Universitatea din Bacău
Facultatea de Ştiinţe
Catedra de Matematică şi Informatică
Specializarea Informatică
Forma de învăţământ cu frecvenţă redusă
Anul I, semestru 1
Disciplina PROGRAMARE PROCEDURALĂ
Titular de curs BOGDAN PĂTRUŢ

BOGDAN PĂTRUŢ

PROGRAMARE PROCEDURALĂ

Curs pentru Informaică

IFR

Bacău, 2008

Cuvânt înainte

Cursul îşi propune să familiarizeze studenții cu principalele noțiuni despre programare procedurală. După ce se definesc conceptele fundamentale despre date, algoritmi, se face o introducere în conceptele de bază ale programării structurate. Următoarele capitole se referă la subprograme, apoi se trec în revistă tehnicile de programare (greedy, backtracking, divide et imper), noțiuni despre complexitatea algoritmilor şi lucrul cu structurile de date dinamice.

Cursul se adresează studenților din anul I, profilul Informatică şi este urmat de aceştia în semestrul I al anului I de facultate, având o prelegere de două ore pe săptămână. Este însoțit de lucrări de laborator, ce se vor desfăşura în şedințe săptămânale de câte două ore.

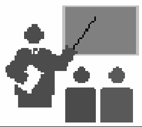
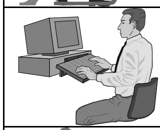
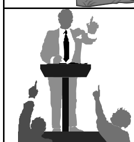
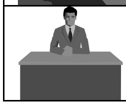
Cursul poate fi studiat şi de alți studenți de la secțiile economice, inginerie etc., precum şi de orice persoană dornică să pătrundă în tainele programării procedurale.

Parcurgerea cursului presupune o bună cunoaştere a modului de operare cu un calculator electronic şi, de asemenea, noțiuni de limba engleză.

Deoarece aceasta este o carte despre tehnica programării, limbajul de programare folosit este mai puțin important.

În cadrul textului pot fi observate diferite pictograme care pot ghida cititorul (profesor sau elev) în procesul e predare/învățare.

Astfel, următoarele simboluri semnifică tipul de lecție sau metodă didactică indicată a fi folosită cu precădere în cadrul lecției, pe care le recomandăm profesorului:

	*Metode comunicative*
	*Metode conversative*
	*Lectura independentă a elevilor*
	*Lecție de laborator*
	*Lecție de verificare şi notare a elevilor*
	*Observații şi comentarii în legătură cu obiectivele specifice ale unui capitol sau lecție*

Alte simboluri sunt folosite pentru a marca diferitele activități ale studenților:

☞	*Observație menită să lămurească chestiuni de finețe sau deosebite în cadrul textului*
🔔	*Atenție! Averizarea cititorului asupra unor chestiuni importante; altfel, cititorul grăbit riscă să le trateze superficial.*
?	*Întrebări și exerciții de autoverificare, cu patru grade de dificultate, după cum urmează:* ☺ (nivel scăzut) 😐 (nivel mediu) ☹ (nivel ridicat) 💣 (nivel foarte ridicat)
🗎	*Rezumatul capitolului, cu scopul de a sistematiza materia predată şi de a sedimenta cunoştințele acumulate.*

Ne exprimăm dorința că aceste simboluri, alături de figurile şi explicațiile care însoțesc fiecare problemă studiată vor face din această carte un manual util elevilor, dar şi acelor specialişti care vor să pătrundă în tainele tehnicilor de programare, acelora care nu vor să se rezume doar la cunoaşterea unui limbaj de programare.

Simbolurile menționate nu indică obligații din partea profesorului sau a elevilor săi, ci sunt doar recomandări pe care le facem cititorilor.

Încheiem cu speranța că această lucrare va fi de folos tuturor cititorilor săi.

Autorul

Referenți ştiințifici:

- *conf. univ. dr. Mihai Talmaciu, Universitatea din Bacău, decanul Facultății de Ştiințe*
- *conf. univ. dr. Elena Nechita, Universitatea din Bacău, Facultatea de Ştiințe, şeful Catedrei de Matematică şi Informatică*

Cuprins

Tematica seminariilor şi a lucrărilor practice

Acestea se vor desfăşura în laboratorul de informatică, despre care se presupune că poate asigura lucrul a maxim doi studenţi la un calculator performant. Orele de practică trebuie să le succeadă pe cele de teorie, de la curs. Pe toată durata cursului, este recomandabil ca studenţii să lucreze suplimentar în laboratorul de informatică, mai ales când vor trebui să-şi realizeze aplicaţia proprie (discutată în şedinţele de laborator 9-13).

Şedinţa 1a. Recapitulare - şedinţă de tip seminar
Studenţii vor fi testaţi asupra modului în care ştiu să utilizeze un calculator, ca operatori ai sistemului de operare şi al unor produse de birotică (procesor de texte, de tabele, editor grafic). Laboratorul poate să constea în elaborarea unei lucrări complexe, care să îmbine lucrul cu diferite pachete de programe.

Şedinţa 1b. Date şi modalităţi de reprezentare a datelor - şedinţă de tip seminar
În discuţie se va aborda tema noţiunii de dată. Se vor clasifica datele, se vor da exemple de modalităţi de reprezentare a datelor, modelând situaţii din lumea reală.

Şedinţa 2a. Algoritmi - şedinţă de tip seminar
Se va discuta cu studenţii, sub forma unor studii de caz, care sunt etapele rezolvării unei probleme. Se vor formula diferite probleme şi se va studia care din ele pot fi rezolvate prin algoritmi şi care nu. Se vor enunţa caracteristicile algoritmilor şi se va discuta pe marginea acestei teme.

Şedinţa 2b. Elementele programării structurate - şedinţă de tip seminar+laborator
Se vor trece în revistă structurile de control folosite în programarea procedurală, exemplificându-se prin scrierea unor algoritmi de calcule matematice şi financiar-contabile, precum şi a unor algoritmi de căutare şi sortare. Algoritmii vor fi implementaţui sub forma unor simple programe în limbajul utilizat de mediul de programare ce va fi predat ulterior (Pascal, C, Visual Basic, Delphi, Visual FoxPro etc.). În final, vor fi analizaţi algoritmii din punct de vedere al complexităţii lor.

Şedinţele 3 şi 4. Metoda backtracking - şedinţă de tip laborator
Vor fi tratate diferite aspecte ale metodei backtracking şi se vor rezolva probleme

Şedinţele 5 şi 6. Recursivitate - şedinţă de tip laborator
Se vor realiza programe simple care să folosească recursivitatea. Programele vor fi scrise în limbajul de programare ales. Se va folosi metoda divide et impera sau backtracking recursiv

Şedinţele 7 şi 8. Metoda greedy - şedinţă de tip laborator
Vor fi tratate diferite aspecte ale metodei greedy şi se vor rezolva probleme

Şedinţele 9 şi 10. Structuri dinamice de date - şedinţă de tip laborator
Vor fi implementate structurile de listă simplu înlănţuită, coadă, stivă, listă dublu înlănţuită, arbore binar, arbore oarecare şi se vor rezolva diferite probleme cu aceste structuri.

Şedinţele 11, 12 şi 13. Lucrare practică - şedinţe de tip laborator
Pe parcusul acestor şedinţe, studenţii vor lucra în echipe de 2-4 persoane pentru realizarea concretă a unei aplicaţii concrete, sub îndrumarea şi coordonarea cadrului didactic. Lucrul studenţilor nu se

va rezuma doar la cele două ore săpămânale, cuprinse în planul de învățământ, ci şi în studiul individual, acasă sau în laboratorul de informatică. La orele de laborator, studenții vor prezenta cadrului didactic, săptămânal, stadiul la care au ajuns cu lucrarea, ce probleme au întâmpinat, iar cadrul didactic îi va ajuta să le soluționeze şi le va sugera îmbunătățiri ce pot fi aduse lucrării. La final, lucrarea practică realizată de studenți trebuie să fie însoțită şi de un document scris, în care se prezintă scopul, modul de realizare şi de utilizare a aplicației.

Şedința 14. Prezentarea lucrării practice - şedință de tip colocviu
În această ultimă lucrare, studenții vor prezenta aplicația realizată comisiei de notare. Comisia va fi formată din cadrul didactic de la seminar, titularul de curs şi doi reprezentanți numiți de studenți şi evidențiați prin aportul lor deosebit la desfăşurarea orelor de laborator.

Mod de notare

Cursul de *Programarea calculatoarelor electronice* se termină cu examen, în care studenții vor primi o notă, calculată ca o medie ponderată după cum urmează:

Forma de verificare (**E**xamen, **C**olocviu)		E	
Modalitatea de susținere (**S**cris şi **O**ral, **O**ral)		SO	Puncte sau procentaj
NOTARE	Răspunsuri la examene, colocviu		30%
	Evaluare activități aplicative (laborator, proiect) – proiect		40%
	Prezență activă la curs şi seminar		10%
	Teme de casă sau studiu individual		20%
	TOTAL PUNCTE SAU PROCENTE		**100%**

Capitolul 1. Introducere

Diferiți autori dau diferite definiții informaticii, dar, în esență, tote aceste definiții fac apel la originea cuvântului. Cuvântul informatică provine din franțuzescul *informatique*, care, la rându-i provine din *information* = informație şi *automatique* = în mod automat, automatică.

Informatica este un complex de discipline ştiințifice care se ocupă de prelucrarea şi transmiterea electronică a informației.

Fireşte, o asemenea definiție presupune ca noi să ştim ce înseamnă atât informația, cât şi ce se înțelege prin prelucrare şi transmitere electronică.

Conceptul de informație este strâns legat de cel de dată.

Datele sunt numere, caractere, imagini sau orice alte modalități de reprezentare (înregistrare) a unor entități reale într-o formă ce poate fi accesată de om sau, în mod special, introdusă într-un calculator, stocată şi procesată acolo sau transmisă pe cale electronică.

O dată nu are ea însăşi un înțeles, decât când este interpretată de un anumit sistem de prelucrare a datelor, care îi dă un înțeles şi atunci data devine **informație**.

Datele pot fi reprezentate pe baza unor şabloane şi putem obține informații, prin interpretarea lor. Informațiile sunt utilizate pentru a ne spori **cunoştințele**.

Exemple:

- 1234567.89 este dată.
- "Contul meu bancar a crescut de 80 de ori, ajungând la 1234567.89 mii lei" este informație.
- "Nimeni nu are atâția bani ca mine." este cunoştință.

Pentru a înțelege mai bine cum stau lucrurile, să considerăm un caz concret din realitate. La o facultate se dă concurs de admitere. Numele candidaților şi notele obținute de ei sunt date. Se realizează o listă a candidaților împreună cu notele acestora.

Lista poate fi considerată o informație, pentru că fiecare element al listei, constituit dintr-un nume şi un număr, are o anumită semnificație.

Lista este ordonată în ordinea descrescătoare a notelor. Acesta este un proces de prelucrare a informațiilor sau a datelor. Se obțin alte informații sau date, care compun lista ordonată. Dacă ordonarea a fost realizată cu ajutorul unui calculator electronic, înseamnă că a avut loc o prelucrare automată a datelor, prin mijloace electronice.

Ulterior, lista ordonată va putea fi transmisă ministerului, fie prin poştă, fie pe căi electronice (de pildă prin poştă electronică).

Unitatea de măsură a datelor/informației este **bitul**, o cifră (engl. *digit*) care poate fi 0 sau 1. S-a constatat că sistemul de numerație binar (care foloseşte doar cele două cifre) poate fi utilizat pentru a reprezenta orice informație. Reprezentarea binară a unei informații se numește și *digitizare*. De fapt, putem observa cu ușurință că toată natura are o organizare binară.

Multiplul bitului este: **octetul** (sau **byte**-ul), care este o grupare de 8 biți. Multiplii octetului sunt:

kilooctetul sau **kilobyte**-ul (notat Kb sau Ko) 1 Kb = 1024 bytes (octeți)
megaoctetul sau **megabyte**-ul (notat Mb sau Mo) 1Mb = 1024 Ko
gigaoctetul sau **gigabyte**-ul (notat Gb sau Go) 1 Gb = 1024 Mb ($1024 = 2^{10}$)

Spuneam că informatica este un complex de discipline științifice ce rezolvă astfel de probleme. Astfel, distingem, în cadrul informaticii următoarele discipline mai importante, care constituie tot atâtea direcții de studiu și cercetare:

- **Arhitectura calculatoarelor** - se ocupă de componentele fizice (hardware) ale unui calculator, de modul lor de funcționare, precum și de legăturile existente între ele;

- **Sisteme de operare** - se ocupă de componenta software de bază a unui calculator (sistemul de operare), de modul de proiectare a acestuia, de felul în care pot fi getionate mai eficient resursele fizice ale calculatorului (memoria, procesorul, discurile);

- **Rețele de calculatoare** - reprezintă o disciplină strâns legată de cele două anterioare, ocupându-se de modul de realizare și configurare software și hardware a unei rețele de calculatoare, de integrarea rețelelor de calculatoare între ele, de tipurile de rețele de calculatoare

- **Structuri de date** - este o disciplină strâns legată de programarea calculatoarelor, ocupându-se de găsirea și îmbunătățirea modalităților de reprezentare a datelor în calculator, în funcție de necesități și de algoritmii ce le vor prelucra

- **Baze de date** - este o disciplină înrudită cu precedenta, ocupându-se, însă, de colecțiile de date mari, cu structuri asemănătoare, ce pot fi stocate pe suporturi fizice externe, precum și de modul de interogare a unor baze de date pentru a realiza selecții

- **Analiza, proiectarea și sinteza algoritmilor** - după cum spune și denumirea, această disciplină se ocupă de modalitățile, tehnicile și strategiile cele mai eficiente de rezolvare a problemelor, de stocare a datelor, de prelucrare a informațiilor în vederea obținerii altor informații, precum și de transmitere eficientă a lor pe căi electronice

- **Programarea calculatoarelor** - se ocupă de implementarea algoritmilor proiectați pe calculatoarele electronice, astfel încât aceștia să poată fi puși la lucru, deci este arta și știința de a crea programe de calculator

- **Limbaje de programare** - un algoritm proiectat va fi implementat sub forma unui program, iar programul este scris într-un anumit limbaj de programare, adică o convenție de simboluri și cuvinte, precum și reguli de îmbinare a acestora, împreună cu înțelesurile lor ce pot fi recunoscute de calculator; disciplina se ocupă și de evoluția și studiul limbajelor de programare, pentru a determina cel mai potrivit limbaj de programare utilizabil într-o anumită situație

- **Limbaje formale și construcția compilatoarelor** - o disciplină care studiază (sub o formă algebrică) modalitățile prin care un program scris într-un limbaj de programare poate fi recunoscut ca fiind corect de către un automat sau o gramatică de descriere a limbajului de programare respectiv; de asemenea, disciplina se ocupă și cu construirea compilatoarelor, adică a acelor translatoare între limbajul de programare și limbajul procesorului (cod mașină)

- **Ingineria software** - este un domeniu strâns legat de programarea calculatoarelor, de data aceasta la un nivel mai ridicat, al proiectării programelor complexe

- **Grafică computaţională** - se ocupă de tehnicile de reprezentare grafică pe calculator a curbelor, suprafeţelor, corpurilor, de modalităţile de ascundere a suprafeţelor nevizibile, domeniu strâns legat de geometrie

- **Inteligenţă artificială** - se ocupă cu rezolvarea unor probleme pentru care se folosesc algoritmi specifici; domeniul doreşte să simuleze pe calculator unele componente ale inteligenţei umane, cum ar fi recunoaşterea textelor scrise, a vorbirii, deducţia, găsirea răspunsurilor creative, capacitatea de a învăţa din experienţă şi capacitatea de a trage concluzii pe baza unor informaţii incomplete.

- **Cercetere operaţională** - se ocupă, în general, de rezolvarea unor probleme de decizie şi conducere ce apar în economie

- **Analiză numerică** - este un domeniu de graniţă între informatică şi matematică, ocupându-se de rezolvarea pe cale numerică a unor probleme de analiză matematică (derivabile, integrale, interpolări, rezolvarea de ecuaţii, calcule cu matrice şi determinanţi)

- **Teoria grafurilor** - este un domeniu de graniţă între informatică şi matematică, care se ocupă de rezolvarea problemelor legate de grafuri, cu aplicabilitate în diferite domenii ale ştiinţei şi tehnicii (management, proiectare în construcţii etc.).

Obiectul de studiu al disciplinei *Programarea calculatoarelor electronice* va fi, aşadar reprezentat de arta şi ştiinţa creării de programe, pe baza unor algoritmi, scrise într-un limbaj de programare. De aceea, acest curs va trece în revistă, mai întâi câteva modalităţi esenţiale de reprezentare a datelor, apoi vom vorbi despre algoritmi şi proprietăţile lor.

Cursul va continua cu prezentarea principalelor stiluri (paradigme) de programare folosite în prezent, pe care le vom prezenta într-o succesiune gradată, până vom ajunge la programarea vizuală.

Conceptele programării vizuale şi o scurtă introducere practică în domeniu va constitui partea a doua a cursului, ce se va încheia cu un capitol referitor la proiectarea, realizarea şi întreţinerea produselor softare şi o recapitulare pentru examen.

X

delta

3

x-3*delta

Capitolul 2. Date

2.1. Constante şi variabile. Expresii

În primul capitol am precizat ce sunt datele, iar în acest capitol ne vom ocupa de reprezentarea lor internă, adică în memoria calculatorului şi pe suporturi externe, fizice, în fişierele de pe discuri.

Datele apar în cadrul unor programe scrise într-un limbaj de programare sau altul, reprezentate prin nişte cuvinte de identificare, numite **identificatori**. În mai toate limbajele de programare, un identificator este un şir de litere sau cifre, eventual şi alte simboluri (cum ar fi "_"), ce începe cu o literă.

În cadrul programului, datele pot fi declarate ca fiind constante sau variabile. O **constantă** este o dată a cărei valoare *nu se poate modifica* pe parcursul execuției programului, deci rămâne constantă. O **variabilă** este o dată a cărei valoare *se poate modifica* pe parcursul execuției programului, deci ea poate varia, dar acest lucru nu este obligatoriu. Astfel, se poate declara o dată ca fiind variabilă în cadrul unui program, apoi ea să primească o anumită valoare, iar această valoare să rămână asociată respectivei variabile până la terminarea programului.

Evident, atunci când se va declara o dată constantă, se va preciza şi valoarea ei, iar când se va declara o dată variabilă, se subînțelege că ulterior, pentru a putea fi folosită, această variabilă va primi o anumită valoare. Majoritatea limbajelor de programare asignează o valoare inițială variabilelor, o dată cu declararea lor. Astfel, şirurile de caractere sunt inițializate la şirul vid, iar numerele sunt considerate cu valoarea zero.

Fireşte, atât constantele cât şi variabilele au o anumită structură, mai simplă sau mai complicată, şi o anumită natură, dată de mulțimea valorilor posibile pentru o dată. Cu ele se pot face anumite operații, în funcție de natura şi structura lor. Astfel, vom spune că o dată are un anumit tip.

Prin **tip de date** vom înțelege o mulțime de valori, împreună cu operațiile ce se pot executa cu ele. Fiecărei variabile, la declarare, i se va asocia un anumit tip. Tipul unei constante poate fi determinat implicit din valoarea constantei, sau poate fi precizat explicit ca în cazul variabilelor.

Astfel, dacă constanta K are valoarea numerică 7, putem trage concluzia că ea este de tip întreg, sau de tip real, nu şi logic sau şir de caractere. Totuşi, există şi limbaje în care se fac anumite convenții, de pildă că orice număr diferit de zero este considerat ca fiind cu valoarea de adevăr *adevărat*, iar numărul zero are valoarea de adevăr *fals*.

Unele limbaje de programare permit declararea unor variabile fără a se preciza tipul lor, considerându-se astfel ca având un anumit tip general. Astfel, atunci când va fi folosită, variabila respectivă va fi considerată ca având cel mai adecvat tip cu putință, în situația concretă respectivă. De pildă, dacă este declarată o variabilă X, iar la un moment dat i se atribuie valoarea 3,. atunci ea poate fi considerată ca având un tip numeric. Dacă ulterior, variabila X va primi valoarea "abc", adică un şir de caractere, se poate considera că X este de tip şir de caractere.

Pe baza constantelor şi variabilelor se formează **expresii**. Bineînțeles, în formarea expresiilor se vor folosi acei operatori, precum şi acele funcții, permise de tipurile valorilor asupra cărora se operează. Expresiile mici pot conduce la elaborarea de expresii mai mari, din ce în ce mai complexe.

Pentru a înțelege cum stau lucrurile, vom considera limbajul Visual Basic, iar exemplele ce vor urma vor fi date în acest limbaj.

Să considerăm următoarele declarații de variabile:

Dim X As Integer, Y As Integer, S As String
Dim V

Astfel, X şi Y sunt variabile întregi (cu valori în mulțimea numerelor întregi), S este variabilă de tip şir de caractere (cuprinse între ghilimele), iar V este o variabilă a cărui tip nu a fost precizat. Visual Basic pune astfel la dispoziție tipul de date Variant, care reuneşte, sub un cadru general, toate celelalte tipuri de date.

Următoarele expresii sunt corecte din punct de vedere sintactic, în limbajul Visual Basic:

2, X, X+5, X+Y, X+4*Sqr(Y), S, V, V+3, V+X, S+S, X+Len(S), Left(S,2)+Right(S,3)

X, Y, V şi S pot primi valori în două feluri: prin atribuire directă sau prin citire (de la tastatură sau dintr-un fişier).

Atribuirea se face cu instrucțiunea de atribuire, care are forma:

Variabilă = Expresie   sau Let Variabilă = Expresie

Citirea valorilor se poate face folosind o operație de citire, ca de pildă:

Input Variabilă

Să considerăm următoarele operații prin care se atribuie valori variabilelor declarate anterior:

```
Y = 16
S = "abcd"
Y = 7
Input V
X = Y + Len(S) + 1
```

Inițial lui Y i se atribuie valoarea 16, dar apoi el primeşte valoarea 7, renunțându-se la 16. Dacă de la tastatură se va da valoarea 8 lui V, atunci V va fi considerat ca fiind de tip întreg. X va lua valoarea 12, adică suma 7 + 4 + 1, 4 fiind lungimea şirului de caractere S ("abcd"). În cadrul secvenței anterioare apar 4 constante şi anume: numărul 16, şirul "abcd", numerele 7 şi 1.

