HANGANU MICȘUNICA

EFICIENȚA ALGORITMILOR ÎN REZOLVAREA PROBLEMELOR DE MATEMATICĂ

GHID METODIC

EDITURA EVOMIND

EFICIENȚA ALGORITMILOR ÎN REZOLVAREA PROBLEMELOR DE MATEMATICĂ

Copyright © 2021
Autor: HANGANU MICȘUNICA

Toate drepturile rezervate.

ISBN 978-606-9734-07-0

Editura Evomind, 2021

https://evomind.org/

INTRODUCERE

Lucrarea încearcă să abordeze una din problemele de programare cele mai importante atât din punct de vedere teoretic cât şi practic, şi anume rearanjarea articolelor într-o ordine crescătoare sau descrescătoare. Ce dificilă ar fi consultarea unui dicţionar , dacă nu ar avea cuvintele aranjate în ordine alfabetică. Analog, ordinea articolelor din memoria calculatorului are implicaţii majore asupra vitezei şi simplităţii algoritmilor care le prelucrează. Chiar dacă dicţionarele definesc sortarea ca pe un proces de separare şi aranjare al lucrărilor după clase şi fel, uzual, programatorii de calculatoare folosesc cuvântul sortare în sens de aranjare a lucrurilor într-o ordine ascendentă sau descendentă, S-ar putea utiliza şi termenul de ordonare, dar datorită înţelesurilor diferite ataşate acestuia se preferă termenul de sortare. Şi în matematică termenul ordine are multiple sensuri (relaţii de ordine, ordinul unei permutări, al unui grup, etc). Nu voi insista asupra multelor aplicaţii ale sortării, dar vreau să menţionez că fabricanţii de calculatoare estimează că peste 25% din timpul de rulare al calculatoarelor este ocupat de sortare în condiţiile în care toate solicitările sunt luate în considerare. Evident că uneori acest timp de rulare creşte mult dacă nu se utilizează algoritmi eficienţi de sortare. Algoritmii ingenioşi de sortare ce au fost descoperiţi şi multitudinea de probleme fascinante nerezolvate în acest domeniu justifică pe deplin actualitatea abordării acestei teme.

În lucrare sunt prezentaţi mai mulţi algoritmi care rezolvă următoarea problemă de ordonare:

Intrare: O secvenţă ($v_1 \ldots v_n$) cu componente vi dintr-o mulţime total ordonată,

Ieşire: O permutare π astfel încât $v_{\pi(1)} <, v_{\pi(2)} < \ldots < v_{\pi(n)}$ şi rearanjarea elementelor din secvenţă conform ordinii din permutare.

Echivalent ca formulare poate fi:

Intrare *:* O secvenţă de înregistrări ($R_{1 \ldots\ldots} R_n$), fiecare înregistrare R_j având o valoare *cheie* k_i, iar peste mulţimea cheilor este definită o relaţie de ordine totală.

Ieşire: O permutare π asttel încât $k_\pi(i) \leq k_\pi(2) \leq .. \leq k_\pi(n)$ şi rearanjarea înregistrărilor din secvenţă în ordinea R_π (1), R_π(2)..... R_π(n).

Secvenţa de intrare este de obicei reprezentată printr-o listă liniară (tablou de n elemente sau listă simplu înlănţuită). Dacă lista e memorată în memoria internă a calculatorului atunci spunem că avem **sortare internă,** iar dacă e memorată într-un fişier de pe un periferic avem **sortare externă.** Lucrarea de faţă se ocupă numai de sortarea internă, unii dintre algoritmi fiind studiaţi la disciplina informatică conform noilor programe pentru învăţământul liceal.

Lucrarea are cinci capitole şi prezintă cei mai semnificativi algoritmi de sortare internă, neavând însă intenţia de a-i trece în revistă pe toţi. Implementarea algoritmilor este făcută în limbajul C++ .

CAP. I. NOŢIUNI GENERALE LEGATE DE ALOGORITMI

Se pare că termenul de algoritm provine de la numele unei persoane, respectiv Abu Ja`far Mohammed ibn Musa al-Khowarizmi (autor persan, sec. VIII-IX), care a scris o carte de matematică cunoscută în traducere latină ca *Algorithmi de numero indorum*, iar apoi ca *Liber algorithmi.* Deci, *algorithm* provine de la al-Khowarizmi, ceea ce literal înseamnă din oraşul Khowarizm. În prezent, acest oraş se numeşte Khiva şi se află în Uzbechistan. În timpul lui Adam Riese (sec. XVI), algoritmii foloseau la: dublări, înjumătăţiri, înmulţiri de numere. Alţi algoritmi apar în lucrările lui Stifer (*Arithmetica integra*) şi Cardano (*Ars magna sive de reguli algebraicis*). Termenul a rămas totuşi multă vreme cu o întrebuinţare destul de restrânsă, chiar şi în domeniul matematicii.

Kronecker (în 1886) şi Dedekind (în 1888) semnează actul de naştere al teoriei funcţiilor recursive. Conceptul de recursivitate devine indisolubil legat de cel de algoritm. Dar abia în deceniile al treilea şi al patrulea ale secolului nostru, teoria recursivităţii şi algoritmilor începe să se constituie ca atare, prin lucrările lui Skolem, Ackermann, Sudan, Church, Kleene, Turing Peter şi alţii.

Este surprinzătoare transformarea gândirii algoritmice, dintr-un instrument matematic particular, într-o modalitate fundamentală de abordare a problemelor în domenii care aparent nu au nimic în comun cu matematica. Această universalitate a gândirii algoritmice este rezultatul conexiunii dintre algoritm şi calculator. Astăzi, înţelegem prin algoritm o metoda generală de rezolvare a unui anumit tip de problemă, metoda care se poate implementa pe calculator. În acest context, un algoritm este esenţa absolută a unei rutine.

Cel mai faimos algoritm este desigur algoritmul lui Euclid pentru aflarea celui mai mare divizor comun a două numere întregi. Alte exemple de algoritmi sunt metodele învăţate în şcoală pentru a înmulţi/împarţi două numere. Ceea ce dă însă generalitate noţiunii de algoritm este faptul că el poate opera nu numai cu numere.

Există astfel algoritmi algebrici şi algoritmi logici. Pană şi o reţetă culinară este în esenţa un algoritm. Practic, s-a constatat că nu există nici un domeniu, oricât ar părea el de imprecis şi de fluctuant, în care să nu putem descoperi sectoare funcţionând algoritmic.

Un algoritm este compus dintr-o mulţime finita de paşi, fiecare necesitând una sau mai multe operaţii. Pentru a fi implementabile pe calculator, aceste operaţii trebuie să fie în primul rând definite, adică să fie foarte clar ce anume trebuie executat. În al doilea rând, operaţiile trebuie să fie efective, ceea ce înseamnă că în principiu, cel puţin o persoană dotată cu creion şi hârtie trebuie să poată efectua orice pas într-un timp finit.

Apariţia primelor calculatoare electronice a constituit un salt uriaş în direcţia automatizării activităţii umane. Nu există astăzi domeniu de activitate în care calculatorul să nu îşi arate utilitatea.

Calculatoarele pot fi folosite pentru a rezolva probleme, numai dacă pentru rezolvarea acestora se concep programe corespunzătoare de rezolvare. Termenul de program (programare) a suferit schimbări în scurtă istorie a informaticii. Prin anii '60 problemele rezolvate cu ajutorul calculatorului erau simple şi se găseau algoritmi nu prea complicaţi pentru rezolvarea lor. Prin program se înţelegea rezultatul scrierii unui algoritm într-un limbaj de programare. Din cauza creşterii complexităţii problemelor, astăzi pentru rezolvarea unei probleme adesea vom concepe un sistem de mai multe programe.

Dar ce este un algoritm? O definiţie matematică, riguroasă, este greu de dat, chiar imposibilă fără a introduce şi alte noţiuni. Vom încerca în continuare o descriere a ceea ce se înţelege prin algoritm.

Ne vom familiariza cu noţiunea de algoritm prezentând mai multe exemple de algoritmi şi observând ce au ei în comun.

Evident, vom prezenta mai mulţi algoritmi, cei mai mulţi fiind legaţi de probleme accesibile absolvenţilor de liceu.

Vom constata că un algoritm este un text finit, o secvenţă finită de propoziţii ale unui limbaj. Din cauză că este inventat special în acest scop, un astfel de limbaj este numit *limbaj de descriere a algoritmilor*. Fiecare propoziţie a limbajului precizează o anumită regulă de calcul, aşa cum se va observa atunci când vom prezenta limbajul *Pseudocod*.

Oprindu-ne la semnificaţia algoritmului, la efectul execuţiei lui, vom observa că fiecare algoritm defineşte o funcţie matematică. De asemenea, din toate secţiunile următoare va reieşi foarte clar că un algoritm este scris pentru rezolvarea unei probleme. Din mai multe exemple se va observa însă că, pentru rezolvarea aceleaşi probleme, există mai mulţi algoritmi.

Pentru fiecare problemă P există date presupuse cunoscute (date iniţiale pentru algoritmul corespunzător, A) şi rezultate care se cer a fi găsite (date finale). Evident, problema s-ar putea să nu aibă sens pentru orice date iniţiale. Vom spune că datele pentru care problema P are sens fac parte din domeniul D al algoritmului A. Rezultatele obţinute fac parte dintr-un domeniu R, astfel că executând algoritmul A cu datele de intrare $x \in D$ vom obţine rezultatele $r \in R$. Vom spune că $A(x)=r$ şi astfel algoritmul A defineşte o funcţie

A : D ---> R .

Algoritmii au următoarele caracteristici: *generalitate, finitudine şi unicitate*.

Prin *generalitate* se înţelege faptul că un algoritm este aplicabil pentru orice date iniţiale $x \in D$. Deci un algoritm A nu rezolvă problema P cu nişte date de intrare, ci o rezolvă în general, oricare ar fi aceste date. Astfel, algoritmul de rezolvare a unui sistem liniar de n ecuaţii cu n necunoscute prin metoda lui *Gauss*, rezolvă orice sistem liniar şi nu un singur sistem concret.

Prin *finitudine* se înţelege că textul algoritmului este finit, compus dintr-un număr finit de propoziţii. Mai mult, numărul transformărilor ce trebuie aplicate unei informaţii admisibile $x \in D$ pentru a obţine rezultatul final corespunzător este finit.

Prin *unicitate* se înţelege că toate transformările prin care trece informaţia iniţială pentru a obţine rezultatul $r \in R$ sunt bine determinate de regulile algoritmului. Aceasta înseamnă că fiecare pas din execuţia algoritmului dă rezultate bine determinate şi precizează în mod unic pasul următor. Altfel spus, ori de câte ori am executa algoritmul, pornind de la aceeaşi informaţie admisibilă $x \in D$, transformările prin care se trece şi rezultatele obţinute sunt aceleaşi.

În descrierea algoritmilor se folosesc mai multe limbaje de descriere, dintre care cele mai des folosite sunt:

- limbajul *schemelor logice*;
- limbajul *Pseudocod*.

În continuare vom folosi pentru descrierea algoritmilor limbajul Pseudocod care va fi definit în cele ce urmează. În ultima vreme schemele logice sunt tot mai puţin folosite în descrierea algoritmilor şi nu sunt deloc potrivite în cazul problemelor complexe. Prezentăm însă şi schemele logice, care se mai folosesc în manualele de liceu, întrucât cu ajutorul lor vom preciza în continuare semantica propoziţiilor Pseudocod.

1.2 Scheme logice

Schema logică este un mijloc de descriere a algoritmilor prin reprezentare grafică. Regulile de calcul ale algoritmului sunt descrise prin blocuri (figuri geometrice) reprezentând operaţiile (paşii) algoritmului, iar ordinea lor de aplicare (succesiunea operaţiilor) este indicată prin săgeţi. Fiecărui tip de operaţie îi este consacrată o figură geometrică (un bloc tip) în interiorul căreia se va înscrie operaţia din pasul respectiv.

Prin **execuţia** unui algoritm descris printr-o schemă logică se înţelege efectuarea tuturor operaţiilor precizate prin blocurile schemei logice, în ordinea indicată de săgeţi.

În descrierea unui algoritm, deci şi într-o schemă logică, intervin variabile care marchează atât datele cunoscute iniţial, cât şi rezultatele dorite, precum şi alte rezultate intermediare necesare în rezolvarea problemei. Întrucât **variabila** joacă un rol central în programare este bine să definim acest concept. Variabila defineşte o mărime care îşi poate schimba valoarea în timp. Ea are un nume şi, eventual, o valoare. Este posibil ca variabila încă să nu fi primit valoare, situaţie în care vom spune că ea este neiniţializată. Valorile pe care le poate lua variabila aparţin unei mulţimi D pe care o vom numi domeniul variabilei. În concluzie vom înţelege prin variabilă tripletul

(*nume, domeniul D, valoare*)

Blocurile delimitatoare *Start* şi *Stop* (Fig.1.2.1.a şi 1.2.1.b) vor marca începutul respectiv sfârşitul unui algoritm dat printr-o schemă logică. Descrierea unui algoritm prin schemă logică va începe cu un singur bloc *Start* şi se va termina cu cel puţin un bloc *Stop*.

Blocurile de intrare/ieşire *Citeşte* şi *Tipăreşte* (Fig. 1.2.1.c şi 1.2.1.d) indică introducerea unor *Date de intrare* respectiv extragerea unor *Rezultate finale*. Ele permit precizarea datelor iniţiale cunoscute în problemă şi tipărirea rezultatelor cerute de problemă. Blocul *Citeşte* iniţializează variabilele din lista de intrare cu valori corespunzătoare, iar blocul *Tipăreşte* va preciza rezultatele obţinute (la execuţia pe calculator cere afişarea pe ecran a valorilor expresiilor din lista de ieşire).

Blocurile de atribuire (calcul) se utilizează în descrierea operaţiilor de atribuire (:=). Printr-o astfel de operaţie, unei variabile *var* i se atribuie valoarea calculată a unei expresii *expr* (Fig.1.2.1.e).

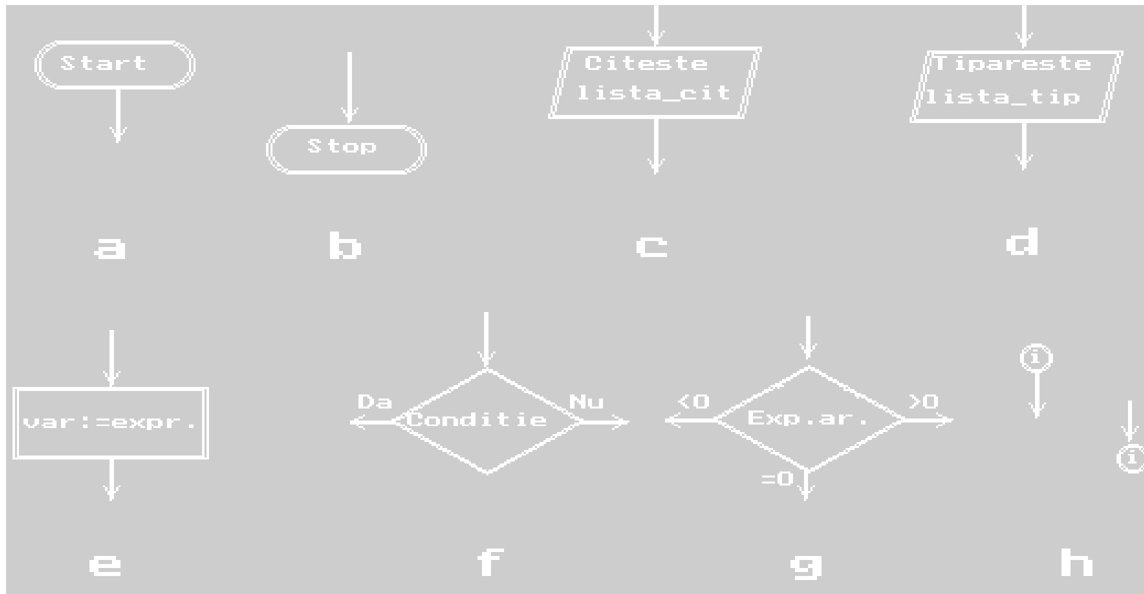


Fig.1.2.1. Blocurile schemelor logice

Blocurile de decizie marchează punctele de ramificaţie ale algoritmului în etapa de decizie. Ramificarea poate fi dublă (*blocul logic*, Fig.1.2.1.f) său triplă (*blocul aritmetic*, Fig. 1.2.1.g). *Blocul de decizie logic* indică ramura pe care se va continua execuţia algoritmului în funcţie de îndeplinirea (ramura **Da**) său neîndeplinirea (ramura **Nu**) unei condiţii. Condiţia care se va înscrie în blocul de decizie logic va fi o expresie logică a cărei valoare poate fi una dintre valorile "adevărat" său "fals". *Blocul de decizie aritmetic* va hotărî ramura de continuare a algoritmului în funcţie de semnul valorii expresiei aritmetice înscrise în acest bloc, care poate fi negativă, nulă său pozitivă.

