e soluţie optimă pentru problema acoperirii sumei $C_2 = C - o_1 v_1$ cu monede de valori $v_2, v_3, \ldots, v_n$. Presupunem că $O_{(2)}$ nu este optimă, adică există $O'_{(2)} = (o'_2, \ldots, o'_n)$ cu proprietăţile $\sum_{i=2}^{n} o'_i v_i = \sum_{i=2}^{n} o_i v_i = C_2$ şi $\sum_{i=2}^{n} o'_i < \sum_{i=2}^{n} o_i$. Atunci $O' = (o_1, o'_2, \ldots, o'_n)$ verifică $o_1 v_1 + \sum_{i=2}^{n} o'_i v_i = C$ şi $o_1 + \sum_{i=2}^{n} o'_i < o_1 + \sum_{i=2}^{n} o_i$, adică $O'$ este mai bună decât $O$, ceea ce contrazice ipoteza că $O$ este optimă.

### 7.2.2 Analiza complexităţii

Strategia greedy conduce în general la algoritmi eficienţi. Din structura generală (`greedy1`) se observă că prelucrarea care determină ordinul de complexitate este cea de selecţie. Dacă aceasta este simplă se pot obţine chiar algoritmi de complexitate liniară. În general însă aplicarea strategiei necesită o sortare prealabilă a elementelor lui $A$ astfel că se ajunge la complexitate de tipul $O(n^2)$ sau $O(n \lg n)$ (dacă se foloseşte un algoritm eficient de sortare).

În cazul algoritmului `submulţime` dacă $m$ este mic în raport cu $n$ atunci nu este justificat să se ordoneze întregul set $A$ ci să se alegă la fiecare etapă valoarea maximă din cele ce nu au fost încă selectate. Se ajunge astfel la un ordin de complexitate $O(mn)$. În algoritmul `monede` sortarea domină celelalte prelucrări astfel că aceasta impune ordinul de complexitate ($\mathcal{O}(n^2)$ sau $\mathcal{O}(n \lg n)$).

## 7.3 Aplicaţii

### 7.3.1 Varianta continuă a problemei rucsacului

Este o problemă clasică ce are multe aplicaţii şi diverse variante. Se consideră un set $A$ de obiecte caracterizate fiecare prin profitul ($p$) şi dimensiunea lor ($d$): $A = ((p_1, d_1), \ldots, (p_n, d_n))$. Se mai consideră un rucsac de capacitate maximă $C$ şi se pune problema selectării unui subset de obiecte, $((p_{i_1}, d_{i_1}), \ldots, (p_{i_k}, d_{i_k}))$ care să nu depăşească capacitatea rucsacului ($\sum_{j=1}^{k} d_{i_j} \leq C$) şi să asigure un profit maxim ($\sum_{j=1}^{k} p_{i_j}$ este maximă). Două dintre cele mai cunoscute variante ale problemei sunt:

- *Varianta discretă (0-1).* Obiectele nu pot fi divizate: un obiect este fie preluat în întregime fie nu este preluat.
- *Varianta continuă (fracţionară).* Este posibil să fie transferate şi fracţiuni din obiecte, profitul asigurat fiind proporţional cu fracţiunea.

Doar pentru varianta fracţionară se poate obţine o soluţie optimă aplicând strategia greedy. În acest caz alegerea local optimală este cea care corespunde *profitului relativ* maxim (profitul relativ al obiectului $i$ este $p_i/d_i$). Aplicând strategia greedy se ajunge la a efectua următoarele prelucrări:

(i) se ordonează $A$ descrescător după profitul relativ;

(ii) se transferă obiectele în ordinea în care apar în $A$, în întregime cu excepţia ultimului obiect din care se ia doar o fracţiune, atât cât să se umple rucsacul.

Pentru a descrie algoritmul facem următoarele convenţii: $A[i].p$ şi $A[i].d$ reprezintă profitul respectiv dimensiunea obiectului $i$. Soluţia va fi de forma: $S = (S[1], \ldots, S[n])$ cu $S[i] \in [0, 1]$, iar elementele sale vor fi interpretate astfel: $S[i] = 0$ dacă obiectul $i$ nu este selectat, $S[i] = 1$ dacă obiectul $i$ este selectat în întregime, $S[i] \in (0, 1)$ - este selectată doar o fracţiune $s[i]$ din obiectul $i$. Cu aceste convenţii algoritmul este descris în 7.1.

Soluţia obţinută aplicând acest algoritm va fi de forma: $S = (1, 1, \ldots, 1, f, 0, \ldots, 0)$. În plus $\sum_{i=1}^{n} s_i d_i = C$, astfel că în continuare considerăm că restricţia problemei este de tip egalitate. Demonstrăm că `rucsac_fracţionar` generează o soluţie optimă arătând că orice soluţie optimă trebuie să respecte criteriul alegerii greedy.

Considerăm că obiectele sunt ordonate descrescător după valoarea profitului relativ: $p_1/d_1 > p_2/d_2 > \ldots > p_n/d_n$ (în cazul când inegalitatea nu este strictă alegerea de tip greedy nu este unică şi complică puţin demonstraţia). Considerăm $O = (o_1, o_2, \ldots, o_n)$ o soluţie optimă şi demonstrăm că este de tip greedy prin reducere la absurd. Presupunem că $O$ nu este de tip greedy şi considerăm că $O' = (o'_1, \ldots, o'_n)$ este generată aplicând tehnica greedy. Fie

**Algoritmul 7.1** Varianta fracţionară a problemei rucsacului

```
rucsac_fracţionar(A[1..n],C)
A[1..n] ← sortare_descrescătoare(A[1..n]) // sortare după profitul relativ
for i ← 1, n do
   S[i] ← 0
end for
i ← 1;
while C > 0 and i ≤ n do
   if S[i].d ≤ C then
      S[i] ← 1; C ← C − A[i].d
   else
      S[i] ← C/A[i].d; C ← 0
   end if
   i ← i + 1
end while
return S[1..n]
```

$B_+ = \{i | o'_i \geq o_i\}$, $B_- = \{i | o'_i < o_i\}$ iar $k = \min B_-$ (cel mai mic indice pentru care $o'_i < o_i$). Din structura unei soluţii de tip greedy rezultă că dacă $i \in B_+$ şi $j \in B_-$ atunci $i < j$. Din restricţia problemei rezultă $\sum_{i=1}^{n} o_i d_i = \sum_{i=1}^{n} o'_i d_i$ adică $\sum_{i \in B_+} (o'_i - o_i) d_i = \sum_{i \in B_-} (o_i - o'_i) d_i$. Calculăm profiturile corespunzătoare celor două soluţii: $P = \sum_{i=1}^{n} o_i p_i$ şi $P' = \sum_{i=1}^{n} o'_i p_i$. Diferenţa lor este:

$$P' - P = \sum_{i=1}^{n} (o'_i - o_i) p_i = \sum_{i \in B_+} (o'_i - o_i) d_i \frac{p_i}{d_i} - \sum_{i \in B_-} (o_i - o'_i) d_i \frac{p_i}{d_i}.$$

Se observă că $p_i/d_i > p_k/d_k$ pentru orice $i \in B_+$ iar $p_i/d_i \leq p_k/d_k$ pentru orice $i \in B_-$. Prin urmare

$$P' - P > \frac{p_k}{d_k} \sum_{i \in B_+} (o'_i - o_i) d_i - \frac{p_k}{d_k} \sum_{i \in B_-} (o_i - o'_i) d_i = 0$$

Deci $P' > P$ ceea ce contrazice ipoteza că $O$ este optimă. Deci $O$ are o structură de tip greedy. Proprietatea de substructură optimă rezultă imediat prin reducere la absurd.

*Observaţie.* Pentru a ilustra faptul că pentru varianta discretă această strategie nu conduce la soluţia optimă considerăm exemplul: $A = ((60, 10), (100, 20), (120, 30))$, $C = 50$. Profiturile relative sunt: 6, 5, 4, iar $A$ este deja ordonat descrescător după acest criteriu. Aplicând strategia greedy s-ar transfera primele două obiecte ducând la un profit egal cu 160. Dacă s-ar transfera al doilea şi al treilea obiect s-ar ajunge la un profit mai mare, 220.

Deşi nu conduce la soluţia optimă, strategia greedy poate fi aplicată şi pentru varianta discretă conducând la o soluţie sub-optimală (se poate demonstra că soluţia obţinută aplicând tehnica greedy satisface: $\sum_{i=1}^{k} s_i p_i \geq P_{opt} - \max_i p_i$, $P_{opt}$ fiind profitul corespunzător soluţiei optime). În schimb se poate obţine o soluţie optimă aplicând tehnica programării dinamice.

### 7.3.2 Problema selectării activităţilor

Se consideră un set de activităţi care au nevoie de o anumită resursă şi la un moment dat o singură activitate poate beneficia de resursa respectivă (de exemplu activitatea poate fi un examen, iar resursa o sală de examen). Presupunem că pentru fiecare activitate, $A_i$, se cunoaşte momentul de începere, $p_i$ şi cel de finalizare $t_i$ ($t_i > p_i$). Presupunem că activitatea se desfăşoară în intervalul $[p_i, t_i)$. Două activităţi $A_i$ şi $A_j$ se consideră *compatibile* dacă intervalele asociate sunt disjuncte. Se cere să se selecteze un număr cât mai mare de activităţi compatibile.

O soluţie a acestei probleme constă într-un subset de activităţi $S = (a_{i_1}, \ldots, a_{i_m})$ care satisfac $[p_{i_j}, t_{i_j}) \cap [p_{i_k}, t_{i_k}) = \emptyset$ pentru orice $j \neq k$.

Criteriul de selecţie ar putea fi: (i) cea mai mică durată; (ii) cel mai mic moment de începere; (iii) cel mai mic moment de sfârşit; (iv) intervalul care se intersectează cu cele mai puţine alte intervale. Dintre aceste variante cea pentru care se poate demonstra că asigură obţinerea soluţiei optime este a treia: la fiecare etapă se alege activitatea care se termină cel mai devreme.

Notând cu $A[i].p$, $A[i].t$ momentul de începere respectiv cel de finalizare a activităţii $A[i]$ algoritmul bazat pe strategia greedy este descris în 7.2.

**Algoritmul 7.2** Problema selectării activităţilor

```
selecţie_activităţi(A[1..n])
A[1..n] ← sortare_crescătoare(A[1..n]) // sortare după timpul de finalizare
S[1] ← A[1]
i ← 2; k ← 1
while i ≤ n do
  if S[k].t ≤ A[i].p then
    k ← k + 1; S[k] ← A[i];
  end if
  i ← i + 1
end while
return S[1..k]
```

Demonstrăm că algoritmul de mai sus conduce la soluţia optimă. După ordonarea crescătoare după timpul de finalizare avem $t_1 \leq t_2 \leq \ldots \leq t_n$. Considerăm o soluţie optimă $O = ((p_{i_1}, t_{i_1}), \ldots, (p_{i_m}, t_{i_m})$. Evident $t_{i_1} \geq t_1$ prin urmare înlocuind $A_{i_1}$ cu $A_1$ în $O$, se obţine o soluţie în care există acelaşi

număr de activităţi compatibile dar prima activitate este aleasă conform strategiei greedy. Deci problema satisface proprietate alegerii local optimale. O dată aleasă prima activitate, problema se reduce la planificarea optimă a activităţilor compatibile cu prima. Dacă $O$ este soluţia optimă a problemei iniţiale atunci $O' = ((p_{i_2}, t_{i_2}), \ldots, (p_{i_m}, t_{i_m})$ este soluţie optimă a subproblemei la care se ajunge după stabilirea primei activităţi. Astfel problema satisface şi proprietatea substructurii optime.

Să considerăm cazul $A = ((1,8),(2,5),(6,8),(5,6))$. Sortând $A$ în ordinea crescătoare a duratei activităţilor se obţine ca soluţie: $S = ((5,6),(6,8),(2,5))$. Prin selecţie după momentul de începere a activităţilor se obţine $((1,8))$, în schimb prin selecţie după momentul de finalizare se obţine o soluţie, $S = ((2,5),(5,6),(6,8))$, în care activităţile sunt deja ordonate în ordinea în care vor fi efectuate.

### 7.3.3 Problema planificării activităţilor

Se consideră un set de $n$ prelucrări ce trebuie executate de către un procesor. Durata execuţiei prelucrărilor este aceeaşi (se consideră egală cu 1). Fiecare prelucrare $i$ are asociat un termen final de execuţie, $t_i \leq n+1$ şi un profit, $p_i$. Profitul unei prelucrări intervine în calculul profitului total doar dacă prelucrarea este executată (dacă prelucrarea nu poate fi planificată înainte de termenul final de execuţie profitul este nul). Se cere să se planifice activităţile astfel încât să fie maximizat profitul total (acesta este corelat cu numărul de prelucrări planificate).

O soluţie constă în stabilirea unui "orar" de execuţie a prelucrărilor $S = (s_1, s_2, \ldots, s_n)$, $s_i \in \{1, \ldots, n\}$ fiind indicele prelucrării planificate la momentul $i$. Pentru rezolvarea problemei se poate aplica o tehnică de tip greedy caracterizată prin:

- se sortează activităţile în ordinea descrescătoare a profitului;
- fiecare activitate se planifică într-un interval liber cât mai apropiat de termenul final de execuţie.

Algoritmul este descris în 7.3. Se observă că dacă pentru fiecare $i \in \{1, \ldots, n\}$ numărul activităţilor care au termenul final cel mult $i$ este cel mult egal cu $i$ ($\text{card}\{j | t_j \leq i\} \leq i-1$) atunci toate activităţile vor fi planificate, altfel vor exista activităţi ce nu pot fi planificate.

Să considerăm $n = 4$ activităţi având termenele finale: $(2,3,4,2)$ şi profiturile corespunzătoare $(4,3,2,1)$. Atunci aplicând tehnica greedy se obţine soluţia $(4,1,2,3)$. Dacă însă termenele de execuţie sunt: $(2,4,1,2)$ şi aceleaşi profituri se obţine soluţia $(3,1,0,2)$ iar activitatea 4 nu este planificată.

**Algoritmul 7.3** Problema planificării activităţilor

```
planificare(A[1..n])
A[1..n] ← sortare_descrescătoare(A[1..n]) // sortare după profit (A[i].p)
for i ← 1, n do
  S[i] ← 0
end for
for i ← 1, n do
  poz ← A[i].t − 1 // termenul final de execuţie
  while S[poz] ≠ 0 and poz > 1 do
    poz ← poz − 1
  end while
  if S[poz] = 0 then
    S[poz] ← i // se planifica activitatea
  else
    "activitatea i nu poate fi planificată"
  end if
end for
return S[1..n]
```

### 7.3.4 Problema împachetării

Se consideră un set de numere $a_1, a_2, \ldots, a_n$ cu proprietatea că $a_i \in (0, C]$. Se cere să se grupeze în cât mai puţine subseturi ($k$) astfel încât suma elementelor din fiecare subset să nu depăşească valoarea $C$. Este cunoscută şi sub numele de "bin-packing problem". Există mai multe variante ce folosesc principiul alegerii greedy însă toate sunt *sub-optimale* (nu garantează obţinerea optimului ci doar a unei soluţii suficient de apropiate de cea optimă).

*Varianta 1.* Se construiesc succesiv submulţimile: se transferă (în ordinea în care se află în set) în primul subset atâtea elemente cât este posibil, din elementele rămase se transferă elemente în al doilea subset etc. (nu e garantată optimalitatea soluţiei însă s-a demonstrat că $k < 2k_{opt}$, $k$ fiind numărul de subseturi generat prin strategia greedy de mai sus, iar $k_{opt}$ este numărul optim de subseturi).

*Varianta 2.* Se iniţializează $n$ subseturi vide. Se parcurge setul de numere şi fiecare număr se transferă în primul subset în care "încape". Numărul de subseturi nevide astfel constituite $k$, are proprietatea $k < 1.7k_{opt}$.

*Varianta 3.* Dacă se ordonează descrescător setul iniţial şi se aplică varianta 2 se obţine o îmbunătăţire: $k < 11/9k_{opt} + 4$.

Considerăm că soluţia va fi reprezentată printr-o matrice $R[1..n, 1..n]$ în care linia $i$ corespunde subsetului $i$ iar $R[i, j]$ conţine fie 0 fie indicele unui element din $a$ care trebuie plasat în subsetul $i$. Pentru a controla modul de completare a subseturilor se pot folosi tablourile $K[1..n]$ ($K[i]$ specifică indicele ultimului element completat în linia $i$) şi $S[1..n]$ care conţine suma curentă a

elementelor de pe linia $i$. Algoritmul este descris în 7.4.

**Algoritmul 7.4** Problema împachetării

```
impachetare(integer a[1..n])
R[1..n, 1..n] ← 0; K[1..n] ← 0; S[1..n] ← 0;
a[1..n] ←sortare_descrescatoare(a[1..n])
for i ← 1, n do
   j ← 1
   while S[j] + a[i] > C do
      j ← j + 1
   end while
   K[j] ← K[j] + 1; R[j, K[j]] ← a[i]; S[j] ← S[j] + a[i]
end for
return R[1..n, 1..n]
```

## 7.4 Probleme

**Problema 7.1** Se consideră un rucsac de capacitate $C$ şi un set de obiecte având toate aceeaşi dimensiune, $d$, însă valori diferite. Să se determine un subset de obiecte care să încapă în rucsac şi a căror valoare totală să fie maximă.

*Indicaţie.* Se selectează $C/d$ obiecte în ordinea descrescătoare a valorilor.

**Problema 7.2** Se consideră un rucsac de capacitate $C$ şi un set de obiecte având toate aceeaşi valoare, $v$, însă dimensiuni diferite. Să se determine un subset de obiecte care să încapă în rucsac şi a căror valoare totală să fie maximă.

*Indicaţie.* Se selectează obiecte în ordinea crescătoare a dimensiunilor până se depăşeşte capacitatea rucsacului (ultimul obiect selectat este ignorat).

**Problema 7.3** (*Stocare optimală pe suport cu acces secvenţial*) Se consideră un set de $n$ fişiere de dimensiuni $d_1$,$d_2$,...,$d_n$. Se pune problema stocării acestora pe un suport cu acces secvenţial astfel încât timpul total de regăsire a fişierelor să fie cât mai mic. Procesul de regăsire satisface următoarele restricţii: (i) fişierele trebuie regăsite în ordinea inversă a stocării lor; (ii) timpul necesar regăsirii fişierului $k$ este proporţional cu suma dimensiunilor tuturor fişierelor stocate înainte de acest fişier la care se adaugă dimensiunea fişierului ($d_k$).

*Indicaţie.* Să considerăm că permutarea $(p(1), p(2), \ldots, p(n))$ indică ordinea în care sunt stocate fişierele. Atunci timpul necesar accesării fişierului $k$ este $\tau(k) = d_{p(1)} + \ldots + d_{p(k)}$. Prin urmare timpul de accesare al tuturor fişierelor este:

$$T(n) = \sum_{k=1}^{n} \tau(k) = nd_{p(1)} + (n-1)d_{p(2)} + \ldots + 2d_{p(k-1)} + d_{p(k)}$$

Se observă că $T(n)$ este cu atât mai mic cu cât fişierele de dimensiuni mai mici sunt stocate mai la început. Suma este minimă când $(d_{p(1)}, d_{p(2)}, \ldots, d_{p(n)})$ este ordonat crescător.

**Problema 7.4** Fie $\{x_1, x_2, \ldots, x_n\}$ o mulţime de numere reale. Să se determine un set minimal de intervale închise de lungime 1 care conţine toate elementele mulţimii.

*Indicaţie.* Se ordonează crescător elementele mulţimii. Se stabileşte ca prim interval cel care începe în $x_1$. Al doilea interval va începe în $x_i$ cu proprietatea că $x_i$ este cel mai mic element din mulţimea ordonată care satisface: $x_i > x_1+1$. Procedeul se aplică în continuare până se găsesc intervale care acoperă toate elementele mulţimii.

**Problema 7.5** Fie $\{x_1, x_2, \ldots, x_n\}$ o mulţime de numere reale şi $I$ o mulţime de intervale. Să se determine o submulţime de intervale astfel că oricare dintre numerele $x_i$ să fie inclus cel puţin într-un interval şi numărul de intervale să fie cât mai mic.

**Problema 7.6** (*Acoperirea unei mulţimi*) Se consideră $S_1$,$S_2$, ..., $S_m$ submulţimi ale mulţimii $X = \{1, 2, \ldots, n\}$. Să se selecteze dintre cele $m$ mulţimi un număr cât mai mic care au proprietatea că reuniunea lor coincide cu $X$.

*Indicaţie.* Se selectează prima dată mulţimea cu cele mai multe elemente şi se marchează în $X$ elementele existente în mulţimea selectată. In continuare, la fiecare etapă, se selectează dintre mulţimile neselectate încă pe cea care acoperă cele mai multe dintre elementele nemarcate. Strategia nu este optimală însă numărul de submulţimi selectate este de cel mult $\ln m$ ori mai mare decât cel optim.

**Problema 7.7** Se consideră $S_1$,$S_2$, ..., $S_m$ submulţimi ale mulţimii $X = \{1, 2, \ldots, n\}$. Să se selecteze dintre cele $m$ mulţimi un număr cât mai mare de submulţimi disjuncte cu proprietatea că reuniunea lor coincide cu $X$.

*Indicaţie.* Se selectează submulţimea cu cele mai puţine elemente şi se marchează toate submulţimile din set cu care are intersecţie nenulă. În continuare se selectează dintre submulţimile nemarcate cea cu cele mai puţine elemente, se marchează submulţimile cu care se intersectează ş.a.m.d. Procesul se opreşte când nu mai există submulţimi nemarcate. Ca şi în cazul problemei precedente strategia greedy este sub-optimală.

**Problema 7.8** Fie $A$ o mulţime de valori naturale. Să se descompună mulţimea $A$ în două submulţimi $B$ şi $C$ astfel încât $A = B \cup C$, $B \cap C = \emptyset$ şi suma elementelor din $B$ este cât mai apropiată de suma elementelor din $C$.

*Rezolvare.* Se ordonează descrescător elementele lui $A$. Se transferă primul element din $A$ în $B$ după care se transferă elemente în $C$ până când suma

elementelor din $C$ devine cel puţin egală cu suma elementelor din $B$. Procesul de transfer continuă până când se epuizează elementele din $A$:

```
descompunere(integer a[1..n])
integer k1, k2, b[1..k1], c[1..k2], S1, S2, i
k1 ← 1; b[k1] ← a[1]; S1 ← a[1]; k2 ← 0; S2 ← 0
a[1..n] ← sortare(a[1..n])
for i ← 2, n
    if S1 < S2 then k1 ← k1 + 1; b[k1] ← a[i]; S1 ← S1 + a[i];
    else k2 ← k2 + 1; c[k2] ← a[i]; S2 ← S2 + a[i];endif
return b[1..k1], c[1..k2]
```

În acest caz tehnica greedy folosită mai sus nu conduce întotdeauna la o soluţie optimă ci doar la una suboptimală. De exemplu aplicând strategia propusă setului de valori $\{10, 8, 7, 5, 4, 2\}$ se obţine descompunerea în $\{10, 5, 4\}$ respectiv $\{8, 7, 2\}$ pe când o soluţie optimă ar fi $\{10, 8\}$ şi $\{7, 5, 4, 2\}$.

**Problema 7.9** Fie $X = \{x_1, \ldots, x_n\}$ şi $Y = \{y_1, \ldots, y_n\}$ două mulţimi. Să se împerecheze elementele din cele două mulţimi (elementul $x_i$ din $X$ se împerechează cu elementul $y_{p(i)}$ din $Y$, $p$ fiind o permutare de ordin $n$) astfel încât suma $\sum_{i=1}^{n} |x_i - y_{p(i)}|$ să fie minimă.

*Indicaţie.* Se ordonează crescător $X$ şi $Y$ şi se împerechează elementele de pe poziţiile corespunzătoare (cel mai mic element din $X$ cu cel mai mic element din $Y$, al doilea element din mulţimea $X$ ordonată cu al doilea element din mulţimea $Y$ ordonată ş.a.m.d).

# Capitolul 8

# Tehnica programării dinamice

Programarea dinamică (introdusă în 1950 de R. Bellman pentru rezolvarea problemelor de optimizare) este o metodă bazată pe construirea şi utilizarea unor tabele cu informaţii. La construirea tabelelor pentru completarea unui element se folosesc elemente completate anterior (construirea se realizează în manieră dinamică). Această tehnică se bazează pe descompunerea unei probleme în subprobleme şi este adecvată rezolvării problemelor (în particular celor de optimizare) care au următoarele proprietăţi:

- *Proprietatea de substructură optimă.* Orice soluţie optimă este constituită din soluţii optime ale subproblemelor. Această proprietatea este specifică şi problemelor ce pot fi rezolvate prin tehnica greedy. Pentru a verifica dacă o problemă posedă această proprietate se poate folosi metoda reducerii la absurd.
- *Proprietatea de suprapunere a problemelor.* Pentru ca programarea dinamică să fie eficientă este necesar ca numărul subproblemelor ce trebuie efectiv rezolvate în scopul obţinerii soluţiei problemei iniţiale să fie relativ mic (polinomial în raport cu dimensiunea problemei). Aceasta înseamnă că în procesul de descompunere a problemei se ajunge de mai multe ori la aceeaşi subproblemă care însă va fi rezolvată o singură dată iar soluţia ei va fi reţinută într-un tabel. Ideea descompunerii problemei în subprobleme este folosită şi în metoda divizării însă în acel caz era important ca subproblemele să fie independente.