Următoarele expresii nu sunt corecte din punct de vedere sintactic, deci, de fapt, ele nu sunt expresii:

X +, X****, X Y, S +

Există şi cazuri speciale, cum ar fi X + S sau S + V. Unele limbaje de programare consideră asemenea entităţi ca fiind incorecte, pe când altele le consideră expresii perfect corecte. De pildă, în primul caz, X considerat ca fiind număr, iar S ca fiind şir de caractere, rezultatul acelei adunări poate fi considerat fie concatenarea dintre X convertit la şir de caractere şi S, fie ca numărul obținut din adunarea lui X cu şirul S, convertit la număr. În general, se alege în funcție de tipul ce se doreşte a-l avea expresia.

De pildă, dacă X are valoarea 5, S are valoarea "123", atunci printr-o atribuire de genul S = X + S vom putea înțelege că S va primi valoarea "5123" (adică "5" concatenat cu "123"), iar printr-o atribuire de genul X = X + S, vom considera că X primeşte valoarea 128 (adică suma 5 + 123).

Majoritatea limbajelor de programare definesc expresiile după un sistem de reguli sintactice, care, în general sunt următoarele:

1. orice constantă este expresie;
2. orice variabilă este expresie;
3. dacă E este expresie, atunci şi (E), -E, +E, F(E) sunt expresii, unde F este numele unei funcții aplicabile expresiei E;
4. dacă E1 şi E2 sunt expresii, atunci şi E1+E2, E1-E2, E1*E2, E1/E2 sunt expresii.

Acum, pe baza regulilor de mai sus putem construi expresii foarte complexe, pornind de la constante şi variabile. Astfel, să considerăm entitatea (3+A)*(5/(-B+C)) şi să verificăm dacă ea este expresie sau nu. Să presupunem că A, B şi C sunt variabile numerice întregi.

Cum 3 este constantă, conform regulii 1, ea este şi expresie. A, fiind variabilă este, conform regulii 2 expresie. Acum, conform regulii 4, 3+A este expresie, iar (3+A) este tot expresie, conform regulii 3. După simbolul înmulțirii (reprezentat adesea prin *), avem: 5 este expresie, fiind constantă, B, C, apoi -B şi -B+C sunt expresii. În fine, conform regulii 3, (-B+C) este tot expresie, apoi şi (5/(-B+C)) este expresie, în conformitate cu regula 4, şi, tot după această regulă, şi (3+A)*(5/(-B+C)) este expresie.

2.2. Tipuri de date simple

Spuneam că fiecare constantă, variabilă sau expresie are un anumit tip de date. Tipul unei date determină comportamentul acesteia, pentru că el limitează sau extinde modul de operare asupra sa. În general, se acceptă că datele pot fi considerate ca fiind *simple, primare*, adică având o structură atomică, indivizibilă, sau *structurate*, construite pe baza altor date, cu ajutorul unor constructori speciali.

În general, sunt acceptate ca fiind atomice sau simple, următoarele tipuri de date: mulțimea numerelor întregi şi operațiile cu numere întregi, care dau rezultat întreg; mulțimea numerelor reale împreună cu operațiile ce se pot executa cu ele; mulțimea caracterelor reprezentabile în calculator şi operațiile cu ele; mulțimea valorilor de adevăr, *adevărat* şi *fals*, ce constituie tipul logic de date. Personal, consider şi tipul şir de caractere ca fiind un tip simplu, datorită faptului că este foarte necesar în elaborarea unor algoritmi simpli, de bază, şi pentru că asupra lui se poate acționa cu operații directe, întâlnite şi la celelalte tipuri de date. Totuşi, tipul şir de caractere este un tip structurar.

Tipurile de date poartă şi ele nume, deci sunt denumite prin identificatori. De obicei, tipul întreg se numeşte Integer, dar pot exista mai multe tipuri întregi, în funcție de necesități particulare. Astfel, în Visual Basic, Integer reprezintă numerele întregi din intervalul -32768 .. 32767, iar Byte reprezintă numerele întregi între 0 şi 255. De asemenea, Long este tot un tip întreg, cu valori între

-2147483648 şi 2147483647. În funcţie de mărimea numerelor de reprezentat, o valoare întreagă poate fi stocată folosind 1, 2 sau 4 octeţi. Astfel, de pildă, o dată de tip Byte se memorează pe un octet, una de tip Integer pe doi octeţi, iar una de tip Long pe 4.

În mod similar, Single stochează valori reale pe 4 octeţi, iar Double valori reale pe 8 octeţi. uuuuuu

Pentru şiruri de caractere se va folosi tipul String, iar caracterele individuale sunt considerate ca fiind şiruri de caractere de lungime 1. În Pascal, de pildă, există însă tipul de date Char pentru reprezentarea caracterelor.

Boolean este tipul de date logic, cuprinzând valorile constante True şi False. În Visual Basic există şi tipul general Variant, despre care am mai vorbit, precum şi alte tipuri speciale, cum ar fi Decimal, Currency sau Date.

În general, cu datele numerice pot fi realizate operaţiile specifice numerelor, cum ar fi adunarea, scăderea, înmulţirea (reprezentată de *), împărţirea (reprezentată de \ la numere întregi, în Visual Basic, de pildă, sau / la numere reale). Există şi unele funcţii matematice cum ar fi funcţiile trigonometrice (Sin, Cos, Tan, Atan), funcţia logaritm zecimal (Log), funcţia radical (Sqr) sau funcţia modul (Abs).

Cu datele de tip logic (Boolean) se realizează operaţii specifice, cum ar fi conjuncţia, disjuncţia (inclusivă şi exclusivă), negaţia, a căror tabele sunt prezentate mai jos (A = adevărat, F = fals):

and (şi) - conjuncţia logică
A and A = A
A and F = F and A = F and F = F

or (sau) - disjuncţia logică inclusivă
F or F = F
A or F = F or A = A or A = A

xor (sau exclusiv) - disjuncţia logică exclusivă
A and F = F and A = A
A and A = F and F = F

not (non, nu) - negaţia logică
not A = F
not F = A

În Visual Basic, cu datele de tip Boolean se pot realiza şi operaţiile de implicaţie şi echivalenţă logică, prin operatorii Imp, respectiv Eqv.

Conversiile de la un tip la altul se pot realiza fie explicit, folosind funcţii speciale, fie implicit, după caz. Astfel, în Visual Basic, dacă X, Z sunt de tip Integer, iar Y este de tip Byte, dacă scriem Z=X+Y, atunci automat Y este convertit la tipul Integer, înainte ca să se realizeze adunarea.

Conversii mai importante se fac în cazuri ca acestea:

```
Dim X as Integer
Dim Y as Double
X = Y
sau
Dim X as Integer
Dim S as String
```

```
S = X sau X = S
```

În primul caz, evident că X va primi partea întreagă a lui Y, iar în celelalte două cazuri, fie X se converteşte la şir, fie se preia din S numărul.

Totuşi, putem folosi şi funcțiile Int pentru a extrage partea întreagă dintr-un număr real, respectiv Str pentru a converti un număr la un şir şi Val pentru a obține un număr dintr-un şir.

Cu datele de oricare tip de date se pot realiza comparații. Astfel, de obicei se folosesc notațiile următoare:
<, > pentru mai mic şi mai mare;
<=, >= pentru mai mic sau egal, respectiv mai mare sau egal;
<> sau != pentru diferit
= sau == pentru egal

2.3. Tipuri de date structurate

Pe baza datelor simple se pot construi date structurate. De pildă, putem să ne imaginăm situația în care dorim să stocăm lista numelor şi notelor candidaților la concursul de admitere în facultate. O modalitatea foarte ineficientă este de a păstra câte o variabilă pentru numele fiecărei persoane şi câte o variabilă pentru nota obținută de fiecare persoană. Dar nu vom şti câte persoane vom avea, de aceea este practic imposibil să procedăm astfel.

Mult mai bine este să folosim un tip de date care să ne permită declararea a două variabile, să zicem Nume şi Nota, care să păstreze cele două liste. Acest tip de date se numeşte **tablou** şi permite gruparea de date de acelaşi tip sub un singur nume. Componentele vor putea fi referite printr-un număr de ordin, numit indice. Pentru ca operarea să fie eficientă, va trebui ca nota persoanei cu indicele i din tabloul de nume să fie pe poziția i în tabloul de note.

Mult mai natural ar fi să avem un singur tablou, în loc de două. Astfel, în loc de a păstra un tablou pentru numele persoanelor şi unul pentru notele lor, mai bine am avea un singur tablou de persoane. Astfel, ar trebui ca fiecare componentă a tabloului mare să conțină un articol, care să grupeze la un loc atât numele, cât şi nota unei persoane. În acest sens ne vine în sprijin tocmai tipul de date înregistrare (sau articol) ce poate încapsula sub un singur nume date de tipuri diferite.

Cu toate că un tablou de articole poate stoca eficient şi natural datele despre candidații la un concurs de admitere, aceasta nu este întotdeauna soluția cea mai bună. Datele dintr-un tablou se stochează în memoria internă a calculatorului şi, cu toate că pot fi accesate rapid şi direct, ele nu sunt păstrează de la o rulare a programului la alta sau când calculatorul este oprit. Iată şi motivul pentru care ar fi necesar ca datele să fie păstrate pe un suport fizic extern, sub forma unui fişier pe disc. Un alt motiv pentru care trebuie utilizat fişierul în locul tabloului este că un fişier poate avea dimensiuni mult mai mari decât un tablou.

Structurile de date complexe se creează pe baza celor simple sau complexe create anterior, aplicând nişte constructori speciali.

În Visual Basic, un tip **înregistrare** se defineşte astfel:

```
Type NumeDeTip
      câmp1 As Tip1
      câmp2 As Tip2
      ......................
End Type
```

Pot exista în cadrul definiției unui tip de date înregistrare şi câmpuri de tip tablou.

Exemplu:

```
Type Candidat
      nume As String
      nota As Single
End Type
```

În continuare, putem declara o variabilă de tip Candidat prin

```
Dim C as Candidat
```

Pentru a face referirea la un anumit câmp se foloseşte notația cu punct, sau operatorul punct, astfel: NumeDeVariabilăÎnregistrare.NumeDeCâmp: C.nume este numele candidatului C, iar C.nota este nota aceluiaşi candidat. Nu se poate citi ansamblul C în întregime, prin Input C, ci doar pe componente, prin Input C.nume şi Input C.nota.

Nu există o definiție specială pentru un tip tablou, dar faptul că X este o variabilă de tip tablou se poate scrie astfel:

```
Dim X(n) as Tip
```

Astfel, X s-a declarat ca fiind o variabilă de tip tablou, cu n componente, numerotate de la 0 la n-1. Fiecare componentă este de tipul Tip. Dacă se doreşte ca numerotarea să se facă de la o anumită valoare v1 la o altă valoare v2 se va scrie:

```
Dim X(v1 To v2) as Tip
```

Exemplu:

Astfel, de pildă, Dim X(1 to 10) as Integer declară un tablou cu 10 numere întregi.

Referirea elementelor unui tablou se face cu ajutorul operatorului ( ) (la alte limbaje se folosesc parantezele pătrate [ ] în loc de cele rotunde). Astfel, X(1) este prima componentă a tabloului X, iar X(10) este ultima.

Exemplu:

Să considerăm următoarea declarație:

```
Dim Cand(1 To 500) as Candidat
```

Astfel, am declarat 500 de candidați, iar Cand(i).nume este numele candidatului al i-lea, pe când Cand(i).nota este nota acestuia.

Până acum am vorbit despre tablouri unidimensionale, numite şi **vectori**. Există însă şi posibilitatea de a declara tablouri bidimensionale (numite **matrice**) sau chiar cu mai multe dimensiuni.

Astfel, declarația:

```
Dim A (1 To 10, 1 To 15)
```

defineşte o matrice cu numele A, cu 10 linii şi 15 coloane. Elementul de la intersecția liniei i cu coloana j se va referi prin A(i,j).

De ce AND si nu OR?

Tipul Boolean. Operatorii logici. Legile lui De Morgan. Logica matematica

- Adevarat sau fals. A treia varianta nu exista!

In urma cu multi, multi ani, oamenii si-au pus problema "sa se joace cu adevarul", adica sa faca rationamente, pornind de la propozitii simple. Aristotel a inventat logica ce-i poarta numele, iar mai apoi Boole, un matematician englez, a formalizat lucrurile obtinand ceea ce ne intereseaza pe noi in programare si anume logica booleana. In aceasta logica avem de a face cu doua valori de adevar: fals si adevarat. A treia varianta nu exista.

La un monent dat, ceva (o afirmatie, o propozitie) poate fi fie adevarat, fie fals, dar niciodata amandoua. Unii spun despre o anumita afirmatie ca este "in general adevarata". In programare, in logica booleana, asa ceva nu exista. O expresie booleana nu poate fi in general adevarata. Cand spui "in general adevarat" inseamna, de fapt, fals. Pentru ca ce nu este "intotdeauna" adevarat este fals. De exemplu, afirmatia (propozitia) logica "programatorii sunt buni la matematica" nu poate fi in general adevarata, atata timp cat o privim din punct de vedere logic, boolean. Daca nu exista nici un programator care sa nu fie bun la matematica, atunci propozitia noastra este adevarata. Daca insa exista cel putin un programator care sa nu fir bun la matematica, propozita noastra este falsa. Cum este propozitia noastra, din punct de vedere logic? Falsa, pentru ca am cunoscut eu un programator care nu se pricepea deloc la matematica! Din punctul de vedere al vietii cotidiene, dar nu din punct de vedere logic, despre afirmatia "programatorii sunt buni la matematica" se poate spune ca este in general adevarata.

Sa luam acum un exemplu mai din programare. Acolo operam cu variabile, constante si expresii. Constantele nu-si modifica valorile, deci vor avea una (si numai una) din cele doua valori, care, in multe limbaje de programare sunt notate cu True si respectiv False. In alte limbaje de programare (de exemplu in C si C++), in loc de False se foloseste 0 (zero), iar orice alt numar in afara de 0 corespunde valorii True.

Sa consideram doua variabile X si Y. Sa zicem ca X are valoarea 3, iar Y are valoarea 5. Afirmatia "X este mai mic decat Y" (notata X<Y) este adevarata, pe cand afirmatia "X este mai mare decat Y" este falsa. Dar si arirmatia "X este mai mic sau egal cu Y-2" este adevarata, asa cum se va vedea mai tarziu.

- Negatia

Ceva poate fi adevarat sau fals, dar niciodata amandoua simultan. Cand spunem ceva, in programare ne referim, este clar, la variabile, la constante si expresii. Prin negatie obtinem cealalta valoare, adica valoarea de adevar opusa valorii curente. Astfel, daca P este o propozitie adevarata, atunci negatia lui P (notata adesea non P sau ~P sau ¬P sau not P (in Pascal), sau !P (in C/C++)) este o propozitie falsa. Invers, daca Q este falsa, non Q este adevarata.

De multe ori in programare operam cu relatii intre diferite variabile numerice, de exemplu scriem X<Y. O asemenea expresie este una logica, deci de tip Boolean. Multi programatori incepatori nu stiu cum sa nege o asemenea expresie. Desi se poate scrie not (X<Y) (in Pascal), se poate scrie mai simplu X>=Y (cu sensul ca X este mai mare sau egal cu Y). Este gresit sa se creada ca negatia lui X<Y este X>Y, pentru ca daca X nu este mai mic decat Y atunci fie este mai mare, fie cele doua numere sunt egale. Deci, atentie!

- Si

Conjunctia este o operatie cu valori logice. Ea se refera la operatorul "si" (notat cu ∧ in logica, sau cu *and* in unele limbaje de programare (ex. Pascal, Basic), respectiv cu "&&" in C/C++.).

Ce inseamna, de fapt, P si Q? Inseamna ca se intampla si P si Q, iar P si Q este o expresie adevarata daca si numai daca P este adevarata si Q este adevarata (in acelasi timp). Astfel, o afirmatie de forma "X>0 and Y>0" va fi adevarata doar daca ambele valori X si Y vor fi pozitive nenule. Daca macar una dintre ele este falsa, atunci intreaga conjunctie este falsa. La fel si daca amandoua sunt false.

- Sau

De obicei conjunctia nu ridica probleme. Nu acelasi caz este in cazul disjunctiei, reprezentata prin operatorul "sau". Acesta se noteaza cu "$\vee$" in logica matematica, cu *or* in Pascal si Basic si cu || in limbajul C/C++. O expresie X or Y este adevarata in trei din cele patru cazuri posibile: cand X este adevarata si Y este falsa, cand X este falsa si Y este adevarata, dar si in cazul cand X este adevarata si Y este adevarata. Acest lucru nu prea se intelege in limbajul curent. De exemplu, daca un om este intrebat ce va face in concediu si el raspunde prin "Ma duc la mare sau la tara.", noi intelegem ca fie se va duce la mare, fie la tara, dar nu ne gandim ca s-ar putea duce in ambele locuri. Daca spunem ca o femeie este fie frumoasa, fie desteapta, excludem in mod gresit ca este posibil ca o femeie sa fie si frumoasa si desteapta in acelasi timp! Aceasta excludere, daca are loc atunci cand programam, inseamna ca noi confundam operatia "sau" cu operatia "sau exclusiv", pe care o vom prezenta in continuare.

- Sau exclusiv versus Sau

Prin "sau" in programare se intelege, de fapt, un "sau inclusiv", adica daca atat P, cat si Q sunt adevarata, atunci si P sau Q este adevarata. Prin "sau exclusiv" (notat in Pascal si Basic prin *xor*) se intelege ca ori P este adevarata, ori Q este adevarata, dar niciodata amandoua simultan. Astfel, P xor Q este o propozitie adevarata numai daca exact una dintre propozitiile P si Q este adevarata. Daca P si Q sunt amandoua false, sau amandoua sunt adevarate (atentie!), atunci P xor Q este o propozitie falsa. Asadar, de multe ori, in limbajul curent intelegem prin cuvantul "sau" ceea ce in programare intelegem prin operatorul de disjunctie exclusiva *xor*.

- Proprietati ale operatorilor logici

Operatorii logici (not, and, or si xor) au si ei niste proprietati, ca si operatorii aritmetici (+, - etc.). Astfel, prioritatea cea mai mare o are negatia (not), urmata de conjunctie si apoi de *or* si *xor*. Daca nu ne convine ordinea in care se vor efectua operatiile, nu avem decat sa folosim paranteze, de cate ori avem nevoie. Astfel, stim de prin clasa a II-a ca daca avem de calculat 2+3*4 rezultatul este 14 si nu 20. Ni se pare evident ca rezultatul este 14 si ne intrebam de ce ar fi 20, dar uitam ca "ne-a intrat in sange" sa efectuam mai intai operatia de inmultire (3*4 = 12) si apoi adunarea cu 2, ca sa ne dea 14. Daca am fi facut intai operatia de adunare (tinand cont ca este prima intalnita), am fi avut 2+3=5 si apoi, efectuand inmultirea am fi obtinut 5*4=20. Daca am fi dorit sa obtinem acest rezultat, trebuia sa folosim paranteze, astfel: (2+3)*4 si totul era OK.

Asa stau lucrurile si cu operatiile *or* si *and*, de exemplu. Daca scriem X or Y and Z, aceasta expresie este echivalenta cu X or (Y and Z) si nu cu (X or Y) and Z. Astfel, daca X ar fi falsa, Y adevarata si Z adevarata, atunci expresia in discutie este una adevarata (corect). Daca in mod gresit am face operatiile de la stanga la dreapta, fara a tine cont de prioritatea lui "and" fata de "or", expreia ar iesi, in mod gresit, una falsa.

Spuneam ca negatia are oricum prioritatea cea mai mare. Astfel, daca scriem "not X or Y", inseamna ca am scris ceva echivalent cu "(not X) or Y" si nu cu "not (X or Y)". Sa nu ne mire atunci ca daca X si Y sunt adevarate, expresia "not X or Y" este tot adevarata si nu falsa!

- Legile lui De Morgan

Legile lui De Morgan le-am intalnit la operatii cu multimi, dar lucruri similare se intampla si in cazul operatorilor logici. Daca facem analogii cu multimile, atunci in loc de complementara unei multimi avem negatia, in loc de intersectie avem conjunctia, iar in loc de reuniune avem disjunctia. Nu e de mirare, atunci, ca si simbolurile seamana intre ele: Pe de o parte, reuniunea se noteaza cu $\cup$, iar disjunctia cu $\vee$, iar pe de alta parte intersectia se noteaza cu $\cap$ si conjunctia cu $\wedge$.

Legile lui De Morgan in cazul logicii booleene ne ajuta pentru a scrie unele expresii logice ceva mai elegant sau mai usor de urmarit. Ele sunt urmatoarele:

$$\neg (P \vee Q) = \neg P \wedge \neg Q$$

si

$$\neg (P \wedge Q) = \neg P \vee \neg Q$$

Acum sa consideram un caz concret din programare. Sa zicem ca avem de exprimat faptul ca X nu apartine intervalului [A,B). Putem scrie not (X>=A and X<B), dar putem scrie si not (X>=A) or not (X<B). Simplificam lucurile si ajungem la varianta X<A or X>=B, care este cea mai eleganta. Incepatorii gresesc, pentru ca ar scrie X<A and X>=B, or asta nici nu este adevarat, nici macar nu se poate, pentru ca A este mai mic decat B!

Asadar, cu AND si cu OR nu ne jucam cum vrem noi, pentru ca intotdeauna trebuie sa rationam corect si logic.

Capitolul 3. Algoritmi

3.1. Etapele rezolvării unei probleme

Rezolvarea unei **probleme** nu trebuie realizată niciodată la întâmplare. Este mult mai bine să se procedeze sistematic, decât haotic. Este bine să se determine anumite reguli care, urmate, să conducă la obținerea soluției.

Rezolvarea unei probleme cu ajutorul calculatorului electronic presupune mai multe etape.