Blocurile de conectare marchează întreruperile săgeţilor de legătură dintre blocuri, dacă din diverse motive s-au efectuat astfel de întreruperi (Fig.1.2.1.h).

Pentru exemplificare vom da în continuare două scheme logice, corespunzătoare unor algoritmi pentru rezolvarea problemelor P1.2.1 şi P1.2.2.

P1.2.1. Să se rezolve ecuaţia de grad doi $aX^2+bX+c=0$ (a,b,c∈R _i a≠0).

Metoda de rezolvare a ecuaţiei de gradul doi este cunoscută. Ecuaţia poate avea rădăcini reale, respectiv complexe, situaţie recunoscută după semnul discriminantului $D=b^2-4ac$.

Algoritmul de rezolvare a problemei va citi mai întâi datele problemei, marcate prin variabilele *a, b* şi *c*. Va calcula apoi discriminantul *D* şi va continua în funcţie de valoarea lui *D*, aşa cum se poate vedea în fig.1.2.2

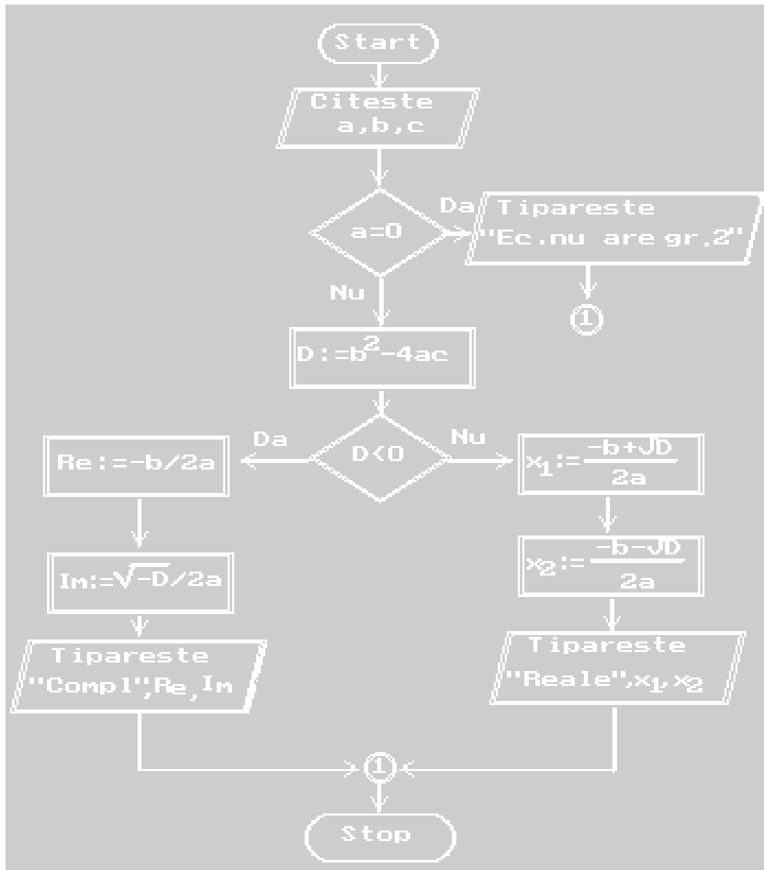


Fig.1.2.2. Algoritm pentru rezolvarea ecuaţiei de gradul doi.

P1.2.2. Să se calculeze suma elementelor pozitive ale unui şir de numere reale dat.

Schema logică (dată în Fig.1.2.3) va conţine imediat după blocul **START** un bloc de citire, care precizează datele cunoscute în problemă, apoi o parte care calculează suma cerută şi un bloc de tipărire a sumei găsite, înaintea blocului **STOP**. Partea care calculează suma *S* cerută are un bloc pentru iniţializarea cu 0 a acestei sume, apoi blocuri pentru parcurgerea numerelor: x_1, $x_2 ... x_n$ şi adunarea celor pozitive la suma *S*. Pentru această parcurgere se foloseşte o variabilă contor *i*, care este iniţializată cu 1 şi creşte mereu cu 1 pentru a atinge valoarea *n*, indicele ultimului număr dat.

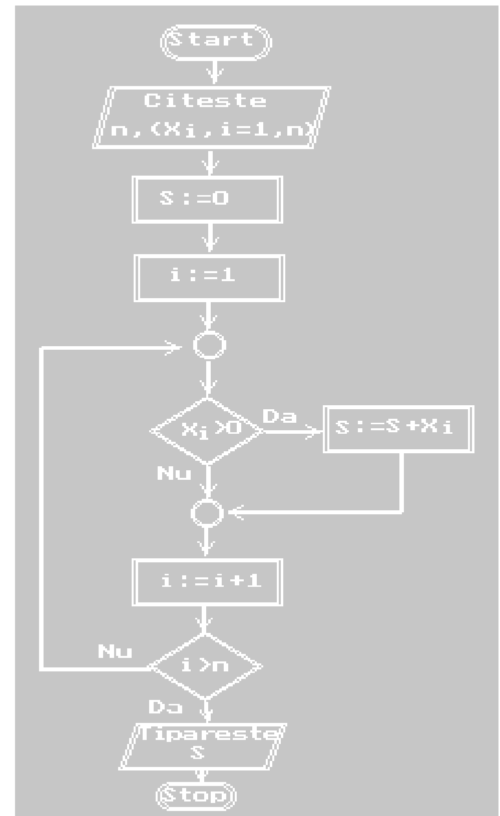


Fig.1.2.3.Algoritm pentru calculul unei sume

Schemele logice dau o reprezentare grafică a algoritmilor cu ajutorul unor blocuri de calcul. Execuţia urmează sensul indicat de săgeată, putând avea loc reveniri în orice punct din schema logică. Din acest motiv se poate obţine o schemă logică încâlcită, greu de urmărit. Rezultă importanţa compunerii unor scheme logice structurate (D-scheme, după Djikstra), care să conţină numai anumite structuri standard de calcul şi în care drumurile de la **START** la **STOP** să fie uşor de urmărit.

1.3. Limbajul PSEUDOCOD

Limbajul Pseudocod este un limbaj inventat în scopul proiectării algoritmilor şi este format din propoziţii asemănătoare propoziţiilor limbii române, care corespund structurilor de calcul folosite în construirea algoritmilor. Limbajul Pseudocod are două tipuri de propoziţii: **propoziţii standard**, care vor fi prezentate fiecare cu sintaxa şi semnificaţia (semantica) ei şi **propoziţii nestandard**. **Propoziţiile standard** ce co-

respund structurilor de calcul prezentate în figura 1.3.1 încep cu un cuvânt cheie, iar **propoziţiile nestandard** sunt texte care descriu părţi ale algoritmului încă incomplet elaborate, nefinisate, asupra cărora urmează să se revină.

Pe lângă aceste propoziţii standard şi nestandard, în textul algoritmului vom mai introduce propoziţii explicative, numite **comentarii** (puse între /*). Prin **execuţia unui algoritm** descris în Pseudocod se înţelege efectuarea operaţiilor precizate de propoziţiile algoritmului, în ordinea citirii lor.

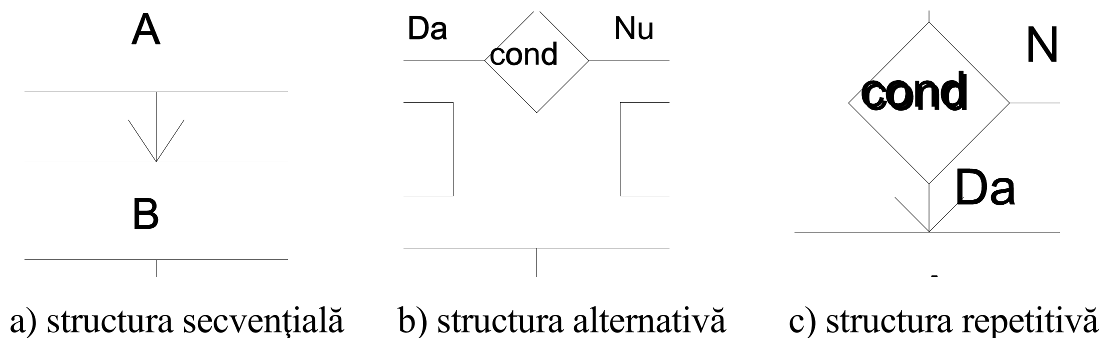
În figura 1.3.1, prin A, B s-au notat subscheme logice, adică secvenţe de oricâte structuri construite conform celor trei reguli menţionate în continuare.

Structura secvenţială (fig.1.3.1.a) este redată prin concatenarea propoziţiilor, simple său compuse, ale limbajului Pseudocod, care vor fi executate în ordinea întâlnirii lor în text. Propoziţiile simple din limbajul Pseudocod sunt **CITEŞTE**, **TIPAREŞTE**, **FIE** şi apelul de subprogram. Propoziţiile compuse corespund structurilor alternative şi repetitive.

Structura alternativă (fig.1.3.1.b) este redată în Pseudocod prin propoziţia DACĂ, prezentată în secţiunea 1.3.2, iar **structura repetitivă** din fig.1.3.1.c este redată în Pseudocod prin propoziţia **CÂT TIMP**, prezentată în secţiunea 1.3.3.

Bohm şi Jacopini au demonstrat că orice algoritm poate fi descris folosind numai aceste trei structuri de calcul.

Propoziţiile **DATE** şi **REZULTATE** sunt folosite în faza de specificare a problemelor, adică enunţarea riguroasă a acestora.



a) structura secvenţială b) structura alternativă c) structura repetitivă

Figura 1.3.1. Structurile elementare de calcul

Acum putem preciza mai exact ce înţelegem prin cunoaşterea completă a problemei de rezolvat. Evident, o problemă este cunoscută atunci când se ştie care sunt datele cunoscute în problemă şi ce rezultate trebuiesc obţinute. Deci pentru cunoaşterea unei probleme este necesară precizarea variabilelor care marchează date considerate cunoscute în problemă, care va fi reflectată printr-o propoziţie **DATE** şi cunoaşterea exactă a cerinţelor problemei, care se va reflecta prin propoziţii **REZULTATE**. Variabilele prezente în aceste propoziţii au anumite semnificaţii, presupuse cunoscute. Cunoaşterea acestora, scrierea lor explicită, formează ceea ce vom numi în continuare **specificarea problemei**. Specificarea unei probleme este o activitate foarte importantă dar nu şi simplă.

De exemplu, pentru rezolvarea ecuaţiei de gradul al doilea, specificarea problemei, scrisă de un începător, poate fi:

DATE: a,b,c; { Coeficienţii ecuaţiei }

REZULTATE: x1,x2; { Rădăcinile ecuaţiei }

Această specificaţie este însă incompletă dacă ecuaţia nu are rădăcini reale. În cazul în care rădăcinile sunt complexe putem nota prin *x1, x2* partea reală respectiv partea imaginară a rădăcinilor. Său pur şi simplu, nu ne interesează valoarea rădăcinilor în acest caz, ci doar faptul că ecuaţia nu are rădăcini reale. Cu alte cuvinte avem nevoie de un mesaj care să ne indice această situaţie (vezi schema logică 1.2.2), sau de un indicator, fie el *ind*. Acest indicator va lua valoarea 1 dacă rădăcinile sunt reale şi valoarea 0 în caz contrar. Deci specificaţia corectă a problemei va fi:

DATE *a,b,c; { Coeficienţii ecuaţiei }*

REZULTATE: *- ind, {Un indicator: 1=rădăcini reale, 0=complexe}*

-x1,x2; { Rădăcinile ecuaţiei, în cazul ind=1,}

{respectiv partea reală şi cea }

{imaginară în cazul ind=0}

Evident că specificarea problemei este o etapă importantă pentru găsirea unei metode de rezolvare şi apoi în proiectarea algoritmului corespunzător. Nu se poate

rezolva o problemă dacă aceasta nu este bine cunoscută, adică nu avem scrisă specificarea problemei. **Cunoaşte complet problema** este prima regulă ce trebuie respectată pentru a obţine cât mai repede un algoritm corect pentru rezolvarea ei.

CAP.2. METODE DE ELABORARE A ALGORITMILOR

2.1 Elaborarea algoritmilor

Prin elaborarea (proiectarea) unui algoritm înţelegem întreaga activitate depusă de la enunţarea problemei până la realizarea algoritmului corespunzător rezolvării acestei probleme.

În elaborarea unui algoritm deosebim următoarele activităţi importante:

- specificarea problemei;
- descrierea metodei alese pentru rezolvarea problemei;
- proiectarea propriu-zisă. Ea constă în descompunerea problemei în subprobleme, obţinerea algoritmului principal şi a tuturor subalgoritmilor apelaţi
- verificarea algoritmului obţinut.

2.2 Proiectarea ascendentă şi proiectarea descendentă

Există două metode generale de proiectare a algoritmilor, a căror denumire provine din modul de abordare a rezolvării problemelor: metoda descendentă şi metoda ascendentă.