## 8.1 Principiul tehnicii

La aplicarea tehnicii programării dinamice se parcurg următoarele etape:

(a) Analiza structurii unei soluţii optime şi a proprietăţilor acesteia prin punerea în evidenţă a relaţiei existente între soluţia problemei şi soluţiile subproblemelor (se verifică dacă problema posedă proprietatea de substructură optimă).

(b) Stabilirea unei relaţii de recurenţă referitoare la criteriul de optimizat sau la valoarea ce trebuie calculată. Aceasta descrie legătura dintre valoarea criteriului de optim corespunzător problemei şi cele corespunzătoare subproblemelor.

(c) Calculul valorii asociate soluţiei optime dezvoltând relaţia de recurenţă într-o manieră *ascendentă* şi reţinând valorile calculate asociate subproblemelor într-o structură tabelară. În funcţie de problemă, în această etapă se pot reţine şi alte informaţii care vor fi utilizate în momentul construirii soluţiei.

(d) Construirea unei soluţii optime folosind informaţiile determinate şi reţinute la etapa anterioară.

Deşi se bazează pe descompunerea unei probleme în subprobleme la fel ca tehnica divizării, specificul programării dinamice este că subproblemele nu sunt independente (ci se suprapun). În schimb tehnica divizării conduce la algoritmi eficienţi doar dacă subproblemele sunt independente.

**Dezvoltarea ascendentă şi descendentă a unei relaţii de recurenţă.** Suprapunerea problemelor face ca obţinerea valorii prin dezvoltarea relaţiei de recurenţă să nu fie eficientă dacă este abordată în manieră descendentă ("top-down") prin implementarea recursivă a relaţiei. Aceasta deoarece o aceeaşi valoare poate fi utilizată de mai multe ori şi, de fiecare dată când este utilizată, este recalculată.

Să considerăm problema determinării celui de-al $m$-lea element din şirul lui Fibonacci ($f_1 = f_2 = 1$, $f_n = f_{n-1} + f_{n-2}$, $n \geq 3$). O abordare descendentă conduce la algoritmul recursiv:

```
fib_rec(m)
if (m = 1) or (m = 2) then
  return 1
else
  return fib_rec(m − 1)+fib_rec(m − 2)
end if
```

Numărul de adunări efectuate pentru a determina pe $f_m$, $T(m)$ verifică relaţiile: $T(1) = T(2) = 0$, $T(m) = T(m-1) + T(m-2) + 1$. Prin urmare $T(m) \geq f_{m-1}$, pentru $m \geq 5$. Cum $f_m \in \Theta(\phi^m)$ cu $\phi = (1+\sqrt{5})/2$ rezultă că `fib_rec` are complexitate exponenţială.

Considerăm acum varianta generării lui $f_m$ in manieră ascendentă: se calculează si se *reţin* valorile lui $f_1, f_2, \ldots f_m$. În acest caz algoritmul poate fi descris prin:

```
fib_asc(m)
f[1] ← 1; f[2] ← 1
for i ← 3, m do
   f[i] = f[i − 1] + f[i − 2]
end for
return f[m]
```

Numărul de adunări efectuate este de această dată $T(m) = m - 2$ (complexitate liniară). Această variantă de implementare necesită însă utilizarea unui spaţiu auxiliar de memorie de dimensiune $m$. Algoritmul poate fi uşor transformat astfel încât să nu folosească decât trei variabile pentru reţinerea elementelor şirului dar să rămână cu complexitate liniară:

```
fib_asc2(m)
f1 ← 1; f2 ← 1
for i ← 3, m do
   f3 ← f1 + f2; f1 ← f2; f2 ← f3
end for
return f3
```

Numărul variabilelor de lucru poate fi redus chiar la două în modul următor:

```
fib_asc3(m)
f1 ← 1; f2 ← 1
for i ← 3, m do
   f2 ← f1 + f2; f1 ← f2 − f1;
end for
return f2
```

Un exemplu similar este cel al calculului combinărilor, $C_n^k$ $(n \geq k)$, pornind de la relaţia de recurenţă:

$$C_n^k = \begin{cases} 1 & \text{dacă } k = 0 \text{ sau } n = k \\ C_{n-1}^k + C_{n-1}^{k-1} & \text{altfel} \end{cases} \tag{8.1}$$

Algoritmul în varianta recursivă este descris în continuare.

```
comb_rec(n, k)
if (k = 0) or (n = k) then
  return 1
else
  return comb_rec(n − 1, k)+comb_rec(n − 1, k − 1)
end if
```

Algoritmul `comb_rec` are complexitate exponenţială întrucât utilizând arborele de apel se poate demonstra că $T(n,k) \geq 2^{\min\{n-k,k\}}$.

În abordarea ascendentă ideea este să se reţină toate valorile $C_i^j$ calculate pentru $0 \leq j \leq i \leq n$. Aceasta s-ar putea realiza prin completarea elementelor $a_{ij}$ din zona inferior triunghiulară a unei matrici $n \times n$. De fapt este suficient să se completeze până la coloana $k$. Dacă $n = k$ zona triunghiulară astfel

completată este cunoscută sub denumirea de *triunghiul lui Pascal*. Algoritmul este descris în 8.1.

**Algoritmul 8.1** Calculul combinărilor folosind triunghiul lui Pascal

```
comb_asc(n, k)
for i ← 0, n do
  for j ← 0, min{i, k} do
    if (i = j) or (j = 0) then
      a[i, j] ← 1
    else
      a[i, j] ← a[i − 1, j] + a[i − 1, j − 1]
    end if
  end for
end for
return a[n, k]
```

Numărul de adunări efectuate, $T(n,k)$ satisface

$$T(n,k) = \sum_{i=1}^{k}\sum_{j=1}^{i-1} 1 + \sum_{i=k+1}^{n}\sum_{j=1}^{k} 1 = \sum_{i=1}^{k}(i-1) + \sum_{i=k+1}^{n} k = \frac{k(k-1)}{2} + k(n-k)$$

deci $T(n,k) \in \Theta(nk)$. Se observă că zona auxiliară utilizată poate fi redusă, întrucât pentru completarea liniei $i$ se folosesc doar valorile din linia $i-1$.

**Exemplul 8.1** (*Problema determinării celui mai lung subşir strict crescător*) Se consideră un şir de valori reale $a_1, a_2, \ldots, a_n$ şi se caută un subşir $a_{j_1}, a_{j_2}, \ldots, a_{j_k}$ cu $1 \le j_1 < j_2 < \ldots < j_k \le n$, $a_{j_1} < a_{j_2} < \ldots < a_{j_k}$ şi astfel încât $k$ să fie cât mai mare. Un astfel de subşir nu este neapărat unic. De exemplu pentru şirul $(2,5,1,3,6,8,2,10)$ există două subşiruri strict crescătoare de lungime 5: $(2,3,6,8,10)$ şi $(1,3,6,8,10)$. Pe de altă parte nu toate subşirurile de aceeaşi lungime se termină cu acelaşi element. De exemplu în şirul $(2,1,4,3)$ există patru subşiruri crescătoare de lungime maximă (2): $(2,4)$, $(2,3)$, $(1,4)$, $(1,3)$.

a) *Caracterizarea soluţiei optime.* Fie $a_{j_1} < a_{j_2} < \ldots < a_{j_{k-1}} < a_{j_k}$ un subşir de lungime maximă. Considerăm că $a_{j_k} = a_i$ iar $a_{j_{k-1}} = a_p$ (evident $p < i$). Prin reducere la absurd se poate arăta că $a_{j_1} < a_{j_2} < \ldots < a_{j_{k-1}}$ este cel mai lung dintre subşirurile crescătoare ale şirului parţial $(a_1, a_2, \ldots, a_p)$ care au proprietatea că ultimul element este chiar $a_p$. Prin urmare problema are proprietatea de substructură optimă şi lungimea unui subşir crescător care se termină în $a_i$ poate fi exprimată în funcţie de lungimea subşirurilor crescătoare care se termină cu elemente $a_p$ ce satisfac $a_p < a_i$.

b) *Stabilirea unei relaţii de recurenţă între lungimile subşirurilor.* Fie $(B_i)_{i=\overline{1,n}}$ un tablou ce conţine pe poziţia $i$ numărul de elemente al celui mai lung subşir strict crescător al şirului $(a_1, \ldots a_n)$ care are pe $a_i$ ca ultim element (spunem

că subşirul se termină în $a_i$). Ţinând cont de proprietăţile soluţiei optime se poate stabili o relaţie de recurenţă pentru $B_i$:

$$B_i = \begin{cases} 1 & i = 1 \\ 1 + \max\{B_j | 1 \leq j < i \text{ şi } a_j < a_i\} & i > 1 \end{cases}$$

Dacă mulţimea $M_i = \{B_j | 1 \leq j < i \text{ şi } a_j < a_i\}$ este vidă atunci $\max M_i = 0$. De exemplu pentru şirul iniţial $a = (2, 5, 1, 3, 6, 8, 4)$ tabloul lungimilor subşirurilor optime care se termină în fiecare element al lui $a$ este: $B = (1, 2, 1, 2, 3, 4, 3)$. Cea mai mare valoare din $B$ indică numărul de elemente din cel mai lung subşir crescător.

c) *Dezvoltarea relaţiei de recurenţă în manieră ascendentă.* Se completează un tablou cu elementele lui $(B_i)_{i=\overline{1,n}}$ aşa cum este descris în Algoritmul 8.2.

**Algoritmul 8.2** Dezvoltarea relaţiei de recurenţă pentru determinarea celui mai lung subşir crescător

```
completare(a[1..n])
B[1] ← 1
for i ← 2, n do
  max ← 0
  for j ← 1, i − 1 do
    if (a[j] < a[i]) and (max < B[j]) then
      max ← B[j]
    end if
  end for
  B[i] ← max + 1
end for
return B[1..n]
```

.

Luând în considerare doar comparaţia dintre două elemente ale şirului ($a[j] < a[i]$) costul algoritmului este $T(n) = \sum_{i=2}^{n} \sum_{j=1}^{i-1} 1 = \sum_{i=2}^{n} (i-1) = n(n-1)/2$ deci completarea tabelului $B$ este de complexitate $\Theta(n^2)$.

d) *Construirea unei soluţii optime.* Se determină cea mai mare valoare din $B$. Aceasta indică numărul de elemente ale celui mai lung subşir crescător iar poziţia pe care se află indică elementul, $a[m]$, din şirul iniţial cu care se termină subşirul. Pornind de la acest element subşirul se construieşte în ordinea inversă a elementelor sale căutând la fiecare etapă un element din şir mai mic decât ultimul completat în subşir şi căruia îi corespunde în $B$ o valoare cu 1 mai mică decât cea curentă. Algoritmul este descris în 8.3.

Dacă după completarea elementului $a[m]$ în subşir există două subşiruri din $a[1..m-1]$ de lungime maximă egală cu $B[m] - 1$: unul care se termină în $a[p]$ şi unul care se termină în $a[q]$ (astfel încât $a[p] < a[m]$ şi $a[q] < a[m]$),

**Algoritmul 8.3** Construirea unui cel mai lung subşir crescător

```
construire(a[1..n], B[1..n])
// determinarea valorii şi poziţiei maximului în B
imax ← 1
for i ← 2, n do
  if B[imax] < B[i] then
    imax ← i
  end if
end for
m ← imax
k ← B[m]
x[k] ← a[m]
while B[m] > 1 do
  i ← m − 1
  while (a[i] ≥ a[m]) or (B[i] ≠ B[m] − 1) do
    i ← i − 1
  end while
  m ← i
  k ← k − 1 // se adaugă un nou element în subşir
  x[k] ← a[m]
end while
return x[1..k]
```

.

algoritmul va selecta ca element pentru subşir pe cel cu indicele mai apropiat de $m$, adică mai mare. Astfel dacă $p < q$ va fi selectat $q$. Aplicând algoritmul de construire şirului $(2, 5, 1, 3, 6, 8, 4)$ se va porni cu $m = 6$ ($B[m] = 4$) ultimul element din subşir fiind astfel 8. Cum $B[5] = 3$ şi $6 < 8$ penultimul element va fi 6. Continuând procesul se selectează în continuare elementele 3 şi 1 obţinându-se subşirul crescător $(1, 3, 6, 8)$ de lungime 4.

Întrucât în ciclul de construire căutarea continuă de la poziţia noului element selectat, chiar dacă sunt două cicluri **while** suprapuse ordinul de complexitate este $\mathcal{O}(n)$. Cum şi determinarea maximului lui $B$ are acelaşi ordin de complexitate rezultă că algoritmul de construire are complexitate liniară.

## 8.2 Aplicaţii

### 8.2.1 Înmulţirea optimală a unui şir de matrici

Fie $A_1$, $A_2$, ..., $A_n$ un şir de matrici având dimensiuni compatibile pentru a putea calcula produsul: $A_1 \cdot A_2 \cdots A_n$. Să presupunem că aceste dimensiuni sunt: $(p_0, p_1, \ldots, p_n)$, adică matricea $A_i$ are $p_{i-1}$ linii şi $p_i$ coloane. Pentru

a calcula produsul $A = A_1 \cdot A_2 \cdots A_n$ trebuie stabilit un mod de grupare a factorilor astfel încât să se pună în evidenţă ordinea în care vor fi efectuate înmulţirile, la fiecare etapă înmulţindu-se două matrici. Considerăm că produsul a două matrici se efectuează după metoda clasică astfel că la produsul $A_i \cdot A_{i+1}$ se efectuează $p_{i-1}p_ip_{i+1}$ înmulţiri scalare.

Modul de grupare a factorilor (plasarea parantezelor pentru a indica ordinea de efectuare a înmulţirilor) nu influenţează rezultatul final (înmulţirea este asociativă) însă poate influenţa numărul de operaţii efectuate.

Să considerăm cazul a trei matrici $A_1$, $A_2$ şi $A_3$ având dimensiunile $(2, 20)$, $(20, 5)$ şi $(5, 10)$. În acest caz există două modalităţi de plasare a parantezelor:

1. $A_1 \cdot A_2 \cdot A_3 = (A_1 \cdot A_2) \cdot A_3$. În acest caz se efectuează $2 \cdot 20 \cdot 5 + 2 \cdot 5 \cdot 10 = 300$ înmulţiri scalare.

2. $A_1 \cdot A_2 \cdot A_3 = A_1 \cdot (A_2 \cdot A_3)$. În acest caz se efectuează $20 \cdot 5 \cdot 10 + 2 \cdot 20 \cdot 10 = 1400$ înmulţiri scalare.

Pentru $n$ mare, numărul de *parantezări* (variante de plasare a parantezelor) poate fi foarte mare astfel că este exclusă generarea tuturor parantezărilor posibile şi alegerea celei mai bune. Pentru calculul produsului $A_1 \cdot A_2 \cdots A_n$ notând cu $K(n)$ numărul de parantezări posibile obţinem:

$$K(n) = \begin{cases} 1 & n = 1 \\ \sum_{i=1}^{n-1}(K(i)K(n-i)) & n > 1 \end{cases} \tag{8.2}$$

întrucât ultimul nivel de paranteze poate fi aplicat în oricare dintre poziţiile $i \in \{1, 2, \ldots, n-1\}$ şi fiecare dintre cele $K(i)$ parantezări ale produsului $A_1 \cdots A_i$ poate fi combinată cu fiecare dintre cele $K(n-i)$ parantezări ale produsului $A_{i+1} \cdots A_n$. Se poate demonstra prin inducţie matematică că $K(n) = \dfrac{C_{2(n-1)}^{n-1}}{n}$ şi $K(n) \in \Omega(4^{n-1}/(n-1)^{3/2})$.

Problema înmulţirii optimale a unui şir de matrici (un caz particular al problemelor de planificare optimală a prelucrărilor) constă în a determina parantezarea (ordinea de efectuare a înmulţirilor) care conduce la un număr minim de înmulţiri scalare.

Aplicăm pentru rezolvarea problemei etapele specifice programării dinamice.

a) *Caracterizarea soluţiei optime.* Pentru a specifica produsele parţiale introducem notaţia $A_{i..j} = A_i \cdot A_{i+1} \cdots A_j$. Fie o soluţie optimă caracterizată prin faptul că parantezele cele mai exterioare sunt plasate pe poziţiile 1, $k$ şi $n$, adică ultima înmulţire efectuată este: $A_{1..k} \cdot A_{k+1..n}$. Atunci parantezările asociate lui $A_{1..k}$ şi $A_{k+1..n}$ trebuie să fie şi ele optime (în caz contrar ar există o parantezare mai bună pentru $A_{1..n}$).

b) *Stabilirea unei relaţii de recurenţă pentru numărul de înmulţiri scalare.* Fie $c_{ij}$ numărul de înmulţiri scalare efectuate pentru calculul lui $A_{i..j}$. Valorile

$c_{ij}$ au sens doar pentru $i \leq j$ iar relaţia de recurenţă dedusă din proprietatea de substructură optimă este:

$$c_{ij} = \begin{cases} 0 & i = j \\ \min_{i \leq k < j}(c_{ik} + c_{k+1,j} + p_{i-1}p_k p_j) & i < j \end{cases}$$

Relaţia se bazează pe faptul că dintre toate parantezările posibile ale lui $A_{i..j}$ se alege cea mai bună (de cost minim).

c) *Dezvoltarea relaţiei de recurenţă.* Valorile $c_{i,j}$ pot fi reţinute în porţiunea superior triunghiulară a unei matrici. Construirea în manieră recursivă a matricii este ineficientă. Construirea în manieră ascendentă se bazează pe ideea completării matricii astfel încât în momentul calculului lui $c_{ij}$ toate elementele $c_{ik}$ şi $c_{k+1,j}$ pentru $k \in \{i, i+1, \ldots, j-1\}$ să fie deja completate. Din acest motiv elementele se completează în ordinea crescătoare a diferenţei $j-i$: prima dată se completează elementele diagonalei principale ($i = j$), apoi se completează elementele aflate imediat deasupra diagonalei principale ($j = i+1$) ş.a.m.d. (Algoritmul 8.4).

**Algoritmul 8.4** Dezvoltarea relaţiei de recurenţă în cazul înmulţirii optimale a unui şir de matrici

```
completare(p[0..n])
for i ← 1, n do
  c[i, i] = 0
end for
for l ← 1, n − 1 do
  for i ← 1, n − l do
    j ← i + l
    c[i, j] ← c[i, i] + c[i + 1, j] + p_{i−1} p_i p_j
    s[i, j] ← i
    for k ← i + 1, j − 1 do
      cost = c[i, k] + c[k + 1, j] + p_{i−1} p_k p_j
      if cost < c[i, j] then
        c[i, j] ← cost;
      end if
      s[i, j] ← k
    end for
  end for
end for
return c[1..n, 1..n], s[1..n, 1..n]
```

O dată cu completarea matricii de costuri se reţine şi poziţia în care se descompune un produs în doi factori: $s[i, j] = k$ indică faptul că produsul $A_{i..j}$ se descompune în $A_{i..k} \cdot A_{k+1..j}$. Aceste poziţii sunt utile la construirea soluţiei. Cum pentru determinarea valorii fiecăruia dintre cele $n(n-1)/2$ e-

lemente este necesară determinarea unui minim dintr-un tablou cu cel mult $n$ elemente rezultă că ordinul de complexitate al algoritmului de completare este $\mathcal{O}(n^3)$.

Pentru exemplul celor trei matrici cu dimensiunile $(2, 20)$, $(20, 5)$ şi $(5, 10)$ se obţin matricile:

$$c = \begin{bmatrix} 0 & 200 & 300 \\ - & 0 & 1000 \\ - & - & 0 \end{bmatrix} \qquad s = \begin{bmatrix} 0 & 1 & 2 \\ - & 0 & 2 \\ - & - & 0 \end{bmatrix}$$

d) În funcţie de cerinţele problemei se poate determina: (i) numărul minim de operaţii; (ii) rezultatul înmulţirii matricilor; (iii) modul de descompunere a produsului (parantezare).

(i) Numărul minim de înmulţiri este chiar $c_{1n}$.

(ii) Calculul produsului poate fi efectuat în manieră recursivă în Algoritmul 8.5. În acest algoritm se presupus că pot fi accesate matricile din şir (matricea $A_i$ este elementul $i$, $A[i]$, al tabloului tridimensional ce contine toate matricile) precum şi elementele matricii $s$. Algoritmul `produs` calculează produsul a două matrici.

**Algoritmul 8.5** Înmulţirea optimală a unui şir de matrici

```
produs_optimal(i, j)
if i = j then
  return A[i]
else
  X ← produs_optimal(i, s[i, j])      // calcul A_{i..s[i,j]}
  Y ← produs_optimal(s[i, j] + 1, j)  // calcul A_{(s[i,j]+1)..j}
  R ← produs(X, Y)      // produsul a două matrici
end if
return R
```

(iii) Ordinea în care se efectuează înmulţirile poate fi determinată prin Algoritmul 8.6.

**Algoritmul 8.6** Determinarea poziţiilor în care trebuie plasate parantezele în cazul înmulţirii optimale a unui şir de matrici

```
parantezare(i, j)
if i < j then
  parantezare(i, s[i, j])
  write s[i, j]
  parantezare(s[i, j] + 1, j)
end if
```

### 8.2.2 Varianta discretă a problemei rucsacului

Considerăm varianta discretă a problemei rucsacului. Presupunem că obiectele se caracterizează prin dimensiunile $(d_1, d_2, \ldots, d_n)$ şi profiturile (valorile): $(p_1, p_2, \ldots, p_n)$ iar capacitatea maximă a rucsacului este $C$. Se pune problema selectării unui subset de obiecte, $((p_{i_1}, d_{i_1}), \ldots, (p_{i_k}, d_{i_k}))$ care să nu depăşească capacitatea rucsacului ($\sum_{j=1}^{k} d_{i_j} \leq C$) şi să asigure un profit maxim ($\sum_{j=1}^{k} p_{i_j}$ este maximă). Se consideră că dimensiunea fiecărui obiect este mai mică decât cea maximă ($d_i \leq C$).

Pentru a uşura aplicarea tehnicii programării dinamice vom considera că $(d_i)_{i=\overline{1,n}}$ şi $C$ sunt valori naturale. În varianta 0-1 a problemei un obiect poate fi selectat doar în întregime. În acest caz tehnica greedy nu garantează obţinerea soluţiei optimale.

a) *Analiza structurii unei soluţii optime.* Problema selectării dintre cele $n$ obiecte poate fi redusă la problema rucsacului corespunzătoare primelor $n-1$ obiecte şi a capacităţii maxime $1 \leq C_{n-1} \leq C$. O soluţie optimă a problemei iniţiale va fi constituită dintr-o soluţie optimă a subproblemei corespunzătoare primelor $n-1$ obiecte la care se adaugă eventual al $n$-lea obiect (dacă $C_{n-1} + d_n \leq C$). Dacă soluţia subproblemei nu ar fi optimă atunci ar putea fi înlocuită cu una mai bună ceea ce ar conduce la o soluţie mai bună a problemei iniţiale.

b) *Stabilirea unei relaţii de recurenţă.* Fie $V(i, j)$ valoarea obiectelor selectate în soluţia optimă a problemei corespunzătoare primelor $i$ obiecte şi capacităţii $j$. Cum soluţia optimă a problemei corespunzătoare lui $i$ poate fi exprimată prin soluţia optimă a problemei corespunzătoare lui $i-1$ rezultă următoarea relaţie de recurenţă pentru $V(i, j)$:

$$V(i,j) = \begin{cases} V(i-1,j) & \text{dacă } j - d_i < 0 \\ \max\{p_i + V(i-1, j-d_i), V(i-1,j)\} & \text{dacă } j - d_i \geq 0 \end{cases}$$

pentru $i = \overline{1,n}$, $j = \overline{1,C}$ iar $V(0,j) = 0$ pentru $j = \overline{0,C}$ şi $V(i,0) = 0$ pentru $i = \overline{1,n}$.

Cele două cazuri din relaţia de recurenţă provin din faptul că o soluţie optimă a problemei asociate primelor $i$ obiecte şi capacităţii $j$ poate să nu conţină sau să conţină obiectul $i$. În primul caz dimensiunea obiectului $i$ este prea mare pentru ca acesta să poată fi selectat. Astfel valoarea este identică cu cea corespunzătoare subproblemei primelor $i-1$ obiecte, $V(i-1, j)$.