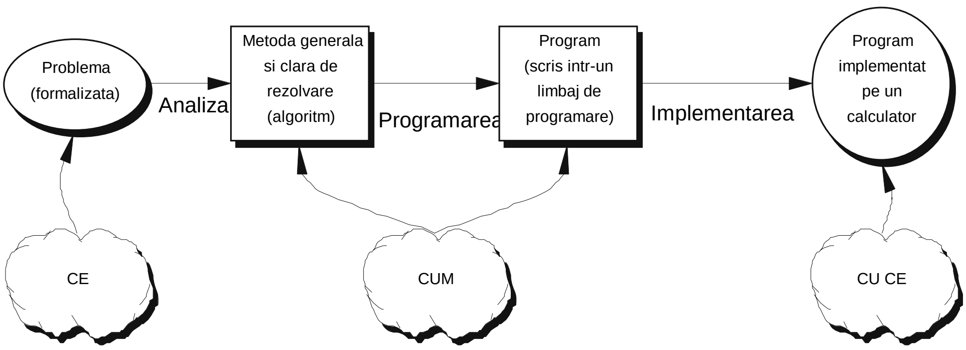
În primul rând, trebuie ca problema să fie formalizată. Aceasta presupune să clarificăm ce se dă şi ce se cere: care sunt datele de intrare şi care sunt datele de ieşire. O dată cunoscute acestea, se vor determina structurile de date cele mai potrivite reprezentării lor în calculator.

Al doilea pas este analiza problemei. Ea se bazează pe aplicarea unei gândiri algoritmice, bazată pe raționamente de tip matematic şi logic, în urma cărora se obține o metodă generală de rezolvare, descrisă clar, folosind reprezentări formale (scheme, limbaje, tabele). Astfel se obține un **algoritm**.

O altă etapă foarte importantă a rezolvării problemei este **programarea**. Astfel, soluția este reprezentată sub o formă acceptată de calculatorul electronic, adică un program.

Implementarea este ultima etapă, ce constă în aplicarea în practică a soluției date, adică scrierea programului pe un calculator.

Schematic, rezolvarea unei probleme cu ajutorul calculatorului se realizează astfel:



3.2. Definiția algoritmului

Un **algoritm** este o metodă generală de rezolvare a unei probleme. El este constituit dintr-o succesiune finită de **paşi** sau etape.

Prin **pas** de algoritm se înțelege o secvență finită de operații (acțiuni) care se pot efectua într-o unitate stabilită de timp. Un pas conține cel puțin o acțiune.

După fiecare pas urmează exact un pas, în execuția algoritmului, eventual în funcție de anumite condiții.

Să considerăm problema ordonării crescătoare a n elemente de acelaşi tip, comparabile între ele, care ar fi stocate sub forma unui vector X cu n elemente, numerotate de la 1 la n. Aceste elemente se află, inițial, într-o ordine oarecare, iar la sfârşit ar trebui să îndeplinească condiția ca fiecare să fie mai mic sau egal decât succesorul său: X(1)<=X(2)<=...X(n). Astfel, am rezolvat etapa formalizării problemei.

A doua etapă este analiza problemei şi proiectarea algoritmului de rezolvare. Metoda pe care o vom folosi (cunoscută sub denumirea de "sortarea prin bule") constă în interschimbarea, pe rând, a câte două elemente succesive din şir, până când şirul va fi ordonat.

Astfel, se compară X(1) cu X(2) şi, dacă nu sânt în ordine (adică X(1)>X(2)), atunci se interschimbă. Apoi se procedează analog cu X(2) şi X(3), X(3) şi X(4) ş.a.m.d., până la X(n-1) şi X(n). Procesul se reia până când, la o anumită parcurgere a vectorului, nu are loc nici o interschimbare, ceea ce reprezintă faptul că vectorul este ordonat.

Metoda de rezolvare trebuie acum scrisă sub forma unui algoritm. Pentru a realiza acest lucru se poate folosi una din modalitățile de reprezentare a algoritmilor.

- limbaje de tip pseudocod;
- scheme logice.

Limbajele de tip pseudocod sunt acelea care folosesc diferite cuvinte numite **cuvinte cheie**, preluate dintr-un limbaj natural, care au un înțeles strict, ele neputând fi folosite în alt context. Exemple: dacă, atunci, altfel, cât timp, execută, repetă, până când, pentru ş.a.. Acestea formează **lexicul** (vocabularul) limbajului. Regulile de formare a instrucțiunilor, pe baza cuvintelor cheie, împreună cu alte cuvinte sau simboluri, determină **sintaxa** limbajului. **Instrucțiunea** este considerată cea mai mică entitate executabilă dintr-un limbaj de programare, dar, prin generalizare, ea poate fi considerată chiar un pas de algoritm, când acesta se reprezintă sub formă de limbaj pseudocod. De fapt, limbajele de tip pseudocod seamănă foarte mult cu cele de programare. **Semantica** limbajului este dată de înțelesurile pe care le capătă instrucțiunile ce alcătuiesc un algoritm.

Limbajele de tip pseudocod folosesc o sintaxă mult mai liberă decât cele folosite de limbajele de programare, acesta fiind cele care permit scrierea unor programe recunoscute de un calculator. Ele au şi avantajul că un algoritm scris într-un limbaj pseudocod poate fi înțeles de toți programatorii, în timp ce unul scris într-un limbaj de programare va fi înțeles doar de cunoscătorii respectivului limbaj.

Folosind o reprezentare de tip pseudocod, metoda de ordonare descrisă se poate scrie ca algoritm astfel:

```
Citeşte(n, X)
repetă
        ordonat = Adevărat;
        pentru i de la 1 la n-1 execută
                dacă X(i)>X(i+1) atunci
                        ordonat = Fals;
                        interschimbă pe X(i) cu X(i-1)
până când ordonat=Adevărat
```

O altă modalitate de reprezentare a algoritmilor o constituie utilizarea **schemelor logice**.

Schemele logice sunt nişte scheme grafice, realizate cu ajutorul unor simboluri speciale care definesc instrucțiuni sau condiții, pentru reprezentarea unui algoritm. Fiecare schemă logică începe cu un dreptunghi având colțurile rotunjite, în care este scris cuvântul START. Schema logică se termină cu unul sau mai multe dreptunghiuri rotunjite la colțuri, ce conține cuvântul STOP. Elementele constituente ale unei scheme logice sunt legate între ele prin săgeți, care indică sensul desfăşurării calculelor.

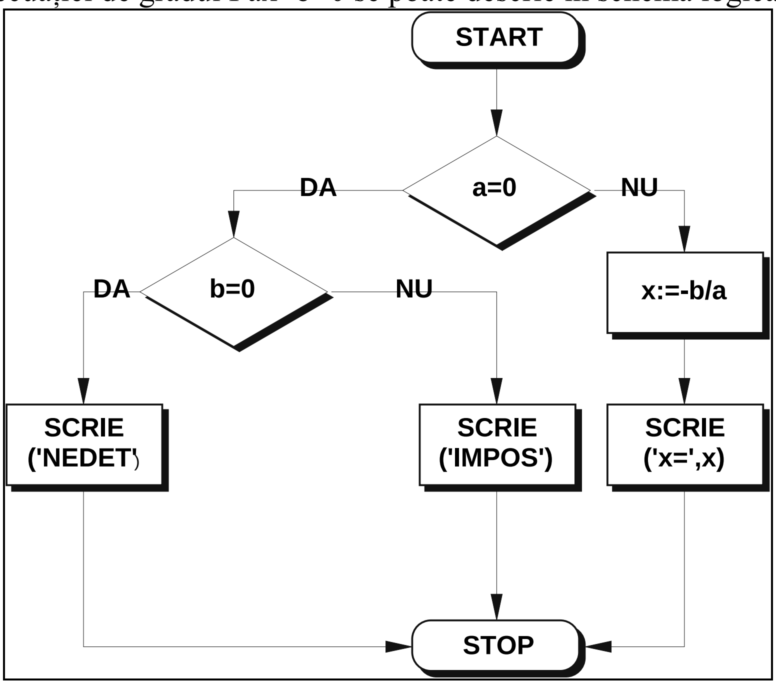
Instrucțiunile simple sunt reprezentate prin dreptunghiuri, în interiorul cărora se scrie acțiunea realizată de respectiva instrucțiune. Într-un dreptunghi intră cel puțin o săgeată, dar iese exact una.

Romburile sunt folosite pentru a reprezenta condiții. Unii autori preferă să utilizeze triunghiuri în locul romburilor. În orice caz, prin unul din colțuri se intră cu o săgeată, iar prin alte două colțuri se iese cu câte o săgeată. Cele două săgeți au ataşate cuvintele DA, respectiv NU, sau ADEV|RAT, respectiv FALS, care indică în ce fel este condiția din interiorul rombului (triunghiului).

Din dreptunghiul START pleacă o săgeată, care indică începutul execuției algoritmului, iar în dreptunghiul STOP intră săgeți, pentru terminarea algoritmului.

Exemplu:

Rezolvarea ecuației de gradul I ax+b=0 se poate descrie în schemă logică astfel:

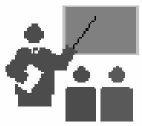


În limbaj natural, putem spune că rezolvarea ecuației se face astfel: se testează dacă a este 0 sau nu; dacă a=0, atunci dacă şi b este zero, ecuația este nedeterminată, iar altfel este imposibilă. Dacă a nu este zero, soluția ecuației este unică, x = -b/a. Descrieți dumneavoastră rezolvarea acestei ecuații în limbaj pseudocod. Realizați, de asemenea, schema logică a algoritmului de ordonare a vectorului X cu n numere.

Observație

Algoritmul problemei ecuației de gradul I presupune, totuşi, citirea mai întâi a valorilor lui a şi b, iar la sfârşit afişarea unui text sau a valorii lui x. Acestea sunt instrucțiuni speciale, pentru realizarea operațiilor de intrare şi ieşire (citire/scriere). Unii autori folosesc romburi pentru a reprezenta asemenea operații, în interiorul cărora figurează "citeşte ... ", respectiv "scrie ...". Alții preferă ca operațiile de citire (introducere) de date să fie reprezentate prin trapeze isoscele cu baza mare sus, iar scrierile să fie reprezentate prin trapeze isoscele cu baza mare jos, în interiorul trapezelor scriind doar valorile, expresiile sau variabilele în cauză.

3.3. Caracteristicile algoritmului



Am observat că algoritmul de rezolvare a problemei ordonării a n numere, ca şi cel de rezolvare a ecuației de gradul I sunt **generali**, în sensul că ei rezolvă cele două probleme pe orice caz.. Se spune că algoritmul rezolvă o clasă întreagă de probleme înrudite.

Când spunem că un algoritm **rezolvă** o problemă înțelegem că se va ajunge la soluție după un număr finit (chiar dacă foarte mare) de paşi. De asemenea, observăm că algoritmii noştri nu au fost descrişi cu ambiguitate, deci sunt descrişi **clar**. Observăm, de asemenea, că rezolvarea celor două probleme este **posibilă** cu resursele pe care le avem la dispoziție.

Toate aceste trăsături caracterizează un **algoritm**, deosebindu-l astfel de orice altă metodă de rezolvare a unei probleme.

Aşadar, un algoritm este o metodă de soluționare a unei clase de probleme, metodă reprezentată de o succesiune finită de paşi, care are următoarele caracteristici:

- este descris clar, fără ambiguități în privința ordinei de execuție a instrucțiunilor;
- este corect, deci este o metodă care rezolvă problema pe orice caz, deci rezolvă o întreagă clasă de probleme;
- este finit, deci se termină după un număr finit de paşi, indiferent câți de mulți;
- este realizabil cu resursele disponibile

Denumirea de algoritm vine de la numele matematicianului persan Abu Ja'far ibn Musa *al Khowarizmi*, adică din oraşul Khowarazm (astăzi Khiva, în Uzbekistan). Este acelaşi care a introdus denumirea de algebră în matematică.

Dându-se o anumită problemă, se pune întrebarea: există un algoritm care să o rezolve? Avem trei răspunsuri posibile: DA, caz în care se construieşte un algoritm; NU, caz în care se poate demonstra (destul de dificil) că nu există o soluție algoritmică pentru respectiva problemă; NU ŞTIM dacă există sau nu, caz în care s-ar putea ca problema să aibă o soluție algoritmică, iar noi o căutăm.

Fireşte, pentru a rezolva o problemă, orice algoritm găsit poate fi acceptat, însă se preferă cei care consumă resurse spațio-temporale mai mici. În disputa spațiu-timp, de obicei se dă câştig de cauză timpului.

Un programator va căuta întotdeauna să scrie algoritmi cât mai performanți din punct de vedere al timpului de execuție, determinat ca o formulă în funcție de cantitatea datelor de intrare. Mărimea datelor de intrare se numeşte **dimensiunea problemei**, iar timpul necesitat de un algoritm în funcție de dimensiunea problemei poartă denumirea de **complexitate a algoritmului**.

Un exemplu concludent de comparare a complexității a doi algoritmi îl oferă problema determinării apartenenței unui număr p la un şir de n numere $a_1, a_2, ..., a_n$ ordonat (crescător). Astfel, o metodă generală de căutare, cum este cea a **căutării secvențiale**, este mult mai înceată, în general, decât **căutarea binară**. În căutarea secvențială se pleacă dintr-un capăt al şirului către celălalt, până se găseşte elementul p sau până se ajunge la celălalt capăt al şirului. Această metodă este foarte folositoare în cazul în care nu se cunoaşte nimic despre aranjarea elementelor din şir. Pe de altă parte, căutarea binară profită de ordonarea deja existentă în cadrul şirului $a_1, ... a_n$, procedând după cum urmează:

- dacă numărul din mijloc este mai mic decât numărul căutat, atunci căutăm în a doua jumătate;
- dacă numărul din mijloc este mai mare ca numărul căutat, atunci căutăm în prima jumătate;
- dacă numărul din mijloc este egal cu numărul căutat, înseamnă că am găsit numărul în cauză şi trebuie să oprim căutarea.

Căutarea în jumătatea aleasă se face tot la fel, deci se va înjumătăți şi această zonă ş.a.m.d, până se termină zona de căutare.

De un real interes se bucură aşa numiții **algoritmi recursivi**, care au proprietatea că în descrierea lor se autoapelează (de obicei pentru probleme de dimensiuni mai mici). Chiar şi algoritmul de căutare binară poate fi scris sub o formă recursivă.

Ce e important si ce nu in programare?

Despre algoritmi si programe. Limbaje si medii de programare. Cum alegem un mediu de programare

- Daca inlocuiesti stiloul cu pana nu ajungi poet.

E mai mult decat evident! Au fost atatia scriitori care au scris noaptea, la lumina lumanarii si cu pana si au lasat omenirii opere literare de mare valoare. Asa cum un amator netalentat poate folosi stiloul cu penita de aur cel mai scump sau chiar calculatorul si imprimanat si sa nu realizeze nimic valoros din punct de vedere literar. Asa stau lucrurile si in programare. Poti apela la un *mediu de programare* foarte performant si sa nu poti realiza nimic valoros, sub aspect *informatic*, pe cand- poate - folosind un mediu de programare mai putin dezvoltat, sa realizezi *programe* care sa-i incante pe toti. Mediul de programare este pentru programator, precum pana sau stiloul pentru scriitor. Iar programul este precum un roman sau o poezie.

Fireste, daca folosesti un toc a carui penita o inmoi mereu in calimara cu cerneala, poti avea diverse probleme. Poate ca din greseala rastorni calimara si patezi tot ce ai scris sau poate penita va scrie in unele locuri mai ingrosat, iar in altele mai subtire. Asta, insa, nu va afecta nicicum valoarea operei literare si nu va schimba cu nimic opinia criticilor despre ea. Daca scriitorul va folosi un stilou cu rezerve de cerneala sau multe din problemele sale se vor rezolva. Daca va apela la un calculator, va putea corecta cu usurinta greselile, adauga noi fragmenet de text printre cele scrise deja si va putea imprima lucrarea folosind caractere diferite. Va lucra mult mai comod, deci schimbarea instrumentului este in favoarea scriitorului si nu a cititorului. Asa se intampla si cu programele. Folosind medii de programare avansate, programatorul isi va usura o importanta parte din munca sa, dar pentru a veni in intampinarea "cititorilor" sau, va trebui sa dea dovada de mult talent, de multa pricepere si imaginatie, pentru ca "opera" sa sa fie deosebita si sa raspunda cerintelor beneficiarilor.

- Ce este algoritmul? De ce trebuie sa inveti sa sofezi in general si nu sa conduci Dacia?

Pentru ca daca inveti sa conduci Dacia, s-ar putea sa nu te descurci decat cu autoturismul Dacia, eventual pe un alt model, dar nu si pe o masina la volanul careia nu te-ai asezat niciodata. Sa presupunem ca intr-o buna zi vei avea un Mercedes. Daca vei stapani tehnica sofatului in general si nu vei invata pe de rost cateva comenzi de la masina Dacia, vei reusi sa te adaptezi cu usurinta noului tau autoturism si, fara experienta prea mare la volanul sau, vei putea, in cateva zile, sa conduci acest Mercedes asa cum conduceai Dacia.

Programarea se rezolva cu rezolvarea de probleme. Fireste, nu orice gen de probleme, ci acelea care opereaza cu informatii si pot fi modelate pe calculator. Daca nu stim o metoda generala de rezolvare a problemelor, ci doar un numar de rezolvari de probleme particulare, nu vom putea sa ne descurcam cu usurinta in situatii noi. Noi trebuie sa stapanim tehnica rezolvarii de probleme de programare si nu sa invatam pe de rost cum se rezolva problema X sau problema Y. Daca stim sa realizam un program prin care sa desenam, pe ecranul calculatorului, un patrat rosu si un patrat verde nu trebuie sa fim multumiti! Trebuie sa vedem cum putem desena un patrat de orice culoare si in orice pozitie a ecranului. Adica sa determinam *algoritmul* de rezolvare a problemei desenarii patratelor.

Algoritmul nu este altceva decat o metoda de rezolvare a unei clase de probleme, adica a unor probleme foarte asemanatoare intre ele. In general aceste probleme difera intre ele prin *dimensiunea* lor, exprimata adesea printr-un numar natural *n*. Pentru a intelege mai bine, vom exemplifica. Mai intai, vom considera doua probleme foarte asemanatoare.

Sa consideram ca avem un pahar cu vin si unul cu suc si un pahar gol. Pentru a interschimba continutul primelor doua pahare, putem turna vinul in paharul al treilea. Acum primul pahar este gol si putem turna in el continutul celui de-al doilea pahar, adica sucul. Paharul al doilea devine gol si turnand din paharul al treilea in el, vom avea aici vinul.

A doua problema este exact ca prima, doar ca in loc de vin si suc avem apa si bere. Fireste, problema se rezolva la fel. In general, pe programator nu-l intereseaza ce se gaseste in cele doua pahare, el pur si simplu doreste sa gaseasca metoda de interschimbare a continuturilor celor doua pahare, pur si simplu. Astfel, din punct de vedere informatic, cele doua probleme prezentate mai sus nu sut doua probleme de programare diferite, ci doar una. Asta deoarece informaticianul nu ia niciodata in considerare "continuturile paharelor", deci nu-l (prea) intereseaza *valorile datelor* pe care le prelucreaza. Pentru el e foarte putin important daca in cele doua pahare se afla vin, bere, suc sau chiar acid clorhidric sau sulfuric. Lucruri esentiale pentru cei ce utilizeaza un program pot fi fara nici o importanta pentru programator!

Spuneam, totusi, ca in general problemele rezolvate de acelasi algoritm difera printr-un numar natural *n*. Astfel, in cele doua cazuri de mai sus avem de a face cu o singura problema de informatica, care se numeste *interschimbarea valorilor a doua variabile*. Nu se poate spune despre cele doua probleme ca formeaza o clasa de probleme, dar sa dam un alt exemplu.

Mai multe persoane candideaza la un concurs de admitere la un liceu sau la o facultate. Candidatii pot fi aranjati in ordinea descrescatoare a mediilor sau in ordinea alfabetica a numelor lor sau in functie de orice alt criteriu. Candidatii pot fi oricati, fie 100, fie 2000 fie chiar mai multi, deci putem nota numarul lor cu *n*. Acum avem de a face cu o clasa de probleme, care pot fi rezolvate prin acelasi algoritm. Daca vom gasi metoda generala de rezolvare, adica algoritmul, vom sti sa aranjam 100 candidati, descrescator dupa note, sau 2000 de candidati, in ordine alfabetica.

Asadar, algoritmul este o metoda generala de rezolvare a unei clase de probleme. Nu este de ajuns. Exista multe asemenea metode de rezolvare, dar trebuie sa le consideram doar pe cele care se termina intr-un timp finit, sau intr-un timp util pentru noi. De asemenea, esential este faptul ca un algoritm sa fie descris clar, fara ambiguitati, pentru a putea fi inteles de oricine.

Cum am putea descrie o metoda de rezolvare a unei probleme asa incat sa poata fie inteleasa exact de catre oricine? Sunt mai multe cai, dar folosind limba romana, sau engleza sau orice alta limba naturala, vorbita, este posibil ca sa nu fim intelesi de toata lumea si intotdeauna. Cu atat mai mult de catre un calculator, care nu este atat de inteligent incat sa inteleaga o limba naturala. El poate invata o limba (un limbaj) simplu, cu un vocabular redus si cu putine reguli de sintaxa, dar pe care trebuie sa le invatam (si noi si el) si sa le respectam cu mare rigurozitate.

In limbajul natural pot aparea ambiguitati. Un exemplu este celebra fraza "Am vazut un om pe deal cu un telescop". Aceasta fraza poate fi inteleasa in trei feluri: "Folosind un telescop, am vazut un om, care era pe deal.", "Am vazut pe dealul pe care era un telescop, un om" sau "Am vazut pe deal un om, care avea un telescop la el".