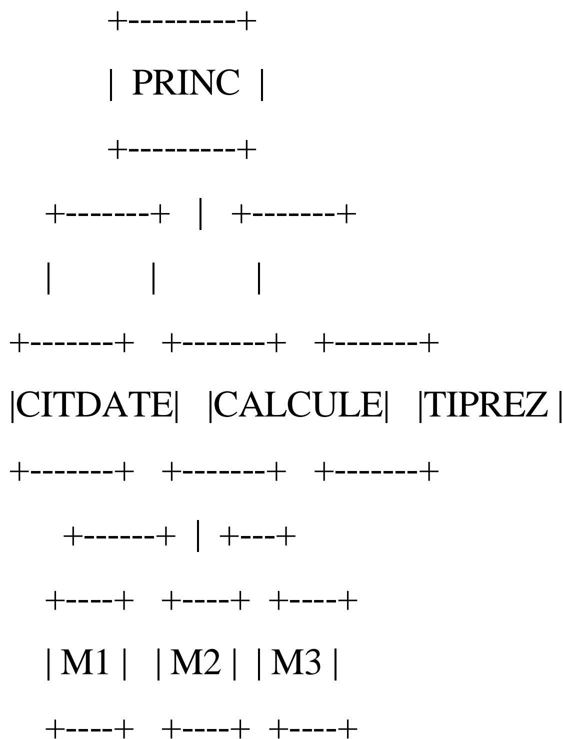
Proiectarea descendentă (top-down) porneşte de la problema de rezolvat, pe care o descompune în părţi rezolvabile separat. De obicei aceste părţi sunt subprobleme independente, care la rândul lor pot fi descompuse în subprobleme. La prima descompunere accentul trebuie pus pe algoritmul (modulul) principal nu asupra subproblemelor. La acest nivel nu ne interesează amănunte legate de rezolvarea subproblemelor, presupunem că le ştim rezolva, eventual că avem deja scrişi subalgoritmi pentru rezolvarea lor. Urmează să considerăm pe rând fiecare subproblemă în parte şi să proiectăm (în acelaşi mod) un subalgoritm pentru rezolvarea

ei. În final, se va descrie subalgoritmul de rezolvare al fiecărei subprobleme, dar şi interacţiunile dintre aceşti subalgoritmi şi ordinea în care ei sunt folosiţi.

Noţiunea de **modul** va fi definită în secţiunea următoare. Deocamdată înţelegem prin modul orice subalgoritm sau algoritmul principal. Legătura dintre module se prezintă cel mai bine sub forma unei diagrame numită arbore de programare. Fiecărui modul îi corespunde în arborele de programare un nod, ai cărui descendenţi sunt toate modulele apelate direct. Nodul corespunzător algoritmului principal este chiar nodul rădăcină.

Astfel, în arborele de programare din fig.2.1.1 există un algoritm principal (modulul **PRINC**), care apelează trei subalgoritmi (modulele **CITDATE**, **CALCULE** şi **TIPREZ**). La rândul său, modulul **CALCULE** apelează trei subalgoritmi (modulele **M1, M2 şi M3**).

Fig.2.1.1. Arbore de programare



În multe cărţi metoda **top-down** este întâlnită şi sub denumirea **stepwise-refinement**, adică **rafinare în paşi succesivi**. Este vorba de un proces de detaliere pas cu pas a specificaţiei, denumit proiectare descendentă. Algoritmul apare în diferite versiuni succesive, fiecare fiind o detaliere a versiunii precedente.

Scopul urmărit este acelaşi: concentrarea atenţiei asupra părţilor importante ale momentului şi amânarea detaliilor pentru mai târziu. Dacă ar fi necesar să le deosebim am spune că metoda top-down se referă la nivelul **macro** iar metoda rafinării succesive la nivel **micro**. La nivel macro se doreşte descompunerea unei probleme complexe în subprobleme. La nivel micro se doreşte obţinerea unui modul în versiune finală. Într-o versiune intermediară pot fi prezente numai părţile importante ale acestuia, urmând să se revină asupra detaliilor în versiunile următoare după ce aspectele importante au fost rezolvate.

Avantajele proiectării top-down (cunoscută şi sub denumirea ***"Divide et impera***") sunt multiple. Avantajul principal constă în faptul că ea permite programatorului să reducă complexitatea problemei, subproblemele în care a fost descompusă fiind mai simple, şi să amâne detaliile pentru mai târziu. În momentul în care descompunem problema în subprobleme nu ne gândim cum se vor rezolva subproblemele ci care sunt ele şi conexiunile dintre ele.

Proiectarea descendentă permite lucrul în echipe mari. Prin descompunerea problemei în mai multe subprobleme, fiecare subproblemă poate fi dată spre rezolvare unei subechipe. Fiecare subechipă nu cunoaşte decât subproblema pe care trebuie să o rezolve.

Metoda "**Divide et Impera**" poate fi folosită nu numai la împărţirea problemei în subprobleme ci şi la împărţirea datelor în grupe mai mici de date. Un astfel de procedeu este folosit de subalgoritmul **Quicksort**.

Metoda ascendentă (bottom-up) porneşte de la propoziţiile limbajului şi de la subalgoritmi existenţi, pe care îi asamblează în alţi subalgoritmi pentru a ajunge în final la algoritmul dorit. Cu alte cuvinte, în cazul metodei ascendente va fi scris mai

întâi subalgoritmul apelat şi apoi cel care apelează. Ca rezultat al proiectării ascendente se ajunge la o mulţime de subalgoritmi care se apelează între ei. Este important să se cunoască care subalgoritm apelează pe care, lucru redat printr-o diagramă de structură, ca şi în cazul programării descendente.

Această metodă are marele dezavantaj că erorile de integrare vor fi detectate târziu, abia în faza de integrare. Se poate ajunge abia acum la concluzia că unii subalgoritmi, deşi corecţi, nu sunt utili.

De cele mai multe ori nu se practică o proiectare ascendentă sau descendentă pură ci o combinare a lor, o **proiectare mixtă**.

2.3. Proiectarea modulară

Prin ***proiectare (programare) modulară*** înţelegem metoda de proiectare (programare) a unui algoritm pentru rezolvarea unei probleme prin folosirea modulelor.

Dar ce este un modul? **Modulul** este considerat o unitate structurală de sine stătătoare, fie program, fie subprogram, fie o unitate de program. Un modul poate conţine sau poate fi conţinut într-un alt modul. Un modul poate fi format din mai multe submodule. Astfel, în Pseudocod fiecare subalgoritm şi algoritmul principal sunt considerate module. În limbajele de programare cu structură de bloc **UNIT**-urile pot fi considerate module. La compilarea separată un grup de subprograme compilate deodată constituie un modul, dar acest modul poate fi considerat ca o mulţime de submodule din care este compus.

Este însă important ca fiecare modul să-şi aibă rolul său bine precizat, să realizeze o funcţie în cadrul întregului program. El apare în mod natural în descompunerea top-down.

Indiferent că privim modulul ca un singur subalgoritm, un grup de subalgoritmi, sau un algoritm de sine stătător ce apelează alţi subalgoritmi, considerăm modulele

relativ independente, dar cu posibilităţi de comunicare între ele. Astfel, un modul nu trebuie să fie influenţat de maniera în care se lucrează în interiorul altui modul. Orice modificare ulterioară în structura unui program, dacă funcţia pe care o realizează un modul *M* încă este necesară, acest modul trebuie să fie util şi folosit în continuare fără modificări.

Rezultă că programarea modulară se bazează pe descompunerea problemei în subprobleme şi proiectarea şi programarea separată a subalgoritmilor corespunzători. De altfel, considerăm că într-o programare serioasă nu se poate ajunge la implementare fără a avea în prealabil algoritmii descrişi într-un limbaj de descriere (la noi Pseudocod). Deci programarea modulară se referă în primul rând la proiectarea modulară a algoritmilor şi apoi la traducerea lor în limbajul de programare ales, ţinând seama de specificul acestui limbaj. Programarea modulară este strâns legată de programarea ascendentă şi de programarea descendentă, ambele presupunând folosirea subalgoritmilor pentru toate subproblemele întâlnite.

Avantajele programării modulare sunt multiple. Menţionăm în cele ce urmează câteva dintre ele. Descompunerea unei probleme complexe în subprobleme este un mijloc convenabil şi eficient de a reduce complexitatea (**Principiul Divide et impera acţionează şi în programare**). Este evident că probabilitatea apariţiei erorilor în conceperea unui program creşte cu mărimea programului, lucru confirmat şi de experienţa practică. De asemenea, rezolvând o problemă mai simplă, testarea unui modul se poate face mult mai uşor decât testarea întregului algoritm.

Apoi, faptul că trebuiesc proiectate mai multe subprograme pentru subproblemele întâlnite, permite munca mai multor programatori. S-a ajuns astfel la munca în echipă, modalitate prin care se ajunge la scurtarea termenului de realizare a produsului program.

Modulele se pot refolosi ori de câte ori avem nevoie de ele. Astfel, s-a ajuns la compilarea separată a subprogramelor şi la păstrarea subprogramelor obţinute în biblioteci de subprograme, de unde ele se pot refolosi la nevoie. Sunt cunoscute astăzi

multe astfel de biblioteci de subprograme. *Reutilizabilitatea* acestor subprograme este o proprietate foarte importantă în activitatea de programare. Ea duce la mărirea productivităţii în programare, dar şi la creşterea siguranţei în realizarea unui produs corect.

Uneori, în timpul proiectării algoritmului sau a implementării lui, se ajunge la concluzia că proiectarea a fost incompletă sau că unele module sunt ineficiente. şi în această situaţie programarea modulară este avantajoasă, ea permiţând înlocuirea modulului în cauză cu altul mai performant.

Una din activităţile importante în realizarea unui program este verificarea corectitudinii acestuia. Experienţa a arătat că modulele se pot verifica cu atât mai uşor cu cât sunt mai mici. Abilitatea omului de a înţelege şi analiza corectitudinea unui subalgoritm este mult mai mare pentru texte scurte. În unele cărţi chiar se recomandă a nu se folosi subalgoritmi mai mari decât 50 de propoziţii. Sigur că o astfel de limită nu există, dar se recomandă descompunerea unui subalgoritm în alţi subalgoritmi oricând acest lucru este posibil în mod natural, deci aceşti noi subalgoritmi rezolvă subprobleme de sine stătătoare, sau realizează funcţii bine definite.

2.4. Programarea structurată

Programarea structurată este un stil de programare apărut în urma experienţei primilor ani de activitate. Ea cere respectarea unei discipline de programare şi folosirea riguroasă a câtorva structuri de calcul. Ca rezultat se va ajunge la un algoritm uşor de urmărit, clar şi corect.

Termenul *programare*, folosit în titlul acestei secţiuni şi consacrat în literatura de specialitate, este folosit aici în sens larg şi nu este identic cu cel de programare propriu-zisă. Este vorba de întreaga activitate depusă pentru obţinerea unui program, deci atât proiectarea algoritmului cât şi traducerea acestuia în limbajul de programare ales.

Bohm şi Jacopini au demonstrat că orice algoritm poate fi compus din numai trei structuri de calcul:

- structura *secvenţială*;
- structura *alternativă*;
- structura *repetitivă*.

Fiecare din aceste structuri, ca parte dintr-o schemă logică, are o singură intrare şi o singură ieşire.

Knuth consideră programarea structurată ca fiind un mijloc de a face produsele program mai uşor de citit. De asemenea, programarea structurată este definită ca fiind programarea în care abordarea este top-down, organizarea muncii este făcută pe principiul echipei programatorului şef, iar în proiectarea algoritmilor se folosesc cele trei structuri de calcul definite de Bohm-Jacopini.

Alţi autori consideră programarea structurată nu ca o simplă metodă de programare ci ansamblul tuturor metodelor de programare cunoscute. Dar programarea modulară, programarea top-down, sau bottom-up (ascendentă sau descendentă) au apărut înaintea programării structurate. Important este faptul că programarea structurată presupune o disciplină în activitatea de programare.

Considerăm că programarea structurată se poate întâlni:

- la nivel **micro**, privind elaborarea unui subalgoritm;
- la nivel **macro**, privind dezvoltarea întregului produs informatic (algoritm).

La nivel micro programarea structurată este cea în care autorul este atent la structura fiecărui modul în parte, cerând claritate şi ordine în scriere şi respectarea structurilor de calcul definite mai sus.

La nivel macro programarea structurată presupune practicarea proiectării top-down, a programării modulare şi a celorlalte metode de programare, cerând ordine în întreaga activitate şi existenţa unei structuri clare a întregii aplicaţii, precizată prin diagrama de structură a aplicaţiei.

În acest scop am definit limbajul Pseudocod, care are structurile de calcul menţionate. Schemele logice obţinute dintr-o descriere în Pseudocod a unui algoritm, conform semanticii propoziţiilor Pseudocod, se numesc **D-scheme** (de la Dijkstra) sau **scheme logice structurate**.

Referitor la faza de codificare într-un limbaj de programare a unui algoritm obţinut în urma unei proiectări structurate se cere respectarea structurii acestui algoritm, ceea ce este posibil şi uşor de realizat dacă limbajul de programare are structurile de calcul respective. În acest caz programul va fi o copie fidelă a algoritmului proiectat.

CAP III. METODE DE SORTARE ŞI CĂUTARE

Căutarea şi ***Sortarea*** sunt două dintre cele mai des întâlnite subprobleme în programare. Ele constituie o parte esenţială din numeroasele procese de prelucrare a datelor. Operaţiile de căutare şi sortare sunt executate frecvent de către oameni în viaţa de zi cu zi, ca de exemplu căutarea unui cuvânt în dicţionar sau căutarea unui număr în cartea de telefon.

Căutarea este mult simplificată dacă datele în care efectuăm această operaţie sunt *sortate* (ordonate, aranjate) într-o anumită ordine (cuvintele în ordine alfabetică, numerele în ordine crescătoare sau descrescătoare).

Sortarea datelor constă în rearanjarea colecţiei de date astfel încât un câmp al elementelor colecţiei să respecte o anumită ordine. De exemplu în cartea de telefon fiecare element (abonat) are un câmp de nume, unul de adresă şi unul pentru numărul de telefon. Colecţia aceasta respectă ordinea alfabetică după câmpul de nume.

Dacă datele pe care dorim să le ordonăm, adică să le sortăm, sunt în memoria internă, atunci procesul de rearanjare a colecţiei îl vom numi ***sortare internă***, iar dacă datele se află într-un fişier (colecţie de date de acelaşi fel aflate pe suport extern), atunci procesul îl vom numi ***sortare externă***.

Fiecare element al colecţiei de date se numeşte ***articol*** iar acesta la rândul său este compus din unul sau mai multe componente. O ***cheie C*** este asociată fiecărui articol şi este de obicei unul dintre componente. Spunem că o colecţie de n articole este *ordonat crescător* după cheia C dacă $C(i) \leq C(j)$ pentru $1 \leq i < j \leq n$, iar dacă $C(i) \geq C(j)$ atunci şirul este *ordonat descrescător.*

3.1. ALGORITMI DE CĂUTARE

În acest subcapitol vom studia câteva tehnici elementare de căutare şi vom presupune că datele se află în memoria internă, într-un şir de articole. Vom căuta un articol după un câmp al acestuia pe care îl vom considera cheie de căutare. În urma procesului de căutare va rezulta poziţia elementului căutat (dacă acesta există).

Notând cu $k_1, k_2, ..., k_n$ cheile corespunzătoare articolelor şi cu a cheia pe care o căutăm, problema revine la a găsi (dacă există) poziţia p cu proprietatea $a = k_p$.

De obicei articolele sunt păstrate în ordinea crescătoare a cheilor, deci vom presupune că: $k_1 < k_2 < .... < k_n$.

Uneori este util să aflăm nu numai dacă există un articol cu cheia dorită ci şi să găsim în caz contrar locul în care ar trebui inserat un nou articol având cheia specificată, astfel încât să se păstreze ordinea existentă.

Deci *problema căutării* are următoarea specificare:

Date $a,n,(k_i, i=1,n)$;

Precondiţia: $n \in N$, $n \geq 1$ *şi* $k_1 < k_2 < .... < k_n$;

Rezultate p;

Postcondiţia: $(p=1$ *şi* $a \leq k_1)$ *sau* $(p=n+1$ *şi* $a > k_n)$

sau $(1<p\leq n)$ *şi* $(k_{p-1} < a \leq k_p)$.

Pentru rezolvarea acestei probleme vom descrie mai mulţi subalgoritmi.

O primă metodă este căutarea **secvenţială**, în care sunt examinate succesiv toate cheile.