În al doilea caz obiectul $i$ ar putea fi selectat însă el va fi selectat doar dacă conduce la o valoare totală mai mare decât dacă nu ar fi fost selectat. Din acest motiv se analizează ambele valori şi se alege cea maximă. Valoarea corespunzătoare cazului în care obiectul este selectat este obţinută adăugând valoarea celui de-al $i$-lea obiect ($p_i$) la valoarea soluţiei problemei corespunzătoare celor $i-1$ obiecte şi capacităţii $j-d_i$ (capacitatea rămasă disponibilă după selectarea obiectului $i$). $V(n, C)$ reprezintă valoarea maximă a unui set de obiecte a căror

dimensiuni însumate nu depăşeşte pe $C$ adică valoarea asociată soluţiei optime a problemei.

c) *Dezvoltarea relaţiei de recurenţă.* Dacă $C$ şi $(d_i)_{i=\overline{1,n}}$ sunt valori întregi atunci valorile lui $V(i,j)$ pot fi reţinute într-o matrice cu $n+1$ linii şi $C+1$ coloane. Algoritmul de construire a tabelului de valori este descris în 8.7.

**Algoritmul 8.7** Dezvoltarea relaţiei de recurenţă în cazul problemei rucsacului

```
tabel_valori(p[1..n],d[1..n],C)
for i ← 0, n do
  V[i, 0] ← 0
end for
for j ← 1, C do
  V[0, j] ← 0
end for
for i ← 1, n do
  for j ← 1, C do
    if j < d[i] then
      V[i, j] = V[i − 1, j]
    else
      V[i, j] = max(V[i − 1, j], p[i] + V[i − 1, j − d[i]])
    end if
  end for
end for
return V[0..n, 0..C]
```

Să considerăm cazul în care $n = 5$, $C = 5$, dimensiunile obiectelor sunt $(2, 4, 2, 1, 3)$ iar valorile asociate $(10, 20, 13, 15, 18)$. În acest caz matricea de valori este cu 6 linii şi 6 coloane (indiciate de la 0 la 5) şi conţine elementele:

$$V = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 10 & 10 & 10 & 10 \\ 0 & 0 & 10 & 10 & 20 & 20 \\ 0 & 0 & 13 & 13 & 23 & 23 \\ 0 & 15 & 15 & 28 & 28 & 38 \\ 0 & 15 & 15 & 28 & 33 & 38 \end{bmatrix}$$

Pornind de la elementul $V[5,5]$ se poate construi soluţia astfel. Se observă că $V[5,5] = V[4,5]$. Aceasta înseamnă că obiectul $o_5$ nu este selectat. În schimb $V[4,5] \neq V[3,5]$ adică obiectul $o_4$ este selectat. Aceasta reduce problema la selecţia dintre obiectele $\{o_1, o_2, o_3\}$ pentru un rucsac de capacitate $5 - d_4 = 4$. Valoarea unei soluţii optime a acestei subprobleme este $V[3,4] = 23$ (se observă că diferenţa $V[5,5] - V[3,4] = 38 - 23 = 15$ reprezintă chiar valoarea obiectului $o_4$). În continuare, se observă că $V[3,4] \neq V[2,4]$ adică obiectul $o_3$ este selectat.

Astfel problema se reduce la subproblema corespunzătoare setului $\{o_1, o_2\}$ şi capacităţii $4 - d_3 = 2$. Valoarea unei soluţii optime a acestei subprobleme este $V[2,2] = 10$. Cum $V[2,2] = V[1,2]$ şi $V[1,2] \neq 0$ rezultă că $o_2$ nu este selectat în schimb este selectat $o_1$. Astfel soluţia problemei iniţiale este $\{o_1, o_3, o_4\}$ cu dimensiunea totală 5 şi valoarea totală 38.

d) *Construirea soluţiei.* Pentru construirea soluţiei se analizează tabelul de valori construit la etapa anterioară iniţiind parcurgerea din $V[n, C]$ după strategia descrisă în exemplul de mai sus. Algoritmul este descris în 8.8.

**Algoritmul 8.8** Rezolvarea variantei discrete a problemei rucsacului

```
construire_soluţie(V[0..n, 0..C],d[1..n],C)
i ← n; j ← C
k ← 0
while j > 0 do
  while V[i, j] = V[i − 1, j] and i ≥ 1 do
    i ← i − 1
  end while
  k ← k + 1; s[k] = i
  j = j − d[i]
  i ← i − 1
end while
return s[1..k]
```

### 8.2.3 Tehnica funcţiilor de memorie

Principalul dezavantaj al abordării ascendente este faptul că se bazează pe completarea unui întreg tabel de valori. În aplicaţii e posibil ca anumite valori din tabel să nu fie necesare nici în calculul celorlalte valori şi nici în construcţia soluţiei (de exemplu elementele $V[5,1]$, $V[5,2]$, $V[5,3]$ şi $V[5,4]$ din tabelul construit pentru problema rucsacului).

Dacă se foloseşte abordarea top-down atunci se vor calcula doar valorile asociate subproblemelor care sunt necesare pentru a obţine valoarea corespunzătoare problemei iniţiale. Dar dacă o subproblemă este întâlnită de mai multe ori atunci ea este rezolvată de fiecare dată.

O soluţie de compromis care îmbină avantajele abordării ascendente cu a celei descendente este de a dezvolta relaţia de recurenţă într-o manieră recursivă reţinând însă fiecare dintre valorile calculate. În felul acesta dacă o valoare deja calculată este necesară din nou ea nu va fi recalculată ci se va folosi valoarea reţinută anterior. Această variantă hibridă este cunoscută ca *tehnica funcţiilor de memorie* sau *tehnica memoizării* (termenul în engleză este "memoization"). Aplicarea ei în practică constă în:

- iniţializarea tabelului cu o valoare *virtuală* ce va fi diferită de valorile ce

se vor obţine din calcule; în felul acesta se va putea verifica dacă o poziţie din tabel a fost calculată deja sau nu;

- calculul valorii corespunzătoare soluţiei în manieră recursivă (cu reţinerea valorilor calculate); aceasta presupune că tabelul de valori este o structură globală ce poate fi accesată la fiecare apel recursiv.

Aplicată la construirea tabelului de valori pentru problema rucsacului această tehnică constă în utilizarea algoritmilor `initializare` şi `calcul_valori` descrişi în continuare.

```
iniţializare(n,C)
for i ← 1, n do
  for j ← 1, C do
    V[i, j] ← −1 // iniţializare cu o valoare virtuală
  end for
end for
for j ← 0, C do
  V[0, j] ← 0
end for
for i ← 0, n do
  V[i, 0] ← 0
end for
return V[0..n, 0..C]
```

şi

```
calcul_valori(i,j)
if i = 0 or j = 0 then
  return 0
end if
if V[i, j] < 0 then
  if j < d[i] then
    val ← calcul_valori(i − 1, j)
  else
    val ← max(calcul_valori(i − 1, j), p[i]+calcul_valori(i − 1, j − d[i]))
  end if
  V[i, j] ← val // reţinerea valorii calculate
end if
return V[i, j]// returnarea valorii preluate din tabel sau calculate
```

Algoritmul `calcul_valori` are acces la talourile $d[1..n]$ şi $V[0..n, 0..C]$. Apelul `calcul_valori`$(n, C)$ se plasează după iniţializarea tabloului.

### 8.2.4 Problema închiderii tranzitive

Nu doar problemele de optimizare pot fi rezolvate cu tehnica programării dinamice ci şi alte probleme a căror soluţie poate fi exprimată în funcţie de

soluţiile unor subprobleme.

Considerăm o relaţie binară $R \subset \{1,\ldots,n\} \times \{1,\ldots,n\}$. Închiderea sa tranzitivă este o relaţie binară $R^* \subset \{1,\ldots,n\} \times \{1,\ldots,n\}$ având proprietatea: dacă pentru $i,j \in \{1,\ldots,n\}$ există $i_1,\ldots,i_m \in \{1,\ldots,n\}$ cu proprietăţile: $(i_1,i_2) \in R$, $(i_2,i_3) \in R$, ... ,$(i_{m-1},i_m) \in R$ iar $i = i_1$ şi $j = i_m$ atunci $(i,j) \in R^*$.

Pentru a construi $R^*$ se consideră următorul set de relaţii binare $R^0 = R$, $R^1$, ..., $R^n = R^*$, definite astfel: dacă pentru $i,j \in \{1,\ldots,n\}$ există $i_1,\ldots,i_m \in \{1,\ldots,k\}$ cu proprietăţile: $(i_1,i_2) \in R$, $(i_2,i_3) \in R$, ..., $(i_{m-1},i_m) \in R$ iar $i = i_1$ şi $j = i_k$ atunci $(i,j) \in R^k$. Relaţiile pot fi descrise prin recurenţe astfel: $(i,j) \in R^k$ dacă $(i,j) \in R^{k-1}$ sau $(i,k) \in R^{k-1}$ şi $(k,j) \in R^{k-1}$.

Pentru a elabora algoritmul de construire a lui $R^*$ considerăm relaţiile binare pe $\{1,2,\ldots,n\}$ reprezentate prin matrici ale căror elemente sunt definite astfel:

$$r_{ij} = \begin{cases} 1 & \text{dacă } (i,j) \in R \\ 0 & \text{dacă } (i,j) \notin R \end{cases}$$

Relaţiile de recurenţă pentru construirea matricilor asociate relaţiilor binare $R^0, R^1, \ldots, R^n$ sunt:

$$r_{ij}^k = \begin{cases} 1 & \text{dacă } r_{ij}^{k-1} = 1 \vee (r_{ik}^{k-1} = 1 \wedge r_{kj}^{k-1} = 1) \\ 0 & \text{altfel} \end{cases} \qquad k = \overline{1,n}$$

cu $r_{ij}^0 = r_{ij}$. Folosind aceste relaţii de recurenţă se poate construi matricea asociată relaţiei binare $R^n$, utilizând doar două matrici auxiliare în modul descris în Algoritmul 8.9.

Algoritmul pentru determinarea închiderii tranzitive este cunoscut şi sub numele de algoritmul lui Warshall.

## 8.3 Probleme

**Problema 8.1** (*Problema subşirului strict crescător de sumă maximă.*) Fie $a_1, a_2, \ldots, a_n$ un şir de valori reale pozitive. Să se determine un subşir strict crescător pentru care suma elementelor este maximă.

*Rezolvare.* Se parcurg etapele aplicării metodei programării dinamice:

(a) *Analiza structurii unei soluţii optime.* Fie $a_{j_1} < \ldots < a_{j_{k-1}} < a_{j_k}$ un subşir strict crescător pentru care suma elementelor este maximă. Presupunând că $a_{j_{k-1}} = a_i$ rezultă că $a_{j_1} < \ldots < a_{j_{k-1}}$ este un subşir strict crescător de lungime maximă care se termină în $a_i$ (în caz contrar ar exista un subşir strict crescător al şirului iniţial pentru care suma elementelor este mai mare decât $\sum_{l=1}^{k} a_{j_l}$, contrazicând ipoteza că $a_{j_1}, \ldots, a_{j_{k-1}}, a_{j_k}$ este soluţie optimă). Deci o soluţie

**Algoritmul 8.9** Determinarea închiderii tranzitive a unei relaţii binare

```
închidere_tranzitivă(R[1..n, 1..n])
R2[1..n, 1..n] ← R[1..n, 1..n]
for k ← 1, n do
  R1[1..n, 1..n] ← R2[1..n, 1..n]
  for i ← 1, n do
    for j ← 1, n do
      if (R1[i, j] = 1) or (R1[i, k] = 1 and R1[k, j] = 1) then
        R2[i, j] ← 1
      else
        R2[i, j] ← 0
      end if
    end for
  end for
end for
return R2[1..n, 1..n]
```

optimă a problemei generale $P(i)$ a determinării unui subşir strict crescător care se termină în $a_i$ şi care are suma elementelor maximă conţine o soluţie optimă a uneia dintre subproblemele $P_j$ pentru care $j < i$ şi $a_j < a_i$.

(b) *Deducerea unei relaţii de recurenţă.* Valoarea care trebuie maximizată este suma elementelor subşirului, astfel că relaţia de recurenţă ce specifică legătura între soluţia problemei şi soluţiile subproblemelor:

$$S_i = \begin{cases} a_i & i = 1 \\ a_i + max\{S_j | a_j < a_i, j = \overline{1, i-1}\} & i > 1 \end{cases}$$

(c) *Dezvoltarea relaţiei de recurenţă.* Se completează tabelul $S[1..n]$, unde $S[i]$ este suma maximă corespunzătoare subşirurilor strict crescătoare care se termină în $a[i]$. În tabloul $P[1..n]$ se reţin indicii pentru care se atinge valoarea maximă. Algoritmul pentru completarea tabelelor $S$ şi $P$ este descris în 8.10.

(d) *Construirea soluţiei.* Subşirul se construieşte pornind de la ultimul element. Acesta corespunde valorii maxime din tabloul $S[1..n]$. După completarea unui element în subşir se identifică elementul imediat anterior folosind tabloul $P[1..n]$. La final elementele tabloului în care este reţinută soluţia se inversează (Algoritmul 8.11).

**Problema 8.2** (*Problema celui mai lung subşir comun a două şiruri.*) Fie $A = (a_1, a_2, \ldots, a_m)$ şi $B = (b_1, b_2, \ldots, b_n)$ două şiruri. Să se determine cel mai lung subşir comun al celor două şiruri, adică $(c_1, c_2, \ldots, c_l)$ cu proprietatea că există $i_1 < i_2 < \ldots < i_l$ şi $j_1 < j_2 < \ldots < j_l$ astfel încât $c_k = a_{i_k} = b_{j_k}$ pentru $k = \overline{1, l}$.

**Algoritmul 8.10** Dezvoltarea relaţiei de recurenţă în cazul determinării subşirului strict crescător de sumă maximă

```
calcul(real a[1..n], S[1..n], P[1..n])
S[1] ← a[1]; P[1] ← 0
for i ← 2, n do
  imax ← 0; max ← 0
  for j ← 1, i − 1 do
    if a[j] < a[i] and max < S[j] then
      imax ← j; max ← S[j]
    end if
  end for
  P[i] ← imax
end for
```

*Rezolvare.* De exemplu pentru şirurile $a = (2, 1, 4, 3, 2)$ şi $b = (1, 3, 4, 2)$ există două subşiruri comune de lungim maximă (3) şi anume: $(1, 4, 2)$ şi $(1, 3, 2)$.

(a) *Analiza structurii unei soluţii optime.* Fie $c = (c_1, c_2, \ldots, c_{l-1}, c_l)$ o soluţie optimă. Presupunem că $c_l = a_i = b_j$. $c$ poate fi considerată soluţie optimă a problemei $P(i, j)$ ce constă în determinarea celui mai lung subşir comun al şirurilor $a_{1..i} = (a_1, \ldots, a_i)$ şi $b_{1..j} = (b_1, \ldots, b_j)$. Se poate demonstra prin reducere la absurd că $(c_1, c_2, \ldots, c_{l-1}$ este soluţie optimă a subproblemei $P(i - 1, j - 1)$.

(b) *Deducerea unei relaţii de recurenţă.* Se notează cu $L(i, j)$ numărul de elemente al celui mai lung subşir comun al lui $(a_1, a_2, \ldots, a_i)$ şi $(b_1, b_2, \ldots, b_j)$. Relaţia de recurenţă pentru calculul lui $L(i, j)$ este:

$$L(i,j) = \begin{cases} 0 & \text{dacă } i = 0 \text{ sau } j = 0 \\ L(i-1, j-1) + 1 & \text{dacă } a_i = b_j \\ \max\{L(i-1, j), L(i, j-1)\} & \text{în celelalte cazuri} \end{cases}$$

În cazul exemplului de mai sus matricea $L$ are următoarea structură:

$$\begin{array}{ccccc} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 2 & 2 \\ 0 & 1 & 2 & 2 & 2 \\ 0 & 1 & 2 & 2 & 3 \end{array}$$

(c) *Dezvoltarea relaţiei de recurenţă.* Este descrisă în Algoritmul 8.12.

(d) *Construirea soluţiei.* Dacă $a_i = b_j$ atunci soluţia problemei $P(i, j)$ se obţine din soluţia subproblemei $P(i - 1, j - 1)$ prin adăugarea valorii comune celor

**Algoritmul 8.11** Construirea subşirului crescător de sumă maximă

```
solutie(real a[1..n])
calcul(a[1..n], S[1..m], P[1..m])
imax ← 1
for i ← 2, n do
  if S[imax] < S[i] then
    imax ← i
  end if
end for
k ← 1; R[k] ← a[imax]; i ← P[imax]
while i > 0 do
  k ← k + 1
  R[k] ← a[i]
  i ← P[i]
end while
R[1..k] ←inversare(R[1..k])
```

**Algoritmul 8.12** Dezvoltarea relaţiei de recurenţă pentru determinarea celui mai lung subşir comun

```
construire (integer a[1..m], b[1..n])
for i ← 0, m do
  L[i, 0] ← 0
end for
for j ← 0, n do
  L[0, j] ← 0
end for
for i ← 1, m do
  for j ← 1, n do
    if a[i] = b[j] then
      L[i, j] ← L[i − 1, j − 1] + 1
    else
      if L[i − 1, j] > L[i, j − 1] then
        L[i, j] ← L[i − 1, j]
      else
        L[i, j] ← L[i, j − 1]
      end if
    end if
  end for
end for
return L[0..m, 0..n]
```

două şiruri $a_i = b_j$. Dacă $a_i \neq b_j$ atunci soluţia lui $P(i,j)$ coincide cu soluţia lui $P(i-1,j)$ (dacă $L[i-1,j] \geq L[i,j-1]$) respectiv cu soluţia lui $P(i,j-1)$ (dacă $L[i-1,j] < L[i,j-1]$). Presupunând că tabloul în care se colectează soluţia ($x$) şi variabila ce conţine numărul de elemente ale subşirului comun ($k$) sunt variabile globale (iar $k$ este iniţializată cu 0), algoritmul pentru construirea soluţiei este descris recursiv în 8.13.

**Algoritmul 8.13** Determinarea celui mai lung subşir comun a două şiruri

```
solutie(i, j)
if L[i, j] ≠ 0 then
  if a[i] = b[j] then
    solutie(i-1,j-1)
    k ← k + 1
    x[k] ← a[i]
  else
    if L[i − 1, j] ≥ L[i, j − 1] then
      solutie(i − 1, j)
    else
      solutie(i, j − 1)
    end if
  end if
end if
```

**Problema 8.3** (*Problema monedelor.*) Se consideră monede de valori $d_n > d_{n-1} > \ldots > d_1 = 1$. Să se găsească o acoperire minimală (ce foloseşte cât mai puţine monede) a unei sume date $S$.

*Rezolvare.* Întrucât există monedă de valoare 1 orice sumă poate fi acoperită exact. In cazul general, tehnica greedy (bazată pe ideea de a acoperi cât mai mult posibil din suma cu moneda de valoarea cea mai mare) nu conduce întotdeauna la soluţia optimă. De exemplu dacă $S = 12$ şi se dispune de monede cu valorile 10, 6, 1 atunci aplicând tehnica greedy s-ar folosi o monedă de valoare 10 şi două monede de valoare 1. Soluţia optimă este însă cea care foloseşte două monede de valoare 6.

(a) *Analiza structurii unei soluţii optime.* Considerăm problema generică $P(i,j)$ care constă în acoperirea sumei $j$ folosind monede de valori $d_1 < \ldots < d_i$. Soluţia optimă corespunzătoare acestei probleme va fi un şir, $(s_1, s_2, \ldots, s_k)$ de valori corespunzătoare monedelor care acoperă suma $j$. Dacă $s_k = d_i$ atunci $(s_1, s_2, \ldots, s_{k-1})$ trebuie să fie soluţie optimă pentru subproblema $P(i, j-d_i)$ ($i$ rămâne nemodificat întrucât pot fi folosite mai multe monede de aceeaşi valoare). Dacă $s_k \neq d_i$ atunci $(s_1, s_2, \ldots, s_{k-1})$ trebuie să fie soluţie optimă a subproblemei $P(i-1, j)$.

(b) *Deducerea relaţiei de recurenţă.* Fie $R(i,j)$ numărul minim de monede de valori $d_1,\ldots,d_i$ care acoperă suma $j$. $R(i,j)$ satisface:

$$R(i,j)=\begin{cases}0 & j=0\\ j & i=1\\ R(i-1,j) & j<d_i\\ \min\{R(i-1,j),1+R(i,j-d_i)\} & j\geq d_i\end{cases}$$

Pentru exemplul de mai sus se obţine matricea:

| 0 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| 6 | 0 | 1 | 2 | 3 | 4 | 5 | 1 | 2 | 3 | 4 | 5 | 6 | 2 |
| 10 | 0 | 1 | 2 | 3 | 4 | 5 | 1 | 2 | 3 | 4 | 1 | 2 | 2 |

Pentru a uşura construirea soluţiei se poate construi încă o matrice $Q[1..n,0..S]$ caracterizată prin:

$$Q(i,j)=\begin{cases}1 & d_i \text{ se foloseşte pentru acoperirea sumei } j\\ 0 & d_i \text{ nu se foloseşte pentru acoperirea sumei } j\end{cases}$$

In cazul exemplului analizat matricea $Q$ va avea următorul conţinut:

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 6 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 10 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |

(c)*Dezvoltarea relaţiei de recurenţă.* Matricile $R$ şi $Q$ pot fi completate după cum este descris în Algoritmul 8.14 ($R[n,s]$ reprezintă numărul minim de monede necesare pentru a acoperi suma $S$).

(d)*Construirea solutiei.* Considerând că tabloul $a$ în care se colectează soluţia şi variabila $k$ ce conţine numărul de elemente ale lui $a$ sunt variabile globale, soluţia poate fi construită în manieră recursivă aşa cum e descris în algoritmul 8.14.

**Problema 8.4** (*Problema submulţimii de sumă dată.*) Fie $A=\{a_1,\ldots,a_n\}$ o mulţime de valori naturale şi $c$ un număr natural. Să se determine o submulţime $S\subset A$ pentru care suma elementelor nu depăşeşte valoarea $c$ dar este maximală ($\sum_{s\in S}s\leq c$ şi $\sum_{s\in S}s$ este maximă).

*Rezolvare.* In cazul general problema $P(i,j)$ se referă la determinarea unei submulţimi a mulţimii $\{a_1,\ldots,a_i\}$ pentru care suma elementelor este cât mai apropiată de $j$. Notând cu $S(i,j)$ suma elementelor unei soluţii optimale a problemei $P(i,j)$ relaţia de recurenţă corespunzătoare este:

**Algoritmul 8.14** Dezvoltarea relaţiei de recurenţă în cazul problemei monedelor şi construirea soluţiei

```
completare(d[1..n],S)
for i ← 1, n do
  R[i, 0] ← 0
end for
for j ← 1, s do
  R[1, j] ← j
end for
for i ← 1, n do
  for j ← 1, s do
    if j < d[i] then
      R[i, j] ← R[i − 1, j]; Q[i, j] ← 0
    else
      if R[i − 1, j] < 1 + R[i, j − d[i]] then
        R[i, j] ← R[i − 1, j]; Q[i, j] ← 0
      else
        R[i, j] ← R[i, j − d[i]] + 1; Q[i, j] ← 1
      end if
    end if
  end for
end for
```

```
solutie(i, j)
if j ≠ 0 then
  if Q[i, j] = 1 then
    k ← k + 1
    a[k] ← d[i]
    solutie(i, j − d[i])
  else
    solutie(i − 1, j)
  end if
end if
```

$$S(i,j) = \begin{cases} 0 & i = 0 \text{ sau } j = 0 \\ S(i-1,j) & a_i > j \\ \max\{S(i-1,j), S(i-1,j-a_i) + a_i\} & a_i \leq j \end{cases}$$

O dată completată matricea $S[0..n, 0..c]$, soluţia $R[1..k]$ poate fi construită ca în Algoritmul 8.15.

**Algoritmul 8.15** Determinarea unei submulţimi de sumă dată

```
solutie(i, j)
if i ≠ 0 and j ≠ 0 then
  if a[i] > j or (a[i] ≤ j and S[i − 1, j] > S[i − 1, j − a_i] + a_i) then
    solutie(i − 1, j)
  else
    solutie(i − 1, j − a[i]); k ← k + 1; R[k] ← a[i]
  end if
end if
```

**Problema 8.5** (*Calculul distanţei de editare.*) Fie $x[1..m]$ şi $y[1..n]$ două şiruri. Distanţa de editare (distanţa Levenshtein) dintre cele două şiruri se

defineşte ca fiind numărul minim de operaţii de înlocuire/ inserţie/ ştergere element necesare pentru a transforma unul dintre şiruri în celălalt. De exemplu cuvântul "carte" este la distanţa 1 faţă de cuvintele "caste" (un caracter înlocuit), "care" (un caracter eliminat) sau "cartel" (un caracter inserat), la distanţa 2 faţă de cuvântul "castel" ("carte"→"caste"→"castel") şi la distanţa 3 faţă de cuvântul "caiet" ("carte"→"cairte" → "caiete" → "caiet"). Aceasta distanţă este utilizată în bioinformatică, la compararea secvenţelor ADN şi în "spell checking".