Limbajul natural a fost folosit cu mult succes de invatatoarea noastra, atunci cand ne-a invatat adunarea si scaderea numerelor, cu mai multe cifre (numerele, nu invatatoarea!) Cand am invatat scaderea, ne-a spus sa asezam numerele unul sub altul si e foarte probabil ca toata lumea a inteles ca trebuie sa aseze al doilea numar (scazatorul) sub primul (descazutul) si nu invers. Apoi ne-a explicat ca trebuie sa scadem cifra din cifra, iar cand nu ne ajunge sa ne "imprumutam" de la cifra din dreapta. Noi am inteles, mai repede sau mai tarziu, cum se procedeaza, care cifra din care se scade, de la cine ne imprumutam. Am reusit sa invatam din doua motive: pentru ca suntem inteligenti (iar calculatorul, atentie!, nu este) si pentru ca am exersat pe mai multe exemple (ceea ce nu se pune problema in cazul calculatorului).

Astfel, pentru a nu aparea ambiguitati, vom reprezenta algoritmii folosind limbaje artificiale, create de om, ca sa vina in sprijinul calculatorului. Cele mai folosite modalitati de reprezentare ale algoritmilor sunt limbajele pseudocod si schemele logice. Despre ele se discuta in manualul de informatica.

- Ce este programul? Ce este un limbaj de programare?

Nu e de ajuns sa gasesti un algoritm pentru a rezolva o anumita problema. Trebuie sa-l si descrii, adica sa-l reprezinti. Cel mai bine este sa folosesti totusi, un limbaj de programare, adica un limbaj de compromis intre om si calculator. Programul este, practic, reprezentarea intr-un asemenea limbaj a unui algoritm. Daca vrei ca algoritmul tau sa poata fi inteles usor de multa lume, care stie sau nu stie un limbaj de programare sau altul, este bine sa apelezi la limbajele pseudocod. Ele seamana mult cu limbajele de programare, dar au mai putine restrictii sintactice si ofera mai multa libertate programatorului.

Dar cel mai bine este sa scrii direct programul, folosind un limbaj sau altul de programare, adecvat problemei pe care doresti sa o rezolvi prin respectivul algoritm. In general, programatorii stiu sa "citeasca" un program scris intr-un limbaj de programare pe care nu-l cunosc, deoarece limbajele de programare seamana mult intre ele, sunt cam la fel gandite si realizate. Seamana intre ele mai mult decat limbile vorbite. Exista, fireste, si limbaje de programare speciale, care nu se aseamana cu celelalte, dar sunt mai putine si nu ne vom referi la ele in aceasta lucrare.

Pascal, C, C++, Java, Basic sunt limbaje de programare. Ele au fost inventate de oameni, cu creionul pe hartie. E ca si cum ai inventa tu acum o limba noua. Inventezi cateva cuvinte (care sa defineasca substantive, verbe, adjective), apoi reguli de scriere, de sintaxa a propozitiei si a frazei si intelesurile (semantica) unor asemenea constructii gramaticale. Asta nu inseamna ca cineva va sti sa si vorbeasca limba inventata de tine!

- Ce este un mediu de programare?

Asa stau lucrurile si cu Pascal, C, C++ si celelalte. Ele au fost inventate de oameni ca Niklaus Wirth, Denis Ritchie, Bjarne Stroupstroup, dar apoi a trebuit sa se realizeze implementari ale lor. Adica niste programe speciale (scrise in alte limbaje de programare, mai vechi si mai primitive) care puse pe calculator sa stie sa inteleaga (compileze sau interpreteze) un text (program) in Pascal, C, C++ etc. O implementare a unui limbaj de programare devine un mediu de programare. Dar un mediu de programare de astazi are mult mai multe functii decat cea principala, de interpretare si executare a programului scris de noi. Un mediu de programare modern te ajuta sa depanezi programul realizat, sa-i schimbi ordinea de executie a instructiunilor din el, sa vizualizezi diferite aspecte legate de datele de intrare, de rezultate sau sa realizezi o serie de prelucrari necesare bunei functionari a programului. De asemenea, un mediu de programare vine cu modificari si imbunatatiri la limbajul de programare de baza, standard.

Turbo Pascal, Delphi sunt medii de programare bazate pe limbajul Pascal, C++ Builder si Visual C++ sunt medii de programare bazate pe limbajele C si C++, iar Visual Basic este un mediu de programare bazat pe limbajul Basic.

- Concluzie! Cine stie sa sofeze poate conduce un Mercedes!

Asadar, am plecat de la problema de programare. Ea prelucreaza informatii pentru a obtine altele. Pentru a rezolva o problema (si toate asemenea ei), avem nevoie de un algoritm. Ca algoritmul sa fie descris fara ambiguitati si sa fie inteles de un calculator, cel mai bine este sa folosim un limbaj de programare adecvat. Algoritmul devine program. Pentru ca totul sa mearga repede, frumos si bine si ca sa punem programul la lucru, vom folosi un mediu de programare corespunzator.

Limbajele de programare sunt cu sutele, iar mediile de programare cu zecile. Fiecare mediu de programare se comercializeaza impreuna cu o carte groasa, numita *documentatia* sa, in care este descris si nimeni nu va putea sa stie pe de rost tot ce este acolo, nici macar cei care au creat mediul de programare si au scris cartea. Nici nu trebuie sa-ti bati capul prea mult cu asta. Tu trebuie sa stii sa rezolvi probleme, deci sa elaborezi algoritmii potriviti problemei date. Vei alege apoi cu mare usurinta limbajul de programare si mediul de programare adecvat si te vei adapta la momentul potrivit lui. Nu vei putea tine niciodata pasul cu evolutia impresionanta a tehnologiei din ultimii ani, de aceea nici nu trebuie sa-ti bati capul cu noile medii de programare care apar pe piata. Ele sunt simple produse comerciale, mai mult sau mai putin performante. Tu invata baza, adica sa programezi, ceea ce inseamna in primul rand sa rezolvi probleme si sa elaborezi algoritmi. Cine stie sa sofeze va putea sa conduca si cel mai sofisticat Mercedes!

Capitolul 4. Elementele programării structurate

4.1. Structurile de bază

O caracteristică importantă a descrierii în limbaj pseudocod a algoritmilor este utilizarea grupurilor de cuvinte: dacă...atunci...altfel... şi atât timp cât... execută... .

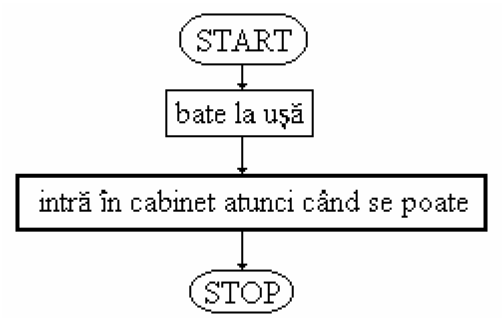
De asemenea, într-un algoritm, instrucțiunile se execută într-o ordine bine definită, care se exprimă în limbajul pseudocod prin instrucțiuni scrise una după alta, de sus în jos, adică *secvențial*. În cele ce urmează vom descrie în limbaj psudocod un algoritm care foloseşte toate aceste elemente.

Să considerăm problema intrării unui bolnav într-un cabinet medical. Putem aprecia că această problemă se rezolvă astfel: mai întâi, bolnavul va bate la uşă; dacă medicul răspunde afirmativ (prin "poftim!"), atunci pacientul va intra, altfel va aştepta până când se va elibera cabinetul, după care va intra.

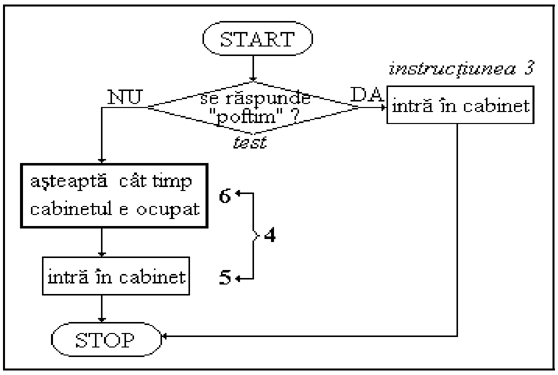
În limbaj pseudocod am putea scrie:

```
1. bate la usă;
2. dacă răspunsul = 'poftim !' atunci
            2.1. intră în cabinet
    altfel
            2.2.1. atât timp cât cabinetul este ocupat execută
                       2.2.1.1. aşteaptă;
            2.2.2. intră în cabinet
```

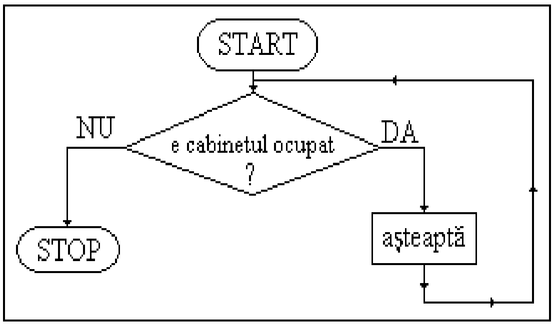
Putem reprezenta schematic succesiunea anterioară de instrucțiuni astfel:



Primul dreptunghi corespunde instrucțiunii 1, al doilea instrucțiunii mari 2. Aceasta este compusă din mai multe instrucțiuni:2.1, 2.2.1 şi 2.2.2. Instrucțiunea 2.2.1 este compusă din anumite cuvinte speciale şi din instrucțiunea 2.2.1.1. De aceea am desenat îngroşat cel de al doilea dreptunghi. Instrucțiunea respectivă se detaliază astfel:

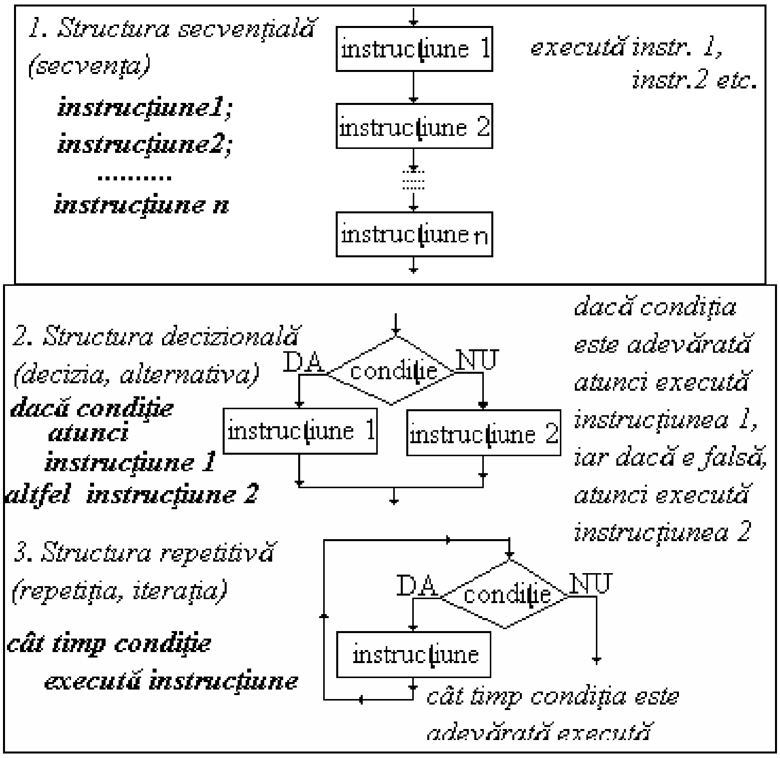


Aşadar, condiţia încadrată de cuvintele *dacă* şi *atunci* este reprezentată în schema alăturată printr-un romb, cu o intrare şi două ieşiri spre cele două ramuri: NU şi DA. Ramura NU conţine, la rândul ei o instrucţiune complexă:



Şi instrucţiunea *atât timp cât... execută...* conţine o condiţie dintr-un romb. Ea este încadrată de grupurile de cuvinte *atât timp cât* şi *execută*.

Exemplul anterior ne-a permis să evidenţiem trei **structuri de control de bază**, cu ajutorul cărora se pot descrie algoritmii: **secvenţa**, **decizia** şi **repetiţia condiţionată anterior**.

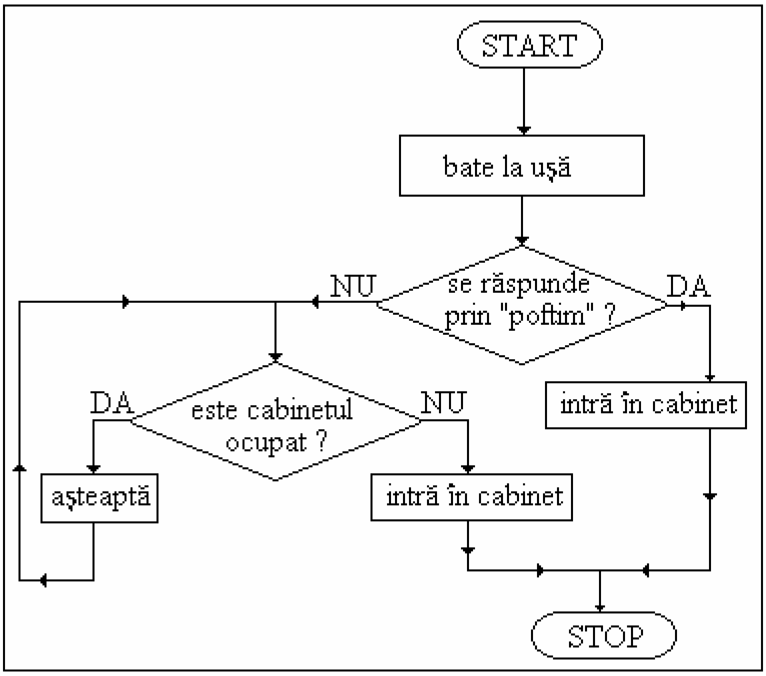


Fireşte, prin instructiune (pas de algoritm) se poate înţelege, în toate cele trei cazuri, fie o instrucţiune simplă, fie una compusă.

O instrucţiune compusă este formată dintr-o *secvenţă* de instrucţiuni, eventual încadrate de cuvintele *început* şi *sfârşit*. Instrucţiunile componente pot conţine, la rândul lor, blocuri de *alternativă* şi *repetiţie*.

Cele trei structuri prezentate sunt numite **structuri de bază**. Ultima este, de fapt, o repetiţie cu test iniţial, sau condiţionată anterior. Vom vedea mai târziu că există şi alt gen de repetiţii, când vom învăţa despre structurile auxiliare.

Acum putem da schema completă a problemei intrării în cabinetul medical, care foloseşte cele trei structuri de bază:



Prin generalizare, putem spune că **programarea** este arta şi tehnica realizării de algoritmi, care ulterior vor fi descrişi şi implementaţi în programe pe calculator, scrise într-un limbaj de programare.

Programarea pe baza celor trei structuri (şi a celor auxiliare, ce vor fi prezentate) se numeşte **programare structurată**.

Informaticienii Böhm şi Jacopini au formulat un **principiu al programării structurate**, sub forma unei teoreme: cele trei structuri (secvenţa, decizia şi repetiţia condiţionată anterior) sunt suficiente pentru a descrie orice algoritm.

Oricărei scheme logice i se pot adăuga noi instrucţiuni şi noi condiţii (predicate) astfel încât să se obţină o schemă logică structurată, echivalentă primeia.

Pentru a înţelege ce ar însemna programare nestructurată, să reconsiderăm algoritmul de intrare în cabinetul medical. Repetiţia *atât timp cât cabinetul este ocupat execută aşteaptă* poate fi rescris astfel:

```
A: dacă cabinetul este ocupat atunci
             aşteaptă
             treci la pasul A
```

Această întoarcere ("treci la pasul A") poate deranja, deoarece revenirea înseamnă şi nerespectarea regulii conform căreia instrucțiunile se execută secvențial, una după alta. Trecerea execuției programului la o anumită instrucțiune etichetată poate avea loc şi peste un număr foarte mare de alte instrucțiuni, iar o astfel de instrucțiune, numită **instrucțiunea de salt necondiționat** trebuie evitată. Programarea se numeşte nestructurată dacă foloseşte instrucțiunea de salt necondiționat.

Dacă saltul ar fi către o instrucțiune foarte îndepărtată, atunci ar fi mai greu de urmărit algoritmul. Informaticienii au demonstrat că nici *nu e nevoie* să fie folosită această instrucțiune de salt, deoarece cele trei elemente ale programării structurate sunt *suficiente* pentru a descrie orice algoritm. Dacă, însă, folosirea instrucțiunii de salt ar face procedura mai lizibilă, atunci se poate apela la această instrucțiune.

Observație

De remarcat că structurile secvențială, alternativă şi repetitivă condiționată anterior sunt suficiente, nu şi neapărat necesare pentru proiectarea structurată a unui algoritm.

De asemenea, structura alternativă se poate elimina, folosind două structuri repetitive, în combinație cu introducerea unei variabile logice (un element ce poate fi adevărat sau fals), după cum urmează:

dacă c atunci i_1 altfel i_2

se înlocuieşte, folosind variabila logică b, cu:

```
b=Adevărat;

cât timp (b=adevărat) şi (c) execută

i1; b=Fals;

cât timp (b=adevărat) şi (c) execută

i2; b:=fals sfârsit
```

4.2. Structurile auxiliare

Să revenim asupra algoritmului de ordonare a unui vector X cu n elemente, pe care l-am prezentat în paragraful 3.2:

```
Citeşte(n, X)
repetă
      ordonat = Adevărat;
      pentru i de la 1 la n-1 execută
            dacă X(i)>X(i+1) atunci
                  ordonat = Fals;
                  interschimbă pe X(i) cu X(i-1)
până când ordonat=Adevărat
```

Observați că descrierea algoritmului în pseudocod foloseşte trei structuri de control asemănătoare celor prezentate deja, dar au anumite diferențe față de ele. Astfel de **structuri de control**, numite **auxiliare**, au fost introduse tocmai pentru a uşura scrierea algoritmilor, dar ele pot fi substituite de celelalte.

Prima dintre ele este **decizia cu ramură vidă**, adică:

```
dacă condiție atunci
      instrucțiune
```

adică nu mai există ramura cu *altfel*

În acest caz, dacă este adevărată condiția, atunci se execută instrucțiunea, iar dacă este falsă condiția, nu se mai execută nimic. Ați observat, probabil, că s-a folosit această structură în algoritmul de sortare prezentat mai sus.

Există şi o formă specială de decizie, numită **decizia multiplă**:

```
în caz că e este
          e1: instrucțiune1;
          e2: instrucțiune2;
          ..........................
```

Astfel, se evaluează expresia e, iar dacă valoarea ei este identică cu valoarea uneia dintre expresiile e_1, e_2 ş.a.m.d., atunci se execută instrucțiunea corespunzătoare.

Algoritmul de sortare descris mai înainte foloseşte două structuri speciale: **structura repetitivă condiționată posterior**:

```
repetă
      instrucțiune1;
      instrucțiune2;
      .....................
până când condiție
```

Aceasta înseamnă că se execută în mod repetat secvența de instrucțiuni specificată, până la îndeplinirea condiției de la final. Spre deosebire de structura repetitivă condiționată anterior, aici secvența de instrucțiuni se execută cel puțin o dată. În cazul lui *cât timp ... execută ...*, dacă de la bun început nu era îndeplinită condiția, atunci instrucțiunea sau instrucțiunile din ciclu nu se execută.

Astfel, structura *repetă* poate fi scrisă, pe baza structurilor de bază astfel:

```
instrucțiune1;
instrucțiune2;
....................
cât timp not condiție execută
         instrucțiune1;
         instrucțiune2;
         ....................
```

Atât în cazul repetiției condiționată anterior, cât şi în cea condiționată posterior, prezentate mai înainte, numărul de paşi ai ciclului nu poate fi calculat aprioric. Asemenea structuri repetitive se mai numesc şi **cu număr necunoscut de paşi**.

Structura specială pentru este o **structură repetitivă cu număr cunoscut de paşi**. Astfel, numărul de paşi ai ciclului poate fi calculat aprioric pe baza celor trei expresii ce compun structura:

```
pentru v de la e1 la e2 [cu pasul e3] execută
          instrucțiune
```

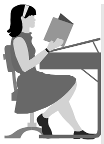
Structura *pentru* poate fi considerată fie condiționată anterior, fie condiționată posterior, iar acest lucru depinde de implementarea limbajului în care structura se regăseşte sub forma unei instrucțiuni. În general, ea este o structură condiționată anterior, iar semnificația ei este:

Dacă e_3 lipseşte, atunci se consideră e_3=1.
Dacă (e_3 > 0 şi e_1>e_2) sau (e_3<0 şi e_1<e_2), atunci nu se execută nimic.
În orice alt caz, structura este echivalentă cu:

```
v = e1;
cât timp v<>e2 execută
     instrucțiune;
     v = v + e3
```

Aşadar, orice structură de control de tip *pentru*, ca şi celelalte structuri auxiliare, poate fi scrisă pe baza celor de bază.

4.3. Teorema programării structurate



Există o serie de teoreme care se referă la programarea structurată, dar pentru noi are importanță teorema de structură a lui Böhm şi Jacopini care se poate enunța astfel: orice schemă logică nestructurată poate fi înlocuită cu una echivalentă, dar structurată, prin adăugarea de noi acțiuni şi condiții. Aşadar, orice program poate fi pus sub o formă structurată, adică să conțină doar structurile de bază şi/sau structurile auxiliare, prin utilizarea unor variabile boolene asociate unor funcții suplimentare.

Să exemplificăm pe cazul unui algoritm scris în limbaj pseudocod. Algoritmul următor determină cel mai mic element (notat min) din şirul de n elemente X:

```
           Citeste(n, X);
           i = 2;
           min = X(1);
A:       dacă i>n atunci
                   treci la pasul B
           altfel
                   dacă X(i)<min atunci
                           min = X(i);
                   i = i+1;
                   treci la pasul A
B:       Scrie(min)
```

Observați că algoritmul nu este scris sub o formă structurată (apare instrucțiunea de salt necondiționat "treci la pasul"). În schimb, următoarea formă reprezintă acelaşi algoritm şi este structurată:

```
Citeste(n, X);
min = X(1);
pentru i de la 2 la n execută
          dacă X(i)<min atunci
                 min = X(i);
Scrie(min)
```

Aceasta este cea mai elegantă formă de scriere a algoritmului de determinare a minimului. O altă formă, care foloseşte doar structurile de bază ar fi:

```
Citeste(n, X);
min = X(1);
i = 2;
```

```
cât timp i<= n execută
          dacă X(i)<min atunci
                  min = X(i);
          i = i+1
Scrie(min)
```

4.4. Instrucțiunea de atribuire. Operații de intrare şi ieşire

Instrucțiunea de atribuire

Vom reveni în acest paragraf şi vom insista asupra unor instrucțiuni esențiale în orice limbaj de programare.