Subalgoritmul *CautSecv(a,n,K,p)* ***este***: *{*$n \in N$, $n \geq 1$ *şi}*

{$k_1 < k_2 < .... < k_n$*}*

{Se caută p astfel ca:}

{$(p=1$ *şi* $a \leq k_1)$ *sau* $(p=n+1$ *şi* $a>k_n)$*}*

{sau $(1<p\leq n)$ *şi* $(k_{p-1} < a \leq k_p)$.

Fie *p:=0; {Cazul "încă negăsit"}*

Dacă *$a \le k_1$* ***atunci*** *p:=1* ***altfel***

Dacă *$a > k_n$* ***atunci*** *p:=n+1* ***altfel***

Pentru *i:=2; n* ***execută***

Dacă *(p=0)* ***şi*** *($a \le k_i$)* ***atunci*** *p:=i* ***sfdacă***

sfpentru

sfdacă

sfdacă

sf*-CautSecv*

Se observă că prin această metodă se vor executa în cel mai nefavorabil caz *n*-1 comparări, întrucât contorul *i* va lua toate valorile de la 2 la *n*. Cele *n* chei împart axa reală în *n*+1 intervale. Tot atâtea comparări se vor efectua în *n*-1 din cele n+1 intervale în care se poate afla cheia căutată, deci complexitatea medie are acelaşi ordin de mărime ca şi complexitatea în cel mai rău caz.

Evident că în multe situaţii acest algoritm face calcule inutile. Atunci când a fost deja găsită cheia dorită este inutil a parcurge ciclul pentru celelalte valori ale lui *i*. Cu alte cuvinte este posibil să înlocuim ciclul **PENTRU** cu un ciclu **CÂT TIMP**. Ajungem la un al doilea algoritm, dat în continuare.

Subalgoritmul *CautSucc(a,n,K,p)* ***este****:{$n \in N$, $n \ge 1$ şi}*

{$k_1 < k_2 < .... < k_n$}

{Se caută p astfel ca:}

{(p=1 şi $a \le k_1$) sau (p=n+1 şi $a > k_n$)}

{sau ($1 < p \le n$) şi ($k_{p-1} < a \le k_p$).

Fie *p:=1;*

Dacă *$a > k_1$* ***atunci***

Câttimp *$p \le n$ şi $a > k_p$* ***execut*** *p:=p+1* ***sfcât***

sfdacă

sf*-CautSecv*

O altă metodă, numită **căutare binară**, care este mult mai eficientă, utilizează tehnica "**divide et impera**" privitor la date. Se determină în ce relaţie se află cheia articolului aflat în mijlocul colecţiei cu cheia de căutare. În urma acestei verificări căutarea se continuă doar într-o jumătate a colecţiei. În acest mod, prin înjumătăţiri succesive se micşorează volumul colecţiei rămase pentru căutare. Căutarea binară se poate realiza practic prin apelul funcţiei *BinarySearch(a,n,K,1,n)*, descrisă mai jos, folosită în subalgoritmul dat în continuare.

Subalgoritmul *CautBin(a,n,K,p)* ***este:*** *{$n \in N$, $n \geq 1$ şi $k_1 < k_2 < .... < k_n$}*

{Se caută p astfel ca: ($p=1$ şi $a \leq k_1$) sau} {($p=n+1$ şi $a>k_n$) sau ($1<p\leq n$) şi ($k_{p-1} < a \leq k_p$)}

Dacă *$a\leq k_1$* ***atunci p:=1 altfel***

Dacă *$a>k_n$* ***atunci*** *p:=n+1* ***altfel***

p:=BinarySearch(a,n,K,1,n)

sfdacă

sfdacă

sf*-CautBin*

Funcţia *BinarySearch (a,n,K,St,Dr)* ***este:***

Dacă *St≥Dr-1*

atunci *BinarySearch:=Dr*

altfel *m:=(St+Dr) Div 2;*

Dacă *a≤K[m]*

atunci *BinarySearch:=BinarySearch(a,n,K,St,m)*

altfel *BinarySearch:=BinarySearch(a,n,K,m,Dr)*

sfdacă

sfdacă

sf*-BinarySearch*

În funcţia *BinarySearch* descrisă mai sus, variabilele *St* şi *Dr* reprezintă capetele intervalului de căutare, iar *m* reprezintă mijlocul acestui interval.

Se observă că funcţia *BinarySearch* se apelează recursiv. Se poate înlătura uşor recursivitatea, aşa cum se poate vedea în următoarea funcţie:

Funcţia *BinSeaNerec (a,n,K,St,Dr)* ***este:***

  Câttimp *Dr-St>1* ***execută***

    m:=(St+Dr) Div 2;

    Dacă *a≤K[m]*

        atunci *Dr:=m*

        altfel *St:=m*

    sfdacă

  sfcât

  BinSeaNerec:=Dr

sf*-BinSeaNerec*

3.2. ALOGORITMI DE SORTARE

1.Algoritmul de sortare prin metoda bulelor

Prin această metodă se parcurge vectorul şi se compară fiecare element cu succesorul său. Dacă nu sunt în ordine cele două elemente se interschimbă între ele. Vectorul se parcurge de mai multe ori, până când la o parcurgere completă nu se mai execută nici o interschimbare între elemente (înseamnă că vectorul este sortat).

Se folosesc următoarele **variabile de memorie:**

- ✓ variabila *a* pentru vector şi variabila n pentru lungimea logică a vectorului.
- ✓ variabila auxiliară *aux* pentru a interschimbă cele două elemente alăturate,
- ✓ variabila *i* la parcurgerea vectorului pentru a verifica dacă două elemente alăturate sunt în relaţia de ordine dorită,

- ✓ variabila *terminat* pentru a şti dacă s-a făcut cel puţin o operaţie de interschimbare la parcurgerea vectorului (deci vectorul încă nu este ordonat).

Variabila *terminat* este o variabilă logică ce se iniţializează cu valoarea 1 la începutul parcurgerii (valoarea logică *True* - se presupune că vectorul este sortat) şi, dacă în timpul parcurgerii se execută o interschimbare, variabilei *terminat* i se atribuie valoarea 0 (valoarea logică *False* - vectorul nu este sortat). Parcurgerea repetată a vectorului se termină atunci când, la sfârşitul unei parcurgeri, variabila *terminat* nu îşi modifică valoarea (îşi păstrează valoarea *1).*

Paşii algoritmului sunt:

Pas 1. Se iniţializează variabila *terminat* cu valoarea 1.

Pas 2. Se iniţializează variabila /cu 0 (i=0) corespunzătoare primului element din vector.

Pas 3. Se compară elementul din poziţia /(a[i]) cu succesorul său (a[i+1]).

Dacă a[i+1]<a[i] cele două elemente se interschimbă (aux=a[i]; a[i]= a[i+1]; a[i+1]=aux;) şi variabilei *terminaţi* se atribuie valoarea 0.

Pas 4. Se trece la următoarea poziţie din vector prin incrementarea lui /(i=i+1)

Pas 5. Se compară /cu numărul de poziţii ale vectorului care trebuie parcurs.

Dacă i<n-1 se revine la **Pas 3**; altfel, se trece la pasul următor.

Pas 6. Se verifică valoarea variabilei *terminat.* Dacă *terminat=0 se* revine la **Pas 1**.

Pas 7. Vectorul este sortat crescător.

Verificarea modului în care se **execută algoritmul:**

prima parcurgere	i=0	4	3	2	1	interschimbare; *terminat=0*
terminat=1	i=1	3	4	2	1	interschimbare; *terminat=0*
	i=2	3	2	4	1	interschimbare; ***terminat=0***
a doua parcurgere	i=0	3	2	1	4	interschimbare; *terminat=0*
terminat=1	i=1	2	3	1	4	interschimbare; ***terminat=0***
	i=2	2	1	3	4	
a treia parcurgere	i=0	2	1	3	4	interschimbare; *terminat=0*
terminat=1	i=1	1	2	3	4	***terminat=0***
	i=2	1	2	3	4	
a patra parcurgere	i=0	1	2	3	4	
terminat=1	i=1	1	2	3	4	
	i=2	1	2	3	4	***terminat=1*** → **terminare**

Secvenţa de instrucţiuni pentru **algoritmul de sortare prin metoda bulelor** este:

```
int i,n,terminat=0, aux, a[50];
cout<<"n= ";cin>>n;
for (i=0;i<n;i++)
{cout<<"a["<<i+1<<"]= ";cin>>a[i];}
     while (! terminat)
       {terminat =1;
          for (i=0;i<n-1;i++)
               if (a[i]>a[i+1])  {aux=a[i];
                                  a[i]=a[i+1];
                                  a[i+1]=aux;
                                  terminat=0;}     }
 for(i=0;i<n;i++)
  cout<<a[i]<<" " ;
getch();}
```

2. Algoritmul de sortare prin metoda inserării

A. Inserare directă

Prin această metodă se împarte vectorul în doi subvectori:

- ✓ subvectorul sursă: a[i], a[i+1],..., a[n-1];
- ✓ subvectorul destinaţie: a[0], a[1],..., a[i-1].

Elementul a[i] din subvectorul sursă este inserat în subvectorul destinaţie conform relaţiei de ordine. Din această cauză vectorul destinaţie va fi tot timpul un vector ordonat.

Se folosesc următoarele **variabile de memorie:**

- ✓ variabila *a* pentru vector şi variabila *n* pentru lungimea fizică a vectorului;
- ✓ variabila *i* la împărţirea vectorului în subvectori: iniţial (la prima împărţire în subvectori) ea are valoarea 1 (subvectorul destinaţie are un singur element), la fiecare nouă împărţire în subvectori se incrementează cu 1, iar operaţia de împărţire în subvectori se termină atunci când variabila *i* are valoarea n-1 (vectorul sursă nu mai conţine nici un element);
- ✓ variabila *j* la parcurgerea secvenţei destinaţie de la dreapta la stânga, pentru a găsi poziţia în care trebuie inserat elementul a[i] conform relaţiei de ordine; la fiecare nouă operaţie de împărţire în subvectori, ea va fi iniţializată cu indicele ultimului element din vectorul destinaţie (i-1);
- ✓ variabila auxiliară *aux,* pentru a salva elementul a[i], care se pierde în urma deplasării spre dreapta în vectorul destinaţie a elementelor care urmează după poziţia în care va fi inserat.

Paşii algoritmului sunt:

Pas 1. Se iniţializează variabila *i* cu valoarea 1 (se face prima împărţire în subvectori).

Pas 2. Se salvează elementul a[i] în variabila aux: aux=a[i].

Pas 3. Se iniţializează variabila *j* cu valoarea i-1 (indicele ultimului element din vectorul destinaţie).

Pas 4. Se compară variabila *aux* cu elementul din poziţia *j* (a[j]) din vectorul destinaţie). Dacă aux<a[j], se deplasează elementul a[j] spre dreapta: a[j+1]= afj]; altfel, se trece la **Pas 7.**

Pas 5. Se decrementează valoarea lui *j* cu 1: j=j-1

Pas 6. Se compară *j* cu indicele primului element din vectorul destinaţie. Dacă j>0 se revine la **Pas 4**; altfel, se trece la pasul următor.

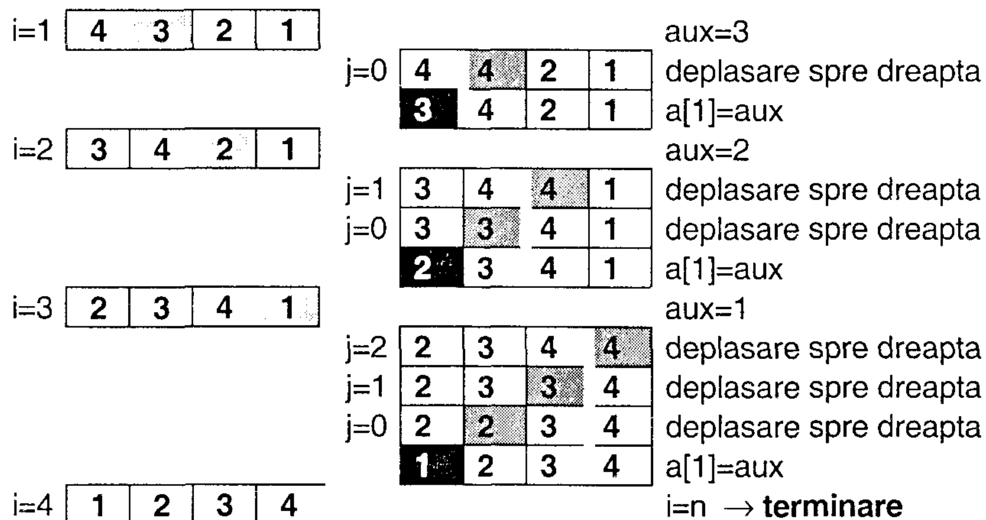
Pas 7. Se compară variabila *aux* cu elementul din poziţia *j* (a[j]) din vectorul destinaţie. Dacă aux>=a[j] se inserează elementul *aux* în poziţia j+1: a[j+1]=aux; altfel, se deplasează elementul din poziţia 0 în poziţia 1 (a[1]=a[0]) şi se inserează elementul *aux* în poziţia 0: a[0]=aux.

Pas 8. Se face următoarea împărţire în subvectori prin incrementarea lui *i* (i=i+1).

Pas 9. Se compară *i* cu indicele ultimului element din vectorul sursă. Dacă i<n-1 se revine la **Pas 2**; altfel, se trece la pasul următor.

Pas 10. Vectorul este sortat crescător.

Verificarea modului în care se **execută algoritmul:**




```
int  i, j, n, aux, a[50];
 cout<<"n =";   cin>>n;
    for   (i=0; i<n; i++)
```

```
{cout<< "a["<<i+1<<" ] =  ";   cin>>a[i];}
      for   (i=1;i<n;i++)  {aux=a[i];
                                  j=i-1;
  while   (j>0  && aux<a[j])
    {a[j+1]=a[j];j--;}
    if (aux>=a[j])   a[j+1]=aux;
      else{a[i]=a[j];
             a[0]=aux;} }
for   (i=0;i<n; i++)
cout<<a[i]<<" ";}
```

B. Sortarea prin inserţie cu diminuarea incrementului (ShellSort)

Principiu: reprezintă o prefecţionare a metodei de sortare prin inserţie. Se alege o secvenţa de numere naturale h1,h2,.. , ht numite incremenţi, care satisfac condiţiile : ht=1 si h(i+1) Se realizează *t* treceri asupra tabloului, la fiecare trecere *i* luându-se in considerare elementele a[1],a[1+hi],a[1+2*hi] …etc. Aceste elemente se sortează aplicând metoda inserţiei, cu utilizarea fanionului. S-a demonstrat că eficienţa algoritmului creşte dacă valorile incremenţilor nu sunt puteri ale lui 2. Pentru a putea folosi tehnica fanionului la această metodă este necesar să se prelungească tabloul *a* spre stânga cu încă h1 elemente.

```
{int i,j,k,h,v;
   int cols[6]={112, 48,21,7,3,1}, a[100],n;
    cout<<"n=" ;cin>>n;
 for (i=1;i<=n;i++)
  {cout<<"a["<<i<<"]= ";cin>>a[i];}
for (k=0;k<6;k++)
```

```
 {h=cols[k];
for (i=h;i<=n;i++) {v=a[i]; j=i;
                        while (j>=h && a[j-h]>v)
                                  {a[j]=a[j-h];j=j-h;}
                              a[j]=v;
}
}
for(i=1;i<=n;i++) cout<<a[i]<<" ";
 getch();}
```

3. Algoritmul de sortare prin metoda selecţiei directe

Prin această metodă se aduce pe prima poziţie elementul cu valoarea cea mai mică din cele *n* elemente ale vectorului, apoi se aduce pe poziţia a doua a elementul cu cea mai mică valoare din ultimele *n-1* elemente ale vectorului, apoi se aduce pe poziţia a treia elementul cu cea mai mică valoare din ultimele *n-2* elemente ale vectorului … etc.