*Rezolvare.* Problema generică $P(i, j)$ poate fi formulată ca fiind cea a determinării distanţei de editare dintre şirurile parţiale $x[1..i]$ şi $y[1..j]$. La fel ca în cazul problemei determinării celui mai lung subşir comun $P(i, j)$ se reduce la una dintre subproblemele $P(i-1, j-1)$, $P(i, j-1)$ respectiv $P(i-1, j)$. Notând cu $d(i, j)$ distanţa de editare dintre $x[1..i]$ şi $y[1..j]$, relaţia de recurenţă asociată poate fi descrisă prin:

$$d(i,j) = \begin{cases} i & j = 0 \\ j & i = 0 \\ d(i-1, j-1) & x[i] = y[j] \\ \min\{d(i-1, j-1)+1, d(i, j-1)+1, d(i-1, j)+1\} & x[i] \neq y[j] \end{cases}$$

În cazul în care valoarea minimă se obţine pentru $d(i-1, j-1)+1$ înseamnă că e necesară înlocuirea elementului $x[i]$ cu elementul $y[j]$. Dacă valoarea minimă se obţine pentru $d(i, j-1)+1$ înseamnă că e necesară înserţia elementului $y[j]$, iar dacă valoarea minimă se obţine pentru $d(i-1, j)+1$ atunci e necesară eliminarea elementului $x[i]$.

Matricea de distanţe pentru cazul cuvintelor "carte" şi "caiet" este:

| | | c | a | i | e | t |
|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 |
| 0 | 0 | 1 | 2 | 3 | 4 | 5 |
| c 1 | 1 | 0 | 1 | 2 | 3 | 4 |
| a 2 | 2 | 1 | 0 | 1 | 2 | 3 |
| r 3 | 3 | 2 | 1 | 1 | 2 | 3 |
| t 4 | 4 | 3 | 2 | 2 | 2 | 2 |
| e 5 | 5 | 4 | 3 | 3 | 2 | 3 |

Matricea de distanţe poate fi completată în mod clasic sau se poate folosi tehnica memoizării care permite completarea doar a elementelor necesare pentru calculul lui $d(m, n)$. Tehnica se caracterizează prin faptul că matricea $d$ este iniţializată cu o valoare virtuală (de exemplu, $-1$) iar completarea elementelor se face in manieră top-down aşa cum este ilustrat în Algoritmul 8.16.

Pentru exemplul de mai sus matricea de distanţe completată folosind tehnică memoizării are următorul conţinut:

**Algoritmul 8.16** Utilizarea tehnicii memoizării în calculul distanţei de editare

```
memo(integer i, j)
if i >= 0 and j >= 0 then
  if d[i, j] = −1 then
    if i = 0 then
      d[i, j] ← j
    else
      if j = 0 then
        d[i, j] ← i
      else
        if x[i] = y[j] then
          d[i, j] ← memo(i − 1, j − 1)
        else
          d[i, j] ← min(memo(i − 1, j − 1) + 1,memo(i, j − 1) + 1,
                  memo(i − 1, j) + 1)
        end if
      end if
    end if
  end if
  return d[i, j]
end if
```

| | | c | a | i | e | t |
|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 |
| 0 | 0 | 1 | 2 | 3 | 4 | -1 |
| c 1 | 1 | 0 | 1 | 2 | 3 | -1 |
| a 2 | 2 | 1 | 0 | 1 | 2 | -1 |
| r 3 | 3 | 2 | 1 | 1 | 2 | -1 |
| t 4 | 4 | 3 | 2 | 2 | 2 | 2 |
| e 5 | -1 | -1 | -1 | -1 | 2 | 3 |

Vizualizarea operaţiilor de editare prin care şirul $x[1..m]$ poate fi transformat în şirul $y[1..n]$ poate fi realizată prin Algoritmul 8.17.

**Problema 8.6** Fie $a[1..n, 1..n]$ o matrice de valori reale. Se defineşte un traseu în matrice ca o succesiune de elemente $a[i_1, 1], a[i_2, 2], \ldots, a[i_n, n]$ cu proprietatea că $i_{k+1} \in \{i_k, i_k-1, i_k+1\}$ (evident dacă $i_k = 1$ atunci $i_{k+1} \in \{i_k, i_k+1\}$ iar dacă $i_k = n$ atunci $i_{k+1} \in \{i_k, i_k-1\}$). Un astfel de traseu vizitează câte un element de pe fiecare coloană trecând de la un element curent la unul "vecin" de pe coloană următoare (modulul diferenţei dintre indicii de linie este cel mult 1). Să se determine un traseu având suma elementelor maximă.

*Rezolvare.* Fie $P(i, j)$ problema determinării unui traseu de sumă maximă care

**Algoritmul 8.17** Determinarea operaţiilor corespunzătoare distanţei de editare

```
transformari(integer x[1..m], y[1..n], d[1..m, 1..n])
i ← m; j ← n
while i > 0 and j > 0 and d[i, j] > 0 do
  if x[i] = y[j] then
    i ← i − 1; j ← j − 1;
  else
    if d[i, j] = d[i − 1, j − 1] + 1 then
      "inlocuire x[i] cu y[j]"; i ← i − 1; j ← j − 1;
    else
      if d[i, j] = d[i, j − 1] + 1 then
        "inserare y[j]"; j ← j − 1;
      else
        "stergere x[i]"; i ← i − 1;
      end if
    end if
  end if
end while
if i = 0 then
  while j > 0 and d[i, j] > 0 do
    "inserare y[j]"; j ← j − 1
  end while
end if
if j = 0 then
  while i > 0 and d[i, j] > 0 do
    "stergere x[i]"; i ← i − 1
  end while
end if
```

se termină în $a[i, j]$. Soluţia optimă a acestei probleme conţine soluţia optimă a uneia dintre subproblemele $P(i-1, j-1)$, $P(i, j-1)$ respectiv $P(i+1, j-1)$. Dacă notăm cu $V(i, j)$ suma asociată traseului optim care se termină în $a[i, j]$ atunci relaţia de recurenţă asociată este:

$$V(i,j) = \begin{cases} a(i,j) & j = 1 \\ \max\{V(i-1,j-1) + a(i,j), V(i,j-1) + a(i,j), & j > 1 \\ \qquad V(i+1,j-1) + a(i,j)\} & \end{cases}$$

Să considerăm următorul exemplu:

$$A = \begin{bmatrix} 10 & 4 & 1 & 6 \\ 3 & -2 & 8 & 2 \\ 2 & 15 & -6 & 4 \\ 7 & -3 & 4 & 9 \end{bmatrix}$$

Printr-o abordare de tip greedy (în care se porneşte de la cel mai mare element de pe prima coloană şi la fiecare etapă se alege elementul cel mai mare din cele vecine aflate pe coloana următoare) s-ar obţine traseul (10,4,8,6) având suma 28. Aplicând relaţia de recurenţă de mai sus se obţine matricea:

$$V = \begin{bmatrix} 10 & 14 & 15 & 36 \\ 3 & 8 & 30 & 32 \\ 2 & 22 & 16 & 34 \\ 7 & 4 & 26 & 35 \end{bmatrix}$$

ceea ce conduce la traseul: (7,15,8,6) având suma 36.

O dată completată matricea $V$, soluţia ($s = (i_1, i_2, \ldots, i_n)$) poate fi construită începând de la ultimul element aşa cum este descris în Algoritmul 8.18.

**Algoritmul 8.18** Determinarea unui traseu de sumă maximă

```
solutie(integer a[1..n, 1..n], s[1..n], V[1..n, 1..n])
s[n] ← imax(V[1..n, n]) // indicele celui mai mare element din ultima
                        //coloană a matricii V
for k ← n − 1, 1, −1 do
  if V[s[k + 1], k + 1] = V[s[k + 1], k] + a[s[k + 1], k + 1] then
    s[k] ← s[k + 1]
  else
    if s[k+1] > 1 and V[s[k+1], k+1] = V[s[k+1]−1, k]+a[s[k+1], k+1]
    then
      s[k] ← s[k + 1] − 1
    else
      if s[k+1] < n and V[s[k+1], k+1] = V[s[k+1]+1, k]+a[s[k+1], k+1]
      then
        s[k] ← s[k + 1] + 1
      end if
    end if
  end if
end for
```

**Problema 8.7** (*Problema turistului în Manhattan.*) Se consideră o grilă pătratică $n \times n$ (care poate fi interpretată ca harta străzilor din Manhattan). Fiecare muchie în cadrul grilei are asociată o valoare (ce poate fi interpretată

ca fiind câştigul turistului care parcurge porţiunea respectivă de stradă). Se caută un traseu care porneşte din nodul (1,1), se termină în nodul $(n, n)$, şi care are proprietatea că la fiecare etapă se poate trece din nodul $(i, j)$ fie în nodul $(i, j + 1)$ (deplasare la dreapta) fie în nodul $(i + 1, j)$ (deplasare în jos). În plus valoarea asociată traseului trebuie să fie maximă.

*Rezolvare.* Presupunem că valorile asociate muchiilor grilei sunt stocate în două matrici: $C_D[1..n, 1..n-1]$ ($C_D[i, j]$ conţine valoarea asociată trecerii din nodul $(i, j)$ în nodul $(i, j + 1)$) iar $C_J[1..n-1, 1..n]$ conţine valoarea asociată trecerii din nodul $(i, j)$ în nodul $(i + 1, j)$.

Dacă notăm cu $P(i, j)$ problema determinării unui traseu optim care se termină în $(i, j)$ atunci soluţia optimă a acestei probleme conţine soluţia optimă a uneia dintre subproblemele $P(i, j-1)$ respectiv $P(i-1, j)$. Notând cu $C(i, j)$ valoarea asociată soluţiei optime a problemei $P(i, j)$, relaţia de recurenţă asociată este:

$$C(i,j) = \begin{cases} 0 & i = 1, j = 1 \\ C(i, j-1) + C_D(i, j-1) & i = 1, j > 1 \\ C(i-1, j) + C_J(i-1, j) & i > 1, j = 1 \\ \max\{C(i, j-1) + C_D(i, j-1), & i > 1, j > 1 \\ \quad C(i-1, j) + C_J(i-1, j)\} & \end{cases}$$

După completarea matricii $C$ (folosind fie varianta clasică fie cea bazată pe tehnica memoizării) elementul $C(n, n)$ indică valoarea asociată soluţiei optime. Traseul propriu-zis poate fi determinat simplu dacă o dată cu construirea matricii $C$ se construieşte şi o matrice $P$ ale cărei elemente indică direcţia asociată ultimului pas făcut pentru a ajunge în nodul respectiv: "dreapta" sau "jos":

$$P(i,j) = \begin{cases} \text{"}dreapta\text{"} & i = 1, j > 1 \\ \text{"}jos\text{"} & i > 1, j = 1 \\ \text{"}dreapta\text{"} & i > 1, j > 1, \\ & C(i, j-1) + C_D(i, j-1) > C(i-1, j) + C_J(i-1, j) \\ \text{"}jos\text{"} & i > 1, j > 1, \\ & C(i, j-1) + C_D(i, j-1) < C(i-1, j) + C_J(i-1, j) \end{cases}$$

In varianta recursivă algoritmul poate fi descris prin:

```
traseu(integer i, j)
if i > 1 or j > 1 then
  if P[i, j] = "jos" then
    traseu(i, j − 1); write "jos"
  else
    traseu(i − 1, j); write "dreapta"
  end if
end if
```

Pentru a obţine traseul se apelează algoritmul `traseu`$(n, n)$.

**Problema 8.8** Se consideră un şir de valori reale (pozitive şi negative). Să se determine subşir de elemente cu semne alternate (un element pozitiv este urmat de un element negativ iar un element negativ este urmat de un element pozitiv) pentru care suma valorilor modulelor este maximă.

**Problema 8.9** Propuneţi o variantă iterativă pentru construirea soluţiei în cazul problemei monedelor.

**Problema 8.10** Se consideră un set de valori naturale nenule $A = (a_1, a_2, \ldots, a_n)$ cu proprietatea că $\sum_{i=1}^{n} a_i$ este număr par. Să se descompună $A$ în două subseturi $B$ şi $C$ astfel încât sumele elementelor din cele două subseturi să fie cât mai apropiate.

*Indicaţie.* Se reduce la un caz particular al problemei rucsacului: selectia unor obiecte care să maximizeze gradul de umplere al unui rucsac având capacitatea egală cu $S = \lfloor \frac{1}{2} \sum_{i=1}^{n} a_i \rfloor$ (criteriul de optim nu este legat de valorile ci de dimensiunile obiectelor). Obiectele selectate vor reprezenta subsetul $B$ iar cele neselectate subsetul $C$. Minimizând diferenţa dintre suma elementelor din $B$ şi valoarea $S$ se minimizează de fapt modulul diferenţei dintre sumele elementelor celor două subseturi.

**Problema 8.11** Justificaţi, folosind arborele de apel, că algoritmul `comb_rec` are complexitate exponenţială.

**Problema 8.12** Propuneţi un algoritm de calculare în manieră ascendentă a lui $C_n^k$ pe baza relaţiei de recurenţă (8.1) folosind cât mai puţin spaţiu auxiliar.

**Problema 8.13** Aplicaţi tehnica memoizării pentru şirul lui Fibonacci şi studiaţi complexitatea algoritmului obţinut.

**Problema 8.14** Aplicaţi tehnica memoizării pentru calculul costului minim în cazul înmulţirii unui şir de matrici.

# Capitolul 9

# Tehnici de parcurgere a spaţiului soluţiilor

Multe dintre problemele întâlnite în informatică se bazează pe căutarea în spaţiul soluţiilor. In această categorie intră atât problemele care necesită identificarea unei configuraţii care satisface anumite restricţii cât şi cele care urmăresc optimizarea unui criteriu. Deşi spaţiul soluţiilor este finit, dimensiunea acestuia creşte de regulă exponenţial cu dimensiunea problemei (cum se întâmplă de exemplu în cazul în care spaţiul soluţiilor potenţiale ale unei probleme de dimensiune $n$ este reprezentat de ansamblul tuturor submulţimilor unei mulţimi cu $n$ elemente). Generarea tuturor configuraţiilor care sunt potenţiale soluţii şi alegerea dintre acestea a celor ce satisfac restricţiile şi/sau criteriul de optim este o abordare total ineficientă.

În astfel de situaţii este necesară o tehnică de căutare controlată a spaţiului soluţiilor, care sa evite pe cât posibil redundanţă şi să permite abandonarea unor configuraţii în momentul în care se poate decide că acestea nu conduc la soluţii ale problemei. Astfel de tehnici sunt *căutarea cu revenire* (backtracking) şi cea de tip *ramifică şi mărgineşte* (branch and bound). Principiile celor două tehnici sunt asemănătoare însă diferă domeniile de aplicabilitate. Căutarea cu revenire este folosită în special pentru probleme de satisfacere a restricţiilor pe când căutarea de tip "ramifică şi mărgineşte" este folosită pentru rezolvarea problemelor de optimizare.

Trebuie menţionat faptul că deşi aceste tehnici conduc de regulă la algoritmi mai eficienţi decât tehnica forţei brute, totuşi complexitatea lor este ridicată astfel că pot fi aplicaţi efectiv doar pentru probleme de dimensiune relativ mică. Specific acestor tehnici este faptul că pentru anumite instanţe ale unei probleme este posibil ca soluţia problemei să fie obţinută prin parcurgerea unei porţiuni mici a spaţiului soluţiilor pe când pentru alte instanţe ale aceleiaşi probleme este posibil ca porţiunea parcursă să fie semnificativ mai mare.

## 9.1 Principiul căutării cu revenire

Tehnica căutării cu revenire se utilizează pentru rezolvarea problemelor printr-o parcurgere controlată a spaţiului soluţiilor. În ultimă instanţă este o îmbunătăţire a metodei căutării exhaustive (metoda forţei brute) care permite reducerea numărului de soluţii potenţiale analizate.

Majoritatea problemelor ce pot fi rezolvate prin "backtracking" pot fi reduse la determinarea unei submulţimi a unui produs cartezian de forma $A_1 \times A_2 \times \ldots \times A_n$ (cu mulţimile $A_k$ finite). Fiecare element al submulţimii poate fi văzut ca o soluţie (metoda fiind astfel adecvată în special în situaţiile în care se doreşte determinarea tuturor soluţiilor unei probleme, nu numai a uneia dintre ele). O soluţie este de forma $s = (s_1, s_2, \ldots, s_n)$ cu $s_k \in A_k = \{a_1^k, a_2^k, \ldots, a_{m_k}^k\}$, $m_k = \text{card} A_k$. În majoritatea cazurilor nu orice element al produsului cartezian este soluţie ci doar cele care satisfac anumite *restricţii*. De exemplu, problema determinării tuturor permutărilor de ordin $n$ poate fi reformulată ca problema determinării submulţimii produsului cartezian $\{1, 2, \ldots, n\} \times \ldots \{1, 2, \ldots, n\}$ ($A_1 = A_2 = \ldots = A_n = \{1, 2, \ldots, n\}$) în care elementele au componentele distincte (restricţiile problemei sunt: $s_i \neq s_j$, pentru orice $i \neq j$).

O soluţie $s$ se obţine completând succesiv componentele $s_k$ pentru $k = \overline{1, n}$. Specificul metodei constă în maniera de parcurgere a spaţiului soluţiilor:

- Soluţiile sunt construite succesiv, la fiecare etapă fiind completată câte o componentă (similar cu tehnica greedy însă ulterior se poate reveni asupra alegerii unei componente);
- Alegerea unei valori pentru o componentă se face într-o anumită ordine (aceasta presupune că pe mulţimile $A_k$ există o relaţie de ordine şi se realizează o parcurgere sistematică a spaţiului $A_1 \times A_2 \times \ldots \times A_n$);
- La completarea componentei $k$ se verifică dacă soluţia parţială ($s_1$, $s_2$, ..., $s_k$), verifică condiţiile induse de restricţiile problemei (acestea sunt numite *condiţii de continuare*). O soluţie parţială care satisface condiţiile de continuare este denumită soluţie parţială validă (sau viabilă) întrucât poate conduce la o soluţie a problemei.
- Dacă au fost încercate toate valorile corespunzătoare componentei $k$ şi încă nu a fost găsită o soluţie sau dacă se doreşte determinarea unei noi soluţii atunci se revine la componenta anterioară $(k-1)$ şi se încearcă următoarea valoare corespunzătoare acesteia ş.a.m.d. Această revenire la o componentă anterioară este specifică tehnicii backtracking.
- Procesul de căutare şi revenire este continuat fie până când este găsită o soluţie (în cazul în care este suficientă determinarea uneia) sau până când au fost testate toate configuraţiile posibile.

Strategia de construire a soluţiilor aplicând backtracking este similară construirii unui structuri arborescente ale cărei noduri corespund unor soluţii parţiale valide (nodurile interne) sau soluţiilor finale (nodurile de pe frontieră). Nodurile de pe frontieră pot corespunde unor soluţii parţiale invalide. În figura 9.1 este ilustrat modul de parcurgere a spaţiului soluţiilor în cazul generării permutărilor de ordin 3. Ramurile abandonate în momentul în care condiţiile de continuare nu sunt satisfăcute sunt marcate cu `X`.

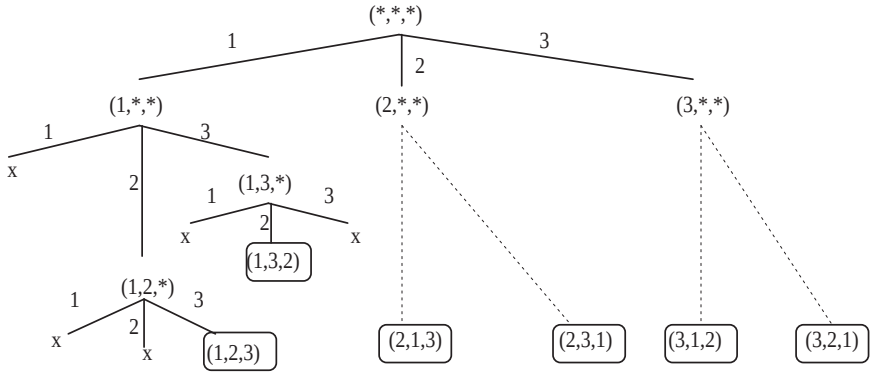


Figura 9.1: Ilustrarea parcurgerii spaţiului soluţiilor în cazul generării permutărilor

În cazul în care sunt specificate restricţii, anumite ramuri ale structurii arborescente asociate parcurgerii sunt abandonate înainte de a atinge lungimea maximă. În aplicarea metodei pentru o problemă concretă se parcurg etapele:

- se alege o reprezentare a soluţiei sub forma unui vector cu $n$ componente;
- se identifică mulţimile $A_1$, $A_2$, ..., $A_n$ şi relaţiile de ordine care indică modul de parcurgere a fiecărei mulţimi;
- pornind de la restricţiile problemei se stabilesc condiţiile de validitate ale soluţiilor parţiale (condiţiile de continuare).
- se stabileşte criteriul în baza căruia se poate decide că s-a obţinut o soluţie finală.

În descrierea structurii generale a algoritmului vom folosi notaţiile: $A_k = \{a_1^k, \ldots, a_{m_k}^k\}$, $m_k$ reprezentând numărul de elemente din $A_k$; $k$ indică componenta curentă din $s$; $i_k$ reprezintă indicele elementului curent din $A_k$. Structura generală a algoritmului este descrisă în 9.1.
**Observaţii.** În aplicaţiile concrete pot interveni următoarele situaţii:

**Algoritmul 9.1** Structura generală a unui algoritm bazat pe tehnica căutării cu revenire

```
backtracking(n, A_1,..., A_n)
k ← 1          // se începe cu completarea primei componente
i_k ← 0        // se pregăteşte indicele de parcurgere a lui A_k
while k > 0 do
  i_k ← i_k + 1          // indicele următorului element din A_k
  valid ← false
        // căutarea unei componente valide pentru poziţia k
  while (valid =false) and (i_k ≤ m_k) do
    s_k ← a^k_{i_k}          // se încearcă valoarea curentă din A_k
    if (s_1, ..., s_k) "satisface condiţiile de continuare" then
      valid ← true
    else
      i_k ← i_k + 1
    end if
  end while
  if valid =true then
    if (s_1, ..., s_k) este soluţie then
      "s-a gasit o soluţie"
    else
      k ← k + 1;
      i_k ← 0 // se pregăteşte completarea următoarei componente
    end if
  else
    k ← k − 1 // se revine la completarea componentei anterioare
  end if
end while
```

1. Mulţimile $A_1,\ldots,\ A_n$ nu sunt neapărat distincte sau elementele lor pot fi generate pe parcursul algoritmului fără a fi necesară transmiterea lor ca argument algoritmului.

2. Condiţiile de continuare se deduc din restricţiile problemei.

3. Pentru unele probleme soluţia este obţinută în cazul în care $k = n$ iar pentru altele este posibil să fie satisfăcută o condiţie de găsire a soluţiei pentru $k < n$ (pentru anumite probleme nu toate soluţiile conţin acelaşi număr de elemente).

4. La găsirea unei soluţii de cele mai multe ori aceasta se afişează sau se reţine într-o zonă dedicată. Dacă se doreşte identificarea unei singure soluţii atunci căutarea se opreşte după găsirea acesteia, altfel procesul de căutare continuă prin analizarea următoarei valori a ultimei componente

completate.

**Exemplul 9.1** *Generarea permutărilor de ordin n.* Caracteristicile acestei probleme sunt:

*Reprezentarea soluţiilor.* O permutare de ordin $n$ este un tablou cu $n$ elemente *distincte* din $\{1, 2, \ldots, n\}$.

*Mulţimile $A_k$.* Mulţimile ce conţin componentele soluţiilor sunt toate egale cu $\{1, 2, \ldots, n\} = A_1 = \ldots = A_n$. Ordinea de parcurgere a elementelor este cea naturală: de la cel mai mic către cel mai mare element.

*Restricţiile şi condiţiile de continuare.* Dacă $s = (s_1, \ldots, s_n)$ este o soluţie atunci ea trebuie să respecte restricţiile: $s_i \neq s_j$ pentru oricare $i \neq j$. Un vector cu $k$ elemente, $(s_1, s_2, \ldots, s_k)$, poate conduce la o soluţie doar dacă satisface condiţiile de continuare $s_i \neq s_j$ pentru orice $i \neq j$ $(i, j \in \{1, \ldots, k\})$. Vom considera că verificarea condiţiilor de continuare va fi efectuată în cadrul unui algoritm de validare. Se observă că la completarea componentei $k$ este suficient să se verifice că $s_k$ este diferit de $s_1$, $s_2$, ..., $s_{k-1}$.

*Condiţia de găsire a unei soluţii.* În acest caz orice vector cu $n$ componente care respectă restricţiile este o soluţie. Deci dacă $k = n$ înseamnă că a fost găsită o soluţie.

*Prelucrarea soluţiilor.* Fiecare soluţie obţinută va fi afişată.