În primul rând, avem **instrucțiunea de atribuire**, numită şi **de calcul** sau **de asignare**, prin care o variabilă primeşte valoarea unei expresii date. În general, în pseudocod, instrucțiunea de atribuire se notează prin v ← e, iar în limbajele de programare astfel:

Basic: v = e sau Let v = e; Pascal: v := e; C/C++: v = e; FoxPro: v = e sau Store e To v.

În toate cazurile, v este o variabilă, iar e este o expresie. Se consideră că v şi e au acelaşi tip sau că există o compatibilitate între tipul lui v şi tipul lui e. Prin această instrucțiune, vechea valoare a variabilei v se pierde (dacă v avea o anumită valoare), expresia e se evaluează, iar valoarea lui e este dată variabilei v; fireşte, expresia e nu suferă nici o modificare.

Exemplu:

Vom da un exemplu în limbajul Basic. Presupunem că avem trei variabile x, y şi z de tip întreg.

Dim x As Integer, y As Integer, z As Integer
x = 3 (x devine egal cu 3)
y = 2 (y devine egal cu 2)
z = x + y (z devine egal cu 3 + 2, deci cu 5)
x = x + 1 (acum x devine egal cu 3 + 1, adică se adună la fosta valoare a lui x (3) valoarea 1, iar rezultatul (4) se atribuie variabilei x)
y = x + z (acum y devine egal cu 4 + 5, deci 9)

Să considerăm un alt exemplu, din limbajul Pascal:
Fie declarațiile de variabile:
var m, n: Integer; x, y: Real; c, d: Boolean;
Putem avea următoarele atribuiri, considerate corecte:
x := 2 + 3 (rezultatul este 5, număr întreg, dar şi real, iar x va deveni egal cu 5)
m := Round(x/3); (rezultatul este 1, adică 5/3 rotunjit la cel mai apropiat număr întreg)
n := 4; (n ia valoarea 4, număr întreg)
c := m = n; (c este o variabilă booleană; comparându-se valorile lui m şi n, se constată că m este diferit de n (1 este diferit de 4), aşadar c va fi False)
d := not c (d va fi True)

În limbajul C se pot face conversii prin chiar instrucțiunea de atribuire, de la un tip la altul, ca în exemplul următor:

```
int m, n;
float x;
char c;
x = 5.3;
m = x;
```

```
n = 65;
c = n;
```

În acest exemplu, prima dată x ia valoarea 5,3, după care m ia valoarea părții întregi a lui x, adică 5; penultima instrucțiune atribuie lui n valoarea numerică 65, iar prin instrucțiunea c = n, caracterul c devine egal cu 'A', pentru că n este 65, iar caracterul având acest cod ASCII este tocmai 'A'.

Operatori de incrementare/decrementare în limbajul C

Limbajul C pune la dispoziția programatorilor nişte operatori speciali pentru realizarea unor operații speciale de atribuire numite incrementări sau decrementări. Dacă valoarea variabilei a creşte cu valoarea 1 putem scrie a++ sau ++a, în loc de clasicul a=a+1. De asemenea, o decrementare cu o unitate se poate scrie a-- sau --a, în loc de a=a-1.

Dacă se doreşte incrementarea, respectiv decrementarea lui a cu o valoare b oarecare, se va scrie:

a+=b, respectiv a-=b. Astfel, a+=b semnifică a=a+b, iar a-=b semnifică a=a-b. Evident, operația a+=1 este echivalentă cu a=a++, iar a-=1 cu a--.

Cu toate că aparent, a++ şi ++a (ca şi a-- şi --a) semnifică acelaşi lucru, adică o incrementare (decrementare) cu o unitate a lui a, totuşi, există cazuri când cele două instrucțiuni au înțelesuri diferite. Astfel, a++ înseamnă "foloseşte-l pe a, apoi incrementează-l pe a cu 1", pe când ++a înseamnă inversarea celor două operații: "incrementează-l pe a cu 1, apoi foloseşte-l".

Astfel, pentru a calcula suma componentelor unui tablou unidimensional cu indici de la 0 la 9 declarat prin int x[10] putem scrie:

for (S=0; i=0;) S+=a[i++] sau for (S=0; i=-1;) S+=a[++i]

În primul caz se foloseşte i ca indice în vectorul a, apoi se incrementează, iar în al doilea caz i se incrementează, apoi se foloseşte pe post de indice în cadrul vectorului a.

Aşadar, S+=a[i++] este echivalent cu S+=a[i]; i=i+1, pe când S+=a[++i] este echivalent cu i=i+1; S+=a[i].

Operații de intrare şi ieşire

Foarte importante într-un limbaj de programare sunt şi **operațiile de intrare şi ieşire**. Prin acestea se realizează comunicarea între om şi calculator. Operațiile de intrare, numite şi **de citire** sunt acelea prin care anumite variabile primeste valori dintr-un mediu extern, fie dintr-un fişier, fie de la tastatură (considerată şi ea ca un fişier). Operațiile de ieşire, numite şi **de scriere** sau **de afişare** sunt acelea prin care într-un mediu extern (fişier, ecran sau imprimantă) se scriu (afişează, imprimă) valorile unor expresii. Există, în multe medii de programare, anumite fişiere standard pentru intrare, respectiv ieşire. De pildă, în limbajul C, prin stdin se înțelege fişierul standard de intrare, iar prin stdout fişierul standard de ieşire. În general, acestea se consideră a fi tastatura, respectiv monitorul, dar pot fi redefinite.

Operațiile de intrare şi ieşire sunt realizate fie prin apelarea unor proceduri speciale (ca în cazul limbajului Pascal sau C/C++), fie prin folosirea unor instrucțiuni speciale (ca în Basic).

În limbaj de tip pseudocod vom scrie:

Citeşte(v_1, v_2, ..., v_n) pentru a citi valori pentru variabilele v_1, v_2 ş.a.m.d;
Scrie(e_1, e_2, ..., e_n) pentru a scrie valorile expresiilor e_1, e_2 ş.a.m.d..

În limbajul de tip pseudocod am folosit operațiile de citire/scriere doar pentru fişierele standard (tastatură/monitor). În limbajele de programare putem avea şi alte fişiere de citire/scriere.

Astfel, în Basic am vorbit deja despre instrucțiunile:

Input #f, v_1, v_2, ..., v_n, prin care de la fişierul deschis cu numărul de identificare f se citesc valori pentru variabilele v_1, ..., v_n

Line Input #f, v_1, v_2, ..., v_n, care lucrează ca şi instrucțiunea precedentă, dar citeşte linii de şiruri de caractere;

Print #f, e_1, e_2, ..., e_n prin care în fişierul deschis cu numărul de identificare f se scriu valorile expresiilor e_1, e_2, ..., e_n

Dacă #f lipseşte, atunci se consideră că operațiile se execută cu tastatura/monitorul.

Să considerăm acum cazul limbajului Pascal. Există două proceduri de citire şi două proceduri de scriere, care pot lucra cu fişiere, respectiv cu tastatura/monitorul.

Astfel, prin Read(v_1, v_2, ..., v_n) se citesc valori pentru cele n variabilele, de la tastatură, fără a se citi şi caracterul Enter, deci fără sfârşitul de rând; pe când prin ReadLn(v_1, v_2, ..., v_n) se citeşte şi caracterul Enter.

În mod similar, Write(e_1, e_2, ..., e_n) afişează, una după alta, valorile celor n expresii, lăsând cursorul de scriere la dreapta ultimei expresii afişate, iar WriteLn(e_1, e_2, ..., e_n) procedează ca şi Write, dar trece cursorul pe următorul rând, deci mai "afişează" un Enter.

Pentru a citi doar Enter-ul putem folosi ReadLn, iar pentru a trece pe următorul rând WriteLn. Apelul ReadLn(x,y) este echivalent cu secvența: Read(x,y); ReadLn, iar apelul WriteLn(x,y,z) este echivalent cu secvența: Write(x,y,z); WriteLn sau cu secvența: Write(x,y); WriteLn(z) sau cu secvența: Write(x); WriteLn(y,z).

Lucrul cu fişierele este similar, în sensul că numele variabilei de tip fişier este trecut ca prim argument în apelul procedurilor de citire/scriere.

Astfel, prin ReadLn(f,s) (în care var f: Text; s: String) se citeşte din fişierul f şirul de caractere s; iar prin WriteLn(f,x,y,x+y) (în care var x,y: Integer; iar x are valoarea 2 şi y valoarea 3) va afişa în fişierul text f numerele: 2 3 5, iar cursorul va trece pe rândul următor, pentru eventuala următoare scriere.

Pentru realizarea operațiilor de intrare în C şi C++ există foarte multe funcții, pe care cititorul le poate descoperi singur consultând un manual al acestor limbaje.

Observație:

Citirea unui singur caracter de la tastatură, fără acționarea tastei Enter după citire, este considerată foarte utilă şi, de aceea, în multe limbaje sunt puse la dispoziție funcții speciale. Astfel, de pildă, în mediul Turbo (Borland) Pascal, există funcțiile speciale (din biblioteca Crt) KeyPressed şi ReadKey. KeyPressed returnează True dacă *tocmai* s-a apăsat o tastă, iar ReadKey dă chiar caracterul ASCII al tastei apăsate. Funcţiile nu lucrează pentru tastele reci: Ctrl, Alt sau Shift, caz în care utilizatorul trebuie să se folosească de alte elemente de programare.

4.5. Implementarea structurilor de control

Elementele (de bază sau auxiliare) programării structurate se regăsesc în toate limbajele de programare, fie că ele sunt procedurale, fie că nu. Limbajele procedurale se bazează pe o anumită paradigmă de programare asupra căreia vom reveni în capitolul următor.

În continuare vom prezenta sintaxa fiecărei instrucţiuni în mai multe limbaje de programare. Am ales Visual Basic, Pascal, C/C++ şi FoxPro.

Instrucţiunea compusă - structura secvenţială

În limbajul Pascal, instrucţiunile sunt separate de simbolul ";", iar o secvenţă de instrucţiuni cuprinse între cuvântul *begin* şi cuvântul *end* formează o instrucţiune compusă. În limbajul C, instrucţiunile simple se termină prin ";", deci nu sunt separate de ";" ca în Pascal, iar instrucţiunile compuse din C sunt încadrate de "{" şi "}".

```
begin
    instrucţiune1;
    instrucţiune2;
    ...................
    instrucţiunen
end
```

```
{
    instrucţiune1
    instrucţiune2
            ...................
    instrucţiunen
}
```

Observaţie:
În limbajul C o instrucţiune compusă poate conţine şi declaraţii (de variabile).

În Visual Basic sau în FoxPro o secvenţă de instrucţiuni nu trebuie încadrată în vreun fel de două cuvinte speciale. De obicei, structurile de control alternative sau repetitive acceptă, în diferitele lor secţiuni, o secvenţă de mai multe instrucţiuni, deci o instrucţiune compusă, aşa cum se va vedea imediat. De asemenea, în FoxPro, instrucţiunile se separă prin *Carriage Return* (*CR* sau *Enter*). În Visual Basic, separatorii sunt *Carriage Return* sau simbolul ":", atunci când trebuie separate instrucţiuni de pe acelaşi rând.

Instrucţiunea decizională IF

În Pascal are forma:

```
if condiţie then
    instrucţiune1
else
    instrucţiune2
```

în care ramura cu *else* poate lipsi. Atenţie că între instrucţiune$_1$ şi cuvântul *else* nu se pune ";", deoarece acest simbol separă două instrucţiuni, or *if-then-else* este o singură instrucţiune.

În C condiția trebuie încadrată de paranteze rotunde şi, de fapt, poate fi orice expresie. Dacă valoarea ei este zero (adică Fals), atunci se execută (dacă există), cea de a doua instrucțiune, în orice alt caz (considerat Adevărat) se execută prima instrucțiune. Cuvântul *then* lipseşte.

```
if (condiție-expresie)
     instrucțiune1
else
     instrucțiune2
```

Evident, dacă instrucțiune$_1$ este o instrucțiune simplă, atunci ea se va termina cu ";", ceea ce ar justifica apariția acestui simbol înainte de *else* în cazul limbajului C.

În FoxPro lipseşte, de asemenea, cuvântul *then*. Forma structurii este:

```
if condiție
   instrucțiuni1
else
   instrucțiuni2
endif
```

Conform sintaxei, putem avea mai multe instrucțiuni, în cele două secțiuni, corespunzătoare celor două ramuri, separate prin *CR* (*Enter*). Am putea spune că forma lui *if* în FoxPro este:

```
if condiție
   instrucțiune1'
   instrucțiune2'
   ....................
else
   instrucțiune1"
   instrucțiune2"
   ....................
endif
```

Cuvântul *endif* termină structura *if*. Dacă apare ramura cu *else*, atunci cuvântul *endif* apare după grupul al doilea de instrucțiuni, iar dacă nu apare ramura cu *else*, cuvântul *endif* apare, fireşte, după primul grup de instrucțiuni ca mai jos:

```
if condiție
   instrucțiune1'
   instrucțiune2'
   ....................
endif
```

Ca şi în Pascal, ramura cu else poate lipsi în C şi în FoxPro.

În Visual Basic există mai multe forme ale acestei instrucțiuni, pe care le prezentăm în continuare:

```
if condiție then instrucțiune1': instrucțiune2': .....

if condiție then instrucțiune1': instrucțiune2': ..... else instrucțiune1": instrucțiune2": .....

sau, ca şi în FoxPro:

if condiție
   instrucțiune1'
   instrucțiune2'
   ....................
else
```

```
   instrucţiune₁"
   instrucţiune₂"
   ...................
endif
```

şi

```
if condiţie
   instrucţiune₁'
   instrucţiune₂'
   ...................
endif
```

Instrucţiunea de selecţie multiplă

În Pascal ea are forma:

```
case expresie of
        caz₁: instrucţiune₁;
        caz₂: instrucţiune₂;
        ............................
        else instrucţiune
end
```

Se evaluează expresia, care trebuie să fie de tip ordinal (Boolean, Char, Integer, subdomeniu sau enumerare, nu şi Real sau String). Dacă ea este egală cu una din expresiile apărute într-unul din cazurile date, atunci se execută instrucţiunea corespunzătoare. Dacă nu, atunci se execută, în caz că există, instrucţiunea de după *else*. Ramura cu *else* este facultativă. Unele variante de Pascal mai vechi foloseau, în cazul instrucţiunii *case*, cuvântul *otherwise* în loc de *else*, dar desemna acelaşi lucru. Prin *caz* înţelegem o listă sau un domeniu de expresii, de acelaşi tip cu expresia în funcţie de care are loc discuţia. Iată un exemplu concret:

```
case zi of
        1..3, 4, 5: WriteLn('Zi lucratoare');
        6: WriteLn('Sambata');
        7: WriteLn('Duminica')
    else WriteLn('Eroare...')
end
```

Instrucţiunea care realizează selecţia multiplă în Basic este Select Case:
Sintaxa ei este:

```
Select Case exp_test
[Case lista_expn
      [instrn]] ...
[Case Else
      [instr]]
End Select
```

exp_test este o parte obligatorie în sintaxă; ea este orice expresie numerică sau şir de caractere;
lista_exp$_n$ este necesară, dacă apare un Case; reprezintă una sau mai multe liste de forma: expresie, expresie To expresie, Is operator_de_comparaţie expresie. Cuvântul cheie To specifică domeniul valorilor. Dacă utilizaţi To, cea mai mică valoare trebuie să apară înainte de To.
Pentru a specifica un domeniu de valori puteţi folosi cuvântul cheie Is împreună cu operatori de comparaţie. Dacă nu este scris, cuvântul Is este automat insertat.

$instr_n$ este o parte opțională, reprezentând una sau mai multe instrucțiuni ce se execută dacă exp_test se potrivește unei părți din $lista\_exp_n$.
instr este una sau mai multe instrucțiuni (opționale) ce se execută dacă exp_test nu se potrivește nici uneia dintre clausele Case.

Observație:

Dacă exp_test se potrivește uneia dintre expresiile dintr-o listă, atunci instrucțiunile ce urmează clausa Case corespunzătoare se execută până la întâlnirea următorului cuvânt Case, sau, pentru ultima clausă, până la End Select. După această execuție, controlul este dat instrucțiunii ce urmează după End Select. Dacă expresia de test (test_exp) se potrivește mai multor expresii din lista lista_exp, doar instrucțiunile ce urmează după prima potrivire se vor executa. Acest lucru este valabil, de altfel, și în limbajul Pascal.

Clausa Case Else este folosită pentru a indica faptul că instrucțiunea instr să se execute dacă nu se gasește nici o potrivire anterioară. Dacă această clausă nu există, iar nici o potrivire nu are loc, atunci nu se va executa nimic.

Exemplu:

Puteți utiliza mai multe expresii sau domenii în fiecare clauză Case.

```
Case 1 To 4, 7 To 9, 11, 13, Is > MaxNumber
```

Un alt exemplu:

```
Case "everything", "nuts" To "soup", TestItem
```

Următorul exemplu determină ce fel de zi a săptămânii este o anumită zi, a cărui număr de ordine se cunoaște:

```
Select Case numar
    Case 1 To 5
        Print "Zi lucratoare"
    Case 6
        Print "Sambata"
    Case 7
        Print "Duminica"
    Case Else
        Print "Nu este un numar corect"
End Select
```

Instrucțiunea C care realizează selecția multiplă este switch. Formatul ei este:

```
switch(exp) {
case lista_expn: instrn ...
[default instr]
}
```

Instrucțiunea switch este similară celor din Pascal și din Visual Basic. Dar, dacă exp se potrivește unei liste de expresii, atunci se execută instrucțiunea corespunzătoare, dar și instrucțiunile următoare, lucru care nu se întâmpla în Pascal, de pildă.

Astfel, o secvență de forma:

```
switch(exp) {
    case e1: sir_instr1;
```

```
    case e2: sir_instr2;
}
```

se va executa aşa:

dacă exp este egală cu e_1, atunci se execută şirul de instrucțiuni sir_instr$_1$ şi apoi sir_instr$_2$ (dacă nu cumva sir_instr$_1$ defineşte el însuşi o altă secvență); dacă exp este egală cu e_2, atunci se execută sir_instr$_2$; dacă exp este diferită atât de e_1, cât şi de e_2, atunci instrucțiunea switch nu este efectivă.

Observație:
Pentru ca să se obțină aceeaşi semnificație ca în limbajul Pascal, fiecare şir de instrucțiuni poate fi terminat cu *break*, care determină întreruperea instrucțiunii switch.

Astfel, o secvență de forma:

```
switch(exp) {
    case e1: sir_instr1; break;
    case e2: sir_instr2; break;
    default: instr3
}
```

se va executa aşa:

dacă exp este egală cu e_1, atunci se execută şirul de instrucțiuni sir_instr$_1$; dacă exp este egală cu e_2, atunci se execută sir_instr$_2$; dacă exp este diferită atât de e_1, cât şi de e_2, atunci se execută instr$_3$.

Propunem cititorului să descopere singur în mediile FoxPro ce instrucțiune se foloseşte pentru selecția multiplă.

Instrucțiunea repetitivă condiționată posterior

În limbajul Pascal, aceasta are forma sintactică următoare:

```
repeat
    instrucţiune1;
    instrucţiune2;
    ...................
    instrucţiunen
until condiţie
```

Semantica este: se execută, în mod repetat, instrucțiunile până când condiția este îndeplinită. Dacă de la bun început condiția era îndeplinită, aceasta înseamnă că instrucțiunile se vor executa, totuşi, măcar o dată.

În limbajul C avem următoarea instrucțiune:

```
do
    instrucţiune
while (condiţie)
```

Instrucțiunea dintre "do" şi "while" poate fi simplă sau compusă (deci cu acolade), iar condiția este o expresie ce poate fi adevărată (nenulă) sau falsă (nulă).

O instrucțiune de forma de mai sus se poate scrie în Pascal astfel:

```
repeat
    instrucţiune
until not condiţie
```

Aşadar, condițiile din cele două forme (Pascal şi C) sunt opuse una alteia.

În Visual Basic avem instrucțiunea:
```
Do
    instrucțiuni
Loop Until condiție
```

sau

```
Do
    instrucțiuni
Loop While condiție
```

Prima formă este echivalentă cu varianta din Pascal, iar cea de a doua cu cea din C.

Instrucțiunea repetitivă FOR

În FoxPro avem instrucțiunea For în două variante:

```
for v = e1 to e2
    instruțiuni
endfor
```

pentru parcurgere crescătoare cu pasul 1 de la expresia e_1 la expresia e_2, respectiv

```
for v = e1 to e2 step p
    instrucțiuni
endfor
```

pentru parcurgere crescătoare (p>0) sau descrescătoare (p<0) de la e_1 la e_2.

De pildă, secvența de program următoare:

```
for i = 10 to 1 step -1
    ?? i
endfor
```

afişează pe un rând numerele de la 10 la 1 (în ordine descrescătoare).

Instrucțiunea repetitivă For din Visual Basic este asemănătoare celei din FoxPro:

```
For v = e1 To e2
    instrucțiuni
Next v    (sau Next)
```

şi:

```
For v = e1 To e2 Step p
    instrucțiuni
Next v    (sau Next)
```

p este pasul de parcurgere care poate fi pozitiv sau negativ, oricum este reprezentat de o expresie întreagă nenulă.

Limbajul Pascal este cam "sărac" în privința instrucțiunii repetitive FOR, pentru că aici pasul poate fi doar 1 sau -1:

```
for v := e1 to e2 do
```

```
      instrucţiune
(pasul este 1)
```

şi:

```
for v := e₁ downto e₂ do
      instrucţiune
(pasul este -1)
```

În orice caz, structura for cu pasul 1 este suficientă pentru a înlocui execuţia unei instrucţiuni FOR cu orice alt pas, pozitiv sau negativ.