Se folosesc următoarele **variabile de memorie:**

- ✓ variabila a pentru vector şi variabila *n* pentru lungimea logică a vectorului,
- ✓ variabila *i* la parcurgerea vectorului pentru a aduce pe poziţia *i* minimul dintre ultimele elemente (ultimele *n-i-1* elemente): se iniţializează cu 0 (se aduce pe prima poziţie valoarea minimă din vector), se incrementează cu 1 (se aduce pe următoarea poziţie valoarea minimă din restul elementelor), iar operaţia de parcurgere se termină atunci când variabila *i* are valoarea n-2 (s-a adus pe penultima poziţie minimul din elementul penultim şi elementul ultim al vectorului);
- ✓ variabila *j* la parcurgerea ultimelor *n-i-1* elemente pentru a găsi valoarea minimă; la fiecare operaţie de parcurgere a ultimelor elemente, ea va fi

inițializată cu indicele elementului care urmează celui în care se aduce minimul (i+1);

- ✓ variabila auxiliară *aux* pentru a interschimba elementul din poziția curentă cu elementul minim găsit.

Pașii algoritmului sunt:

Pas 1. Se inițializează poziția pe care se aduce minimul. Aceasta va fi prima poziție din vector: i=0.

Pas 2. Se aduce pe poziția *i* a vectorului elementul cu valoarea minimă din cele *n-i-1* elemente ale vectorului, astfel:

Pas 2.1. Se inițializează poziția primului element cu care se compară: j=i+1.

Pas 2.2. Se compară elementul din poziția *i* (a[i]) cu elementul din poziția *j* (a[i]). Dacă a[j]<a[i] cele două elemente se interschimba: aux=a[i]; a[i]= a[j]; a[j]=aux;.

Pas 2.3. Se trece la următorul element pentru comparare, prin incrementarea lui j: j=j+1.

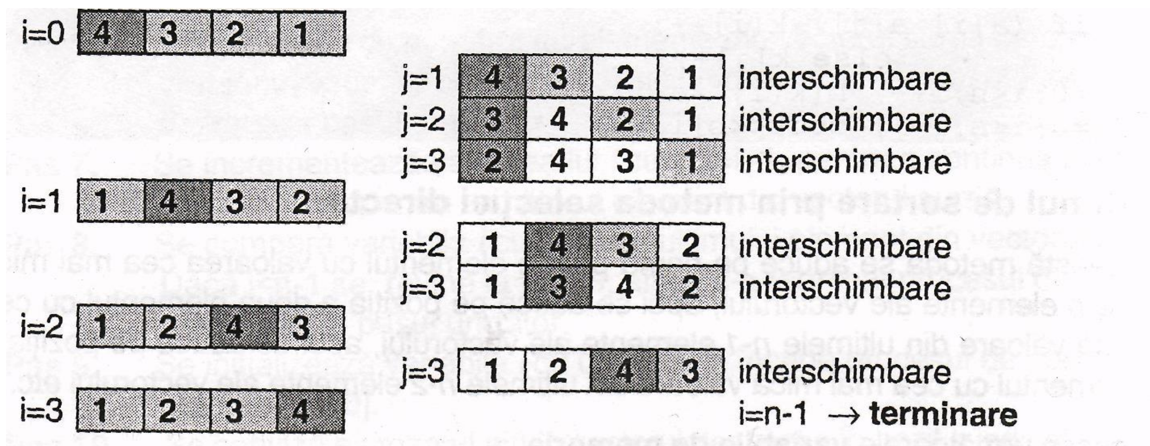
Pas 2.4. Se compară *j* cu numărul de elemente în care se caută minimul pentru a se vedea dacă s-a terminat căutarea minimului în cele *n-i-1* elemente. Dacă j<n se revine la **Pas 2.2.**

Pas 3. Se trece la următoarea poziție din vector prin incrementarea lui *i* (i=i+1) pentru a duce pe această poziție minimul din cele *n-i-1* elemente ale vectorului.

Pas 4. Se compară *i* cu numărul de poziții ale vectorului în care trebuie aduse valorile minime. Dacă i<n-1 se revine la **Pas 2**.

Pas 5. Vectorul este sortat crescător.

Verificarea modului în care se **execută algoritmul:**



Secvenţa de instrucţiuni pentru algoritmul de sortare prin **metoda selecţiei directe** este:

```
{clrscr();
int i, j, n, aux, a[50];
 cout<<"n= ";cin>>n;
   for (i=0;i<n; i++)
    {cout<<"a["<<i+1<<"]= ";cin>>a[i];}
for (i=0;i<n-1; i++)
   for (j=i+1;j<n;j++)
      if (a[j]<a[i]) {aux=a[i]; a[i]=a[j]; a[j]=aux;}
for (i=0;i<n; i++)
cout<<a[i]<<" ";
getch();}
```

4. Algoritmul de sortare prin metoda numărării

În această metodă se folosesc trei vectori:

- ✓ vectorul sursă (vectorul nesortat): a;
- ✓ vectorul destinaţie (vectorul sortat): b;
- ✓ vectorul numărător (vector de contoare): k.

Elementele vectorului sursă a[i] se copiază în vectorul destinaţie prin inserare în poziţia corespunzătoare, astfel încât în vectorul destinaţie să fie respectată relaţia de ordine. Pentru a se cunoaşte poziţia în care va fi inserat fiecare element, se parcurge vectorul sursă a şi se numără în vectorul *k,* pentru fiecare element a[i], câte elemente au valoarea mai mică decât a lui. Fiecare element al vectorului k este un contor: elementul *k[i]* este un contor pentru elementul *a[i].* Valoarea contorului *k[i]* pentru elementul *a[i],* reprezentând câte elemente sunt mai mici decât el, arată de fapt poziţia în care trebuie copiat în vectorul *b.*

Se folosesc următoarele **variabile de memorie:**

- ✓ variabilele *a, b* şi *k* pentru vectori şi variabila *n* pentru lungimea logică a vectorilor; vectorul *k* fiind un vector de contoare, elementele lui se iniţializează cu 0;
- ✓ variabila *i* la parcurgerea vectorului sursă în vederea numărării pentru fiecare element a[i] (se iniţializează cu 0 şi se incrementează până la indicele penultimului element: n-2) şi apoi la parcurgerea vectorului sursă în vederea copierii (se iniţializează cu 0 şi se incrementează până la indicele ultimului element: n-1);
- ✓ variabila *j* la parcurgerea în vectorul sursă a elementelor situate după elementul curent: pentru fiecare element a[i], ea va fi iniţializată cu indicele elementului succesor (i+1) şi se incrementează cu 1 până se ajunge la sfârşitul vectorului a (n-1).

Paşii algoritmului sunt:

Pas 1. Se iniţializează elementele vectorului k cu valoarea 0.

Pas 2. Se iniţializează variabila *i* cu 0 pentru a începe procesul de numărare.

Pas 3. Se iniţializează variabila *j* cu valoarea i+1 (indicele succesorului) pentru a verifica relaţia dintre elementul curent a[i] şi elementele care îi urmează în vector.

Pas 4. Se compară elementul a[i] cu elementul a[j]. Dacă a[i]>a[j], înseamnă că pentru a[i] mai există un element cu valoare mai mică şi se incrementează contorul lui: k[i]= k[i+1]; altfel, înseamnă că pentru a[j] mai există un element cu valoare mai mică şi se incrementează contorul lui: k[j]= k[j+1].

Pas 5. Se incrementează valoarea lui *j* cu 1: j=j+1 pentru a continua procesul de comparare.

Pas 6. Se compară *j* cu numărul de elemente din vectorul sursă care trebuie vizitate în vederea comparării. Dacă j<n se revine la pasul **Pas 4**; altfel, se trece la pasul următor.

Pas 7. Se incrementează valoarea lui *i* cu 1: i=i+1, pentru a continua procesul de numărare pentru următorul element din vectorul sursă.

Pas 8. Se compară variabila *i* cu indicele ultimului element din vectorul sursă.

Dacă i<n-1 se revine la **Pas 3**; altfel, s-a terminat procesul de numărare şi se trece la pasul următor.

Pas 9. Se iniţializează variabila *i* cu 0 pentru a începe procesul de copiere cu elementul a[0].

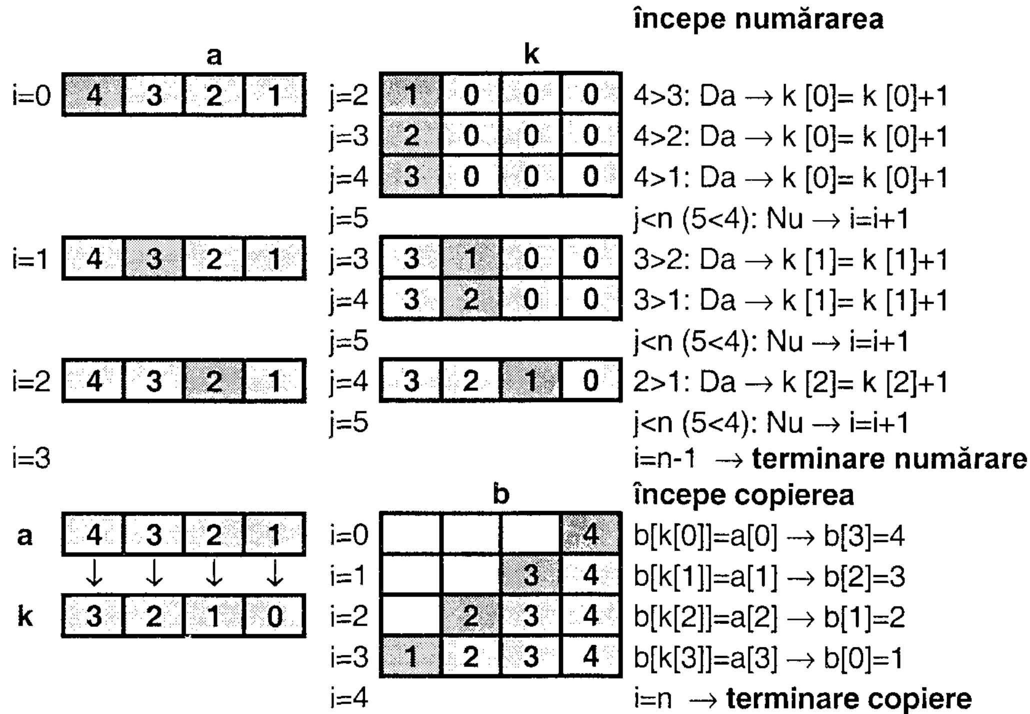
Pas 10. Se copiază elementul a[i] în vectorul destinaţie în poziţia indicată de contorul k[i]: b[k[i]] = a[i].

Pas 11. Se incrementează valoarea lui *i* cu 1: i=i+1, pentru a continua procesul de copiere pentru următorul element din vectorul sursă.

Pas 12. Se compară variabila *i* cu indicele ultimului element din vectorul sursă. Dacă i<n se revine la **Pas 10**; altfel, s-a terminat procesul de copiere şi se trece la pasul următor.

Pas 13. Vectorul b este sortat crescător.

Verificarea modului în care se **execută algoritmul**:



Secvența de instrucțiuni pentru algoritmul de sortare prin **metoda numărării** este:

```
int i,j,n,a[50],b[50],k[50]={0};
 cout<<"n ="; cin>>n;
   for (i=0;i<n;i++)
    {cout<<"a["<<i+1<<"]=";cin>>a[i];}
 for (i=0;i<n-1;i++)
 for (j=i+1;j<n;j++)
   if (a[i]>a[j]) k[i]++;
              else k[j]++;
for (i=0;i<n;i++) b[k[i]]=a[i];
for (i=0; i<n; i++) cout<<b[i]<<" ";
getch();}
```

5. Sortarea prin metoda divide et impera

A. Algoritmul pentru interclasarea a doi vectori sortaţi.

Presupunem că cei doi vectori sunt sortaţi (crescător sau descrescător). Prin acelaşi algoritm se parcurg simultan cei doi vectori sursă pentru a se compara un element dintr-un vector cu un element din celălalt vector. Elementul cu valoarea mai mică respectiv mai mare, este copiat în vectorul destinaţie şi şters din vectorul sursă Procesul continuă până când este epuizat unul dintre vectori. Elementele rămase în celălalt vector se adaugă la sfârşitul vectorului destinaţie.

Se folosesc următoarele **variabile de memorie** pentru:

- ✓ **vectori: a şi b -** vectorii sursă (se presupune că sunt deja sortaţi crescător sau descrescător) şi c - vectorul rezultat (vectorul destinaţie) care reuneşte elementele din vectorii a şi b, ordonate după acelaşi criteriu ca şi cele din vectorii sursă:
- ✓ **lungimile logice ale vectorilor: n** - pentru vectorul a (se introduce de la tastatură) şi m - pentru vectorul *b* (se introduce de la tastatură); lungimea vectorului c se calculează (n+m);
- ✓ **contori pentru parcurgerea** vectorilor: **i** - pentru vectorul a, **j** - pentru vectorul *b* şi k - pentru vectorul *c*.

Paşii algoritmului (pentru vectori sortaţi crescător} sunt:

Pas 1. Se iniţializează variabilele contor, pentru vectorii sursă şi pentru vectorul rezultat, cu 0: i=0; j=0; k=0;.

Pas 2. Se compară elementele a[i] şi b[j]. Dacă a[i]<b[j] se copiază elementul a[i] în vectorul c prin c[k]=a[i] şi se şterge elementul a[i] din vectorul *a*; altfel, se copiază elementul b[j] în vectorul *c* prin c[k]=b[j] şi se şterge elementul b[j] din vectorul *b*. Operaţia de ştergere nu se execută prin eliminarea efectivă a elementului din vector, ci prin incrementarea contorului: dacă se şterge elementul din vectorul *a* se

incrementează contorul i(i=i+1), iar dacă se şterge elementul din vectorul *b* se incrementează contorul j (j-j+1)-

Pas 3. Se incrementează contorul *k* pentru că urmează să se copieze un element în acest vector.

Pas 4. Se compară contorii vectorilor sursă cu lungimea lor. Dacă fiecare contor este mai mic sau egal cu lungimea vectorului (i<n şi j<m) se revine la **Pas 2**; altfel, se trece la pasul următor.