Întrucât $A_k = \{1, \ldots, n\}$, se poate considera că $s_k = i_k$. Cu aceste remarci algoritmul de generare a permutărilor poate fi descris ca în 9.2.

### 9.1.1 Varianta recursivă a algoritmului

Datorită faptului că după completarea componentei $k$ problema se reduce la una similară de completare a componentei $k + 1$ cu una dintre valorile valide, tehnica backtracking poate fi uşor descrisă recursiv. Considerând ca parametru al algoritmului numărul de ordine al componentei care trebuie completată în cadrul apelului curent şi considerând că celelalte date ($n$, mulţimile $A_1$,...,$A_n$) au caracter global, algoritmul poate fi descris ca în 9.3.

Algoritmul recursiv se apelează pentru $k = 1$ (completarea soluţiei începe cu prima componentă): *backtracking_recursiv*(1). Pentru problema generării permutărilor de ordin $n$ varianta recursivă a algoritmului este descrisă în 9.4.

## 9.2 Aplicaţii ale tehnicii căutării cu revenire

### 9.2.1 Generarea tuturor submulţimilor unei mulţimi

Se consideră problema determinării tuturor celor $2^n$ submulţimi ale unei mulţimi $X = \{x_1, x_2, \ldots, x_n\}$. Orice submulţime $S \subset X$ poate fi descrisă prin vectorul

**Algoritmul 9.2** Generarea permutărilor folosind tehnica căutării cu revenire

```
permutări(n)
k ← 1
s[k] ← 0
while k > 0 do
  s[k] ← s[k] + 1
  valid ← false
  while (valid=falseand  (s[k] ≤ n)
  do
    if validare(s[1..k]) =true then
      valid ← true
    else
      s[k] ← s[k] + 1
    end if
  end while
  if valid =true then
    if k = n then
      afisare(s[1..n])
    else
      k ← k + 1; s[k] ← 0
    end if
  else
    k ← k − 1
  end if
end while
```

```
validare(s[1..k])
for i ← 1, k − 1 do
  if s[k] = s[i] then
    return  false
  end if
end for
return  true
```

**Algoritmul 9.3** Varianta recursivă a căutării cu revenire

```
backtracking_recursiv(k)
if (s_1, ..., s_{k−1}) este soluţie then
  "s-a gasit o soluţie" // condiţia de ieşire
else
  for j ← 1, m_k do
    s_k ← a_j^k // se completează componenta k
    if (s_1, ..., s_k) "satisface condiţiile de continuare" then
      // se trece la completarea următoarei componente
      backtracking_recursiv(k + 1)
    end if
  end for
end if
```

**Algoritmul 9.4** Varianta recursivă pentru generarea permutărilor

```
permutari_recursiv(k)
if k = n + 1 then
  afisare(s[1..k])
else
  for j ← 1, n do
    s[k] ← j
    if validare(s[1..k])=true then
      permutari_recursiv(k + 1)
    end if
  end for
end if
```

caracteristic $s = (s_1, \ldots, s_n)$ caracterizat prin:

$$s_k = \begin{cases} 1 & \text{dacă } x_k \in S \\ 0 & \text{dacă } x_k \notin S \end{cases}$$

Astfel $A_1 = \ldots = A_n = \{0, 1\}$ şi orice vector cu componente 0 sau 1 este o soluţie validă (nu se impun restricţii). Rezultă că problema este echivalentă cu a genera elementele produsului cartezian $A_1 \times \ldots \times A_n = \{0, 1\}^n$. Algoritmul poate fi descris ca în 9.5.

**Algoritmul 9.5** Generarea tuturor submulţimilor unei mulţimi

```
submulţimi(n)
k ← 1
s[k] ← −1
while k > 0 do
  s[k] ← s[k] + 1
  if s[k] ≤ 1 then
    if k = n then
      afisare(s[1..n])
    else
      k ← k + 1; s[k] ← −1
    end if
  else
    k ← k − 1
  end if
end while
```

```
subm_rec(k)
if k = n + 1 then
  afisare(s[1..n])
else
  s[k] ← 0; subm_rec(k + 1)
  s[k] ← 1; subm_rec(k + 1)
end if
```

Pornind de la acest algoritm poate fi descris cel pentru generarea tuturor submulţimilor cu $m$ elemente (*combinări de n luate câte m*) ale unei mulţimi $A = \{a_1, \ldots, a_n\}$. Această problemă se caracterizează prin faptul că soluţiile

satisfac restricţia că numărul de componente egale cu 1 (sau echivalent suma tuturor componentelor) este egală cu $m$ ($\sum_{j=1}^{n} s_j = m$). Această restricţie conduce la condiţia de continuare: $\sum_{j=1}^{k} s_j \leq m$. Un vector $(s_1, \ldots, s_k)$ este soluţie dacă $\sum_{j=1}^{k} s_j = m$. Prin modificarea algoritmilor pentru generarea tuturor submulţimilor se obţin variantele din Algoritmul 9.6.

**Algoritmul 9.6** Generarea tuturor submulţimilor cu $m$ elemente

```
combinări(n,m)
k ← 1
s[k] ← −1
while k > 0 do
  s[k] ← s[k] + 1
  if (s[k] ≤ 1) and suma(s[1..k]) ≤ m
  then
    if suma(s[1..k]) = m then
      afisare(s[1..k])
    else
      if k < n then
        k ← k + 1; s[k] ← −1
      end if
    end if
  else
    k ← k − 1
  end if
end while
```

```
comb_rec(k)
if suma(s[1..k − 1]) = m then
  afisare(s[1..k − 1])
else
  if k <= n then
    s[k] ← 0; comb_rec(k + 1)
    s[k] ← 1; comb_rec(k + 1)
  end if
end if
```

Algoritmul `suma`$(s[1..k])$ calculează suma elementelor din $s[1..k]$ iar algoritmul `afisare` apelat pentru $s[1..k]$ afişează afişează elementele $a_i$ pentru care $s[i] = 1$. Se observă că vectorii având toate componentele completate ($k = n$) dar pentru care suma elementelor este diferită de $m$ sunt ignoraţi.

### 9.2.2 Amplasarea damelor pe tabla de şah

O problemă clasică (enunţată de Gauss în 1850) de generare a unor configuraţii ce respectă anumite restricţii este cea a amplasării damelor pe o tablă de şah astfel încât să nu se atace reciproc. Considerăm cazul general în care $n$ dame trebuie amplasate în cadrul unei matrici pătratice $n \times n$ astfel încât pe nici o linie, pe nici o coloană şi pe nici o diagonală să nu se afle două dame.
*Reprezentarea soluţiei.* Întrucât pe fiecare linie se va afla exact o damă şi toate damele sunt identice este suficient să reţinem coloana pe care se află fiecare dintre ele. Astfel soluţia problemei va fi de forma: $(s_1, s_2, \ldots, s_n)$ unde $s_k$ indică coloana pe care se va afla dama de pe linia $k$. Vectorii corespunzători configuraţiilor din figura 9.2 sunt $(3, 1, 4, 2)$ respectiv $(2, 4, 1, 3)$. Pentru valori

mai mari ale lui $n$ numărul configuraţiilor devine din ce în ce mai mare (pentru $n = 8$ există 92 de configuraţii).

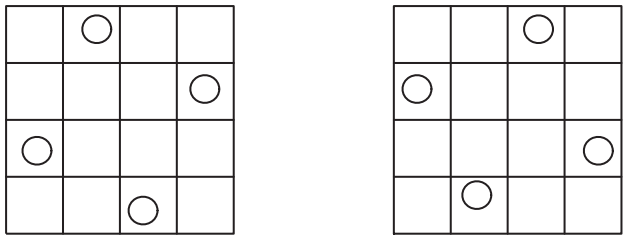


Figura 9.2: Configuraţii valide pentru problema damelor ($n = 4$)

*Restricţii şi condiţii de continuare.* Din modul de reprezentare a soluţiilor restricţia ca damele să nu fie plasate pe aceeaşi linie este implicit satisfăcută. Condiţia ca pe orice coloană să se afle o singură damă este echivalentă cu $s_i \neq s_j$ pentru orice $i \neq j$. În ceea ce priveşte condiţia referitoare la diagonale pornim de la observaţia că două elemente ale unei matrici aflate pe poziţiile $(i_1, j_1)$ respectiv $(i_2, j_2)$ se află pe aceeaşi diagonală dacă $i_1 - j_1 = i_2 - j_2$ sau $i_1 + j_1 = i_2 + j_2$. Astfel condiţia ca două dame să nu se afle pe aceeaşi diagonală este: $i - s_i \neq j - s_j$ şi $i + s_i \neq j + s_j$ pentru orice $i \neq j$. Cele două relaţii sunt echivalente cu $|i-j| \neq |s_i - s_j|$ pentru orice $i \neq j$. La completarea componentei $k$ condiţiile de continuare sunt prin urmare: $s_k \neq s_i$ şi $|k-i| \neq |s_k - s_i|$ pentru orice $k \neq i$.
*Descrierea algoritmului.* Algoritmul este descris în 9.7 atât în variantă iterativă cât şi în variantă recursivă. Se observă că structura algoritmului este cea generală, pentru $A_1 = A_2 = \ldots = A_n = \{1, \ldots, n\}$ iar funcţia `validare` implementează condiţiile de continuare.

### 9.2.3 Colorarea hărţilor

Se consideră o hartă cu $n$ ţări care trebuie colorată folosind $m < n$ culori, astfel încât oricare două ţări vecine să fie colorate diferit. Relaţia de vecinătate dintre ţări este reţinută într-o matrice $n \times n$ ale cărei elemente sunt:

$$v_{ij} = \begin{cases} 1 & \text{dacă } i \text{ e vecină cu } j \\ 0 & \text{dacă } i \text{ nu e vecină cu } j \end{cases}$$

*Reprezentarea soluţiilor.* O soluţie a problemei este o modalitate de colorare a ţărilor şi poate fi reprezentată printr-un vector $(s_1, \ldots, s_n)$ cu $s_i \in \{1, \ldots, m\}$ reprezentând culoarea asociată ţării $i$. Mulţimile de valori ale elementelor sunt $A_1 = \ldots = A_n = \{1, \ldots, m\}$.

**Algoritmul 9.7** Amplasarea damelor

```
plasare_dame(n)
k ← 1
s[k] ← 0
while k > 0 do
  s[k] ← s[k] + 1
  valid ← false
  while (valid =false) and (s[k] ≤ n) do
    if validare(s[1..k]) =true then
      valid ← true
    else
      s[k] ← s[k] + 1
    end if
  end while
  if valid =true then
    if k = n then
      afisare(s[1..n])
    else
      k ← k + 1; s[k] ← 0
    end if
  else
    k ← k − 1
  end if
end while
```

```
dame_rec(k)
if k = n + 1 then
  afisare(s[1..n])
else
  for i ← 1, n do
    s[k] ← i
    for validare(s[1..k]) do
      dame_rec(k + 1)
    end for
  end for
end if
validare(s[1..k])
for i ← 1, k − 1 do
  if (s[k] = s[i]) or (|k − i| = |s[k] − s[i]|) then
    return false
  end if
end for
return true
```

*Restricţii şi condiţii de continuare.* Restricţia ca două ţări vecine să fie colorate diferit se specifică prin: $s_i \neq s_j$ pentru orice $i$ şi $j$ având proprietatea $v_{ij} = 1$. Condiţia de continuare pe care trebuie să o satisfacă soluţia parţială $(s_1, \ldots, s_k)$ este: $s_k \neq s_i$ pentru orice $i < k$ cu proprietatea că $v_{ik} = 1$.
*Descrierea algoritmului.* Variantele iterativă şi recursivă sunt descrise în 9.8 în care este folosit următorul algoritm de validare.

```
validare(s[1..k])
for i ← 1, k − 1 do
  if (s[k] = s[i]) and (v[i, k] = 1) then
    return false
  end if
end for
return true
```

**Algoritmul 9.8** Colorarea hărţilor

```
colorare(n)
k ← 1
s[k] ← 0
while k > 0 do
  s[k] ← s[k] + 1
  valid ← false
  while (valid =false) and (s[k] ≤ m) do
    if validare(s[1..k]) =true then
      valid ← true
    else
      s[k] ← s[k] + 1
    end if
  end while
  if valid =true then
    if k = n then
      afisare(s[1..n])
    else
      k ← k + 1; s[k] ← 0
    end if
  else
    k ← k − 1
  end if
end while
```

```
colorare_rec(k)
if k = n + 1 then
  afisare(s[1..n])
else
  for i ← 1, m do
    s[k] ← i
    if validare(s[1..k]) then
      colorare_rec(k + 1)
    end if
  end for
end if
```

### 9.2.4 Determinarea tuturor drumurilor dintre două oraşe

Se consideră o mulţime de $n$ oraşe $\{o_1, o_2, \ldots, o_n\}$ şi o matrice binară cu $n$ linii şi $n$ coloane care specifică între care oraşe există drumuri directe. Elementele matricii sunt de forma:

$$v_{ij} = \begin{cases} 1 & \text{dacă există drum direct între } i \text{ şi } j \\ 0 & \text{altfel} \end{cases}$$

Se pune problema determinării tuturor drumurilor care leagă oraşul $o_p$ de oraşul $o_q$.

*Reprezentarea soluţiilor.* Spre deosebire de problemele anterioare nu toate soluţiile au aceeaşi lungime. O soluţie a problemei este de forma $(s_1, \ldots, s_m)$ cu $s_i \in \{1, 2, \ldots, n\}$ indicând oraşul care va fi parcurs în etapa $i$ a traseului.

*Restricţii şi condiţii de continuare.* O soluţie $(s_1, \ldots, s_m)$ trebuie să satisfacă: $s_1 = p$ (oraşul de start), $s_m = q$ (oraşul destinaţie), $s_i \neq s_j$ pentru orice $i \neq j$ (nu se trece de două ori prin acelaşi oraş), $v_{s_i s_{i+1}} = 1$ pentru $i = \overline{1, n-1}$ (între oraşele parcurse succesiv există drum direct). La completarea componentei

$k$, condiţia de continuare este $s_k \neq s_i$ pentru $i = \overline{1, k-1}$ şi $v_{s_{k-1}s_k} = 1$. Condiţia de găsire a unei soluţii nu este determinată de numărul de componente completate ci de faptul că $s_k = q$.

Varianta recursivă şi funcţia de validare sunt descrise în Algoritmul 9.9.

**Algoritmul 9.9** Determinarea tuturor traseelor între două oraşe

```
drumuri_rec(k)
if s[k-1] = q then
  afisare(s[1..k-1])
else
  if k ≠ n+1 then
    for i ← 1, n do
      s[k] ← i
      if validare(s[1..k]) =true
      then
        drumuri_rec(k+1)
      end if
    end for
  end if
end if
```

```
validare(s[1..k])
if v[s[k-1], s[k]] = 0 then
  return false
else
  for i ← 1, k-1 do
    if (s[k] = s[i]) then
      return false
    end if
  end for
end if
return true
```

Înainte de apelul algoritmului *drumuri_rec* se completează $s[1]$ cu $p$ iar la apel se specifică: *drumuri_rec*(2).

## 9.3 Tehnica căutării de tip "ramifică şi mărgineşte"

Căutarea cu revenire permite parcurgerea sistematică a spaţiului soluţiilor şi abandonarea căilor care nu conduc la soluţii fezabile. În cazul problemelor de optimizare cu restricţii abandonarea unei ramuri în arborele de parcurgere a spaţiului soluţiilor se poate face nu doar când sunt încălcate restricţiile ci şi atunci când se poate decide că urmând calea respectivă nu este posibil să se obţină o configuraţie mai bună decât cea descoperită până la etapa curentă. Aceasta este ideea tehnicii de căutare bazată pe "ramificare şi mărginire" (branch and bound).

Să considerăm problema determinării unei configuraţii $s = (s_1, s_2, \ldots, s_n) \in S$ care satisface anumite restricţii şi care optimizează o funcţie obiectiv, $f : S \to \mathbb{R}$. La fel ca în cazul tehnicii backtracking soluţia se construieşte succesiv prin completarea componentelor iar la fiecare etapă se verifică dacă soluţia parţială $(s_1, s_2, \ldots, s_k)$ este *promiţătoare*. O soluţie parţială, $s_{(k)} = (s_1, s_2, \ldots, s_k)$, este considerată promiţătoare dacă: (i) nu încalcă restricţiile problemei; (ii) există şansa ca prin completarea celorlalte componente să se ajungă la o configuraţie mai bună decât ceea ce s-a obţinut până în momentul curent al parcurgerii.

La fel ca şi în cazul tehnicii backtracking procesul de construire a soluţiei poate fi ilustrat utilizând un arbore de decizie caracterizat prin faptul că nivelul $k$ corespunde completării componentei $k$ a soluţiei. Dacă nodului curent îi corespunde soluţia parţială $(s_1, \ldots, s_k)$ atunci se parcurg următorii paşi:

*Ramificare.* Se construiesc toate soluţiile parţiale $(s_1, \ldots, s_k, s_{k+1})$ prin completarea succesivă a poziţiei $k+1$ cu toate valorile posibile.

*Mărginire.* Pentru fiecare dintre soluţiile parţiale construite în pasul anterior se verifică dacă satisface restricţiile problemei. Pentru soluţiile ce satisfac restricţiile se estimeaza margini (inferioare şi/sau superioare) corespunzătoare valorii funcţiei obiectiv. Calculul acestor margini depinde de problema de rezolvat şi ele sunt folosite pentru a estima şansa ca o soluţie parţială să conducă la o soluţie mai bună decât cea mai bună soluţie construită până în momentul curent.

In cazul unei probleme de maximizare a doua proprietate specifică faptul că marginea superioară a funcţiei obiectiv a unei configuraţii având primele $k$ componente egale cu $s_{(k)}$ adică $f^*(s_{(k)}) = \max_{(x_{k+1},\ldots,x_n)} f(s_1, \ldots, s_k, x_{k+1}, \ldots, x_n)$ nu este mai mică decât cea mai mare valoare a funcţiei obiectiv corespunzătoare configuraţiilor complete construite până în etapa curentă. In cazul unei probleme de minimizare se estimează marginea inferioară a funcţiei obiectiv şi se urmăreşte ca aceasta să nu fie mai mare decât valoarea asociată celei mai bune soluţii descoperite până în etapa curentă.

O soluţie parţială este abandonată fie dacă nu satisface restricţiile (ca în cazul tehnicii backtracking) fie dacă marginea superioară a funcţiei obiectiv este mai mică decât cea mai bună valoare a unei soluţii (în cazul unei probleme de maximizare) sau dacă marginea inferioară este mai mare decât cea mai bună valoare (în cazul unei probleme de minimizare).

**Exemplul 9.3** (*Problema rucsacului*) Reluăm problema selectării unui subset de obiecte dintre $n$ obiecte având dimensiunile $d_1$,...,$d_n$ şi valorile $v_1$,...,$v_n$ astfel încât suma dimensiunilor obiectelor selectate să nu depăşească capacitatea $C$ a rucsacului iar suma valorilor să fie maximă. Dacă reprezentăm un subset de obiecte printr-un vector $(s_1, \ldots, s_n) \in \{0,1\}^n$ cu proprietatea că $s_k = 1$ înseamnă că obiectul $k$ a fost selectat iar $s_k = 0$ înseamnă că obiectul nu a fost selectat atunci restricţia problemei se exprimă: $\sum_{i=1}^{n} s_i d_i \leq C$ iar funcţia obiectiv este $V(s_1, \ldots, s_n) = \sum_{i=1}^{n} s_i v_i$.
Fiind o problemă de maximizare fiecărei soluţii parţiale $(s_1, \ldots, s_k)$ i se poate asocia ca margine superioară:

$$V^*(s_1, \ldots, s_k) = \sum_{i=1}^{k} s_i v_i + (C - \sum_{i=1}^{k} s_i d_i) \max_{i=\overline{k+1,n}} v_i/d_i \tag{9.1}$$

care reprezintă valoarea care ar fi obţinută dacă zona liberă din rucsac ar fi umplută cu cel mai valoros obiect neselectat încă. Această margine este în

general una largă, în sensul că nu întotdeauna este atinsă. Să considerăm următorul exemplu concret: $C = 4$, $d = (1, 2, 3, 2)$, $v = (8, 14, 15, 10)$. Valorile relative ale obiectelor sunt: $p = (8, 7, 5, 5)$. Considerăm că obiectele sunt ordonate descrescător după valoarea relativă. Acest lucru nu este esenţial dar poate facilita identificarea mai rapidă a unei prime soluţii candidat ce poate fi utilizată ulterior ca referinţă. Se observă că aplicarea tehnicii greedy conduce la soluţia $(1, 1, 0, 0)$ corespunzătoare selectării primelor două obiecte având valoarea totală egală cu 22.

Arborele de decizie corespunzător parcurgerii spaţiului soluţiilor este ilustrat în figura 9.3. Fiecare nod conţine subsetul de obiecte deja selectat, dimensiunea spaţiului încă disponibil în rucsac ($C_r$) şi marginea superioară a valorii totale ($V^*$). Abandonarea unei ramuri din cauza încălcării restricţiei (nu există suficient spaţiu liber în rucsac) este marcată cu **X** iar abandonarea unui nod din cauză că marginea superioară este mai mică decât cea asociată celei mai bune soluţii descoperite ($V_B$) este marcată printr-o linie orizontală. Valoarea $V_B$ este actualizată ori de câte ori se ajunge la o soluţie mai bună.

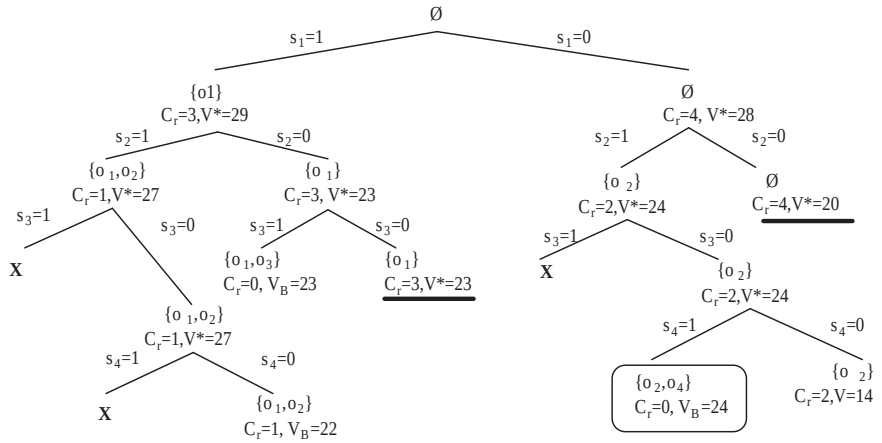


Figura 9.3: Arborele de decizie utilizat în rezolvarea problemei rucsacului folosind tehnica "ramifică şi mărgineşte"

Etapele parcurse în proiectarea unui algoritm bazat pe tehnica branch and bound sunt similare celor de la tehnica backtracking:

- Se alege reprezentarea soluţiei şi se identifică mulţimile de valori corespunzătoare componentelor soluţiei. În cazul problemei rucsacului mulţimile $A_k$ sunt toate egale cu $\{0, 1\}$.
- Se deduc condiţiile de continuare a ramificării (restricţiile pe care trebuie să le satisfacă soluţiile parţiale). Pentru problema rucsacului condiţia de

continuare corespunzătoare unei soluţii parţiale $(s_1, \ldots, s_k)$ este $\sum_{i=1}^{k} s_i d_i < C$.

- Se stabileşte modul de calcul al marginilor corespunzătoare funcţiei obiectiv. Este elementul specific tehnicii "ramifică şi mărgineşte" iar în cazul problemei rucsacului se bazează pe relaţia 9.1.
- Se stabileşte ordinea de parcurgere a soluţiilor parţiale obţinute prin ramificarea unui nod. Nodurile corespunzătoare unui nivel sunt ramificate fie în ordinea în care au fost obţinute fie descrescător după valoarea limitei superioare (în cazul problemelor de maximizare) sau crescător după valoarea limitei inferioare (în cazul problemelor de minimizare). Această strategie nu garanteaza întotdeauna faptul că soluţia va fi descoperită mai devreme (după cum se observă din figura 9.3 soluţia se obţine nu prin ramificarea nodului având $V^* = 29$ ci a celui cu $V^* = 28$).
- Se stabileşte modul în care se decide dacă s-a obţinut o soluţie. În cazul problemei rucsacului s-a ajuns la o soluţie fezabilă fie dacă nu mai există alte obiecte de selectat fie dacă a fost acoperită întreaga capacitate a rucsacului. O dată cu obţinerea unei soluţii fezabile se actualizează $V_B$, dacă valoarea noii soluţii este mai bună decât $V_B$. Pentru a se putea efectua comparaţiile $V_B$ trebuie iniţializată (în cazul unei probleme de maximizare cu o valoare cât mai mică, iar in cazul uneia de minimizare cu o valoare cât mai mare). În cazul problemei rucsacului $V_B$ se poate iniţializa cu 0.