Dacă în Pascal FOR-ul este foarte simplu, în C/C++ el este foarte complex. Aceasta deoarece, în comparaţie cu alte limbaje de programare, în C instrucţiunea for poate executa în mod repetat o instrucţiune atât timp cât o condiţie de continuare este îndeplinită, ceea ce face ca instrucţiunea for din C să fie, de fapt, un "while":

```
for (e₁; e₂; e₃)
      instrucţiune
```

În această instrucţiune repetitivă, e_1 este expresie de start (iniţială), e_2 este o expresie ce reprezintă o condiţie de continuare, iar e_3 este o expresie de reiniţializare a ciclului, care, de obicei, este o instrucţiune ce face repetarea.

Instrucţiunea for din limbajul C se execută conform următorilor paşi:

1. se execută secvenţa de iniţializare definită de expresia e_1;
2. se evaluează e_2; dacă e_2 are valoarea zero, atunci se iese din ciclu, adică se trece la instrucţiunea ce urmează după instrucţiunea for; altfel se execută instrucţiunea din corpul ciclului;
3. după executarea ciclului, se execută secvenţa de reiniţializare definită de expresia e_3; apoi se reia secvenţa de la pasul 2.

Dacă e_2 este nulă de la început, instrucţiunea din corpul lui for poate să nu se execute niciodată.

Expresiile din antetul instrucţiunii for pot fi şi vide, dar simbolul ";" trebuie să apară de două ori.
Pascal

Exemple:

Pentru a calcula suma S = 1+2+3+...+n putem scrie:

```
S = 0;
for (i=1; i<=n; i=i+1)
    S=S+i
```

La fel de bine, putem include S=0 alături de iniţializarea lui i în cadrul primei expresii din antetul lui for:

```
for (S=0; i=1; i<=n; i++)
    S+=i
```

Am folosit operatorii speciali "++" şi "+=" din limbajul C;

După cum spuneam, expresiile pot fi şi vide; astfel, suma anterioară poate fi calculată şi aşa:

```
S=0; i=1;
```

for (; i<=n;)
    S+=i++

Instrucțiunea for din C/C++ este cea mai complexă instrucțiune repetitivă.

4.6. Exemple de algoritmi

În acest paragraf vom prezenta câțiva algoritmi care vor rezolva diferite probleme, de la cele mai simple, până la unele mai complicate, din domeniul matematic, dar şi din domeniul financiar-contabil. De asemenea, vor fi prezentați algoritmi clasici de căutare şi sortare (ordonare).

Algoritmi din matematică

1. Numere prime

Vom începe prin a verifica dacă numărul natural n este sau nu prim. Un număr este prim dacă este divizibil doar prin 1 şi el însuşi. 0 şi 1 le vom considera ca nefiind prime.

Pentru a verifica dacă un număr natural n este prim sau nu se testează, mai întâi, dacă el este 0, 1 sau 2. 2 este număr prim (singurul par). Apoi se verifică paritatea sa. Dacă este par, nu mai poate fi prim. În fine, se împarte n la toate numerele d impare până la partea întreagă a rădăcinii pătrate a lui n, notată cu rad. Dacă în intervalul [3,rad] există un divizor d, atunci numărul nu este prim, altfel este prim. Faptul că numărul este prim sau nu se va memora folosind variabila logică prim.

Pentru că deja cunoaştem unele elemente din limbajul Visual Basic, vom descrie algoritmul prezentat în acest limbaj. Înainte, însă de a proiecta programul, trebuie să precizăm că nu cunoaştem încă modul în care se introduc date de la tastatură şi cum se afişează pe ecran în mediul Visual Basic. De aceea, vom folosi două instrucțiuni speciale, Input pentru citire şi Print pentru afişare. Aceste instrucțiuni fac parte din lexicul limbajului Basic (nevizual) şi, cum deocamdată ne interesează partea algoritmică, le vom folosi pentru a descrie algoritmi, urmând ca în capitolele 7-8, o dată cu învățarea programării vizuale să descoperim şi cum se realizează citirile şi scrierile într-un program vizual.

Algoritmul (programul) este:

```
Dim n As Integer, d As Integer, rad As Integer, prim As Boolean
Print "Dati numarul n:"
Input n
If n=0 or n=1 then
   prim = False
Else
   If n=2 Then
      prim = True
   Else
      If n mod 2 = 0 Then
         prim = False
      Else
         prim = True: d = 3: rad = Abs(Sqr(n))
         While d <= rad And prim = True
               If n Mod d = 0 Then
                  prim = False
               Else
```

```
                d = d +2
            End If
        Wend
      End If
   End If
End If
If prim = True then
   Print "Numarul este prim"
Else
   Print "Numarul nu este prim"
```

Grupul de instrucțiuni încadrat funcționează astfel: se pleacă de la premisa că n ar putea fi prim. Până la proba contrarie (reprezentată de n Mod p = 0) variabila prim rămâne cu valoarea adevărată. Trecerea de la un număr impar la altul se realizează prin instrucțiunea d = d+2. Condiția de test a buclei este o condiție compusă, verificându-se două lucruri:

- că nu s-a ajuns la capăt: $d \le n$;
- că încă nu s-a determinat un divizor d al lui n: prim = True.

2. Algoritmul lui Euclid

Vom calcula cel mai mare divizor comun al două numere folosind algoritmul lui Euclid: pentru a obține cel mai mare divizor comun al două numere întregi a şi b, $b \neq 0$, împărțim a cu b; dacă restul împărțirii r_1 este zero, atunci b este cmmdc; dacă nu, împărțim pe b la restul împărțirii anterioare, r_1, şi obținem restul r_2; apoi împărțim pe r_1 la r_2 şi obținem un nou rest r_3 ş.a.m.d.. Ultimul rest nenul este c.m.m.d.c. al celor două numere.

Justificarea algoritmului lui Euclid este dată de proprietatea: cmmdc(a,b)=cmmdc(b,a mod b).

```
Dim a As Integer, b As Integer, deimp As Integer, imp As Integer
Dim cmmdc As Integer, cmmmc As Integer
Print "Dati a: ": Input a: Print "Dati b: ": Input b
deimp = a: imp = b
While imp<>0
   rest = deimp mod imp
   deimp = imp: imp = rest
Wend
cmmdc = deimp
Print "C.m.m.d.c. = ", cmmdc
cmmmc = a*b \ cmmdc
Print "C.m.m.m.c. = ", cmmmc
```

Algoritmul nostru determină şi cel mai mic multiplu comun al celor două numere după formula: cmmmc = a*b \ cmmdc, în care operatorul \ semnifică împărțirea întreagă, adică câtul împărțirii primului operand la cel de al doilea.

O variantă a algoritmului lui Euclid este următoarea, care foloseşte scăderi în loc de împărțiri: cât timp numerele a şi b sunt diferite între ele, scădem numărul mai mic din cel mai mare; la sfârsit, atât a, cât şi b, a devenit cel mai mare divizor comun al numerelor inițiale.

```
Dim a As Integer, b As Integer, a0 As Integer, b0 As Integer
Dim cmmdc As Integer, cmmmc As Integer
Print "Dati a: ": Input a: Print "Dati b: ": Input b
a0 = a: b0 = b
While a0<>b0
```

```
     If a0>b0 then
         a0 = a0 - b0
     Else
         b0 = b0 - a0
     End If
Wend
cmmdc = a0
Print "C.m.m.d.c. = ", cmmdc
cmmmc = a*b \ cmmdc
Print "C.m.m.m.c. = ", cmmmc
```

3. Şirul lui Fibonacci

Şirul lui Fibonacci este un şir de numere întregi a_1, a_2, a_3, ... definit în felul următor: $a_1 = 1$, $a_2 = 1$, $a_n = a_{n-1} + a_{n-2}$, pentru orice $n \geq 3$. O astfel de definiție, în care un anumit termen se construieşte din termeni anterior determinați, se numeşte definiție recurentă (recursivă).

Pentru a determina al n-lea termen al şirului lui Fibonacci, va trebui să determinăm toți termenii până la al n-1-lea inclusiv. Vom folosi trei variabile: t_2, t_1, t, corespunzătoare termenilor a_{n-2}, a_{n-1} şi a_n.

```
Dim t_1 As Integer, t_2 As Integer, t As Integer
Print "Dati n: ": Input n
If  (n=1) Or (n=2) Then
     t = 1
Else
     t_2 = 1: t_1 = 1
     i = 2
     While i<n
          t = t_2 + t_1
          t_2 = t_1
          t_1 = t
          i = i+1
     Wend
End If
Print "Termenul cerut este: ", t
```

4. Media aritmetică a mai multe numere reale

Să presupunem că avem n numere reale a_1, a_2, ..., a_n. Pentru a determina media aritmetică a lor, va trebui să calculăm suma numerelor şi apoi să o împărțim la n. O altă variantă este să calculăm, de la bun început, suma: $a_1/n + a_2/n + ... + a_n/n$, care este identică mediei aritmetice.

Fireşte, pentru a stoca cele n numere vom declara un tablou unidimensional (vector) de numere reale. Vom folosi instrucțiunea For pentru a citi componentele tabloului şi, de asemenea, pentru a determina suma numerelor.

```
Dim n As Integer, i As Integer
Dim A(1 To 30) As Single
Dim suma As Single, media As Single
Print "Dati n: ": Input n
For i = 1 To n
       Print "Dati componenta nr. ", i
       Input A(i)
Next i
suma = 0
For i = 1 To n
```

```
       suma = suma + A(i)
Next i
media = suma / n
Print "Media aritmetica este: ", media
```

De remarcat că se putea determina pe măsura citirii numerelor suma, ba chiar şi media aritmetică a lor. De asemenea, structura de tablou nu era absolut necesară, putându-se folosi o singură variabilă pentru citirea şi utilizarea în calcule a fiecăruia dintre cele n numere. Am apelat, totuşi, la structura de tablou, pentru a stoca cele n numere, în eventualitatea în care ar fi nevoie de ele pentru calcule ulterioare.

Algoritmi diverşi

1. Maximul dintr-un vector

Vom scrie un algoritm structurat care să determine maximul dintr-un vector de n elemente (de acelaşi fel şi comparabile). Determinarea maximului se face în felul următor: se consideră primul element ca fiind maxim. Apoi, parcurgând vectorul, se compară maximul cu fiecare element curent din vector. Dacă elementul curent este mai mare decât maximul considerat până atunci, atunci se consideră maxim acest element găsit mai mare. La sfârşit, avem maximul din tot vectorul. Determinarea minimului se face similar. De asemenea, determinarea maximului sau a minimului dintr-o matrice se face într-un mod similar, parcurgerea facându-se atât pe linii, cât şi pe coloane, deci pe ambele dimensiuni.

```
Dim n As Integer, i As Integer
Dim X(1 To 50) As Single, maxim As Single
Print "Dati numarul de elemente: ": Input n
For i = 1 To n
       Print "Dati elementul de pe pozitia ", i
       Input X(i)
Next i
maxim = X(1)
For i = 2 To n
       If X(i) > maxim Then maxim = X(i)
Next i
Print "Cel mai mare element din vector este: ", maxim
```

2. Inversarea unui vector

Să considerăm un şir de elemente şi dorim să obţinem inversul său, adică în loc de primul element să fie ultimul, în loc de al doilea să fie penultimul ş.a.m.d., până când în locul ultimului element să fie primul.

Pentru a rezolva problema, să considerăm memorat şirul de caractere sub forma unui şir de elemente de lungime n: s_1, s_2, ..., s_n. Deci s_i este al i-lea element din şirul s. Acesta trebuie interschimbat cu elementul de pe poziția n+1-i din acelaşi şir. Deci are loc o interschimbare între s_i şi s_{n+1-i}, interschimbare ce se realizează până la jumătatea şirului, deoarece, dacă interschimbăm după jumătatea şirului, elementele vor reveni pe pozițiile lor inițiale, aşa încât algoritmul nu va avea nici un efect.

```
Dim n As Integer, i As Integer
Dim S(1 To 30) As Single: Dim aux As Single
Print "Dati n: ": Input n
For i = 1 To n
      Print "Dati elementul al ", i, "-lea"
      Input S(i)
Next i
```

```
For i = 1 To n \ 2
       aux = S(i)
       S(i) = S(n+1-i)
       S(n+1-i) = aux
Next i
For i = 1 To n
       Print S(i)
Next i
```

Interschimbarea între cele două elemente S(i) şi S(n+1-i) este realizată cu ajutorul unei variabile suplimentare, aux, de acelaşi tip cu tipul de bază al vectorului S. Aici am considerat tipul real Single, dar poate fi orice tip.

Algoritmi de căutare şi sortare

Foarte importanți în programare sunt algoritmii de căutare şi se ordonare sau sortare, care apar foarte des în programe. Se spune că în jur de o treime din operațiile pe care le execută un calculator sunt reprezentate de căutări şi sortări. În cele ce urmează vom prezenta câțiva algoritmi clasici şi simpli din acest domeniu.

În general, problemele care se pun sunt:

a) Problema de căutare: se dă un vector X cu n elemente de un anumit tip şi un element A de acelaşi tip. Se cere să se determine dacă elementul A este egal cu cel puțin unul din elementele vectorului X.

b) Problema de sortare: se dă un vector X cu n elemente de un anumit tip, comparabile între ele. Se cere să se aranjeze elementele lui X în ordine crescătoare.

a) Algoritmi de căutare

Căutarea secvențială

În cazul în care nu se cunoaşte nimic special despre vectorul X, deci elementele lui X nu au o anumită proprietate, pentru a permite aplicarea unui algoritm de căutare adecvat, se poate folosi o metodă de căutare secvențială. Aceasta constă în parcurgerea element cu element a vectorului, până când s-a terminat de parcurs, sau până s-a găsit un element identic celui căutat. Căutarea poate fi făcută în orice sens, fie de la stânga la dreapta, fie de la dreapta la stânga. O variabilă specială booleană, notată mai jos prin *gasit*, este inițial cu valoarea fals, urmând ca la găsirea lui A în X să se schimbe în adevărat.

Astfel, algoritmul se poate descrie în Basic astfel:

```
Dim n As Integer, i As Integer
Dim X(1 To 30) As Single
Dim A As Single: Dim gasit As Boolean
Print "Dati numarul n: ": Input n
For i = 1 to n
       Print "Dati elementul de pe pozitia ", i, " din vector: "
       Input X(i)
Next i
Print "Dati elementul cautat: ": Input A
gasit = False
While (i<=n) And (not gasit)
       If X(i) = A Then
          gasit = True
       Else
          i = i + 1
```

```
Wend
If gasit Then
    Print "Elementul a fost gasit pe pozitia ", i
Else
    Print "Elementul nu a fost gasit"
End If
```

Căutarea binară

În ipoteza că elementele vectorului sunt deja aşezate în ordine crescătoare, se poate alege ca algoritm de căutare unul mai eficient. Căutarea binară constă în execuția, în mod repetat, a unei căutări în acea jumătate a vectorului X în care elementul A ar putea exista. Astfel, cum elementele sunt în ordine crescătoare, să comparăm mai întâi elementul A cu elementul X(m), unde m este mijlocul vectorului. Fireşte, dacă ele sunt identice, atunci înseamnă că putem opri căutarea şi am avut succes.

Dacă însă ele diferă, atunci înseamnă că A este mai mic decât X(m) sau mai mare. Dacă este mai mic, evident A nu se poate afla decât în prima parte a vectorului, adică înainte de poziția m. Acest lucru se întâmplă tocmai pentru că ştim că elementele sunt puse în ordine crescătoare în cadrul vectorului, aşadar nu are sens să mai căutăm în partea din dreapta a vectorului.

Similar, dacă A ar fi mai mare decât X(m), atunci se va căuta în partea din dreapta.

Cu zona de căutare nouă se procedează la fel, împărțind-o şi pe ea, făcând comparația cu elementul din mijloc, alegând noua zonă de căutare, mai mică şi aşa mai departe.

Căutarea se opreşte în două cazuri: s-a găsit un m cu X(m) = A (succes) sau zona de căutare rămasă este vidă (eşec). Pentru a implementa algoritmul, vom folosi două variabile, notate prin *s* şi *d*, prin care notăm capătul din stânga, respectiv pe cel din dreapta, al zonei de căutare curente. Fireşte, inițial, s=1 şi d=n, unde n este numărul de elemente ale lui X.

În Basic, algoritmul se descrie astfel:

```
Dim X(1 To 30) Of Single
Dim A As Single
Dim n As Integer, i As Integer, m As Integer
Dim s As Integer, d As Integer
Dim gasit As Boolean
Print "Dati numarul n: ": Input n
For i = 1 to n
       Print "Dati elementul de pe pozitia ", i, " din vector: "
       Input X(i)
Next i
Print "Dati elementul cautat: ": Input A
gasit = False: s = 1: d = n
While (s<=d) and (not gasit)
       m = (s + d) \ 2
       If X(m) = A Then
           gasit = True
       Else
           If A < X(m) Then
               d = m - 1
           Else
               s = m + 1
           End If
       End If
Wend
If gasit Then
    Print "Elementul a fost gasit pe pozitia ", m
```

```
Else
    Print "Elementul nu a fost gasit"
End If
```

b) Algoritmi de sortare

După cum s-a arătat, algoritmul de căutare secvenţială se poate aplica doar dacă vectorul X este deja ordonat crescător. Probleme de ordonare crescătoare sau descrescătoare a elementelor unui vector apar foarte des ca subprobleme în probleme mai complexe, din diferite domenii de activitate. De aceea, programatorii şi-au concentrat foarte mult atenţia asupra problemei sortării, astfel încât astăzi sunt cunoscuţi mai mulţi algoritmi de sortare, mai eficienţi sau nu. Cel mai eficient algoritm de sortare, în cazul în care vectorul X nu are vreo proprietate specială, este algoritmul QuickSort. Descrierea acestuia presupune însă cunoştinţe despre recursivitate, care vor fi prezentate ulterior. De aceeam în cele ce urmează ne vom rezuma la a prezenta doi algoritmi de sortare simpli.

Sortarea prin interschimbare (bubble-sort)

Acest algoritm presupune parcugerea în mod repetat a vectorului X şi interschimbarea, la nevoie, a câte două elemente de pe poziţii succesive, până când, la o anumită parcurgere a vectorului, elementele sunt ordonate crescător. Aşadar, dacă vectorul ar fi ordonat, am avea X(1)<=X(2)<=...<=X(n). Dacă există i între 1 şi n-1 astfel încât X(i)>X(i+1), atunci înseamnă că vectorul nu este ordonat. Se interschimbă X(i) cu X(i+1) şi se continuă procedeul. Variabila *ordonat* de tip Boolean va indica dacă elementele din X sunt sau nu ordonate.

```
Dim X(1 To 30) As Single: Dim aux As Single
Dim n As Integer, i As Integer
Dim ordonat As Boolean
Print "Dati numarul de elemente al vectorului: ": Input n
For i = 1 To n
       Print "Dati X(", i, "): "
       Input X(i)
Next i
Do
      ordonat = True
      For i = 1 To n - 1
             If  X(i) > X(i+1) Then
                 aux = X(i)
                 X(i) = X(i+1)
                 X(i+1) = aux
                ordonat = False
             End If
       Next i
Loop Until ordonat
```

Sortarea prin selecţie directă

Metoda constă în determinarea, la fiecare pas i din cei n-1, a celui mai mic element dintre X(i), X(i+1), X(i+2), ... X(n), care va ocupa poziţia i. Aşadar, în fiecare pas i vom compara pe X(i) cu elementele X(j), cu j de la i+1 la n, iar ori de câte ori se găseşte un element X(j) mai mare decât X(i), cele două elemente se interschimbă. Prin urmare, la pasul i are loc o selecţie directă a elementului ce va ocupa poziţia a i-a în vectorul ordonat.

```
Dim X(1 To 30) As Single: Dim aux As Single
Dim n As Integer, i As Integer
Print "Dati numarul de elemente al vectorului: ": Input n
For i = 1 To n
       Print "Dati X(", i, "): "
       Input X(i)
```

```
Next i
For i = 1 To n - 1
    For j = i + 1 To n
             If  X(i) > X(j) Then
                 aux = X(i)
                 X(i) = X(j)
                 X(j) = aux
             End If
    Next j
Next i
```

Algoritmi din contabilitate

1. Soldul unui cont

Să considerăm un cont cu soldul iniţial (debitor sau creditor) cunoscut şi nişte modificări asupra acestui cont. Se cere să se determine soldul final şi tipul său (debitor sau creditor).

Vom stoca în doi vectori cu numele Debit şi Credit valorile iniţiale din cont, precum şi modificările din debit şi credit. Vom declara Debit şi Credit ca avânt componentele numerotate începând cu indicile zero. Una din valorile Debit(0) şi Credit(0) va conţine soldul iniţial, iar cealaltă va fi zero. Modificările, în număr de n_debit, respectiv n_credit, vor fi trecute în celelalte componente din vectori. Se vor calcula sumele Debit(0) + Debit(1) + ... + Debit(n_debit), respectiv Credit(0) + Credit(1) + ... + Credit(n_credit). Cele două sume vor fi comparate şi scăzute una din alta pentru a determina soldul final.

```
Dim Debit(0 To 20) As Integer, Credit(0 To 20) As Integer
Dim n_debit As Integer, n_credit As Integer, i As Integer
Dim suma_debit As Integer, suma_credit As Integer
Dim sold_initial As Integer, sold_final As Integer
Dim tip as String*1
Print "Dati soldul initial: ": Input sold_initial
Print "Dati tipul soldului initial (D/C) ": Input tip
If (tip = "D") Or (tip = "d") Then
    Debit(0) = sold_initial: Credit(0) = 0
Else
    Debit(0) = 0: Credit(0) = sold_initial
End If
Print "Dati numarul de modificari din debit: ": Input n_debit
For i = 1 To n_debit
     Print "Dati suma nr. ", i, " trecuta la debit: "
     Input Debit(i)
Next i
Print "Dati numarul de modificari din credit: ": Input n_credit
For i = 1 To n_credit
     Print "Dati suma nr. ", i, " trecuta la credit: "
     Input Credit(i)
Next i
suma_debit = 0
For i = 1 To n_debit
      suma_debit = suma_debit + Debit(i)
Next i
For i = 1 To n_credit
      suma_credit = suma_credit + Credit(i)
Next i
If suma_debit > suma_credit Then
   sold_final = suma_debit - suma_credit
   Print "Soldul final este debitor: ", sold_final
Else
   sold_final = suma_credit - suma_debit
```

```
   Print "Soldul final este creditor: ", sold_final
End If
```

2. Calculul anuităților

Când trebuie rambursat un credit obligatar sau când trebuie amortizat un activ concesionat, au loc anumite operațiuni de amortizare. Suma plătită anual de către debitor creditorului se numeşte *anuitate*. Anuitatea cuprinde rata de rambursat şi dobânda. În cazul în care se stabileşte ca anuitatea să fie constantă, ea se determină cu relația: $A = C*r/(1-(1+r)^{-n})$, unde A = anuitatea de rambursat, C = capitalul împrumutat (împrumutul de rambursat inițial); r = rata dobânzii (procentuală), n = durata rambursării împrumutului (în ani).