Pas 5. Se adaugă la sfârşitul vectorului *c* elementele rămase în celălalt, printr-o simplă operaţie de copiere: dacă i<n înseamnă că s-a epuizat vectorul *b* şi se vor adăuga la vectorul *c* elementele rămase în vectorul *a* (se execută c[k]=a[i]; k=k+1; i=i+1; până când *i* devine egal cu lungimea *n* a vectorului *a*); altfel, s-a epuizat vectorul *a* şi se vor adăuga la vectorul *c* elementele rămase în vectorul *b* (se execută c[k]=b[j]; k=k+1; j=j+1; până când *j* devine egal cu lungimea *m* a vectorului *b).*

Secvenţa de instrucţiuni pentru **interclasarea a doi vectori sortaţi crescător** este:

```
{int i, j, k, n, m, a[50], b[50], c[100];
cout<<"n= "; cin>>n;
cout<<"m= "; cin>>m;
for (i=0;i<n; i++)  {cout<<"a["<<i+1<<"]= ";cin>>a[i];}
for (j=0;j<m; j++) {cout<<"b["<<j+1<<"]= " ;cin>>b[j];}
  i = 0;   j=0;   k=0;
while   (i<n &&  j<m)
  {if   (a[i]<b[j])  {c[k]=a[i];j++;}
              else   {c[k]=b[j];j++;}
   k++;}
   if (i<n)
    while (i<n) {c[k] =a[i]; k++; i++;}
 else
```

```
while (j <m) {c[k]=b[j];  k++;   j ++;}
  for (k=0;k<n+m;k++)
    cout<<c[k]<<" " ;}
```

B. Sortare rapidă (Quicksort)

Algoritmul de sortare rapidă a fost propus de către C.A. Hoare şi se bucură de o apreciere foarte bună între diferiţii algoritmi de sortare. Această apreciere este corectă, cu observaţia că acest algoritm funcţionează destul de slab tocmai pentru situaţia în care şirul ce urmează a fi sortat este din start în ordinea corectă. Acest lucru va deveni evident după ce vom lămuri modul în care funcţionează algoritmul. Pentru a exemplifica modul de lucru al algoritmului să considerăm şirul de 8 numere

1,8,5,4,3,6,2,7.

Subprogramele de mai jos au ca intrări: un vector *V* în care sunt memorate numere întregi, indicii *p* şi *q* care precizează poziţia de început respectiv de sfârşit a şirului ce urmează a fi sortat.

Vom spune că indicele *m* separă şirul definit de indicii *p* şi *q* dacă subşirurile 57, 52 definite de indicii *p, m-1* respectiv *m+1,* au proprietatea că orice element din 57 este mai mic decât *V[m]* şi orice element din 52 este mai mare decât *V[m].*

Implementarea recursivă a algoritmului de sortare rapidă poate fi scris în pseudocod astfel:

dacă p < u atunci determină o partiţie 57, 52 *a şirului sortează subşirul* 57 *sortează subşirul* 52.

Pentru a separa şirul definit de *p* şi *q* se atribuie variabilelor *i* şi *j* valorile *p* şi respectiv *q*. Iniţial se fixează semaforul *s* pe 7 *(adevărat).*

La o trecere are loc fie o deplasare spre stânga a indicelui j (dacă *s-l),* fie o deplasare spre dreapta a indicelui i (dacă *s=0).* Dacă *V[i] > V[j],* cele două valori se

schimbă intre ele şi semaforul s este basculat pe valoarea opusă. Procesul se opreşte când *i=j,* moment în care se obţine o separare a şirului prin indicele *m = i,*

Faptul că se obţine o separare a şirului rezultă din observaţia că la fiecare pas valorile care se află în exteriorul segmentului *[i, j]* sunt separate şi orice operaţie din cele descrise mai sus menţin această proprietate.

Se observă de asemenea că pentru a separa un şir este necesar un număr de operaţii proporţional cu numărul de elemente ale acestuia. Dacă la fiecare separare indicele *m* ar ocupa o poziţie de mijloc în vectorul iniţial atunci am putea spune că algoritmul are complexitatea *O(n*ln(n)).* Din nefericire acest lucru nu poate fi garantat. Mai mult, dacă vectorul iniţial este sortat atunci separarea se face la unul din capetele şirului. Acest ultim argument ne îndreptăţeşte să afirmăm că în cea mai proastă situaţie complexitatea algoritmului este $O(n^2)$.

Implementarea în C++ a algoritmului de sortare rapidă.

```
void Sortare_Rapida(int *v, int p, int u)
{ if (p<u)
{
int i=p;
int j=u;
char s=1;
while (i < j)
{
if (v[i]>v[i])
{
int t = v[i];
v[i]=v[j];
v[j]=t;
s=!s;
}
```

```
}
if (p < i-1) Sortare_Rapida(v, p, i-1);
if (i+1 < u) Sortare_Rapida(v, i+1, u);
}
}
```

Capitolul IV. APLICAŢII

1. SORTARE PRIN METODA BULELOR (bubblesort)

Permutări lexicografice (programul este salvat sub denumirea de aplic_bu)
Pentru numărul natural n (n > 1) dat, determinaţi toate permutările mulţimii { l, 2, . . ., n } în ordine lexicografică.

Date de intrare: Se citeşte de la tastatură un număr natural n, 1 < n < 20.

Date de ieşire: în fişierul "permutări.out" afişaţi, câte una pe linie, permutările mulţimii { l, 2, ..., n }.

Exemplu:

Tastatura	**permutări.out**
n = 3	1 2 3 1 3 2 2 1 3 2 3 1 3 1 2 3 2 1

Analiza problemei şi proiectarea soluţiei:

Mulţimea permutărilor de rang *n* este o mulţime ordonată, pe care se poate stabili o relaţie de ordine, în care există deci un prim element (care nu are predecesor) şi un ultim element (care nu are succesor). Vom descrie în continuare această ordine pe mulţimea de permutări:

- primul element este permutarea {l, 2, 3, ...,n}
- ultimul element este permutarea {n, n-1, n-2, ..., 2, 1 }
- aflarea succesorului unei permutări (în afară de ultima):

Fie permutarea $p_1\ p_2\ p_3 \ldots p_k \ldots p_n$.

Succesoarea acestei permutări în cadrul ordinii lexicografice se determină astfel:

Pas 1. se caută primul element de la dreapta la stânga care satisface $p_k > p_{k+1}$ (k<n);

Pas 2. pentru acest *k* fixat se caută primul element de la dreapta la stânga, fie acesta p_t, care satisface $p_t > p_k$ (acesta există pentru că îl avem cel puţin pe p_{k+1} care satisface condiţia);

Pas 3. se interschimbă p_k cu p_t;

Pas 4. se inversează subsecventa p_{k+i} ... p_n.

Permutarea astfel obţinută are proprietatea că este cea mai mică permutare mai mare decât P! p_2 . . . p_n.

Exemplu: presupunem că n = 6 şi că trebuie să determinăm succesoarea permutării:

6 3 5 4 2 1

Primul *k* de la dreapta la stânga pentru care $p_k < p_{k+1}$ este k = 2 (deoarece 3 < 5 şi unice pereche de la dreapta acesteia nu satisface inegalitatea). Pentru acest *k* fixat, primul element de la dreapta la stânga mai mare decât p_2 este $p_4 = 4$. Interschimbăm pe 3 cu 4 şi obţinem permutarea:

6 4 5 3 2 1

Ultimul pas este oglindirea secvenţei de după 6 5 4 3 2 l şi obţinem:

1 2 3 4 5 6

Aceasta este succesoarea permutării date în cadrul ordinii lexicografice. Deoarece lucrăm în C++, vom considera elementele permutării de la 0 la n-1 şi le vom incrementa la afişare

```
#include <iostream.h>
#include <conio.h>
#include <fstream.h>
```

```
  ofstream fişier ("permutari.out");
   void Citeste (int &n)
  {cout  <<" n = ";cin   >>n; }
  void ScriePermutare (int *x, int  n)
{
    for (int  i=0;   i<n;   i++)
      f<< x[i]+1<<" "; f << endl;
}
  void Genereaza ( int n )
{
 int i, k, t, aux;
 int x[100] ;
for ( i=0; i<n; i++ ) x[i] = i;
  ScriePermutare (x, n);
      k=n-2;
while(k>= 0)
{
k=n-2;
// cauta k-ul
  while ( x[k] > x[k+1] && k >= 0 ) k--;
      if (  k >= 0 )
{
//  cauta   t-ul
  t = n-1;
     while (  x[t]   < x[k])   t --;
// interschimba xk cu xt
        aux = x[k];
        x[k]=x[t];
```

```
        x[t]=aux;
//oglindeste subsecventa xk+1 . . . xt
  for(  i=k+1;  i  <=  (k+n)/2;  i++)
{       aux = x[i];
        x[i] = x[n+k-i];
        x[n+k-i] = aux;}
}
if(  k >= 0)
ScriePermutare(  x,   n  );
}
}
void main()
{
 int  n;
 Citeste (n);
  Genereaza (n);
getch();}
```

2.SORTARE PRIN INSERARE DIRECTĂ (program salvat ***apl_inse)***

Se dă un vector. Folosindu-se metoda inserţiei directe să se sorteze crescător vectorul.

```
#include <iostream.h>
#include <conio.h>
void main()
{clrscr();
int  i, j, n, aux, a[50];
```

```
 cout<<"n =";   cin>>n;
    for (i=0;i<n;i++)
       {cout<< "a["<<i+1<<"] = ";cin>>a[i];}
for(i=1;i<n;i++)
    {aux=a[i];  j=i-1;
while   (j>0  && aux<a[j])   {a[j+1]=a[j];   j--;}
    if (aux>=a[j])   a[j+1]=aux;
     else   {a[i]=a[j];   a[0]=aux;} }
for (i=0;i<n; i++)   cout<<a[i]<<" ";
getch();}
```

3. SORTARE PRIN MICŞORAREA INCREMENTULUI (SHELLSORT)

(program salvat ca *apl_shel)*

Dându-se un sir de numere să se sorteze utilizând metoda micşorării incrementului.

```
#include <iostream.h>
#include <conio.h>
void main ()
{clrscr();
int i,j,k,h,v;
int cols[6]={112, 48,21,7,3,1}, a[100],n;
cout<<"n=" ;cin>>n;
for (i=1;i<=n;i++)
   {cout<<"a["<<i<<"]= ";cin>>a[i];}
for (k=0;k<6;k++)
{h=cols[k];
for (i=h;i<=n;i++) {v=a[i]; j=i;
```

```
                        while (j>=h && a[j-h]>v)
                              {a[j]=a[j-h];j=j-h;}
                            a[j]=v;
}
}
for(i=1;i<=n;i++) cout<<a[i]<<" ";
getch();}
```

4. SORTARE PRIN SELECTIE DIRECTĂ

(program salvat ca *apl_sele*)

Dându-se un vector să se afişeze elementele sale, crescător şi decrescător; la fiecare sortare să se afişeze un mesaj.

```
#include<iostream.h>
#include<conio.h>
int a[10],n,i,k,aux,gasit;
void main()
{
clrscr();
cout<<"Introduce numarul de elemente"<<" "<<"n=";cin>>n;
for(i=1;i<=n;i++)
{cout<<"a["<<i<<"]=";cin>>a[i];};
do
 {
 gasit=0;
 for(i=1;i<=n-1;i++)
 if(a[i]>a[i+1]) {aux=a[i] ;
```

```
                a[i]=a[i+1];
                a[i+1]=aux;
                gasit=1;}
  } while(gasit);
cout<<"Listez numerele in ordine crescatoare"<<endl;
for(i=1;i<=n;i++) cout<<a[i]<<" ";
  cout<<endl<<"Listez numerele in ordine descrescatoare"<<endl;
  for(i=n;i>=1;i--)cout<<a[i]<<" ";
getch();
}
```

5. SORTARE PRIN INTERCLASARE

(program salvat ca *crefis şi mergesor*)

Folosind un fişier de intrare cu n<=10001 să se calculeze timpul de executie necesar ordonării numarului de elemente introduse aleatoriu într-un fişier de intrare.

Folosim 2 programe:

- crefis, unde se generează aleatoriu printr-o funcţie randomize() din biblioteca stdlib.h numărul n de elemente dorit (este bine ca acest n să fie foarte mare, până la 10000)
- mergesor unde se definesc timpii de început, de sfârşit şi diferenţa de execuţie de tip structură; se construiesc apoi funcţiile de citire, afişare şi de sortare prin interclasare. Foarte importantă este şi funcţia void timp care prin reportul r face diferenţa între timpul la care începe programul să ruleze şi cel la care se termină.

```
#include<fstream.h>
```

```
#include<dos.h>
#include<stdio.h>
struct  time t1,t2,t,tt;
ifstream f("fis.in");
int n,x[10001];
void cit()
{f>>n;
 for(int i=1;i<=n;i++) f>>x[i];
 f.close();
}
void interclas(int s,int d,int m)
{int i=s,j=m+1,v[10001],k=0;
 while(i<=m && j<=d)
if(x[i]<x[j]) v[++k]=x[i++];
     else v[++k]=x[j++];
 while(i<=m) v[++k]=x[i++];
   while(j<=d) v[++k]=x[j++];
 for(k=1,i=s;i<=d;k++,i++) x[i]=v[k];
}
void mergesort(int s,int d)
{int m;
 if(s<d)
  {m=(s+d)/2;
   mergesort(s,m);
   mergesort(m+1,d);
   interclas(s,d,m);
  };
}
```

```
void afis()
{int i;
 for(i=1;i<n;i++) cout<<x[i]<<' ';
 cout<<'\n';
}
void timp(time t1,time t2)
{int r=0;
 printf("The current time is: %2d:%02d:%02d.%02d\n",
            t1.ti_hour, t1.ti_min, t1.ti_sec, t1.ti_hund);
  printf("The current time is: %2d:%02d:%02d.%02d\n",
            t2.ti_hour, t2.ti_min, t2.ti_sec, t2.ti_hund);
 t.ti_hund=t2.ti_hund-t1.ti_hund;
 tt.ti_hund=0; tt.ti_sec=0;
 if(t.ti_hund<tt.ti_hund)
 {t.ti_hund+=100; r=1;};
 t.ti_sec=t2.ti_sec - t1.ti_sec-r;
 if(t.ti_sec<tt.ti_sec) {t.ti_sec+=60; r=1;} else r=0;
 printf("timpul de executie %02d.%02d\n",t.ti_sec, t.ti_hund);
}
void main()
{gettime(&t1);
 cit();
 mergesort(1,n);
 afis();
 gettime(&t2);
 timp(t1,t2);
}
```

6. SORTARE RAPIDĂ (QUICKSORT)

(program salvat ca quiqsor)

Având un vector de 50 elemente maxim, să se sorteze prin metoda QuickSort

```
#include <iostream.h>
#include<conio.h>
#include <string.h>
void main ()
{int i,j,n,aux,st,dr,mij1 ,a[50];
cout<<"n ="; cin>>n;
for (i = 0;i<n;i++) {cout<<"a["<<i+1<<"]=";cin>>a[i];}
for (i=1;i<n;i++) {aux=a[i];
                   st=0;
                   dr=i-1;}
while (st<=dr) {mij1=(st+dr)/2;
   if (aux<a[mij1])  dr=mij1-1;
              else st=mij1+1;} j=i-1;
while (j>=st)   {a[j+1]=a[j];
                 j=j-1;
                a[st]=aux;}
for (i = 0; i<n; i++)
   cout<<a[i]<<" ";
getch();}
```

APLICAŢIE GENERALĂ PENTRU TOATE METODELE DE SORTARE PREZENTATE ÎN CAPITOLUL III

(program salvat ca multi_so)

Se dă un vector un vector de max 100 elemente. Cerinţe:

- să se afişeze vectorul nesortat
- să se sorteze crescător prin metodele discutate

```
#include <stdio.h>
#include <conio.h>
#define nmax 100
/* ALGORITMI DE SORTARE */

void citire_vector(int n,float a[nmax],float b[nmax])
        /* CITIRE ELEMENTE VECTOR */
        {
         int i;
         printf("\nIntroduceti elementele vectorului de sortat\n");
         for(i=0;i<n;i++)
           {
            printf("a[%d]=",i);
            scanf("%f",&a[i]);
            b[i]=a[i];
           }
        }

        void reconstituire(int n,float a[nmax],float b[nmax])
        /* RECONSTITUIREA INITIALA A VECTORULUI a */
        {
         int i;
         for(i=0;i<n;i++)
           a[i]=b[i];
        }
```

```
void afisare(int n,float a[nmax])
/* AFISARE VECTOR */
{
 int i;
 for(i=0;i<n;i++)
   {
     printf("%8.2f",a[i]);
     if(((i+1) % 10)==0) printf("\n");
   }
}

 void  sort_numarare(int n,float a[nmax])
 /*  SORTAREA PRIN NUMARARE  */
{
 int i,j;
 float b[nmax];
 int c[nmax];
 for(i=0;i<n;i++)
   {
     c[i]=0; /* initializare vector de numarare */
     b[i]=a[i]; /* copiere vector a in b */
   }
 /* numarare */
 for(j=1;j<n;j++)
   for(i=0;i<=j-1;i++)
     if(a[i]<a[j]) c[j]++;
     else c[i]++;
 /* rearanjare vector a */
 for(i=0;i<n;i++)
   a[c[i]]=b[i];
}
```

```
void sort_inserare_directa(int n,float a[nmax])
/* SORTARE PRIN INSERARE DIRECTA */
{
  int i,j;
  float x;
  for(j=1;j<n;j++)
    {
        x=a[j]; i=j-1;
        while((i>=0) && (x<a[i]))
               {
                  a[i+1]=a[i];
                   i=i-1;
                 }
           a[i+1]=x;
      }
 }

 void shell_sort(int n,float a[nmax])
 /* SORTAREA PRIN METODA SHELL */
 {
    int i,j,incr;
    float x;
    incr=1;
    while(incr<n)
              incr=incr*3 + 1;
    while(incr>=1)
      {
               incr=incr / 3;
               for(i=incr;i<n;i++)
                     {
                         x=a[i];j=i;
                         while(a[j-incr]>x)
                                 {
```

```
                                    a[j]=a[j-incr];
                                    j=j-incr;
                                    if(j<incr) break;
                              };
                        a[j]=x;
                  }
      }
}

void sort_metoda_bulelor(int n,float a[nmax])
/* SORTAREA PRIN INTERSCHIMBARE-METODA BULELOR */
{
 int i,j,gata;
 float x;
 j=0;
 do{
          gata=1;
          j=j+1;
          for(i=0;i<n-j;i++)
           if(a[i]>a[i+1]){
                            gata=0;
                            x=a[i];a[i]=a[i+1];a[i+1]=x;
                          };
     }while(gata==0);
}

void quick(int prim,int ultim,float a[nmax])
{
   int i,j;
   float pivot,x;
   i=prim;j=ultim;
   pivot=a[(prim + ultim) / 2];
   do{
```

```
            while(a[i]<pivot) i++;
            while(a[j]>pivot) j--;
            if(i<=j) {
                        x=a[i];a[i]=a[j];a[j]=x;
                        i++;j--;
                     };
         }while(i<=j);
    if(prim<j) quick(prim,j,a);
    if(i<ultim) quick(i,ultim,a);
}

void quicksort(int n,float a[nmax])
/* SORTAREA  RAPIDA QUICKSORT */
{
     quick(0,n-1,a);
 }

 void sort_selectie(int n,float a[nmax])
 /* SORTAREA PRIN SELECTIE */
 {
   int i,j,poz;
   float x;
   for(i=0;i<n-1;i++)
         {
            x=a[i];poz=i;
            for(j=i+1;j<n;j++)
            if(a[j]<x) {
                          x=a[j];
                          poz=j;
                        }
            a[poz]=a[i];a[i]=x;
           }
     }
```

```
void main(void)
{
   int i,n;
   float a[nmax],b[nmax];
   clrscr();
   printf("\nIntroduceti nr.elementelor n=");
   scanf("%d",&n);
   citire_vector(n,a,b);
   printf("\nVECTORUL NESORTAT\n");
   afisare(n,a);
   printf("\nVECTORUL SORTAT PRIN METODA NUMARARII\n");
   sort_numarare(n,a);
  afisare(n,a);
  getch();
  printf("\nVECTORUL SORTAT PRIN METODA INSERARII DIRECTE\n");
  reconstituire(n,a,b);   /* a devine vectorul  nesortat initial */
  sort_inserare_directa(n,a);
  afisare(n,a);
  getch();
  printf("\nVECTORUL SORTAT PRIN METODA SHELL\n");
  reconstituire(n,a,b); /* a devine vectorul  nesortat initial */

   shell_sort(n,a);
   afisare(n,a);
   getch();
   printf("\nVECTORUL SORTAT PRIN METODA BULELOR\n");
   reconstituire(n,a,b); /* a devine vectorul  nesortat initial */
   sort_metoda_bulelor(n,a);
   afisare(n,a);
   getch();
   printf("\nVECTORUL SORTAT PRIN METODA QUICKSORT\n");
```

```
reconstituire(n,a,b); /* a devine vectorul  nesortat initial */
quicksort(n,a);
afisare(n,a);
getch();
printf("\nVECTORUL SORTAT PRIN METODA SELECTIEI DIRECTE\n");
reconstituire(n,a,b); /* a devine vectorul  nesortat initial */
sort_selectie(n,a);
afisare(n,a);
getch();
}
```

CAPITOLUL V. ANALIZA TIMPULUI DE EXECUŢIE

Eficienţa algoritmilor

Există situaţii când sunt mai mulţi algoritmi pentru rezolvarea aceleiaşi probleme şi practic trebuie ales algoritmul cel mai eficient. **Eficienţa unui algoritm este** evaluată prin timpul **necesar executării algoritmului.** Ştim şi că, pentru a compara doi algoritmi care rezolvă aceeaşi problemă, se foloseşte aceeaşi **dimensiune a datelor de intrare** (acelaşi număr de valori pentru datele de intrare) adică una dintre următoarele metode de evaluare a eficienţei:

- **numărul de operaţii elementare ale algoritmului** (dacă se poate preciza) sau
- **timpul mediu de execuţie.**

Este foarte important să determinăm corect dimensiunea datelor de intrare. De exemplu, în cazul sortării unui vector cu *n* componente, dimensiunea datelor de intrare este *n,* iar în cazul determinării tuturor permutărilor unei mulţimi cu *n* elemente, dimensiunea datelor de intrare este *n.*

Pentru a compara doi algoritmi folosind numărul de operaţii elementare, trebuie să stabilim unitatea de măsură pe care o vom folosi, adică **operaţia de bază** executată în cadrul algoritmului, şi să numărăm apoi în cazul fiecărui algoritm de câte ori se execută ea.

Operaţia de bază este o operaţie elementară sau o succesiune de operaţii elementare a căror execuţie nu depinde de valorile datelor de intrare.

În alegerea operaţiei de bază trebuie să se aibă în vedere ca numărul său de execuţii în cadrul algoritmului să se poată calcula funcţie de dimensiunea datelor de intrare *n.*

Mărimile care pot fi folosite în compararea a doi algoritmi sunt:

- **eficienţa**
- **ordinul de complexitate.**

Eficienţa unui algoritm o reprezintă timpul de calcul estimat prin numărul de execuţii ale operaţiei de bază.

Eficienţa se notează cu **T(n)** şi este o funcţie care depinde de variabila *n* - dimensiunea datelor de intrare. Astfel, dacă într-un algoritm se execută $3n^2 + 2n + 3$ operaţii de bază, eficienţa sa va fi $T(n) = 3n^2 + 2n + 3$. În general, pentru a compara doi algoritmi, nu este nevoie să se determine eficienţa lor, ci numai factorul care determină acestei mărimi şi care este denumit **ordinul de complexitate.**

Ordinul de complexitate al unui algoritm îl reprezintă timpul de calcul estimat prin ordinul de mărime al numărului de execuţii ale operaţiei de bază.

Ordinul de complexitate se notează cu **O(f(n)),** unde **f(n)** reprezintă termenul determinant al numărului de execuţii ale operaţiei de bază - termenul determinant din funcţia T(n). Pentru determinarea ordinului de complexitate se porneşte de la funcţia T(n) - în exemplul nostru $T(n) = 3n^2 + 2n + 3$ - şi se execută următoarele operaţii:

1. Coeficienţii termenilor din expresia T(n) se reduc la valoarea 1. În exemplu, $f(n) = n^2 + n + 1$.
2. Se păstrează din expresie termenul determinant, în exemplu, $f(n) = n^2$, deoarece pentru valori foarte mari ale lui n valoarea lui n +1 este neglijabilă faţă de cea a lui n^2.

Aşadar, dacă eficienţa unui algoritm este dată de funcţia $T(n) = 3n^2 + 2n + 3$, spunem că acest algoritm are ordinul de complexitate $O(n^2)$.

Astfel, în funcţie de ordinul de complexitate, există următoarele tipuri de algoritmi:

Ordin de complexitate	Tipul algoritmului
0(n)	Algoritm liniar.
$0(n^m)$	Algoritm polinomial. Dacă m=2, algoritmul m=3, este pătratic, iar dacă algoritmul este cubic.
$0(k^n)$	Algoritm exponenţial. De exemplu, 2^n, 3^n etc. Algoritmul de tip O(n!) este tot de tip exponenţial deoarece: $1*2*3*4*...*n > 2*2*2*...*2 = 2^{n"1}$.

O(logn)	Algoritm logaritmic.
O(n logn)	Algoritm liniar logaritmic.

Tipul algoritmului este foarte important deoarece de el depinde timpul de execuţie şi implicit eficienţa algoritmului. Pentru exemplificare să presupunem că pentru executarea unei operaţii de bază sunt necesare 10^{-9} secunde (adică într-o secundă se execută un miliard de operaţii de bază), iar dimensiunea datelor de intrare este *n.* în următorul tabel puteţi compara timpii de execuţie necesari pentru diferite tipuri de algoritmi şi pentru diferite dimensiuni ale datelor de intrare (valori ale lui n):

0(n)	n=10	n=20	n=30	n=40	n=50
Iog_2n	3,32	4,32			
n	10^{-8} s	$2*10^{-8}$ s	$3*10^{-8}$s	$4*10^{-8}$s	$5*10^{-8}$ s
n^2	10^{-7}s	$4*10^{-7}$s	$9*10^{-7}$s= 10^{-6}s	$16*10^{-7}$ s=1,6* 10^{-5} s	$125*10^{-6}$ s = $12,5*10^{-4}$ s
n^3	10^{-6}s	$8*10^{-6}$s	$27*10^{-6}$ s = 3*1 0^{-5} s	$64*10^{-5}$s=$6,4*10^{-5}$ s	125*10-^{6}s = $1,25*10^{-4}$
2^n	$2^{10}*10^{-9}$ s=10^{-6}s	$2^{20}*10^{-9}$ =10^{-3}s	$2^{30}*10^{-9}$s= 1s	$2^{40}*10^{-9}$ s =10^3s = 16,7 m	$2^{50}*10^{-9}$ s = 10^6s=11,57 zile
3^n	$3^{10}*10^{-9}$ s = $5,9*10^5$s	$3^{20}*10^{-9}$s= 3,5 s	$3^{30}*10^{-9}$ S= $2*10^5$s = 2,31 zile	$3^{40}*10^{-9}$ s= $1,2*10^{10}$ s= 3,85 secole.	$3^{50}*10^{-9}$ s =$7*10^{14}$s = $2,3*10^5$ secole

Concluzii:

1. Cei mai rapizi algoritmi sunt cei logaritmici.
2. Dacă pentru rezolvarea aceleiaşi probleme există algoritmi polinomiali şi exponenţiali, se preferă cei polinomiali.

3. Dacă pentru rezolvarea aceleiaşi probleme există mai mulţi algoritmi polinomiali, se preferă cel cu gradul mai mic.

Aşadar, ordinul de complexitate este o funcţie dependentă de dimensiunea datelor de intrare şi este determinat de structurile repetitive care se execută cu acea mulţime de valori pentru datele de intrare. Astfel:

Structura repetitivă	Numărul de execuţii ale corpului structurii	Tipul algoritmului
for (i=l;i<=n; i=i+k) {....}	f(n)=n/k -» O(n)=n	Liniar
for (i=l;i<=n;i=i*k) {....}	f(n)= log^n -> O(n)= logn	Logaritmic
for (i=n;i>=l;i=i/k) {....}	f(n)= $\log_k n$ -> O(n)= logn	Logartmic
for (i=n; i<=n; i=i+p) {....} for (j=n; j<=n;j=j+q) { ----- }	f(n)=(n/p)*(n/q) = n^/(p*q) -* 0(n)= n^2	Polinomial pătratic
for (i=n; i<=n; i=i++) {....} for (j=i; j<=n;j=j++) {....}	f(n)=1+2+3+ ... +n =(n*(n+1))/2 -> 0(n)= n^2	Polinomial pătratic

Observaţie: în cazul structurilor repetitive imbricate, eficienţa va fi dată de produsul dintre numărul de repetiţii ale fiecărei structuri repetitive. De exemplu, în cazul algoritmului de sortare prin metoda selectării directe, dacă vom considera ca operaţie de bază comparaţia care se execută în cele două structuri repetitive imbricate:

- ✓ for (i=l;i<n-l;i++)
- ✓ for (j=i+l;j<n;j++) if (a[j]<a[i]) {...}

Eficienţa este: f(n)=(n-1) + (n-2) + ... + 2 + 1 =(n*(n-1))/2, iar ordinul de complexitate este O(n)= n^2. Algoritmul este pătratic.

Există cazuri în care nu se poate determina numărul de operaţii de bază. în acest caz se vor putea folosi trei valori:

- **timpul minim de execuţie** - corespunde cazului cel mai favorabil.
- **timpul maxim de execuţie** - corespunde cazului cel mai defavorabil.
- **timpul mediu de execuţie** - calculează media timpilor de execuţie pentru fiecare caz posibil.

De exemplu, dacă vom considera algoritmul de sortare prin metoda bulelor şi vom lua ca operaţie de bază comparaţia, apar următoarele cazuri:

- Cazul cel mai favorabil - vectorul este deja sortat. Se vor executa *n-1* comparaţii (corespunzând unei parcurgeri a vectorului). Timpul minim de execuţie este *n-1*.
- Cazul cel mai defavorabil - vectorul este sortat invers. Se vor executa *(n-1)*n* comparaţii (vectorul se va parcurge de *n* ori şi la fiecare parcurgere se vor executa *n-1* comparaţii). Timpul maxim de execuţie este *(n-1)*n.* Ordinul de complexitate este $O(n)= n^2$, iar algoritmul este pătratic.

STUDIU DE CAZ: ilustrarea eficienţei algoritmilor de sortare implementaţi în limbajul C++

Programul permite compararea timpului de execuţie a trei dintre algoritmii standard de sortare: sortarea prin metoda bulelor (sau selecţie) , sortarea prin inserţie şi sortarea Shell.

Se dă un vector k ce se va umple cu N numere întregi aleatoare. Se poate testa timpul de execuţie pentru fiecare algoritm în parte, dând programului numărul de elemente ale vectorului, sau se poate efectua un test complet (în care se vor folosi toate cele trei metode pe un vector cu 10000 de elemente generate aleator). La final se va afişa timpul de execuţie al fiecărei metode în bătăi de ceas intern şi secunde. Pentru a asigura o testare cât mai aproape de adevăr şi pentru a elimina diferenţele

de caz (favorabil - nefavorabil) determinate de generarea aleatoare se va genera o singură dată vectorul cu numere aleatoare, pentru ca după terminarea sortării cu o metodă, structura vectorului să fie restaurată.