  O dată cu stocarea valorii în $V_B$ se stochează si soluţia corespunzătoare în $S_B$. La sfârşit soluţia problemei se va găsi în $S_B$.

Structura generală a unui algoritm bazat pe tehnica "ramifică şi mărgineşte" este descris în 9.10. Descrierea se bazează pe următoarele ipoteze: (i) se rezolvă o problemă de maximizare; (i) $S_B$ şi $V_B$ sunt variabile globale iar $V_B$ este iniţializată cu o valoare suficient de mică; (iii) algoritmul se apelează pentru $k = 1$ iar rezultatul se colectează în $S_B$.

Principalele diferenţe între tehnica backtracking şi branch and bound sunt:

- Tehnica backtracking se aplică în general pentru problemele de satisfacere a restricţiilor pe când tehnica branch and bound se foloseşte pentru rezolvarea problemelor de optimizare.
- În cazul tehnicii backtracking abandonarea unei ramuri se face doar când sunt încalcate restricţiile pe când în cazul tehnicii branch and bound se face ori de câte ori se ajunge la o configuraţie care nu este promiţătoare (nu este posibil să conducă la o configuraţie pentru care valoarea funcţiei obiectiv să fie mai bună decât cea corespunzătoare soluţiei optime curente).

**Algoritmul 9.10** Structura generală a unui algoritm bazat pe tehnica "ramifică şi mărgineşte"

```
BB(k)
if (s_1, ..., s_{k-1}) este soluţie then
  if V(s_1, ..., s_{k-1}) > V_B then
    S_B ← (s_1, ..., s_{k-1}); V_B ← V(s_1, ..., s_{k-1})
  end if
else
  for j ← 1, m_k do
    s_k ← a_j^k
    if (s_1, ..., s_k) satisface condiţiile de continuare then
      calcul valoare limită V*(s_1, ..., s_k)
    end if
  end for
  ordonează soluţiile parţiale în funcţie de valorile limită
  for toate soluţiile fezabile do
    if V*(s_1, ..., s_k) > V_B then
      BB(k + 1)
    end if
  end for
end if
```

## 9.4 Probleme

**Problema 9.1** (*Problema submulţimii de sumă dată.*) Fie $A = \{a_1, a_2, \ldots, a_n\}$ o mulţime de valori întregi. Să se determine toate submulţimile lui $A$ pentru care suma elementelor este egală cu o valoare dată $V$.

*Rezolvare.*(a)*Reprezentarea soluţiei.* O soluţie a problemei este o submulţime $S$ a mulţimii $A$. Aceasta poate fi reprezentată prin tabloul elementelor sale $S = (s_1, \ldots, s_m)$ cu $m \leq n$ şi $s_i \in A$. Deci mulţimile de valori posibile pentru componentele soluţiei sunt toate egale cu $A$.

(b) *Restricţii şi condiţii de continuare.* O soluţie trebuie să satisfacă următoarele condiţii: $s_i \neq s_j$ pentru orice $i \neq j$ şi $\sum_{i=1}^{m} s_i = V$. Condiţia de continuare dedusă din aceste restricţii este $s_k \neq s_i$ pentru $i = \overline{1, k-1}$. În cazul în care elementele lui $A$ şi valoarea $V$ ar fi valori naturale s-ar putea adăuga ca şi condiţie de continuare $\sum_{i=1}^{k} s_i \leq V$.

(c)*Criteriul de decizie pentru a verifica că s-a obţinut o soluţie finală.* O soluţie parţială $(s_1, \ldots, s_k)$ poate fi considerată soluţie finală dacă $\sum_{i=1}^{k} s_i = V$.

Cu aceste ipoteze algoritmul de generare a tuturor submulţimilor cu suma $V$ este descris în 9.11.

**Algoritmul 9.11** Generarea submulţimilor de sumă dată

```
subm(integer k)
if suma(s[1..k-1]) = V then
  write s[1..k-1]
else
  if k ≤ n then
    for i ← 1, n do
      s[k] ← a[i]
      if validare(s[1..k]) =true then
        subm(k+1)
      end if
    end for
  end if
end if
```

Algoritmul `suma` returnează suma elementelor şirului pentru care este apelat iar algoritmul `validare` verifică dacă $s[k]$ este diferit de elementele şirului $s[1..k-1]$ (acelaşi criteriu de validare ca la problema generării permutărilor). Algoritmul recursiv `subm` va fi apelat pentru $k = 1$. Tablourile $a[1..n]$ şi $s[1..n]$ sunt considerate variabile globale.

**Problema 9.2** (*Problema comis voiajorului (TSP-Traveling Salesman Problem)).* Se consideră un set de $n$ oraşe şi un comis voiajor care trebuie să viziteze toate cele $n$ oraşe pornind din primul oraş, trecând o singură dată prin fiecare oraş şi întorcându-se în primul oraş. Se consideră cunoscută matricea $D[1..n, 1..n]$ în care elementul $D[i, j]$ este 0 dacă nu există drum direct între oraşul $i$ şi oraşul $j$ respectiv lungimea drumului direct dintre cele două oraşe. (i) Să se genereze toate circuitele pe care le poate parcurge comis voiajorul pornind din oraşul 1; (ii) Să se determine cel mai scurt circuit.

*Rezolvare.* În ambele variante soluţia poate fi reprezentată printr-un şir ($s_1$, ..., $s_n$) unde $s_i \in \{1, \ldots, n\}$ reprezintă indicele oraşului vizitat la etapa $i$. Pentru a obţine circuitul trebuie doar adăugat oraşul de pornire (de exemplu, 1). Restricţiile ce trebuie satisfăcute sunt: $s_i \neq s_j$ pentru orice $i \neq j$ (nu se vizitează de două ori acelaşi oraş) şi $D(s_{i-1}, s_i) > 0$ pentru $i = \overline{2, n}$ (există drum direct între oraşele vizitate la etape succesive).

Condiţiile de continuare deduse din restricţiile de mai sus sunt: $s_k \neq s_i$ pentru $i = \overline{1, k-1}$ şi $D(s_{k-1}, s_k) > 0$. În cazul căutării celui mai scurt traseu numărul traseelor generate poate fi redus dacă se completează condiţiile de continuare cu $\sum_{i=2}^{k} D(s_{i-1}, s_i) < Lmin$ unde $Lmin$ este lungimea celui mai scurt traseu generat până la momentul curent. Variabila $Lmin$ se iniţializează înainte de apelul algoritmului de generare cu o valoare suficient de mare ($Lmin = \infty$). In ambele variante ale problemei se consideră că s-a obţinut o soluţie când au fost completate toate cele $n$ componente. Pentru a obţine lungimea circuitului

se adaugă la suma $\sum_{i=2}^{n} D(s_{i-1}, s_i)$ distanţa dintre oraşul $s_n$ şi primul oraş (stocată în $D(s_n, 1)$).

În prima variantă, algoritmul este descris în 9.12.

**Algoritmul 9.12** Problema comis voiajorului. Generarea tuturor circuitelor (stânga). Generarea unui circuit de lungime minimă (dreapta)

```
TSP(integer k)
if k − 1 = n then
  write s[1..n]
else
  for i ← 2, n do
    s[k] ← i
    if validare(s[1..k]) =true
    then
      TSP(k + 1)
    end if
  end for
end if
```

```
1: optTSP(integer k)
2: if k − 1 = n then
3:   if L + D[s[n], s[1]] < Lmin then
4:     Lmin ← L + D[s[n], s[1]]
5:     smin[1..n] ← s[1..n]
6:   end if
7: else
8:   for i ← 2, n do
9:     s[k] ← i
10:    if validare(s[1..k]) =true then
11:      L ← L + D[s[k − 1], s[k]]
12:      optTSP(k + 1)
13:      L ← L − D[s[k − 1], s[k]]
14:    end if
15:  end for
16: end if
```

Înainte de apelul lui `TSP`, $s[1]$ se setează pe 1 (se porneşte întotdeauna din primul oraş) iar algoritmul se apelează pentru $k = 2$. În algoritmul `validare` se verifică faptul că $s[k] \neq s[i]$ pentru $i = \overline{1, k-1}$ şi faptul că $D[s[k-1], s[k]] > 0$. În cazul în care una dintre aceste condiţii este încălcată se returnează **false**.

În a doua variantă se folosesc variabilele globale $smin[1..n]$ şi $Lmin$ pentru a stoca cel mai scurt traseu generat până la momentul curent, respectiv lungimea lui. $Lmin$ se iniţializează cu o valoare suficient de mare. În plus se foloseşte variabila globală $L$ care conţine la fiecare etapă lungimea traseului dintre primul şi ultimul oraş vizitat. Înainte de apel $L$ se iniţializează cu 0. Algoritmul este descris în 9.12. Algoritmul `optTSP` poate fi transformat pe baza ideii de la tehnica "ramifică şi mărgineşte" calculând pentru fiecare soluţie parţială o margine inferioară a lungimii traseului şi efectuând apelul recursiv de la linia 12 doar dacă această limită este mai mică decât $Lmin$ (iniţializată înaintea apelului `optTSP`(1) cu o valoare suficient de mare - de exemplu $n \max_{i,j} D_{ij}$). O variantă de calcul a limitei inferioare pentru lungimea traseului în cazul soluţiei parţiale $(s_1, \ldots, s_k)$ este:

$$L_B = \sum_{i=1}^{k-1} D_{s_i s_{i+1}} + (n-k) \sum_{i=k+1}^{n} d_i + \min_{i=\overline{k+1,n}} D_{i1}$$

unde $d_i$ este media aritmetică a distanţelor dintre oraşul $i$ şi cele mai apropiate alte două oraşe care nu au fost vizitate încă.

**Problema 9.3** (*Problema labirintului.*) Se consideră o grilă pătratică $n \times n$ în care anumite celule sunt libere iar altele sunt ocupate. Celulele libere ale grilei pot fi vizitate prin deplasare din poziţia curentă în oricare dintre poziţiile libere vecine (stânga, dreapta, sus, jos). Presupunând că celulele $(1, 1)$ şi $(n, n)$ sunt libere să se genereze toate traseele care permit trecerea doar prin celule libere de la $(1, 1)$ la $(n, n)$. Un exemplu de labirint si de traseu care uneşte cele două celule este ilustrat în Figura 9.4.

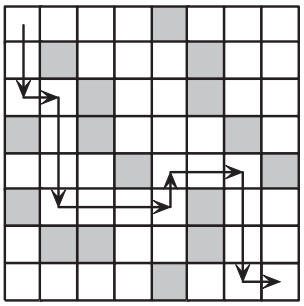

Figura 9.4: Exemplu de labirint şi de traseu care uneşte colţul din stânga sus cu colţul din dreapta jos

*Rezolvare.* Considerăm că informaţia privind starea unei celule este stocată în matricea $M[1..n, 1..n]$ în care $M[i, j]$ este 0 dacă celula e liberă şi 1 în caz contrar. O soluţie a problemei este un şir de perechi de coordonate identificând celulele libere prin care se trece: $s = (s_1, \ldots, s_m)$ cu $s_l = (i_l, j_l)$. Restricţiile ce trebuie satisfăcute de către soluţie sunt: elementele lui $s$ sunt perechi distincte şi se trece doar prin celule libere ($M[s[l].i, s[l].j] = 0$). Când se ajunge în celula $(n, n)$ se consideră că s-a obţinut o soluţie. Dintr-o celulă de coordonate $(i, j)$ se poate trece într-una dintre cele patru celule vecine având coordonatele: $(i-1, j)$, $(i+1, j)$, $(i, j-1)$ şi $(i, j+1)$ (ţinându-se cont de limitările existente pentru celulele de pe frontieră). Algoritmul de validare returnează **false** dacă cel puţin una dintre următoarele condiţii este încălcată:

1. $1 \leq s[k].i \leq n$, $1 \leq s[k].j \leq n$
2. $M[s[k].i, s[k].j] = 0$
3. $s[k] \neq s[l]$, $l = \overline{1, k-1}$

**Algoritmul 9.13** Determinarea unui traseu prin labirint

```
labirint(integer k)
if s[k-1] = (n, n) then
  write s[1..k-1]
else
  s[k] ← (s[k-1].i - 1, s[k-1].j)
  if validare(s[1..k]) then
    labirint (k+1)
  end if
  s[k] ← (s[k-1].i + 1, s[k-1].j)
  if validare(s[1..k]) then
    labirint (k+1)
  end if
  s[k] ← (s[k-1].i, s[k-1].j - 1)
  if validare(s[1..k]) then
    labirint (k+1)
  end if
  s[k] ← (s[k-1].i, s[k-1].j + 1)
  if validare(s[1..k]) then
    labirint (k+1)
  end if
end if
```

Algoritmul care generează traseele este descris în 9.13.

**Problema 9.4** (*Turul calului.*) Se consideră un cal plasat pe o poziţie $(i_0, j_0)$ pe o tablă de şah. Se pune problema determinării unei secvenţe de poziţii prin care trebuie să treacă calul pentru a vizita o singură dată fiecare poziţie.

*Indicaţie.* Soluţia poate fi reprezentată printr-un tablou conţinând 64 de perechi distincte de indici $(i, j)$ $(i, j \in \{1, \ldots, 8\})$. Două perechile consecutive ale soluţiei, $(i, j)$ şi $(i', j')$ trebuie să reprezinte poziţii între care calul se poate deplasa printr-o singură mutare, adică: $|i - i'| = 1$ şi $|j - j'| = 2$ sau $|i - i'| = 2$ şi $|j - j'| = 1$.

**Problema 9.5** (*Partiţionarea unui număr.*) Pentru un număr natural $C$ să se determine toate mulţimile de numere naturale care au proprietatea că suma elementelor lor este egală cu $C$.

*Indicaţie.* Problema este similară cu cea a determinării submulţimilor de sumă dată.

**Problema 9.6** (*Problema asignării sarcinilor*) Se consideră un set de $n$ sarcini de lucru care trebuie asignate unei mulţimi de $n$ lucrători. Asignarea sarcinii $i$ lucrătorului $j$ are costul $C_{ij}$. Să se determine o asignare de cost cât mai mic.

*Indicaţie.* O soluţie este de fapt o permutare $(s_1, \ldots, s_n)$ unde $s_i$ reprezintă lucrătorul asignat sarcinii $i$. Pentru a evita generarea tuturor permutărilor de ordin $n$ se poate folosi ideea de la tehnica "ramifică şi mărgineşte" calculând pentru o soluţie parţială $(s_1, \ldots, s_k)$ o limită inferioară a costului asignării de forma:

$$C^*(s_1, \ldots, s_k) = \sum_{i=1}^{k} C_{is_i} + \sum_{i=k+1}^{n} m_i$$

unde $m_i$ reprezintă valoarea minimă de pe linia $i$ a matricii $C$ care corespunde unui lucrător neselectat încă.

# Capitolul 10

# Algoritmi specifici unor clase de probleme

## 10.1 Operaţii cu numere mari

În anumite aplicaţii, cum sunt cele întâlnite în criptografie, se pune problema operării cu numere întregi având valori care depăşesc valorile maxime reprezentabile prin tipurile standard oferite de către limbajele de programare de uz general (în schimb sistemele software specializate pentru calcule algebrice, cum sunt Maple sau Mathematica permit efectuarea de operaţii cu numere arbitrar de mari).

În astfel de situaţii se poate opta pentru reprezentarea valorilor prin tablouri ale căror elemente sunt "cifre" într-o anumită bază, $b$. În continuare un număr de $n$ cifre va fi reprezentat printr-un tablou $x[0..n-1]$ ale cărui elemente aparţin mulţimii $\{0, \ldots, b-1\}$.

Dacă baza $b$ este mare atunci valorile reprezentate pe $n$ poziţii vor fi suficient de mari. De exemplu dacă baza $b$ este 256 (fiecare "cifră" va fi o valoare între 0 şi 255) valoarea maximă reprezentabilă pe $n$ poziţii va fi $256^n - 1$. De exemplu tabloul $a[0..5] = (120, 35, 180, 252, 78, 102)$ corespunde valorii 112489433146232 reprezentată în baza 256.

Valoarea corespunzătoare unui număr reprezentat în baza $b$ printr-un tablou $x[0..n-1]$ este $x_{n-1}b^{n-1} + x_{n-2}b^{n-2} + \ldots + x_1 b + x_0$. Se observă că aceasta reprezentare este similară reprezentării polinoamelor prin tabloul coeficienţilor, singura diferenţa fiind legată de faptul că în cazul numerelor mari valorile "coeficienţilor" sunt limitate de $b-1$. Această analogie permite utilizarea ideilor de la algoritmii destinaţi efectuării de operaţii asupra polinoamelor.

### 10.1.1 Adunarea a două numere mari

Considerăm două numere $x$ şi $y$ reprezentate în baza $b$ prin tablourile $x[0..n-1]$ şi $y[0..n-1]$. Suma $s = x + y$ va fi reprezentată printr-un un tablou $s[0..n]$. Elementele tabloului sumă pot fi determinate folosind următoarele relaţii:

$$s_i = (x_i + y_i + r_i) \text{ MOD } b, \qquad i = \overline{0, n-1}$$

unde

$$r_0 = 0 \qquad r_{i+1} = (a_i + b_i + r_i) \text{ DIV } b, \qquad i = \overline{0, n-1}$$

$r_i$ reprezintă reportul de la calculul sumei "cifrelor" $x_{i-1}$ cu $y_{i-1}$. Calculul sumei începe de la cifra cea mai puţin semnificativă şi la fiecare etapă se ţine cont de reportul obţinut la etapa anterioară. Algoritmul corespunzător este descris în 10.1. Este uşor de văzut că algoritmul are ordinul de complexitate $\Theta(n)$.

**Algoritmul 10.1** Adunarea a două numere reprezentate prin tablourile cifrelor

```
suma_numere(integer x[0..n − 1],y[0..n − 1],b)
integer i, r,sum, s[0..n]
r ← 0
for i ← 0, n − 1 do
   sum ← x[i] + b[i] + r
   s[i] ← sumMODb
   r ← sumDIVb
end for
if r = 0 then
   return s[0..n − 1]
else
   s[n] ← r
   return s[0..n]
end if
```

### 10.1.2 Înmulţirea a două numere mari

Pentru calculul produsului a două numere $x[0..n-1]$ şi $y[0..n-1]$ se poate aplica algoritmul de înmulţire a două polinoame de grad $n-1$. Singura diferenţă este dată de faptul că fiecare element al tabloului va trebui să aparţină mulţimii $\{0, \ldots, b-1\}$. Produsul va putea fi reprezentat pe $2n$ poziţii iar algoritmul corespunzător este descris în 10.2. Faptul că sunt suficiente $2n$ poziţii pentru reprezentarea produsului rezultă din faptul că la fiecare iteraţie sunt adevărate inegalităţile: $0 \leq t < b^2$ şi $0 \leq r < b$. Aceste inegalităţi pot fi demonstrate prin inducţie matematică. Pentru detalii se poate consulta [9].

**Algoritmul 10.2** Inmulţirea a două numere reprezentate prin tablourile cifrelor

```
produs_numere(integer x[0..n-1],y[0..n-1],b)
integer i, j,r,t, p[0..2n-1]
for i ← 0, 2n-1 do
   p[i] ← 0
end for
for i ← 0, n-1 do
   r ← 0
   for j ← 0, n-1 do
      t ← p[i+j] + x[i] * y[j] + r
      p[i+j] ← t MOD b;
      r ← t DIV p
   end for
   p[i+n] ← r
end for
return c[0..2n-1]
```

Ordinul de complexitate al algoritmului este $\Theta(n^2)$. În anumite cazuri numărul de operaţii poate fi redus analizând pentru fiecare $i$ dacă $x[i]$ este 0 şi evitând înmulţiri inutile. Pentru a reduce ordinul de complexitate se poate proceda la fel ca şi în cazul înmulţirii a două polinoame, utilizând tehnica divizării şi descompunând factorii de $n$ cifre în doi termeni de câte $n/2$ cifre. Astfel factorii $x$ şi $y$ se scriu sub forma: $x = x_{(1)} \cdot b^{n/2} + x_{(0)}$, $y = y_{(1)} \cdot b^{n/2} + y_{(0)}$ iar produsul poate fi scris

$$x \cdot y = x_{(1)} \cdot y_{(1)} \cdot b^2 + ((x_{(1)} + x_{(0)}) \cdot (y_{(1)} + y_{(0)}) - x_{(1)} \cdot y_{(1)} - x_{(0)} \cdot y_{(0)}) \cdot b^{n/2} + x_{(0)} \cdot y_{(0)}$$

În felul acesta se poate ajunge la un algoritm de complexitate $\Theta(n^{\lg 3})$. Un algoritm bazat pe aceasta idee este cunoscut sub numele de algoritmul lui Karatsuba.

### 10.1.3 Împărţirea a două numere mari

Considerăm deîmpărţitul notat cu $x$ iar împărţitorul notat cu $y$. În cazul în care $x < y$ câtul este 0 iar restul este chiar $x$. Dacă $x = y$ câtul este 1 iar restul 0. Rămân de analizat cazurile în care $x > y$. Vom considera în continuare că $x$ este reprezentat prin tabloul $x[0..m+n-1]$ iar $y$ prin tabloul $y[0..n-1]$ (se utilizează cel puţin $n+1$ cifre pentru $x$ chiar dacă $x[n] = 0$). Cifrele câtului se construiesc succesiv prin împarţirea a câte $n+1$ cifre ale deîmpărţitului la cele $n$ cifre ale împărţitorului. Ideea este bazată pe metoda manuală de împărţire care pentru determinarea cifrei $(j-n)$ a câtului împarte numărul reprezentat prin $x[j-n..j]$ la împărţitorul $y[0..n-1]$. Elementul central al metodei este

**Algoritmul 10.3** Înmulţirea şi împărţirea unui număr reprezentat prin tabloul cifrelor în baza $b$ cu un număr, $d$, constituit dintr-o singură cifră

```
multiply1(integer x[0..n-1],d,b)
integer p[0..n],t
p[0..n] ← 0
for i ← 0, n-1 do
   t ← p[i] + x[i] * d
   p[i] ← t MOD b
   p[i+1] ← p[i+1] + t DIV b
end for
return p[0..n]
```

```
divide1(integer x[0..n-1],d,b)
integer c[0..n-1],t,r
r ← 0
for i ← n-1, 0, -1 do
   t ← r * b + x[i]
   c[i] ← t DIV b
   r = t MOD b
end for
return c[0..n-1]
```

determinarea câtului unui număr $a$ reprezentat prin $a[0..n]$ (şi care reprezintă o succesiune de n+1 cifre consecutive din $x[0..m+n-1]$) la împărţitorul $y$ în ipoteza că $a < b \cdot y$. In această ipoteză câtul este constituit dintr-o singură cifră, $q$. O primă estimare a acestei cifre se poate obţine împărţind primele două cifre ale lui $x$ la prima cifră a lui $y$. O astfel de estimare este

$$\hat{q} = \min\{\lfloor (x_n \cdot b + x_{n-1})/y_{n-1} \rfloor, b-1\} \tag{10.1}$$

În [9] este demonstrat faptul că dacă $y_{n-1} \geq \lfloor b/2 \rfloor$ atunci $\hat{q}$ este cel mult cu două unităţi mai mare decât valoarea corectă, $q$ (adică $q \leq \hat{q} \leq q+2$). Condiţia $y_{n-1} \geq \lfloor b/2 \rfloor$ este similară unei condiţii de normalizare şi poate fi asigurată prin înmulţirea atât a împărţitorului cât şi a deîmpărţitului cu cifra $d = \lfloor b/(y_{n-1}+1) \rfloor$. Prin această înmulţire valoarea câtului şi numărul de cifre din $y$ nu se modifică, însă numărul de cifre ale lui $x$ poate creşte cu 1. În plus restul obţinut trebuie împărţit la final la cifra $d$.

Prin urmare sunt necesari algoritmi de înmulţire respectiv împărţire la o cifră. Algoritmii `multiply1` respectiv `divide1` descrişi în 10.3 realizează acest lucru în timp liniar.

Algoritmul 10.5 descrie metoda împărţirii lui $x[0..m+n-1]$ la $y[0..n-1]$ cu obţinerea câtului $q[0..m]$ şi a restului $r[0..n-1]$. Primele 3 prelucrări realizează *normalizarea* valorilor pentru a putea estima cifrele câtului folosind relaţia 10.1. Cum prin înmulţirea cu $d$, numărul de cifre ale lui $x$ poate creşte cu 1 în declararea parametrilor de intrare s-a specificat $x[0..m+n]$ în loc de $x[0..m+n-1]$. Prelucrarea principală o reprezintă ciclul **for** care parcurge cifrele deîmpărţitului începând cu cea mai semnificativă şi determină cifrele câtului împărţind câte $n+1$ cifre din deîmpărţit la împărţitor. Variabila $qq$ conţine la fiecare iteraţie estimarea cifrei corespunzătoare a câtului. Întrucât $qq$ este cu cel mult două unităţi mai mare decât valoarea corectă a câtului, ciclul `while` al cărui rol este să corecteze (parţial) pe $qq$ se execută cel mult de două ori. Algoritmul `subtract` calculează diferenţa a două numere de câte $n+1$ cifre iar rezultatul este stocat în tabloul $d[0..n+1]$. Este posibil că

valoarea reprezentată de subtabloul $x[j-n..j]$ să fie mai mică decât valoarea reprezentată de $p[0..n]$. În acest caz diferenţa se efectuează în mod normal prin efectuarea unui împrumut de la cifra de rang imediat superior, motiv pentru care în $d[n+1]$ se află valoarea $-1$. Algoritmul pentru calculul acestei diferenţe este descris în 10.4.