O dată rambursată o rată de împrumut, mărimea acestuia se diminuează şi se modifică structura anuității, în sensul reducerii dobânzii şi creşterii corespunzătoare a ratei de rambursat. Deci, după achitarea fiecărei anuități se determină suma creditului restant (împrumutul de rambursat la valoarea actuală sau soldul), dobânda aferentă şi, prin diferența dintre anuitate şi dobândă, se calculează rata de rambursat.

Dobânda se calculează, anual, cu relația: $D = C*r/100$, în care D reprezintă suma dobânzii anuale, r este rata dobânzii (ca procent), iar C este suma creditului restant (împrumutul de rambursat la începutul anului respectiv).

În continuare este prezentat un program care realizează aceste calcule şi scrie în fişierul PLAN.TXT planul de rambursare eşalonată a împrumutului.

```
Dim Imprumut As Single, RataDobanda As Single, Produs As Single
Dim Anuitate As Single, TotalR As Single, TotalD As Single
Dim Dobanda As Single, RataRestituita As Single
Dim NrAni As Integer, an As Integer
Open "PLAN.TXT" For Output As #1
Print "Dati capital imprumutat: ": Input(Imprumut)
Print #1, "Capital imprumutat = ", Imprumut
Print "Dati numarul de ani: ": Input NrAni
Print #1, NrAni
Print "Dati rata dobanzii (%): ": Input RataDobanda
RataDobanda = RataDobanda / 100
Print #1, "Rata dobanzii = ", RataDobanda
Produs = 1
For an = 1 to NrAni
       Produs = Produs * (1 + RataDobanda)
Next an
Anuitate = Imprumut * RataDobanda / (1 – 1/Produs)
Print #1, "Anuitatea: ", Anuitate
RataRestituita = 0
Print #1, "Plan de rambursare esalonata a imprumutului"
Print #1, "An    Anuit.    Dobanda    Rata r.   Sold ramas"
For an = 1 to NrAni
       Imprumut = Imprumut – RataRestituita
       Dobanda = Imprumut * RataDobanda
       TotalD = TotalD + Dobanda
       RataRestituita = Anuitate – Dobanda
       TotalR = TotalR + RataRestituita
       Print #1, an, Anuitate, Dobanda, RataRestituita, Imprumut
Next an
Print #1, "TOTAL", NrAni*Anuitate, TotalD, TotalR
Close #1
```

4.7. Complexitatea algoritmilor

Estimarea timpului de calcul

Să considerăm problema determinării celui mai mare element dintr-un vector; algoritmul de rezolvare a acestei probleme a fost deja prezentat:

```
Dim n As Integer, i As Integer
Dim X(1 To 50) As Single, maxim As Single
Print "Dati numarul de elemente: ": Input n
For i = 1 To n
       Print "Dati elementul de pe pozitia ", i
       Input X(i)
Next i
maxim = X(1)
For i = 2 To n
       If X(i) > maxim Then maxim = X(i)
Next i
Print "Cel mai mare element din vector este: ", maxim
```

La sfârşit, variabila max conţine cel mai mare element din vectorul X.
Timpul de execuţie a acestui algoritm depinde, în primul rând, de n, care aici este numărul de componente al vectorului X.

Pentru a estima timpul de calcul necesar, ar trebui să inventariem toate instrucţiunile programului şi să ştim de câte ori se execută fiecare dintre ele (în funcţie de n). Mai mult, ar trebui să cunoaştem cât durează execuţia fiecărui tip de instrucţiune .

Ordinul de complexitate

În orice problemă putem observa că există un anumit număr n de date de intrare de care depinde, de obicei, timpul de execuţie al algoritmului care rezolvă acea problemă. De exemplu, dacă se pune problema determinării tuturor permutărilor unei mulţimi, atunci n este numărul de elemente al acelei mulţimi, dacă se pune problema sortării unui vector, n este numărul de elemente al vectorului.

Deoarece nu putem şti întotdeauna cu exactitate de câte ori se execută o anumită instrucţiune (de pildă atribuirea max = X(i)), este destul de greu de determinat timpul total de execuţie. Totuşi, putem considera că există o proporţionalitate între valoarea n şi numărul de execuţii.

În plus, timpul de execuţie al unei instrucţiuni este dependent de calculatorul utilizat.

Majoritatea instrucţiunilor se execută de un număr de ori destul de mic astfel că timpul afectat lor este neglijabil. De aceea, se alege o operaţie (instrucţiune) esenţială, numită *operaţie de bază* şi se determină de câte ori se execută ea. Cerinţa pentru operaţia de bază este ca numărul de execuţii al acesteia să se poată calcula în funcţie de n, de la început.

Timpul de calcul estimat pentru un algoritm oarecare se numeşte *ordinul său de complexitate*. El se notează astfel:

O(număr estimat de execuţii ale operaţiei de bază)

Dacă, de exemplu, un algoritm efectuează $2n^2+4n+3$ operaţii de bază, vom spune că acesta este un algoritm al cărui ordin de complexitate este $O(n^2)$ sau se mai spune că algoritmul are un timp de execuție pătratic. (Pentru n foarte mare, 4n+3 este o valoare neglijabilă comparativ cu $2n^2$, iar $2n^2$ este comparabilă cu n^2.).

Exemple:
a) Pentru aflarea maximului (minimului) dintr-un vector cu n componente, ordinul de complexitate este O(n).
b) Pentru generarea permutărilor (în cazul în care se consideră ca operație de bază generarea unei permutări) ordinul de complexitate este O(n!).

În cazul unor probleme nu putem preciza nici macăr numărul de execuții ale operației de bază. De exemplu, la sortarea prin interschimbare (bubble-sort), dacă vectorul este deja sortat, se execută n-1 comparații (se parcurge vectorul o singură dată). Dacă vectorul este sortat invers, se execută n(n-1)/2 operații de bază (se parcurge vectorul de n ori).

În astfel de cazuri se poate face o discuție asupra a trei valori:
- timpul minim de calcul;
- timpul mediu de calcul;
- timpul maxim de calcul.

De fiecare dată când se vorbeşte despre timpul estimat, se precizează şi despre care este vorba (minim, mediu sau maxim). În practică prezintă interes timpul mediu şi timpul maxim.

Tipuri de ordine de complexitate

Dacă timpul estimat este sub forma O(n), spunem că algoritmul este în timp liniar.

Dacă timpul estimat este sub forma $O(n^k)$, spunem că algoritmul este în timp polinomial (n=2 - pătratic, n=3 - cubic etc.).

Dacă timpul estimat este sub forma $O(2^n)$, $O(3^n)$ şi aşa mai departe, spunem că algoritmul este în timp exponențial.

Un algoritm în timp O(n!) este asimilat unui algoritm în timp exponențial deoarece $n!=1\times2\times\ldots\times n>2\times2\times\ldots\times2=2^{n-1}$.

În practică nu sunt admişi decât algoritmi în timp polinomial, deşi nu este deloc indiferent gradul polinomului.

În concluzie, ori de câte ori avem de rezolvat o problemă, căutăm pentru aceasta un algoritm de timp polinomial. Mai mult, vom căuta un algoritm care să rezolve problema în timp polinomial de grad minim.

Exemple:
a) Pentru problema aflării maximului, operația de bază va fi comparația (fiind dat n, se fac n-1 comparații pentru calculul maximului).
b) În cazul problemei generării permutărilor, este greu de stabilit o anumită operație de bază. Dacă se consideră comparația, va fi dificil de determinat numărul total de comparații necesar generării permutărilor. De aceea, putem considera drept operație de bază generarea unei permutări. Vom avea astfel n! operații de bază (un număr foarte mare).

Generarea permutarilor se poate face pe baza mai multor algoritmi. Alegerea ca operație de bază a comparației permite ca aceştia să fie comparați între ei, dar alegerea ca operație de bază a generării unei permutări nu permite acest lucru, doarece toți algoritmii vor avea de determinat acelaşi număr de permutări: n!.

În astfel de cazuri (când avem de ales între mai multe operații de bază), vom alege acea operație care corespunde cât mai bine scopului propus.

De ce sunt necesari algoritmii eficienți

Ați observat, probabil, că în ultimii ani viteza de calcul a microprocesoarelor devine din ce în ce mai mare. Ne punem problema dacă este necesară găsirea de algoritmi din ce în ce mai eficienți pentru rezolvarea problemelor, sau ar fi mai bine să aşteptăm următoarele generații de calculatoare.

Să presupune că lucrăm cu un calculator capabil să efectueze un milion de operații (elementare) pe secundă. În tabelul următor sunt indicați timpii necesar efectuării a n^2, n^3 , 2^n , 3^n operații cu ajutorul unui astfel de calculator, pentru diferite valori ale lui n.

O	**n=20**	**n=30**	**n=40**	**n=50**	**n=60**
n^2	0,0004 secunde	0,0009 secunde	0,0016 secunde	0,0025 secunde	0,0036 secunde
n^3	0,001 secunde	0,008 secunde	0,0027 secunde	0,125 secunde	0,216 secunde
2^n	o secundă	17,9 minute	12,7 zile	35,7 ani	366 secole
3^n	58 minute	6,5 ani	3855 secole	2×10^8 secole	$1,3\times10^{13}$ secole

Să presupunem că, pentru o anumită problemă, cunoaştem un algoritm al cărui timp de execuție pentru cazuri de mărime n este de 2^n secunde. Din tabel reiese că pentru n=20 algoritmul are nevoie de o secundă, iar pentru n=60 acelaşi algoritm are nevoie de 366 de secole! Dacă vom cumpara un calculator de 100 de ori mai rapid, atunci timpul de rulare pentru n=60 ar fi 366/100 secole = 3,66 secole (deci un timp, de asemenea, neacceptabil).

Dar ce facem dacă avem de rezolvat cazuri pentru un n mult mai mare decat 50? Soluția nu poate fi alta decât găsirea unui alt algoritm mult mai eficient.

Principalul motiv pentru care unii algoritmi pot să ajungă la timpi de lucru practic infiniți, chiar pentru valori relativ mici ale lui n, îşi găseşte explicația nu în lipsa de performanțe a calculatoarelor, ci în faptul că funcția exponențială $f(n)=a^n$, cu a>1, creşte extraordinar de repede.

Conform celor de mai sus, este foarte indicat ca pentru o problemă dată să elaborăm algoritmi care să nu fie exponențiali. Sunt considerați "buni" algoritmii pentru care numărul operațiilor este polinomial (adică se poate exprima sub forma unui polinom de gradul n, unde n este numărul datelor de intrare). Nu este posibil sa evităm totdeauna algoritmii exponențiali; un exemplu îl constituie problema generării tuturor permutărilor de n elemente sau a submulțimilor unei mulțimi cu n elemente, când numărul rezultatelor este n!, respectiv 2^n şi deci numărul de operații va fi inerent exponențial.

Concret si abstract

Cazuri particulare si generale. De la concret la abstract si invers. Constante si variabile

- Concret si abstract

Un programator trebuie sa gandeasca abstract. Acest lucru este atat de important in programare, incat orice incercare de a invata sa programezi, fara a avea o gandire abstracta, este sortita esecului. Cine nu poate intelege abstractiunile matematice din scoala generala sau liceu, de pilda, nu poate sa ajunga sa programeze (bine). Ca programatori, va trebui sa realizati diferite programe pentru oameni foarte diferiti ca nivel cultural, pregatire. Veti intra in contact cu economisti, ingineri, psihologi, sau oameni de litere, medici sau avocati, care va vor solicita sa le faceti un program pe calculator care sa le rezolve anumite probleme din domeniul lor de activitate. Probabil cu exceptia inginerilor, veti constata ca majoritatea au un anumit mod de a se exprima si de a va prezenta problema de rezolvat incompatibil, intr-o oarecare masura, cu modul dumneavoastra de a vorbi si de a intelege problema. Cel mai bun sfat ar fi acela de a-i lasa sa va explice tot ce vor, fara sa-i intrerupeti, dupa care sa incercati sa "preluati dumneavoastra carma" si, prin intrebari simple, la care interlocutorul sa va raspunda doar prin da sau nu, sa intelegeti esenta problemei pe care trebuie sa o rezolvati.

In general, beneficiarii programului dumneavoastra, vor fi foarte concreti. Ei nu vor prezenta in linii mari, generale, lucrarea pe care vor sa o informatizeze, generale, ci vor da tot soiul de exemple, care nu au nici o relevanta pentru problema, din perspectiva dumneavoastra. Va trebui sa identificati, in explicatiile interlocutorului sau in raspunsurile acestuia, urmatoarele elemente:
- ce se da si ce se cere programului, pentru ca orice program/algoritm prelucreaza anumite date de intrare pentru a obtine niste informatii, drept rezultate;
- cum se vor da datele de intrare si in ce ordine, ce conditionari exista intre ele, pentru a putea proiecta interfata cu utilizatorul, pentru introducerea datelor;
- ce informatii se asteapta de la program si in ce forma, in ce ordine, pentru a sti cum sa proiectati interfata cu utilizatorul, pentru extragerea rezultatelor;
- care sunt formulele de calcul care se folosesc, in ce ordine si ce conditionari exista intre ele.

In privinta interfetei cu utilizatorul, fiti convinsi ca beneficiarul se va razgandi de mai multe ori, mai ales atunci cand programul capata o forma apropiata de cea finala, de aceea nu trebuie sa acordati prea multa atentie acestui aspect, pentru inceput. Concentrati-va asupra formulelor de calcul si e posibil ca aici sa aveti multe dificultati de a le obtine din cauza ca ele nu va vor fi prezentate, pur si simplu! Cei mai multi prefera sa va dea exemple si dumneavoastra sa deduceti singur formulele de calcul, decat sa va spuna care este formula din teorie. Deci, va trebui sa gasiti generalul din cazurile lor particulare si sa abstractizati tot ceea ce va prezinta ei concret.

- Constante si variabile

Sa revenim la problema desenarii pe ecranul calculatorului. Sa presupunem ca dispunem de un mediu de programare, in care, pentru a desena un cerc de raza r, cu centrul cercului in punctul de coordonate x,y trebuie sa folosim instructiunea CIRCLE(x,y,r). De obicei, instructiunile sunt prezentate folosind variabile (precum x, y si r) si nu constante (numere ca 100, 150, 215, 342). Pentru a desena un cerc avand centrul in punctul de coordonate 200, 300, si cu raza de 10 de unitati vom scrie, asadar, CIRCLE(200,300,10), pastrand ordinea celor trei parametri ai instructiunii. La fel, putem particulariza folosirea lui CIRCLE, si pentru cercul de coordonate 200, 300 si de raza 20: CIRCLE(200,300,20). Evident, cele doua cercuri sunt concentrice, pentru ca au aceleasi coordonate pentru centru.

Pentru a desena 15 asemenea cercuri concentrice, de raze de 10, 20, 30 etc., ar trebui sa folosim 15 instructiuni CIRCLE, in care cel de-al treilea parametru sa fie schimbat, pe rand, in 10, 20 s.a.m.d.. Acest mod de rezolvare a problemei desenarii celor 15 cercuri concentrice denota o gandire pur concreta, care se bazeaza pe utilizarea a 15 cazuri particulare de desenare a unor cercuri. Un programator bun nu va proceda asa, el va cauta sa gaseasca o regula pentru desenarea mai usoara a celor 15 cercuri, eventual pomenind o singura data de comanda CIRCLE. Astfel, el va incerca sa inlocuiasca constantele numerice 10, 20, 30, ..., 150, cu o singura data, care sa varieze intre 10 si 150, din 10 in 10. O asemenea data se numeste variabila. Ea isi va schimba valoarea, in functie de necesitati. Astfel, notand cu R acea variabila, cele cincisprezece cazuri concrete vor ajunge cazul abstract CIRCLE(200,300,R), unde R variaza intre 10 si 150, cu pasul 10. Limbajele de programare ofera diferite posibilitati de a-l face pe R sa ia pe rand valorile 10, 20 etc., dar putem sa ne gandim mai departe la o alta variabila I, care sa varieze intre 1 si 15, si sa scriem CIRCLE(200,300,10*I), unde I variaza intre 1 si 15, cu pasul 1. Putem continua, considerand un caz si mai general, deci mai abstract, in care pasul sa nu fie 10, ci un numar oarecare, reprezentat de variabila P. Atunci vom scrie CIRCLE(200,300,P*I), considerandul-l pe I intre 1 si 15. Dar s-ar putea ca sa avem nevoie sa desenam nu doar 15 cercuri, ci 20 sau 50, adica un numar N oarecare. Si poate acestea vor avea centrul intr-un punct de coordonate X, Y, oarecare, iar razele sa inceapa sa creasca de la valoarea T. Astfel, cel mai abstract caz este: CIRCLE(X,Y,T+P*I), unde I ia valori, din 1 in 1, intre 0 si N-1. Astfel, X, Y, T, P si N sunt date de intrare in problema, I este o variabila de lucru, iar rezultatul ar fi cele N cercuri desenate pe ecran.

Procesul de abstractizare este foarte complex si este greu de explicat ce mecanisme intelectuale si psihice intra in joc, atunci cand abstractizam. Trebuie sa dovedim multa imaginatie si sa incercam sa ne gandim si la alte situatii decat cele concrete cu care avem de a face la un moment dat. Pentru a abstractiza cat mai mult o problema si rezolvarea ei, va trebui sa ne punem intrebari de genul "ce-ar fi daca nu as cunoaste aceasta valoare?" sau "ce-ar fi daca as schimba aceasta valoare cu alta?" si sa incercam sa raspundem la asemenea intrebari, rescriind algoritmul.

Capitolul 5. Subprograme

5.1. Definirea subprogramelor

Să considerăm următoarea problemă. Se citesc două numere întregi m şi n. Se cere să se determine numărul m!+n!. Pentru aceasta, va trebui să calculăm cele două produse: m!=1×2×...×m, respectiv n!=1×2×...×n. În limbajul Pascal vom scrie:

```
program SumaFactoriale;
var m,n,mf,nf,i: LongInt;
begin
    Write('Dati m: '); ReadLn(m);
    Write('Dati n: '); ReadLn(n);
    mf:=1; for i:=1 to m do mf:=mf*i;
    nf:=1; for i:=1 to n do nf:=nf*i;
    WriteLn('m!+n!=',mf+nf)
end.
```

Observăm că există două secvenţe de instrucţiuni asemănătoare (scrise aplecat), aşadar, pentru simplificarea calculului în asemenea situaţii, ar fi de preferat să dispunem de o funcţie care să ne dea factorialul unui număr întreg oarecare x. Să presupunem că am dispune de o asemenea funcţie, cu numele *fact*. Atuncim, cele două secvenţe de instrucţiuni evidenţiate în program ar fi înlocuite de:

```
mf:=fact(m); nf:=fact(n);
```

Definirea de funcţii este posibilă în toate limbajele de programare, inclusiv în Pascal.

Funcţiile grupează o serie de instrucţiuni pentru a calcula o anumită valoare, în funcţie de argumentele funcţiei. Există şi cazuri când funcţiile nu au argumente.

De asemenea, majoritatea limbajelor de programare pun la dispoziţia programatorilor şi alte modalităţi de a grupa mai multe instrucţiuni sub un nume, iar aceste instrucţiuni să execute anumite operaţii în funcţie de anumite argumente, eventual să comunice cu exteriorul prin alte argumente. De obicei se numesc *subprograme* sau *rutine* sau *subrutine* sau *proceduri*.

Astfel, în Pascal avem conceptul de procedură şi conceptul de funcţie. Procedurile şi funcţiile poartă denumirea de **subprograme**. Ele se declară şi se definesc în partea declarativă a programului, deci înaintea instrucţiunii compuse din care este constituit programul propriu-zis.

O procedură se defineşte în Pascal astfel:

```
procedure nume_procedura(lista_de_parametri_formali);
declaraţii locale;
begin
    instrucţiuni
end;
```

În lista parametrilor formali sunt precizate argumentele funcţiei, împreună cu tipurile lor, unele argumente putând fi precedate de cuvântul var, ceea ce înseamnă că sunt parametri variabili.

În lista de declarații locale pot fi declarate constante, variabile, tipuri de date, chiar şi altesubprograme.

Exemplu:

Următoarea procedură determină factorialul unui număr n:

```
procedure Factorial(n: LongInt; var f: LongInt);
var i: LongInt;
begin
   f:=1; for i:=1 to n do f:=f*i
end;
```

Similar, o funcție care să calculeze acelaşi factorial se va scrie:

```
function Fact(n: LongInt): LongInt;
var i,f: LongInt;
begin
   f:=1; for i:=1 to n do f:=f*i;
   Fact:=f
end;
```

În program, procedura se va apela (chema, folosi) prin: Factorial(n,nf) şi Factorial(m,mf), iar funcția prin nf:=Fact(n); mf:=Fact(m).

Motivul pentru care apare cuvântul *var* în fața parametrului *f* din procedura *Factorial* va fi explicat ulterior, în paragraful următor.

În Basic avem funcții, iar procedurile se numesc subrutine. Factorialul se va calcula prin subrutina:

```
Sub factorial(ByVal n As Long, ByRef f As Long)
    Dim i As Long
    f = 1
    For i = 1 To n
        f = f * i
    Next i
End Sub
```

Apelul se va realiza prin: factorial m, mf, în care m este o valoare, iar mf o variabilă.
O funcție care să calculeze factorialul lui n este:

```
Function fact(n As Long) As Long
    Dim i As Long, f As Long
    f = 1
    For i = 1 To n
        f = f * i
    Next i
    fact = f
End Function
```

Putem folosi această funcție astfel: mf = fact(m).