Se recomandă alegerea unei dimensiuni a vectorului de peste 3000 de elemente pentru ca testul să fie cât mai edificator (cel mai bine să fie aleasă opţiunea cu test complet, pentru 10000 de elementa generate aleatoriu). Dimensiunea maximă a vectorului poate fi de 10000 de elemente.

Programul conţine în comentarii toate explicaţiile necesare înţelegerii rulării lui şi este salvat sub numele tmp_exec.cpp.

```
/*
Exemplu de testare a eficientei pentru trei dintre algoritmii de sortare standard:
- bubble sort
- insertie
- shell
*/
#include <iostream.h>
#include <stdio.h>
#include <conio.h>
#include <time.h>
#include <stdlib.h>
clock_t t0,t1;
int k[60000],n,i,v,p,opt,k_salvat[60000],t_bule,t_insertie,t_shell;
//================================================
void insertie()
{
int c=0;
cout<<"Sortez vectorul...";
t0=clock(); //citeste timpul initial (referinta)
//Sortare prin insertie
```

```
for(i=2;i<=n;i++)
{
// Insereaza Ki
v=k[i];
p=i-1;
while(p>0 && k[p]>v) // sa fie ordonat crescator
{
k[p+1]=k[p];
p=p-1;
}
k[p+1]=v;
if(c==n/60){ //progresul operatiei
c=0; //reseteaza contorul (s-a afisat o steluta)
cout<<"*";
}
c++; //incrementeaza contorul ce se reseteaza dupa afisarea unei stelute de progres
} //endfor
cout<<"OK\n";

cout<<"Vectorul a fost sortat cu metoda de sortare prin insertie"<<endl;
t1=clock(); //citeste timpul final

cout<<"Sortarea a durat "<<(t1-t0)<<" batai de ceas intern --> aproximativ "<<(t1-
t0)/CLK_TCK<<" secunde !\n\n";
t_insertie=t1-t0;
}
//===============================================
void bubble_sort()
{
cout<<"Sortez vectorul...";
t0=clock(); //citeste timpul initial (referinta)
int aux,c=0;
for(i=1;i<n;i++)
```

```
{
for(int j=i+1;j<=n;j++)
if(k[i]>k[j]) { aux=k[i]; k[i]=k[j];k[j]=aux; }
if(c==n/60){ //progresul operatiei
c=0; //reseteaza contorul (s-a afisat o steluta)
cout<<"*";
}
c++; //incrementeaza contorul ce se reseteaza dupa afisarea unei stelute de progres
}
cout<<"OK\n";
cout<<"Vectorul a fost sortat cu metoda bulelor (bubble-sort)"<<endl;
t1=clock(); //citeste timpul final

cout<<"Sortarea a durat "<<(t1-t0)<<" batai de ceas intern --> apro*imativ "<<(t1-
t0)/CLK_TCK<<" secunde !\n\n";
t_bule=t1-t0;
}
//==================================================
void shell()
{
cout<<"Sortez vectorul...\n";
t0=clock(); //citeste timpul initial (referinta)
int gap=n/2,aux,modificari;
do
{
do
{
modificari=0;
for(i=1;i<n-gap;i++){
if(k[i]>k[i+gap])
{
aux=k[i];
k[i]=k[i+gap];
```

```
k[i+gap]=aux;
modificari=1;
}
}

} while (modificari);
} while(gap=gap/2);
cout<<"Vectorul a fost sortat cu metoda de sortare Shell"<<endl;
t1=clock(); //citeste timpul final

cout<<"Sortarea a durat "<<(t1-t0)<<" batai de ceas intern --> aproximativ "<<(t1-
t0)/CLK_TCK<<" secunde !\n\n";
t_shell=t1-t0;
}
//==================================================
void salvare_vector()
{
//operatie utila pentru testarea eficientei metodelor in aceleasi conditii
Se aplica fiecare metoda pentru aceeasi structura si aceeasi componentză a
vectorului k*/
cout<<"Salvez asezarea elementelor in vector...\n";
for(i=1;i<=n;i++)
k_salvat[i]=k[i];
}
//==================================================
void restaurare_vector()
{
/*Restaureaza asezarea elementelor in vector
Vectorul inainte de apelul acestei functii este sortat*/
cout<<"Restaurez asezarea elementelor in vector...\n";
for(i=1;i<=n;i++)
k[i]=k_salvat[i];
}
```

```
//==================================================
void meniu()
{
clrscr();//functia sterge ecranul

cout<<"Exemplu de testare a eficientei pentru trei dintre algoritmii de sortare
standard:"<<endl;
cout<<"- Bubble-sort\n- Sortare prin Insertie\n- Sortare Shell\n";
cout<<"Vectorul test va fi umplut cu numere intregi generate aleator\n";
cout<<"Pentru testarea optima a fiecarei sortari se recomanda alegerea unei
dimensiuni a vectorului de peste 3000 elemente\n";
cout<<"In cazul testului complet se va lua ca dimensiune a vectorului 10000 de
elemente\n\n";
//desenez meniul
cout<<"1.Test complet\n2.Bubble-sort\n3.Sortare prin insertie\n4.Sortare
Shell\n5.Iesire\n\nOptiunea dvs:";
cin>>opt; //citesc optiunea
clrscr(); //sterg ecranul
}
//===================================================
void test_complet()
{
cout<<"Generez vectorul cu 10000 nr intregi aleatoare...\n";
//numarul de elemente ale vectorului este setat la 10000
n=10000;
//se genereaza elementele vectorului k (nr. intregi aleatoare)
for(i=1;i<=n;i++)
{
k[i]=rand();
}
//se salveaza vectorul inainte de a fi sortat
//se aplica apoi fiecare algoritm de sortare
```

```
salvare_vector();
bubble_sort();
cout<<"Apasati orice tasta...";
getch();

//se restaureaza starea vectorului k initial (nesortata)
restaurare_vector();
insertie();
cout<<"Apasati orice tasta...";
getch();

restaurare_vector();
shell();
cout<<"Apasati orice tasta...\n\n";
getch();
cout<<"Timpi de executie algoritmi de sortare:\n\n";
cout<<"Algoritmul de sortare 'Shell' : aproximativ "<<t_shell/CLK_TCK<<"
secunde\n";
cout<<"Algoritmul de 'Sortare prin Insertie' : aproximativ
"<<t_insertie/CLK_TCK<<" secunde\n";
cout<<"Algoritmul 'Bubble-sort' : aproximativ "<<t_bule/CLK_TCK<<"
secunde\n\n";
}
//=====================================================
main()
{
do
{
meniu(); //desenare ecran principal + meniu
/*daca optiunea nu e 1(Test complet) sau 5(Iesire), ci o valoarea intre aceste doua
se citeste numarul de elemente si apoi se genereaza elementele vectorului*/
if(opt>1 && opt<5 ){
cout<<"Cƒte elemente are vectorul:";cin>>n;
```

```
cout<<"Generez vectorul cu nr intregi aleatoare...\n";
for(i=1;i<=n;i++)
{
k[i]=rand(); //nr intregi aleatoare
}
}

//in functie de valoarea introdusa ca optiune (opt)
switch(opt){
case 1:test_complet();break;
case 2:bubble_sort();break;
case 3:insertie();break;
case 4:shell();break;
case 5:exit(0);
default:"Atentie nu ati specificat o optiune valida!\n";break;
}

//afisare vector sortat
cout<<"Afisez primele 10 elemente din noua structura a vectorului..."<<endl;
for(i=1;i<=10;i++)
printf("k[%d]=%d\n",i,k[i]);

cout<<"Apasati orice tasta...";
getch();

}
while(opt!=5); //programul se termina la alegerea optiunii 5 (Iesire)
/*Iesirea propriu-zisa se face in switch() cu exit(0), conditia de iesire din acest
do while nefiind niciodata un motiv de terminare a programului*/
}
```

CONCLUZII

În domeniul calculatoarelor şi al matematicii, un algoritm de sortare este o metodă prin care se aranjează elementele unui tablou într-o ordine precisă. Cele mai folosite tipuri de date în cadrul acestor metode de sortare sunt tipul de ordin numeric şi tipul lexicografic. Eficienţa metodelor de sortare este importantă deoarece se pune un mare accent pe optimizarea altor algoritmi (cum sunt algoritmii de cautare) care necesită sortarea tablourilor cu care lucrează, devenind astfel mai eficienţi.

Cele mai multe limbaje de programare şi-au implementat propriul algoritm de sortare. Aceste limbaje foloseşc tipic numai un singur algoritm care este imbunataţit şi implementat în sintaxa specifică fiecărui limbaj de programare, şi se comportă eficient în lucrul cu memoria. Cel mai folosit este sortarea rapidă ("quick sort"), iar din punct de vedere al utlizării este urmat de algoritmul de sortare "heap sort" şi e algoritmul de sortare prin interclasare ("merge sort").

Iată câteva exemple de componente software care folosesc algoritmii de sortare:
Limbajul C include **qsort()**,care este o funcţie de librărie standard care realizează o sortare prin comparaţii utilizând un operator de comparare care este folosit ca pointer în cadrul acestei funcţii. Această implementare se comportă eficient, deşi nu este prea folosită, ea bazându-se pe principiul algoritmului de sortare rapidă.

Limbajul C++ foloseşte funcţia **qsort**()însă adusă ca o demostraţie pentru funcţia STL **std :: sort**, care sortează rangurile dintre două iteraţii cu acces aleatoriu. În plus, clasă **std :: list** (liste inlantuite), care nu utilizează accese alatorii este în metodă sort. tipul elementelor sortate trebuie să accepte operatorul de comparare, în caz contrar va fi necesar un operator de comparare adecvat. În plus este uşor să se folosească principuil comparării cu operator de comparare, el adesea performând cu metoda **qsort()**, funcţie prezentă în limbajul C. Limbajul C++ foloseşte şi metodele stable_sort şi **parţial_sort**

Limbajul Java - clasele utilizate în acest limbaj sunt numite **Arrays** şi **Collections** şi includ funcţia **sort**, care prezintă metode statice grupate în lucrul cu structuri de date (tablou,liste).

Utilitarul Microsoft . NET Framework - această componentă software oferă metoda statică **Array. sort()** care poate sorta un tablou de obiecte folosind un indiciu (reper) de comprare arbitrară fiind inspirată din algoritmii de sortare de tip rapid ("quick sort"). În plus se mai oferă şi metoda **sort()**, aflată în cadrul clasei **ArrayList** care şi ea foloseşte principul metodei de sortare rapidă ("quick sort").

Utilitarul Perl - oferă o funcţie sort, care este o funcţie-operator ce returnează valori de tip întreg. În versiunea Perl 5.6 se foloseşte o metodă de sortare rapidă ("quick sort"), însă în versiunile de după 5.6 se mai foloseşte şi metodă de sortare prin interclasare ("merge sort") ca procedeu de sortare secundar. Un prim motiv pentru care se abordează metode de sortare prin interclasare este securitatea aplicaţiilor, deoarece Perl este o unealtă software cu care se pot construi aplicaţii web.

Python - ofera două funcţii de sortare. Una, numită funcţia **sorted,** funcţionează dupa principiul metodelor de sortări albitrare, şi cea de-a doua funcţie funcţionează ca o metode de sortare "pe loc" pentru sortarea elementelor din tablouri.

Common Lisp - Common Lisp defineşte în secţiunea 17.3 două funcţii de sortare denumite **SORT** şi **STABLE-SORT**, însă sunt nefolosite deoarece încalcă principiile programarii structurate prin funcţii de salt. Poate permite în limbaj specific Lisp orice metode de sortare.

Deşi am prezentat doar câteva exemple, se poate afirma fără nici un dubiu că metodele de sortare sunt prezente, practic, în orice software.

Bibliografie

1. Logofătu, Hrinciuc, Doina – C++.Probleme rezolvate şi algoritmi, editura POLIROM, 2001;
2. Cormen, Th.H., Leiserson, C.E, Rivest, R.R. – Introducere în algoritmi, editura Computer Libris Agora, Cluj_Napoca, 2001;
3. Întuneric Ana, Schimim Cristina – Proiectarea Algoritmilor, editura POLIROM, 2004.
4. Livovischi, L.. Georgescu, H. – Sinteza şi analiza algoritmilor,Editura Ştiinţifică şi Encicolpedică, Bucureşti, 1986;
5. Miloşescu Mariana – Informatică, profil Real Cl XI-a, editura EDP, 2004.
6. Miloşescu Mariana – Informatică, profil Real Cl X-a, editura EDP, 2005.
7. Oprescu Daniela – Informatică pentru Cl X-a, editura NICULESCU, 2001.
8. Mitrana Victor – Provocarea Algoritmilor. Culegere de probleme pentru liceu, editura POLIROM, 2001.
9. Pătruţ, Bogdan – Aplicaţii C şi C++, Editura Teora, Bucureşti, 1998;
10. Popescu, Carmen – Culegere de Informatică, editura TIPOTRIB SIBIU, 2006
11. Rancea Daniel – Limbajul C++, editura LIBRIS, 1998.
12. Tudor Sorin – Tehnici de Programare. Manual pentru Cl. X-a, editura L&S, 1996.
13. Udrea Cristian, Ţancu Dan – Informatică, Teorie şi Aplicaţii Cl. X-a, editura ARVES, 2003.
14. Web Sit-ul: http://frankeman.sitesfree.com.
15. Web Sit-ul: http://www.cs.cmu.edu.

CUPRINS

INTRODUCERE **1**

CAP. I. NOŢIUNI GENERALE LEGATE DE ALOGORITMI **5**

1.2 SCHEME LOGICE 8
1.3. LIMBAJUL PSEUDOCOD 12

CAP.2. METODE DE ELABORARE A ALGORITMILOR **16**

2.1 ELABORAREA ALGORITMILOR 16
2.2 PROIECTAREA ASCENDENTĂ ŞI PROIECTAREA DESCENDENTĂ 16
2.3. PROIECTAREA MODULARĂ 19
2.4. PROGRAMAREA STRUCTURATĂ 21

CAP III. METODE DE SORTARE ŞI CĂUTARE **24**

3.1. ALGORITMI DE CĂUTARE 25
3.2. ALOGORITMI DE SORTARE 28
1.Algoritmul de sortare prin metoda bulelor *28*
2. Algoritmul de sortare prin metoda inserării *31*
A. Inserare directă 31
B. Sortarea prin inserţie cu diminuarea incrementului (ShellSort) 33
3. Algoritmul de sortare prin metoda selecţiei directe *34*
4. Algoritmul de sortare prin metoda numărării *36*
5. Sortarea prin metoda divide et impera *40*
A. Algoritmul pentru interclasarea a doi vectori sortaţi. 40
B. Sortare rapidă (Quicksort) 42

CAPITOLUL IV. APLICAŢII **45**

1. SORTARE PRIN METODA BULELOR (BUBBLESORT 45
2.SORTARE PRIN INSERARE DIRECTĂ 48
3. SORTARE PRIN MICŞORAREA INCREMENTULUI (SHELLSORT) 49
4. SORTARE PRIN SELECTIE DIRECTĂ 50
5. SORTARE PRIN INTERCLASARE 51
6. SORTARE RAPIDĂ (QUICKSORT 54
APLICAŢIE GENERALĂ PENTRU TOATE METODELE DE SORTARE PREZENTATE ÎN CAPITOLUL III. 55

CAPITOLUL V. ANALIZA TIMPULUI DE EXECUŢIE **62**

STUDIU DE CAZ: ILUSTRAREA EFICIENŢEI ALGORITMILOR DE SORTARE IMPLEMENTAŢI ÎN LIMBAJUL C++ 66

CONCLUZII **74**

BIBLIOGRAFIE **76**