**Algoritmul 10.4** Calculul diferenţei a două numere reprezentate prin tablourile cifrelor

```
subtract(integer x[0..n], y[0..n], b)
integer z[0..n + 1]
for i ← 0, n − 1 do
   z[i] ← x[i] − y[i]
   if z[i] < 0 then
      z[i] ← z[i] + b; x[i + 1] ← x[i + 1] − 1
   end if
end for
if x[n] ≥ y[n] then
   z[n] ← x[n] − y[n]; z[n + 1] ← 0
else
   z[n] ← b + x[n] − y[n]; z[n + 1] ← −1
end if
return z[0..n + 1]
```

Dacă s-a ajuns la o astfel de situaţie trebuie ajustată din nou valoarea lui $qq$ şi a lui $x[j-n..j]$ prin adunarea împărţitorului. Ordinul de complexitate al algoritmului `divide` este $\mathcal{O}((m+1)n)$ unde $n$ este numărul de cifre ale împărţitorului iar $m$ este diferenţa dintre numărul de cifre al deîmpărţitului şi numărul de cifre ale împărţitorului.

## 10.2 Algoritmi din geometria computaţională

In domenii precum proiectarea asistată de calculator, robotică sau grafica pe calculator intervine necesitatea rezolvării unor probleme de geometrie. Ramura informaticii care se ocupă cu proiectarea, analiza şi implementarea algoritmilor destinaţi rezolvării problemelor de geometrie este cunoscută sub numele de *geometrie computaţională*. În problemele din geometria computaţională se lucrează cu entităţi de tipul: punct, dreaptă, linie poligonală etc. Majoritatea entităţilor geometrice pot fi reprezentate ca perechi de valori (corespunzând coordonatelor unui punct în raport cu un reper cartezian în plan) sau liste de perechi de valori (corespunzând punctelor care definesc o linie poligonală). În cazul obiectelor tridimensionale punctele sunt reprezentate de triplete de valori.

În continuare vom considera puncte în plan reprezentate prin coordonatele lor carteziene, $P(x,y)$. O mulţime de puncte, $(P_1,\ldots,P_n)$ poate fi reprezen-

**Algoritmul 10.5** Împărţirea a două numere reprezentate prin tablourile cifrelor

```
1: divide(integer x[0..m+n], y[0..n-1], b)
2: integer d, q[0..m], r[0..n-1], qq, p[0..n],z[0..n+1]
3: d ← b DIV (y[n-1]+1)
4: x[0..m+n] ←multiply1(x[0..m+n-1],d,b)
5: y[0..n-1] ←multiply1(y[0..n-1],d,b)
6: for j ← m+n, n, -1 do
7:    if x[j] = y[n-1] then
8:       qq ← b-1
9:    else
10:      qq ← (x[j]*b+x[j-1]) DIV y[n-1]
11:      while y[n-2]*qq > (x[j]*b+x[j-1]-qq*y[n-1])*b+x[j-2] do
12:         qq ← qq-1
13:      end while
14:   end if
15:   p[0..n] ← multiply1(y[0..n-1],qq,b)
16:   z[0..n+1] ← subtract(x[j-n..j],p[0..n],b)
17:   if z[n+1] = -1 then
18:      qq ← qq-1
19:      x[j-n..j] ← add(x[j-n..j],y[0..n-1],b)
20:   end if
      q[j-n] ← qq
21: end for
22: r[0..n-1] ← divide1(x[0..n-1],d,b)
23: return q[0..m], r[0..n-1]
```

tată fie printr-un tablou $P[1..n]$ conţinând perechi corespunzătoare celor două coordonate fie prin două tablouri $x[1..n]$ şi $y[1..n]$ conţinând abscisele respectiv ordonatele punctelor.

### 10.2.1 Determinarea celor mai apropiate două puncte

Problema constă în a determina pentru o mulţime de puncte $\{P_1, \ldots, P_n\}$ distanţa minimă dintre oricare două puncte. Pentru două puncte $P_i$ şi $P_j$ date prin coordonatele $(x_i, y_i)$ şi $(x_j, y_j)$ distanţa se calculează cu formula cunoscută: $d(P_i, P_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$.

Abordarea bazată pe tehnica forţei brute presupune calculul a $n(n-1)/2$ distanţe (între oricare două puncte $P_i$ şi $P_j$ cu $i < j$) şi determinarea minimului acestora. Se ajunge astfel la un algoritm de complexitate $\Theta(n^2)$. Aplicând însă tehnica divizării se poate obţine un algoritm de complexitate $\mathcal{O}(n \lg n)$. Această abordare se bazează pe următoarele prelucrări:

*Pre-sortare.* Se ordonează mulţimea de puncte crescător atât după valoarea abscisei cât şi după valoarea ordonatei. Acest lucru se poate realiza utilizând două tablouri de indici: $X[1..n]$ (conţine indicii elementelor din tabloul de puncte $P[1..n]$ în ordinea crescătoare a abscisei) şi $Y[1..n]$ (conţine indicii elementelor din tabloul de puncte $P[1..n]$ în ordinea crescătoare a ordonatei). Dacă se aplică un algoritm de sortare rapidă etapa de pre-sortare are complexitatea $\mathcal{O}(n \lg n)$.

*Divizare.* Se determină valoarea mediană, $M$, a absciselor (abscisa corespunzătoare elementului $X[\lfloor n/2 \rfloor]$). Se împarte mulţimea curentă de puncte în două submulţimi: $P_S$ (conţine punctele din semiplanul stâng în raport cu verticala de ecuaţie $x = M$, deci având abscisele mai mici sau egale cu valoarea medianei) şi $P_D$ (conţine punctele aflate în semiplanul drept). Se determină $X_S$, $X_D$, $Y_S$ şi $Y_D$, adică vectorii ce conţin pentru fiecare semiplan indicii punctelor în ordinea crescătoare a absciselor respectiv a ordonatelor. Toate aceste prelucrări sunt de complexitate liniară în raport cu numărul de puncte din mulţime.

Procesul de divizare se aplică recursiv până se atinge dimensiunea critică $n = 3$ pentru care se calculeaza direct distanţa minimă.

*Combinare.* Prin rezolvarea subproblemelor corespunzătoare submulţimilor $P_S$ şi $P_D$ se obţin două valori: $d_S$ şi $d_D$, reprezentând distanţele minime la care se află punctele din mulţimile $P_S$ respectiv $P_D$. Fie $d^* = \min\{d_S, d_D\}$. Problema este că $\delta$ nu este neapărat cea mai mică distanţă dintre punctele mulţimii $P = P_S \cup P_D$ întrucât pot exista puncte mai apropiate aflate de o parte şi de alta a verticalei $x = M$. Se pune problema identificării acestor perechi în timp liniar în raport cu numărul de puncte. Observaţia cheie este faptul că astfel de perechi se pot afla doar în regiunea $R$ delimitată de verticalele $x = M - d^*$ şi $x = M + d^*$. Pentru fiecare punct din $R \cap P_S$ ar trebui calculate distanţele faţa de punctele din $R \cap P_D$. Separarea mulţimii în două submulţimi şi regiunea de analizat în faza de combinare sunt ilustrate în Figura 10.1 (stânga).

O altă observaţie importantă este că la analiza punctului $P_i \in R \cap P_S$ este suficient să calculeze distanţele dintre $P_i$ şi punctele din $R \cap P_D$ a căror ordonată este între $y_i - d^*$ şi $y_i + d^*$. Identificarea acestor puncte este uşor de făcut atâta timp cât $Y_S$ conţine indicii punctelor în ordinea crescătoare a ordonatei. În plus numărul maxim de puncte din $P_D$ aflate în dreptunghiul delimitat de $(M, y_i - d^*)$, $(M + d^*, y_i - d^*)$,$(M + d^*, y_i + d^*)$,$(M, y_i + d^*)$ este 6. Fiecare punct $P_i$ din $P_S$ este comparat cu cel mult 6 puncte din $R \cap P_D$ (în Figura 10.1 - dreapta sunt marcate punctele care ar trebui analizate în cel mai rău caz). În felul acesta etapa de combinare are complexitate liniară.

Timpul de execuţie corespunzător etapei bazată pe metoda divizării satisface relaţia de recurenţă:

$$T(n) = \begin{cases} 2T(n/2) + \mathcal{O}(n) & \text{dacă } n > 3 \\ \mathcal{O}(1) & \text{dacă } n \leq 3 \end{cases}$$

Aplicând Teorema master rezultă că $T(n) \in \mathcal{O}(n\lg n)$.

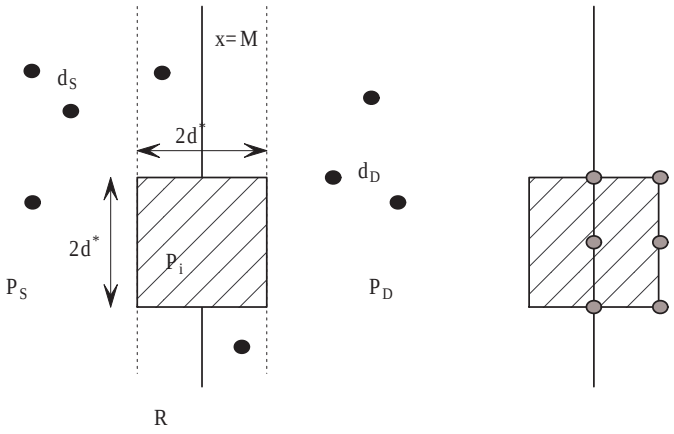


Figura 10.1: Determinarea distanţei minime dintre punctele unei mulţimi

### 10.2.2 Determinarea înfăşurătorii convexe

Un poligon convex are proprietatea că dacă conţine două puncte conţine întreg segmentul de dreaptă determinat de cele două puncte. Înfăşurătoarea convexă, $IC(P)$, a unui set de puncte, $P$, este cel mai mic poligon convex care conţine toate punctele mulţimii (în interior sau pe frontieră). Pentru ca poligonul care conţine punctele să fie cât mai mic este necesar ca vârfurile poligonului să aparţină înfăşurătorii convexe, adică mulţimea de puncte ce reprezintă vârfurile poligonului este inclusă în mulţimea $P$. Acest lucru poate fi uşor motivat intuitiv vizualizând punctele ca fiind cuie înfipte într-o suprafaţă plană şi înfăşurătoarea ca fiind un elastic ce înconjoară toate punctele (Figura 10.2).

Punctele din $P$ care au valori extreme pentru cel puţin una dintre coordonate ($x_{min}$, $x_{max}$, $y_{min}$, $y_{max}$) aparţin întotdeauna înfăşurătorii convexe. Există mai multe variante de determinare a lui $IC(P)$ dar majoritatea se bazează pe a selecta din $P$ punctele ce aparţin lui $IC(P)$ şi de a le returna într-o anumită ordine (de exemplu trigonometrică).

Înainte de a analiza un algoritm de generare a înfăşurătorii convexe să vedem cum se poate ordona o listă de $n$ puncte astfel încât să reprezintă vârfurile unui poligon (nu neapărat convex) dar astfel încât segmentele de dreaptă definite de perechi de puncte consecutive să nu se intersecteze. Acest lucru este echivalent

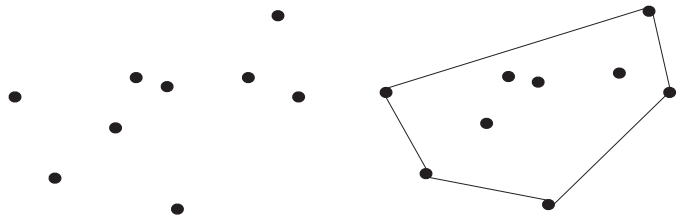

Figura 10.2: Mulţime de puncte în plan şi înfăşuratoarea ei convexă

cu a ordona punctele începând de la unul considerat de referinţă astfel încât ele să fie parcurse în sens trigonometric (în sens invers acelor de ceasornic).

Pentru determinarea ordinii de parcurgere a punctelor în sens trigonometric este important să se poată stabili dacă un punct $P_i$ se află "înainte" sau "după" un alt punct $P_j$ în raport cu un punct de referinţă, $P_1$. O variantă, ilustrată în Figura 10.3, constă în a calcula unghiurile formate de segmentele de forma $\overline{P}_1P_i$ şi semidreapta cu originea in $P_1$ şi paralelă cu semiaxa pozitivă $Ox$. Măsura acestui unghi este $\arctan((y_i - y_1)/(x_i - x_1))$. Dezavantajul acestei variante este faptul că necesită operaţii de împărţire şi utilizarea funcţiei arctan. Există diverse variante care evită aceste calcule [3],[15]. Varianta pe care o vom utiliza în continuare este cea descrisă în [3] şi care se bazează pe noţiunea de *produs încrucişat*. Considerăm punctul $P_1(x_1, y_1)$ ca punct de referinţă (în particular poate să fie originea sistemului de axe de coordonate) şi punctele $P_i(x_i, y_i)$ şi $P_j(x_j, y_j)$. Produsul încrucişat al punctelor $P_i$ şi $P_j$ în raport cu $P_1$ se defineşte prin:

$$P_i \times P_j = (x_i - x_1) \cdot (y_j - y_1) - (x_j - x_1) \cdot (y_i - y_1) \tag{10.2}$$

Dacă $P_i \times P_j > 0$ atunci $P_i$ se află înaintea lui $P_j$ prin parcurgere în sens trigonometric iar dacă $P_i \times P_j < 0$ atunci $P_j$ se află înaintea lui $P_i$ (aceasta este echivalent cu faptul că $P_j \times P_i = -P_i \times P_j > 0$). Aceste situaţii sunt ilustrate în figura 10.4. Dacă produsul este 0 atunci $P_1$, $P_i$ şi $P_j$ sunt coliniare.

Pentru a determina ordinea de parcurgere în sens trigonometric a punctelor este suficient să se ordoneze astfel încât $P_i \times P_{i+1} > 0$ pentru fiecare $i \in \{0, \ldots, n-1\}$. În cazul în care $P_i \times P_{i+1} = 0$ se utilizează, ca al doilea criteriu de sortare, valoarea ordonatei (punctul cu valoarea mai mică a ordonatei este înaintea punctului cu ordonata mai mare). În cazul în care ordonarea punctelor se face cu scopul de a determina înfăşurătoarea convexă dintre punctele cu proprietatea $P_i \times P_{i+1} = 0$ se păstrează doar cel aflat la distanţa cea mai mare faţă de $P_1$. În felul acesta o parte dintre punctele din setul iniţial pot fi eliminate. În continuare vom considera că au rămas punctele $P_1$,$P_2$,...,$P_m$ care au proprietatea că $P_i \times P_{i+1} > 0$ pentru fiecare $i \in \{1, \ldots, m\}$.

Construirea înfăşurătorii convexe constă în parcurgerea punctelor în sens trigonometric începând cu $P_1$. Este uşor de văzut că pe lângă $P_1$ vor face parte

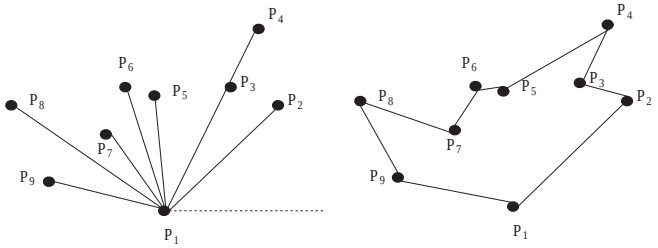


Figura 10.3: Ordonarea punctelor în sens trigonometric pornind de la punctul de ordonată minimă şi linia poligonală determinată de puncte

.

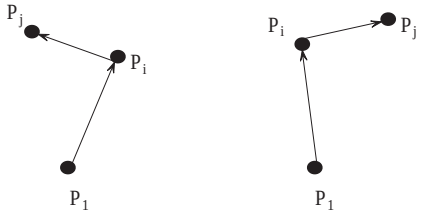


Figura 10.4: Poziţia a două puncte în raport cu punctul de referinţa $P_1$. Stânga: $P_i$ se află înaintea lui $P_j$. Dreapta: $P_j$ se află înaintea lui $P_i$.

din înfăşurătoarea convexă întotdeauna $P_2$ şi $P_m$. Ideea algoritmului, cunoscut sub numele de baleiere de tip Graham ("Graham scan") este de a folosi o *stivă* care să permită revenirea la punctele deja analizate dacă se dovedeşte că ele trebuie eliminate din înfăşurătoarea convexă. Stiva este o structură de date simplă care permite adăugarea secvenţială a unor elemente şi extragerea lor în ordinea inversă inserării (ultimul element inserat este primul element ce poate fi extras). O implementare simplă se bazează pe utilizarea unui tablou şi reţinerea indicelui ultimului element adăugat la tablou. Acest indice indică spre aşa numitul vârf al stivei.

Iniţial în stivă se introduc punctele $P_1$, $P_2$ şi $P_3$. Primele două vor rămâne tot timpul în stivă. În continuare pentru fiecare punct $P_i$, $i > 3$ din mulţimea ordonată în sens trigonometric se verifică dacă unghiul format de punctul aflat pe penultima poziţie în stivă, ultimul punct din stivă (vârful) şi $P_i$ este orientat înspre stânga. Dacă această proprietatea nu este satisfăcută atunci elementul din vârful stivei este eliminat. Procesul de verificare continuă pentru $P_i$ şi cele două elemente curente din vârful stivei pâna când se ajunge la un triplet de puncte ce satisface proprietatea. Dacă notăm cu $P_j$ punctul aflat în vârful

stivei şi cu $P_k$ penultimul punct din stivă atunci condiţia care trebuie satisfăcută pentru a nu elimina $P_j$ din stivă este: $(x_i - x_k) \cdot (y_j - y_k) - (x_j - x_k) \cdot (y_i - y_k) > 0$.

Algoritmul Graham pentru determinarea înfăşurătorii convexe este descris în 10.6. Algoritmul primeşte ca intrare un tablou $P[1..n]$ având ca elemente perechi de coordonate şi construieşte tabloul $S[1..v]$ ce va conţine înfăşurătoarea convexă. Funcţia `sortare_trigonometric` realizează ordonarea în sens trigonometric a punctelor şi elimină punctele despre care se poate decide că nu fac parte din $IC(P)$. Funcţia `stanga` verifică dacă e satisfăcută condiţia $(x_i - x_k) \cdot (y_j - y_k) - (x_j - x_k) \cdot (y_i - y_k) > 0$ unde $(x_i, y_i)$ reprezintă coordonatele lui $P_i$, $(x_k, y_k)$ reprezintă coordonatele lui $S[v-1]$ iar $(x_j, y_j)$ reprezintă coordonatele lui $S[v]$.

**Algoritmul 10.6** Determinarea înfăşurătorii convexe prin baleiere de tip Graham

```
Graham(P[1..n])
P[1..m] ←sortare_trigonometric(P[1..n])
S[1] ← P[1]; S[2] ← P[2]; S[3] ← P[3]; v ← 3
for i ← 4, m do
  while stanga(S[v − 1],S[v],P[i])=false do
    v ← v − 1 // eliminare din stivă
  end while
  v ← v + 1; S[v] ← P[i]
end for
return S[1..v]
```

Întrucât un punct este cel mult o dată introdus în stivă rezultă că numărul total de apeluri ale funcţiei `stanga` (prelucrarea cea mai costisitoare din ciclul **for**) este liniar în raport cu numărul de puncte. Prin urmare complexitatea algoritmului `Graham` este determinată de sortarea punctelor în sens trigonometric care poate fi realizată în $\mathcal{O}(n \lg n)$. O altă variantă este cea bazată pe parcurgerea de tip Jarvis cărei complexitate este $\mathcal{O}(nv)$ unde $v$ este numărul de vârfuri ale înfăşurătorii. Pentru detalii se poate consulta [3].

## 10.3 Algoritmi de căutare în şiruri de caractere

Căutarea unei subsecvenţe într-o secvenţă este o prelucrarea frecvent întâlnită în procesarea textelor (unde secvenţa este un text iar subsecvenţa poate fi un cuvânt). Secvenţa în care se caută este din acest motiv frecvent denumită *text* iar subsecvenţa *cuvânt* sau *şablon*. Vom considera în continuare că secvenţa în care se caută este reprezentată printr-un tablou $t[0..n-1]$ iar subsecvenţa căutată printr-un tablou $p[0..m-1]$. Se pune astfel problema să se verifice dacă şablonul este sau nu prezent în text iar în caz afirmativ să se determine prima poziţie unde este plasat.

Cea mai simplă variantă este cea bazată pe metoda forţei brute şi constă în a compara succesiv subsecvenţa $t[i..i+m-1]$ cu şablonul $p[0..m]$ pentru fiecare $i = \overline{0, n-m-1}$. Aceasta înseamna că textul este baleiat folosind şablonul prin mutarea acestuia din urmă cu câte o poziţie la dreapta la fiecare etapă.

În cazul cel mai defavorabil numărul de comparaţii efectuate este $m(n-m+1)$. Cel mai defavorabil caz este destul de rar întâlnit în practică întrucât presupune ca şablonul să se potrivească până la penultimul caracter. Un astfel de exemplu este cel în care textul este "aaaaaaaaat" iar şablonul este "aaat". Algoritmul forţei brute este acceptabil în practică atâta timp cât lungimea şablonului este suficient de mică. În cazul în care ordinul $\mathcal{O}(mn)$ este inacceptabil se poate folosi unul dintre algoritmii de complexitate liniară ($\mathcal{O}(m+n)$) existenţi. Doi dintre aceşti algoritmi sunt Knuth-Morris-Pratt, respectiv Boyer-Moore-Horspool. Ambii algoritmi se bazează pe o analiză preliminară a şablonului care permite ca în cazul identificării unei nepotriviri şablonul să poată fi deplasat cu mai mult de o poziţie la dreapta.

În ambele situaţii parcurgerea textului se face de la stânga la dreapta dar compararea şablonului se face diferit în cazul celor doi algoritmi: de la stânga la dreapta în cazul algoritmului Knuth-Morris-Pratt, şi de la dreapta la stânga în cazul algoritmului Boyer-Moore-Horspool.

### 10.3.1 Algoritmul Knuth-Morris-Pratt

In cazul aplicării forţei brute, dacă şablonul $p[0..m-1]$ nu coincide în totalitate cu porţiunea de text cu care se compară, $t[i..i+m-1]$, atunci se trece la a compara şablonul cu porţiunea $t[i+1..i+m]$, fără a se ţine cont câte dintre poziţiile din şablon coincid cu cele din porţiunea deja analizată a textului. Să presupunem că primele $k+1$ ($k \in \{0, \ldots, m-2\}$) poziţii din şablon coincid cu textul, adică $p_j = t_{i+j}$ pentru $j \in \{0, \ldots, k\}$ şi $p_{k+1} \neq t_{i+k+1}$.

Întrucât porţiunea $t[i+1..i+k+1]$ a fost deja analizată s-ar putea utiliza această informaţie pentru a deplasa înspre dreapta şablonul cu mai mult de o poziţie. Să considerăm că numărul de poziţii cu care se realizează deplasarea este $s$. Valoarea lui $s$ trebuie aleasă astfel încât să permită identificarea primei potriviri potenţiale. Aceasta înseamnă că cel puţin $p[0..k-s] = t[i+s..i+k]$. Însă se ştie deja că $t[i+s..i+k] = p[s..k]$, prin urmare trebuie ca $p[0..k-s] = p[s..k]$ (vezi Tabelul 10.1). Aceasta înseamnă că pentru fiecare fiecare poziţie $k$ a şablonului deplasarea maximă care garantează faptul că nu se pierde nici o ocurenţă a şablonului este:

$$d(k) = \min\{s > 0 | p[0..k-s] = p[s..k]\}, \quad k = \overline{0, m-1} \tag{10.3}$$

În cazul în care în subşablonul $p[0..k]$ nu există o porţiune iniţială care să coincidă cu porţiunea finală, valoarea lui $d(k)$ este $k+1$ (în acest caz atât secvenţa $p[0..k-s]$ cât şi $p[s..k]$ sunt vide deci coincid). Acest lucru se întâmplă întotdeauna când elementele şablonului sunt distincte astfel că într-o astfel de

situaţie şablonul se deplasează până este aliniat cu elementul din text pentru care a fost detectată neconcordanţa.

| | | $p_0$ | $\dots$ | $p_s$ | $p_{s+1}$ | $\dots$ | $p_k$ | $p_{k+1}$ | $\dots$ | $p_{m-1}$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | $\shortparallel$ | $\dots$ | $\shortparallel$ | $\shortparallel$ | $\dots$ | $\shortparallel$ | $\not\shortparallel$ | | | |
| $t_0$ | $\dots$ | $t_i$ | $\dots$ | $t_{i+s}$ | $t_{i+s+1}$ | $\dots$ | $t_{i+k}$ | $t_{i+k+1}$ | $\dots$ | $\dots$ | $\dots$ |
| | | | | $p_0$ | $p_1$ | $\dots$ | $p_{k-s}$ | $p_{k-s+1}$ | $\dots$ | $\dots$ | $p_{m-1}$ |

Tabelul 10.1: Baleierea textului folosind şablonul

Să considerăm "textul": GATTAATCGATTGA şi şablonul GATTGA. În acest caz valorile deplasamentelor corespunzătoare şablonului şi "parcurgerea textului cu sablonul sunt descrise în tabelul 10.2. Se compară $t[0..5]$ cu $p[0..5]$ şi se detectează prima neconcordanţă pe poziţia $k+1=5$. Aceasta înseamnă că şablonul poate fi deplasat înspre dreapta cu $k=4$ poziţii. Se ajunge astfel la a compara $t[4..9]$ cu $p[0..5]$ detectându-se prima neconcordanţă pe poziţia $k+1=1$ în şablon. Cum $d[0]=1$ şablonul se deplasează cu o poziţie şi se compară şablonul cu $t[5..10]$. În acest caz prima neconcordanţă de detectează pe poziţia $k+1=3$ a şablonului. Deplasamentul corespunzător este $d[2]=3$ deci şablonul de deplasează cu 3 poziţii ajungându-se să se compare şablonul cu $t[8.13]$. Neconcordanţa este chiar pe prima poziţie, deci $k+1=0$ ceea ce conduce la $k=-1$ deci la necesitatea de a avea valoare in tabelul de deplasamente corespunzătoare indicelui $-1$. In acest caz valoarea deplasamentului este întotdeauna egală cu 1 astfel că se deplasează şablonul cu o singură poziţie detectându-se în final concordanţa dintre şablon şi $t[9..14]$.

| | | G | A | T | T | G | A |
|---|---|---|---|---|---|---|---|
| $k$ | -1 | 0 | 1 | 2 | 3 | 4 | 5 |
| $d(k)$ | 1 | 1 | 2 | 3 | 4 | 4 | 4 |

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| G | A | T | T | G | G | A | T | C | G | A | T | T | G | A |
| G | A | T | T | G | A | | | | | | | | | |
| | | | | G | A | T | T | G | A | | | | | |
| | | | | | G | A | T | T | G | A | | | | |
| | | | | | | | | G | A | T | T | G | A | |
| | | | | | | | | | G | A | T | T | G | A |

Tabelul 10.2: Exemplu de aplicare a algoritmului Knuth-Morris-Pratt

Dacă considerăm deplasamentele calculate conform relaţiei 10.3 ca fiind stocate în tabloul $d[-1..m-1]$ algoritmul poate fi Knuth-Morris-Pratt poate fi descris ca în 10.7.

Corectitudinea algoritmului `KMP` poate fi demonstrată folosind ca invariant

**Algoritmul 10.7** Algoritmul Knuth-Morris-Pratt

```
KMP(t[0..n − 1], p[0..m − 1])
integer d[−1..m − 1]
d[−1..m − 1] ← calcul_deplasamente(p[0..m − 1])
i ← 0; j ← 0
while i + m ≤ n do
  while t[i + j] = p[j] do
    j ← j + 1
    if j ≥ m then
      return i
    end if
  end while
  i ← i + d[j − 1]
  j = max{j − d[j − 1], 0}
end while
return −1
```

relaţiile:

$$p[0..j-1] = t[i..i+j-1] \text{ şi } p[0..m-1] \neq t[k..k+m-1], k \in \{0, \ldots, i-1\} \quad (10.4)$$

Prima relaţie specifică faptul că variabila $j$ exprimă numărul de concordanţe dintre şablon şi text, iar a doua relaţie faptul că nu a fost identificată o potrivire anterioară a şablonului cu textul.

Deşi algoritmul conţine două cicluri suprapuse, unul care se execută de maxim $n - m + 1$ ori, iar celălalt care se execută de maxim $m$ ori se poate arăta că ordinul de complexitate nu este $\mathcal{O}(mn)$ întrucât numărul maxim de repetări ale celor două cicluri nu se realizează simultan. Cu cât se fac mai multe comparaţii în ciclul interior cu atât este mai mare valoarea deplasamentului, adică pasul cu care se modifică $i$. Cum $i \in \{0, \ldots, n-m\}$ şi $j \in \{0, \ldots, m-1\}$ sunt corelate este util să se analizeze comportarea lui $i + j$. În cazul unui ciclu dublu de tipul **for** $i \leftarrow 0, n - m$ **do for** $j \leftarrow 0, m - 1$ **do** ..., şirul sumelor $i + j$ obţinute pe parcursul execuţiei ciclurilor nu este crescător (pentru $m = 3$ valorile corespunzătoare sunt 0, 1, 2, 1, 2, 3, 2, 3, 4 etc). În cazul algoritmului `KMP` însă şirul valorilor $i+j$ este crescător, însă nu neapărat strict crescător. În schimb dacă se analizează şirul valorilor lui $2i+j$ se observă că la fiecare evaluare a condiţiei corespunzătoare ciclului `while` interior valoarea corespunzătoare creşte fie cu 1 (în cazul în care condiţia e adevărată) fie cu $d[j-1] > 0$ (dacă condiţia e falsă). Această din urmă afirmaţie este adevărată întrucât $i$ creşte cu $d[j-1]$, $j$ scade cu $d[j-1]$ astfel că $2i + j$ va creşte cu $d[j-1]$. Cum $2i+j$ este mai mic decât $2n+m$ şi cum la fiecare comparaţie a unui element al textului cu un element al şablonului, $2i + j$ este mărit cu un increment nenul rezultă că numărul de comparaţii este cel mult $2m + n$. Deci algoritmul `KMP`

**Algoritmul 10.8** Completarea tabelului cu deplasamente pentru algoritmul Knuth-Morris-Pratt

```
calcul_deplasamente(p[0..m − 1])
integer d[−1..m − 1]
d[−1] ← 1; d[0] ← 1
i ← 1; j ← 0
while i + j < m do
  if p[i + j] = p[j] then
    d[i + j] ← i; j ← j + 1
  else
    if j = 0 then
      d[i] ← i + 1
    end if
    i ← i + d[j − 1]
    j ← max{j − d[j − 1], 0}
  end if
end while
return d[−1..m − 1]
```

este de complexitate $\mathcal{O}(m + n)$. Rămâne de văzut cum poate fi completat tabloul cu deplasamente $d[-1..m-1]$ în timp liniar în raport cu dimensiunea şablonului. Ideea cheie este de a folosi aceeaşi strategie ca la `KMP` cu diferenţa că şablonul este comparat în raport cu el însuşi şi se folosesc ca deplasamente valorile completate deja. Algoritmul de completare este descris în 10.8.

### 10.3.2 Algoritmul Boyer-Moore-Horspool

În cazul algoritmului Knuth-Morris-Pratt compararea şablonului cu porţiunea de text se face de la stânga la dreapta. Cu cât neconcordanţa este detectată mai aproape de sfârşitul şablonului cu atât se efectuează mai multe comparaţii dar şi deplasarea şablonului în text poate fi mai mare. Acest lucru presupune însă compararea tuturor elementelor până la acea poziţie în şablon. Neconcordanţa ar putea fi detectată mai repede dacă s-ar parcurge şablonul dinspre dreapta înspre stânga. Aceasta este ideea de bază a algoritmului Boyer-Moore. Atunci când se compară şablonul $p[0..m-1]$ cu secvenţa de text $t[i.i + m - 1]$ se porneşte de la a compara elementul $p[m-1]$ cu $t[i + m - 1]$. Să presupunem că $j$ este prima poziţie (întâlnită la parcurgerea dinspre dreapta înspre stânga) din şablon pentru care $p[j] \neq t[i + j]$. Procesul de comparare se opreşte şi se pune problema cu câte poziţii se poate deplasa şablonul înspre dreapta astfel încât să nu se piardă nici o ocurenţă a şablonului. Există mai multe variante pentru calculul deplasamentului.

O primă variantă se bazează pe ideea de a deplasa şablonul astfel încât cea mai din dreapta ocurenţă a simbolului $t[i + j]$ din şablon să fie aliniată

cu poziţia $i + j$ a textului. Dacă simbolul $t[i + j]$ nu este prezent în şablon atunci se deplasează şablonul astfel încât $p[0]$ să fie aliniat cu elementul de pe poziţia $i + j + 1$ din text. Să considerăm textul $t[0..9]$ ="GATCGATTCA" şi şablonul $p[0..5]$ ="GATTCA". Comparând $p$ cu $t[0..5]$ de la dreapta spre stânga prima discordanţă se detectează între $p[4]$ şi $t[4]$. Întrucât simbolul $G$ se află în şablon pe poziţia 0 deplasarea şablonului se va face cu 4 poziţii pentru a asigura alinierea lui $t[4]$ cu $p[0]$. Dacă şablonul este "GATTGA" atunci prima discordanţă este între $p[3]$ şi $t[3]$ iar cum simbolul "C" nu se află în şablon acesta se va deplasa până când $p[0]$ se aliniază cu $t[4]$. Aceste două cazuri sunt ilustrate în Tabelul 10.3 (a) şi (b). Dacă s-ar aplica aceeaşi strategie şi pentru şablonul "GCTCGA" (cazul (c) în Tabelul 10.3) s-ar ajunge la situaţia în care şablonul ar trebui deplasat înspre stânga. Pentru a evita astfel de situaţii se poate stabili un deplasament egal cu 1 ori de câte ori s-ar ajunge la o valoare negativă.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| G | A | T | C | G | A | T | T | C | A |
| G | A | T | T | C | A | | | | |
| | | | | G | A | T | T | C | A |

(a)

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| G | A | T | C | G | A | T | T | C | A |
| G | A | T | T | G | A | | | | |
| | | | | G | A | T | T | G | A |

(b)

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| G | A | T | C | G | A | T | T | C | A |
| G | C | T | C | G | A | | | | |
| | G | C | T | C | G | A | | | |

(c)

Tabelul 10.3: Exemple de deplasare a şablonului faţă de text când este detectată o discordanţă pe poziţia $i + j$ în text respectiv poziţia $j$ în şablon. (a) Simbolul $t[i + j]$ se află în şablon iar cea mai din dreapta ocurenţă a sa se află în subşablonul $p[0..j - 1]$; (b) Simbolul $t[i + j]$ nu se află în şablon; (c) Simbolul $t[i + j]$ se află în şablon iar cea mai din dreapta ocurenţă a sa se află în subşablonul $p[j + 1..m - 1]$

Cea de-a doua variantă (specifică algoritmului Horspool) se bazează pe ideea de a realiza alinierea nu în raport cu elementul din text pentru care s-a descoperit prima neconcordanţă ci cu elementul din text aliniat în etapa curentă cu ultimul element din tablou ($p[m - 1]$). Aplicând această idee pentru exemplul din Tabelul 10.3 (c) şi observând că în şablon nu există o altă ocurenţă a simbolului "A" în afara celei de pe ultima poziţie rezultă că şablonul se poate deplasa cu $m = 6$ poziţii. Prin urmare ideea specifică algoritmului Horspool

**Algoritmul 10.9** Algoritmul Boyer-Moore-Horspool

```
KMP(t[0..n − 1], p[0..m − 1])
integer d[1..s]
// calcul deplasamente pentru toate elementele alfabetului
for i ← 1, s do
  d[i] ← m
end for
for i ← 0, m − 2 do
  d[ord(p[i])] ← m − 1 − k
end for
i ← 0
while i + m ≤ n do
  j ← m − 1
  while t[i + j] = p[j] do
    j ← j − 1
    if j < 0 then
      return i
    end if
  end while
  i ← i + d[ord(t[i + m − 1])]
end while
return −1
```

este de a utiliza o funcţie de calcul a deplasamentelor care asociază fiecărui element din alfabetul specific textului o valoare după regula:

$$d(c) = \begin{cases} m - 1 - \max\{0 \le i < m - 1 | p[i] = c\} & \text{dacă } c \text{ este în } p[0..m-2] \\ m & \text{altfel} \end{cases} \tag{10.5}$$

În cazul exemplului din Tabelul 10.3(c) alfabetul utilizat este $\{A, C, G, T\}$ iar valorile deplasamentului corespunzătoare celor 4 simboluri din alfabet şi şablonului $p[0..5] = GCTCGA$ sunt: $d(A) = 6$, $d(C) = 5 - 3 = 2$, $d(G) = 5 - 4 = 1$, $d(T) = 5 - 2 = 3$.

Algoritmul Boyer-Moore-Horspool este descris în 10.9. În prima parte a algoritmului se construieşte tabloul cu deplasamente $d[1..s]$ unde $s$ reprezintă numărul de simboluri din alfabet iar $ord(c) \in \{1, \ldots, s\}$ este numărul de ordine al simbolului $c$ din alfabet.

Pentru un alfabet fixat, costul construirii tabloului cu deplasamente este de ordinul $\Theta(m + s)$. În cel mai defavorabil caz (când la fiecare etapă se efectuează deplasarea şablonului cu o poziţie) căutarea propriu-zisă are ordinul de complexitate $mn$ dar în cazul cel mai favorabil ordinul este $n/m$. In ciuda faptului că în cazul cel mai defavorabil ordinul de complexitate este acelaşi

cu cel al algoritmului bazat pe metoda forţei brute, algoritmul Boyer-Moore-Hoorspool realizează un bun compromis între simplitate şi eficienţă. Pentru detalii suplimentare privind metodele de căutare în şiruri de caractere pot fi consultate [1], [3], [7] sau [11].

# Bibliografie


[1] S. Baase; Computer Algorithms. Introduction to Design and Analysis, Addison Wesley Publishing Company, 2nd edition, 1993

[2] M. de Berg, M. van Kreveld, M. Overmars, O. Schwarzkopf; Computational Geometry. Algorithms and Applications, Springer, 2000.

[3] T.H. Cormen, C.E. Leiserson, R.L. Rivest and C. Stein; Introduction to Algorithms, MIT Press, 2nd edition, 2001.

[4] V.I. Creţu; Structuri de date şi algoritmi, vol. 1, Ed. Orizonturi Universitare, Timişoara, 2000.

[5] J. Edmonds; How to think about Algorithms, Cambridge University Press, 2008.

[6] C.A. Giumale; Introducere în analiza algoritmilor. Teorie şi aplicaţie, Ed. Polirom, 2004.

[7] R. Johnsonbaugh and M. Schaefer; Algorithms, Pearson Education, 2004.

[8] D.E. Knuth; The Art of Computer Programming. Fundamental Algorithms, Addison Wesley, 2nd edition, 1969.

[9] D.E. Knuth; The Art of Computer Programming. Seminumerical Algorithms, Addison Wesley, 2nd edition, 1981.

[10] D.E. Knuth; The Art of Computer Programming. Sorting and Searching, Addison Wesley, 1973.

[11] A. Levitin; Introduction to the Design and Analysis of Algorithms, Addison Wesley Publishing Company, 2003.

[12] D. Lucanu, M. Craus; Proiectarea algoritmilor, Ed. Polirom, 2008.

[13] I. Parberry and W. Gasarch; Problems on Algorithms, Prentice Hall, 2nd edition, 2002.

[14] G. Polya; How to Solve It, 2nd ed., Princeton University Press, 1957.

[15] R. Sedgewick; Algorithms, Addison Wesley, 1983.

[16] S.S. Skiena; The Algorithm Design Manual, Springer-Verlag, 1997.

# Index


Înmulţirea optimală a unui şir de matrici, 179
Şirul lui Fibonacci, 174

Algoritmul
  Boyer-Moore-Horspool, 235
  Euclid, 22, 31, 48, 56, 113
  Knuth-Morris-Pratt, 232
  Warshall, 186

Căutare
  binară, 122
  secvenţială, 63
Codul Gray, 128

Distanţa de editare, 193

Generarea permutărilor
  în ordine lexicografică, 124
  algoritmul Johnson-Trotter, 125
  tehnica backtracking, 203
  tehnica reducerii, 119
Generarea submulţimilor unei mulţimi
  tehnica backtracking, 203
Generarea submulţimilori unei mulţimi
  tehnica reducerii, 127

Invariant, 46

Memoizare, 184
Metoda
  bisecţiei, 134
  Strassen, 151
  substituţiei, 116

Postcondiţii, 42
Precondiţii, 42
Problema
  împachetării, 169
  închiderii tranzitive, 186
  înfăşurătorii convexe, 228
  acoperirii optimale, 171
  amplasării damelor, 206
  celui mai lung subşir comun, 187
  celui mai lung subşir crescător, 176
  clătitelor, 32
  colorării hărţilor, 207
  comis-voiajorului, 215
  determinării tuturor traseelor, 209
  labirintului, 217
  monedelor, 161, 190
  planificării activităţilor, 168
  rucsacului, 165, 182
  selecţiei, 150
  selectării activităţilor, 167
  stocării optimale, 170
  submulţimii de sumă dată, 159, 191, 214
  turistului în Manhattan, 197
  turnurilor din Hanoi, 120
Proprietatea
  alegerii lacome, 162
  substructurii optime, 163, 173

Regula
  funcţiilor netede, 118

Sirul lui Fibonacci, 38
Sortare
  de tip Shell, 93
  Inversarea secvenţelor sufix, 25

prin inserţie, 91
prin interclasare, 140
prin interschimbarea elementelor vecine, 97
prin numărare, 105
prin selecţie, 95
rapidă, 144

Teorema master, 139
Triunghiul lui Pascal, 176

# Cuprins


**1 Introducere** **9**
- 1.1 Rezolvarea algoritmică a problemelor . . . 9
  - 1.1.1 Proprietăţi ale algoritmilor . . . 11
  - 1.1.2 Tipuri de date . . . 13
  - 1.1.3 Tipuri de prelucrări . . . 15
- 1.2 Descrierea algoritmilor . . . 16
  - 1.2.1 Elementele pseudocodului utilizat . . . 16
  - 1.2.2 Specificarea datelor . . . 19
  - 1.2.3 Tehnica rafinării succesive şi subalgoritmi . . . 19
  - 1.2.4 Exemple . . . 20
- 1.3 Clase de probleme si tehnici de rezolvare . . . 27
  - 1.3.1 Clase de probleme . . . 27
  - 1.3.2 Tehnici de rezolvare . . . 28
- 1.4 Probleme . . . 29

**2 Verificarea corectitudinii algoritmilor** **41**
- 2.1 Etapele verificării corectitudinii . . . 41
- 2.2 Elemente de analiză formală a corectitudinii . . . 43
- 2.3 Probleme . . . 49

**3 Analiza complexităţii algoritmilor** **59**
- 3.1 Timp de execuţie . . . 60
  - 3.1.1 Analiza cazurilor extreme . . . 64
  - 3.1.2 Analiza cazului mediu . . . 65
- 3.2 Ordin de creştere . . . 67
- 3.3 Notaţii asimptotice . . . 68
  - 3.3.1 Notaţia $\Theta$ . . . 68
  - 3.3.2 Notaţia $\mathcal{O}$ . . . 70
  - 3.3.3 Notaţia $\Omega$ . . . 71
  - 3.3.4 Analiza asimptotică a principalelor structuri de prelucrare 72
  - 3.3.5 Clase de complexitate . . . 73
- 3.4 Analiza empirică . . . 74

3.5 Analiza amortizată . . . . . 76
3.6 Probleme . . . . . 78

**4 Analiza algoritmilor de sortare 89**
4.1 Problematica sortării . . . . . 89
4.2 Sortare prin inserţie . . . . . 91
4.2.1 Principiu . . . . . 91
4.2.2 Verificarea corectitudinii . . . . . 91
4.2.3 Analiza complexităţii . . . . . 92
4.2.4 Proprietăţi ale sortării prin inserţie . . . . . 93
4.2.5 Sortare prin inserţie cu pas variabil . . . . . 93
4.3 Sortare prin selecţie . . . . . 95
4.3.1 Principiu . . . . . 95
4.3.2 Verificarea corectitudinii . . . . . 96
4.3.3 Analiza complexităţii . . . . . 97
4.3.4 Proprietăţi ale sortării prin selecţie . . . . . 97
4.4 Sortare prin interschimbarea elementelor vecine . . . . . 97
4.4.1 Principiu . . . . . 97
4.4.2 Verificarea corectitudinii . . . . . 98
4.4.3 Analiza complexităţii . . . . . 99
4.4.4 Variante ale sortării prin interschimbarea elementelor vecine 99
4.5 Limite ale eficienţei . . . . . 101
4.6 Sortarea folosind structura de tip "heap" . . . . . 102
4.7 Sortare fără comparare . . . . . 105
4.7.1 Sortarea folosind tabel de frecvenţe . . . . . 105
4.7.2 Sortarea pe baza cifrelor . . . . . 106
4.8 Probleme . . . . . 107

**5 Tehnica reducerii 111**
5.1 Motivaţie . . . . . 111
5.2 Analiza algoritmilor recursivi . . . . . 113
5.2.1 Arbori de apel . . . . . 114
5.2.2 Verificarea corectitudinii algoritmilor recursivi . . . . . 114
5.2.3 Analiza complexităţii algoritmilor recursivi . . . . . 115
5.3 Principiul tehnicii reducerii . . . . . 116
5.4 Aplicaţii . . . . . 119
5.4.1 Generarea permutărilor . . . . . 119
5.4.2 Problema turnurilor din Hanoi . . . . . 120
5.4.3 Căutare binară . . . . . 122
5.5 Probleme . . . . . 124

**6 Tehnica divizării** **137**
6.1 Principiul tehnicii divizării . . . . . 137
6.2 Analiza complexităţii . . . . . 139
6.3 Algoritmi de sortare bazaţi pe tehnica divizării . . . . . 140
6.3.1 Sortare prin interclasare . . . . . 140
6.3.2 Sortare rapidă . . . . . 144
6.4 Alte aplicaţii ale tehnicii divizării . . . . . 150
6.4.1 Problema selecţiei . . . . . 150
6.4.2 Înmulţirea matricilor . . . . . 151
6.4.3 Înmulţirea polinoamelor . . . . . 152
6.5 Probleme . . . . . 155

**7 Tehnica alegerii local optimale** **159**
7.1 Principiul tehnicii . . . . . 160
7.2 Verificarea corectitudinii şi analiza complexităţii . . . . . 162
7.2.1 Verificarea corectitudinii . . . . . 162
7.2.2 Analiza complexităţii . . . . . 164
7.3 Aplicaţii . . . . . 165
7.3.1 Varianta continuă a problemei rucsacului . . . . . 165
7.3.2 Problema selectării activităţilor . . . . . 167
7.3.3 Problema planificării activităţilor . . . . . 168
7.3.4 Problema împachetării . . . . . 169
7.4 Probleme . . . . . 170

**8 Tehnica programării dinamice** **173**
8.1 Principiul tehnicii . . . . . 173
8.2 Aplicaţii . . . . . 178
8.2.1 Înmulţirea optimală a unui şir de matrici . . . . . 178
8.2.2 Varianta discretă a problemei rucsacului . . . . . 182
8.2.3 Tehnica funcţiilor de memorie . . . . . 184
8.2.4 Problema închiderii tranzitive . . . . . 185
8.3 Probleme . . . . . 186

**9 Tehnici de parcurgere a spaţiului soluţiilor** **199**
9.1 Principiul căutării cu revenire . . . . . 200
9.1.1 Varianta recursivă a algoritmului . . . . . 203
9.2 Aplicaţii ale tehnicii căutării cu revenire . . . . . 203
9.2.1 Generarea tuturor submulţimilor unei mulţimi . . . . . 203
9.2.2 Amplasarea damelor pe tabla de şah . . . . . 206
9.2.3 Colorarea hărţilor . . . . . 207
9.2.4 Determinarea tuturor drumurilor dintre două oraşe . . . 209
9.3 Tehnica căutării de tip "ramifică şi mărgineşte" . . . . . 210
9.4 Probleme . . . . . 214

**10 Algoritmi specifici unor clase de probleme** **221**
10.1 Operaţii cu numere mari . . . . . 221
10.1.1 Adunarea a două numere mari . . . . . 222
10.1.2 Înmulţirea a două numere mari . . . . . 222
10.1.3 Împărţirea a două numere mari . . . . . 223
10.2 Algoritmi din geometria computaţională . . . . . 225
10.2.1 Determinarea celor mai apropiate două puncte . . . . . 226
10.2.2 Determinarea înfăşurătorii convexe . . . . . 228
10.3 Algoritmi de căutare în şiruri de caractere . . . . . 231
10.3.1 Algoritmul Knuth-Morris-Pratt . . . . . 232
10.3.2 Algoritmul Boyer-Moore-Horspool . . . . . 235

Consilier editorial: Nina Ceranu
Lector: prof.dr. Dana Petcu
Apărut sub egida Fundaţiei Culturale "Orient Latin", Timişoara