În limbajul C nu există proceduri sau subrutine, ci doar funcții. Totuşi, există funcții de tip vid (void), care nu returnează valori, astfel încât aceste funcții se comportă asemenea unor proceduri din limbajul Pascal.

5.2. Circuitul datelor între subprograme

Programul principal, dar şi subprogramele se mai numesc şi **blocuri** şi, conform unor autori, **module**. Totuşi, există limbaje precum Modula-2 care definesc altfel noţiunea de modul, aşa cum vom vedea şi noi mai târziu.

Subprogramele pot comunica între ele, adică să se apeleze unele pe altele. Unele limbaje de programare (precum Pascal sau C) au drept restricţie faptul că dacă un subprogram P apelează un subrogram Q, atunci trebuie ca Q să fie scris înaintea lui P. În unele limbaje de programare (Pascal, de exemplu), blocurile pot fi cuprinse în alte blocuri.

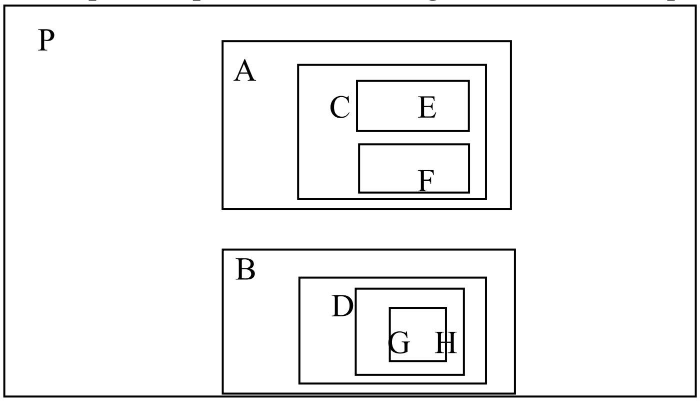
Există trei reguli de valabilitate sau vizibilitate a identificatorilor:

- În cadrul unui bloc, un identificator poate fi declarat o singură dată.

Un acelaşi identificator poate fi declarat în blocuri diferite. În acest caz se aplică regula:

- Natura unui identificator *id* care apare într-o instrucţiune se stabileşte căutând cel mai interior bloc ce include atât instrucţiunea, cât şi declararea lui *id*.
- Nu se poate folosi un identificator în exteriorul blocului în care a fost declarat.

Pentru a exemplifica aplicarea acestor reguli fie schema de program modularizat:



Putem vorbi de o ierarhizare a blocurilor pe nivele:

Bloc	Nivel
P	0
A,B	1
C,D	2
E,F,G	3
H	4

Domeniile de vizibilitate (valabilitate) ale identificatorilor sunt:

Identificatorii declaraţi în blocul:	sunt accesibili în blocul:
P	P, A, B, C, D, E, F, G, H
A	A, C, E, F
B	B, D, G, H
C	C, E, F
D	D, G, H
E	E
F	F
G	G, H
H	H

De exemplu, o variabilă declarată în blocurile B şi G şi apelată în blocul H va avea tipul precizat în declaraţia conţinută de blocul G.

Paradigma de programare care se bazează pe blocuri şi pe instrucţiuni de control ca cele menţionate în capitolul precedent se numeşte **programare procedurală**.

Să revenim la apelul unui bloc Q din cadrul altui bloc P. Primul se numeşte *apelat*, iar al doilea *apelant*. Dacă Q are lista parametrilor formali f_1, f_2, ..., f_n, atunci apelul lui Q se realizează prin Q(e_1, e_2, ..., e_n) (în Basic nu se pun paranteze), în care e_1, ..., e_n sunt aşa numiţii *parametri efectivi*, care sunt fie valorile unor expresii, fie variabile (depinde de modul de transmitere a parametrilor). Între parametrii efectivi şi cei formali se realizează o corespondenţă, în ordine. Astfel, lui f_i îi corespunde e_i.

Pentru a înţelege cum se transmit parametri, menţionăm că există mai multe modalităţi de transmitere a lor, dar cel mai adesea, în limbajele de programare procedurale, vom întâlni *tramsmiterea prin valoare* şi *transmiterea prin referinţă*. În limbajul Pascal, dacă un parametru formal din antetul unui subprogram este precedat de cuvântul *var*, atunci transmiterea se face prin referinţă, altfel prin valoare. La Basic se folosesc cuvintele rezervate ByRef, respectiv ByVal. Limbajul C nu dispune de modalitatea de transmitere a parametrilor prin referinţă, dar aceasta poate fi simulată cu ajutorul transmiterii prin valoare, în care parametrii sunt nişte pointeri, adică adresele de memorie unde sunt stocate variabilele respective.

Când transmiterea parametrilor este prin valoare, înseamnă că parametrii efectivi din blocul apelant sunt nişte expresii. Acestea se evaluează şi se transmit blocului apelat. În blocul apelat, parametrii formali primesc valori de la parametrii efectivi, *adică parametrii formali iau valorile respectivelor expresii*. În interiorul procedurii este posibil ca parametrii formali să-şi modifice valorile, dar aceste modificări nu influenţează în nici un fel valorile expresiilor care formau parametrii efectivi. Aşadar, comunicarea se face de la blocul apelant la cel apelat, nu şi invers.

În transmiterea prin referinţă, parametrii efectivi sunt variabile, din cadrul blocului apelant. În blocul apelat, parametrii formali vor primi, iniţial, valori de la parametrii efectivi, apoi orice modificare asupra parametrilor formali, în cadrul blocului apelat, se va transmite şi asupra variabilelor care reprezintă parametrii formali. Aşadar, comunicarea are loc în ambele sensuri, între blocul apelant şi cel apelat. Aceasta deoarece *parametrii formali se suprapun parametrilor efectivi*, adică, parametrii efectivi vor fi alte nume pentru parametrii formali.

Exemple:

Pentru a înţelege ce este transmiterea parametrilor prin valoare, ca şi transmiterea prin referinţă, vom considera cazul limbajului Pascal.

Următoarea procedură calculează suma a două numere întregi a şi b:

```
program Suma(a,b: Integer; var c: Integer);
begin
    c := a + b
end;
```

Această procedură este blocul apelat. a şi b sunt parametri transmişi prin valoare, deci pot fi orice două expresii, pe când c este parametru transmis prin referinţă, deci va fi doar o variabilă.

În blocul apelant (o altă procedură sau programul principal), să presupunem că avem următoarele declaraţii de variabile:

```
var x,y,z,a,b,c: Integer;
```

Vom putea avea apeluri de genul:

- Suma(2,3,x) - care determină ca x să primească valoarea 2+3=5;
- Suma(2,3,a) - care determină ca a să primească valoarea 2+3=5, unde variabila a este cea din blocul apelant, neavând nici o legătură cu parametrul formal a de la blocul apelat;
- Suma(2,x,y) - care evaluează pe x (să zicem că x este 5), apoi pasează pe 2 lui a, pe 5 (x) lui b, în parametrii formali ai procedurii, unde se calculează valoarea parametrului c, a cărui valoare (7) este transmisă lui y.
- Suma(2+x,3,y), Suma(2+x,3+y,z) sunt apeluri corecte ale procedurii Suma
- Suma(a,b+c,300), Suma(2,3,x+y), Suma(2+x,3,x+2) sau Suma(2+x,3+y,x+4) sunt apeluri greşite, pentru că nici 300, nici x+y, nici x+2, nici x+4 nu sunt variabile.

Să considerăm acum următoarea procedură:

```
procedure Schimba(x,y: Integer);
var aux: Integer;
begin
   aux:=x; x:=y; y:=aux
end;
```

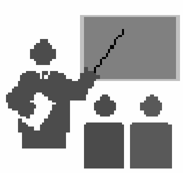
Deşi un apel de forma Schimbă(a,b) este corect, valorile variabilelor x şi y nu se schimbă, deoarece transmiterea se face prin valoare.

Pentru ca într-adevăr valorile celor două variabile să se schimbe, va trebui ca x şi y să fie parametri transmişi prin referinţă:

```
procedure Schimba(var x,y: Integer);
var aux: Integer;
begin
   aux:=x; x:=y; y:=aux
end;
```

Capitolul 6. Metoda backtracking

6.1. Prezentare generală



Există multe probleme a căror soluție se poate reprezenta sub forma unui vector $x=(x_1, ..., x_n) \in S$, în care $S=S_1 \times S_2 \times ... \times S_n$, unde $S_1,...,S_n$ sunt mulțimi finite, cu $|S_i|=s_i$ elemente.

De pildă, să considerăm problema așezării a n dame pe o tablă de șah cu $n \times n$ pătrate. În acest caz, firește vom avea exact o damă pe fiecare coloană. Deci dama k va sta pe coloana k și pe una din cele n linii, pe linia notată x[k]. Așadar, o soluție a acestei probleme poate fi reprezentată doar prin vectorul liniilor pe care sunt așezate cele n dame, coloanele subînțelegându-se.

Pentru fiecare astfel de problemă sunt date anumite relații între componentele $x_1, ..., x_n$ ale vectorului x, numite *condiții interne*. S se numește *spațiul soluțiilor posibile*.

Soluțiile posibile, care satisfac condițiile interne se numesc *soluții rezultat*.

În cazul problemei damelor, se cere ca damele să nu se atace. Acest lucru constituie condițiile intrne. Așezându-le câte una pe fiecare coloană, ele nu se vor ataca vertical. Pentru a nu se ataca orizontal, trebuie ca să nu existe două dame pe aceeași linie (deci două dame i și j cu x[i]=x[j]). De asemenea, trebuie ca damele să nu se atace diagonal.

Astfel de probleme pot fi soluționate folosind **metoda "*backtracking*"** (în caz că nu se cunoaște o metodă mai bună).

Metoda "*Back-tracking*" evită generarea tuturor soluțiilor posibile. Elementele vectorului x primesc, pe rând, valori. Astfel, lui x_k i se atribuie o valoare numai dacă au fost deja atribuite valori lui $x_1, ..., x_{k-1}$. Mai mult, odată o valoare pentru x_k stabilită, nu se trece direct la atribuirea de valori lui x_{k+1}, ci se verifică niște *condiții de continuare* referitoare la $x_1, ..., x_k$. Aceste condiții stabilesc situațiile în care are sens să trecem la determinarea lui x_{k+1}. Neîndeplinirea lor exprimă faptul că, oricum am alege $x_{k+1}, ..., x_n$, nu vom putea ajunge la o soluție rezultat, adică la o soluție pentru care condițiile interne să fie satisfăcute. Dacă condițiile de continuare nu sunt îndeplinite, se va alege alt x_k din S_k, iar dacă S_k s-a epuizat ne întoarcem la S_{k-1}.

În cazul problemei damelor, dacă condițiile interne sunt reprezentate de neatacarea oricăror două dame între ele, condițiile de continuare sunt reprezentate de neatacarea fiecărei dame cu cele dinaintea sa.

Putem vedea vectorul x ca pe o stivă asupra căreia se fac o serie de operații de intrare și ieșire, atât timp cât stiva nu este vidă. Prin urmare, metoda va admite și o variantă recursivă.

- Presupunând stabilite condițiile de continuare (sub forma unei funcții depinzând de valorile stabilite până acum) $\varphi_k(x_1, ..., x_k)$ avem:

```
type vector = array[1..max] of  TIP; var α: TIP;
procedure BackTracking(n: Integer; var x: vector);
var  k: Integer; cont: Boolean;
begin
  k := 1;
  while k > 0 do begin
    cont := False;
```

```
    while (mai există o valoare α∈S_k netestată)
     and (not cont) do
      begin
        x_k := α;
     if φ_k(x_1, ..., x_k) then cont := True
      end;
    if not cont then k := k - 1
    else if k = n then Scrie(x) else k := k + 1
  end
end;
```

- În general, însă, x_k ia valori numere întregi între `a` şi `b`. Astfel, putem rescrie procedura `BackTracking` anterioară sub forma de mai jos, în care valorile sunt date lui `x[k]` în felul următor: mai întâi fiecare `x[k]` primeşte valoarea `a-1`, apoi, la fiecare încercare de a da o nouă valoare, se face incrementarea `x[k]:=x[k]+1`.

```
type vector = array[1..max] of TIP;
procedure BackTracking(n: Integer;
                       var x: vector);
var k: Integer; cont: Boolean;
begin
  k := 1; x[k]:=a-1;
  while k > 0 do
       begin
         cont := False;
         while (x[k] < b) and (not cont) do
                begin
                  x[k]:=x[k]+1
                  if PotContinua(x,k) then
                     cont := True
                end;
          if not cont then
             k := k - 1
          else
             if k = n then Scrie(x)
             else
             begin k:=k+1; x[k]:=a-1 end
        end
end;
```

Printre exemplele celebre de probleme în a căror rezolvare se poate folosi această metodă de programare, amintim: *problema celor opt regine; colorarea vârfurilor unui graf (a unei hărți); generarea permutărilor; problema circuitului hamiltonian (a comisului voiajor)*.

Între condițiile interne şi cele de continuare există o strânsă legătură, alegerea cât mai bună a condițiilor de continuare reducând calculele.

Rezolvitorului îi revine problema soluționării, astfel, a două probleme importante legate de aplicarea metodei "*back-tracking*": memorarea soluției şi scrierea funcției de continuare.

În atenția profesorului

Pe parcursul acestui capitol, se vor urmări: a) formarea la studenți a deprinderilor de a identifica problemele a căror rezolvare optimă necesită folosirea metodei backtracking; b) deprinderea studenților cu determinarea pentru o anumită problemă a variantei optime de back-tracking; c) deprinderea studenților cu alegerea judicioasă a parametrilor formali.

? Întrebări şi exerciții

1 ☺ La ce fel de probleme se poate folosi metoda Back-tracking?
2 ☺ Ce se înțelege prin soluție posibilă? Dar prin soluție rezultat?
3 ☺ Prin ce se caracterizează spațiul soluțiilor ?
4 ☺ Ce se înțelege prin condiții interne? Dar prin condiții de continuare?
5 ☹ Descrieți algoritmul general de Back-tracking.
6 ☹ Descrieți algoritmul de Back-tracking pentru cazul când fiecare element al vectorului soluție ia valori între 1 şi `n`.
7 💣 Ce se întâmplă dacă funcția de continuare returnează întotdeauna adevărat? Rescrieți algoritmul de Back-tracking pentru acest caz (ambele variante).
8 💣 Care sunt condițiile interne în cazul problemei damelor?
9 ☺ Dați exemple de alte probleme ce pot fi rezolvate prin această tehnică de programare.

6.2. Exemple şi aplicații

6.2.1. Problema celor opt dame

Problema constă în a aşeza 8 (sau, mai general, `n`) dame (regine) pe o tablă de şah $n \times n$, astfel încât damele să nu se atace. O soluție pentru $n=5$ este dată în figura următoare.

		3		
				5
	2			
			4	
1				

Vom aşeza, pe rând, câte o damă pe fiecare coloană `k`, dama de pe coloana `k` fiind aşezată pe linia `x[k]`. Astfel, damele nu se vor ataca pe verticală. Mai rămâne ca ele să nu se atace pe orizontală şi pe diagonală. Este de ajuns ca dama `k` să nu se atace, astfel, cu nici una din damele dinaintea sa, deci cu damele `i`, cu $i=1,...,k-1$. Condițiile de continuare sunt, deci:

$x[i] \neq x[k], \forall i=1,...,k-1$ (neatac pe linii (orizontală)) şi

$|x[i]-x[k]| \neq k-i, \forall i=1,...,k-1$ (neatac pe diagonală).

Acestea fiind spuse, putem scrie funcția de continuare astfel:

```
function PotContinua(x: vector; k: integer): Boolean;
var atac: Boolean; i: Integer;
begin
  atac := false; i := 1;
  while (i < k) and (not atac) do
        if (x[i] = x[k]) or (abs(x[i]-x[k]) = k-i)
           then
             atac := True
        else i := i+1;
  PotContinua := not atac
end;
```

Pe baza acestei funcții și a algoritmului de Back-traking se poate scrie programul de mai jos.

☞ **Observație**

Programul are o serie de proceduri şi funcţii, care comunică între ele prin parametri formali. `NrSol` reprezintă numărul de soluţii. Se va observa că următoarele probleme ce vor fi prezentate în acest capitol vor fi scrise fie tot în această manieră, fie se vor folosi variabile globale pentru comunicarea între diferitele blocuri ale programului respectiv.

Subprogramele pe care un astfel de program le foloseşte sunt:

- **`Scrie`** - care afişează soluţia, bazându-se pe componentele vectorului x şi pe semnificaţia acestuia;
- **`PotContinua`** - funcţia de testare a îndeplinirii condiţiilor de continuare; această funcţie va diferi de la problemă la problemă;
- **`Dame`** - procedura de generare prin Back-tracking, deci principalul bloc al programului.

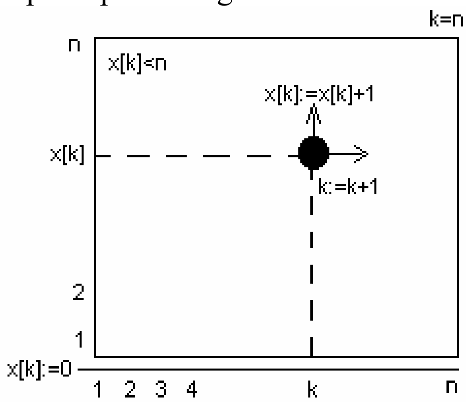
```
program ProblemaDamelor;
uses Crt;
const max=10;
type vector = array[1..max] of Integer;
var NrSol: Integer;

procedure Scrie(n: Integer; x: vector);
var i: Integer;
begin
  Inc(NrSol);
  WriteLn('Solutia nr. ',NrSol);
  for i := 1 to n do
    WriteLn('Dama de pe coloana ',i,' e pe linia ',x[i]);
  WriteLn; ReadLn
end;

function PotContinua(x: vector; k: integer): Boolean;
{ ..... }


procedure Dame(n: Integer; var x: vector);
var k: Integer; cont: Boolean;
begin
  k := 1; x[k] := 0;
  while k > 0 do
    begin
      cont := False;
      while (x[k] < n) and (not cont) do
         begin
            x[k] := x[k] + 1;
            if PotContinua(x,k) then cont := True
         end;
      if not cont then k := k - 1
      else
        if k = n then Scrie(n,x)
        else begin k := k + 1; x[k] := 0 end
   end
end;

var AsezareDame: vector; NrDame: integer;
begin
 ClrScr; WriteLn(' Problema damelor '); WriteLn;
 WriteLn; NrSol:=0;
```

```
Write('Dati nr. de dame: '); ReadLn(NrDame);
Dame(NrDame,AsezareDame)
end.
```

Să explicăm ce înseamnă diferitele atribuiri sau condiții care apar în programul anterior:

`k:=1`	se pleacă cu prima damă (ce se pune pe coloana `1`)
`x[k]:=0`	ea se pune în afara tablei, sub prima linie
`k>0`	mai sunt de aşezat dame, de încercat variante (o atribuire de genul `k:=k-1` nu ne trimite în afara tablei
`x[k]<n`	dama `k` mai poate fi deplasată cu o linie mai sus
`not cont`	dama `k` nu e bine aşezată pe coloana `k` şi linia `x[k]`, deoarece funcția `PotContinua` ne spune că dama `k` se atacă cu una din damele `1`, `2`, ..., `k-1`.
`x[k]:=x[k]+1`	dama k (de pe coloana k) se deplasează cu o linie (x[k]) mai încolo
`k:=k-1`	se revine la dama anterioară
`k=n`	s-a ajuns la ultima damă
`k:=k+1`	se trece la următoarea damă
`x[k]:=0`	noua damă se aşază în afara tablei, sub prima linie (pe linia 0)

Cele explicate mai sus se pot reprezenta grafic astfel:



? Întrebări şi exerciții

1 ☺ Care este problema damelor? Cum se poate reprezenta o soluție rezultat?
2 😐 Scrieți condițiile interne pentru problema damelor, în reprezentarea soluției rezultat conform exercițiului 1?
3 ☹ Scrieți condițiile de continuare şi funcția aferentă pentru problema damelor.
4 ☹ Ce se întâmplă dacă se elimină din funcția de continuare a problemei damelor condiția de atac pe diagonală? Dar dacă se elimină doar cea de atac pe linie?

6.2.2. Generarea funcțiilor injective

Să se afişeze toate funcțiile injective `f: A→B`, unde `A` şi `B` sunt două mulțimi cu `m` respectiv `n` elemente. Se va afişa tabelul de variație al funcției.

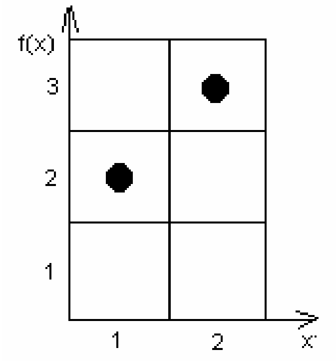
Vom lăsa în seama cititorului să rezolve această problemă pentru cazul general al mulțimilor `A` şi `B`, noi vom considera că `A` cuprinde elementele de la `1` la `m`, iar `B` pe cele de la `1` la `n`.

Astfel, o soluție rezultat va fi dată de un vector `x`, cu semnificația `x[i]=f(i)`, adică valoarea funcției `f` în punctul `i` din `A`. Astfel, `x[i]`$\in$`B`, deci va lua valori între `1` şi `n` (vezi varianta a doua de Back-tracking, din paragraful anterior).

Exemplu, pentru A={1,2} şi B={1,2,3} se vor obține vectorii: (1,2), (1,3), (2,1), (2,3), (3,1) şi (3,2). Vectorul (2,3) corespunde, de pildă, tabelului de variație:

x	1	2
f(x)	2	3

O reprezentare "pe tabla de şah" a acestei funcții ar arăta astfel:



Condițiile interne ale vectorului `x` sunt date de injectivitatea funcției `f`, deci va trebui ca `x[i]`$\neq$`x[j]`, $\forall$`i`$\neq$`j` între `1` şi `m`. Aceste condiții determină ca funcția de continuare din algoritm să verifice ca fiecare `x[k]` să fie distinct de orice `x[i]`, cu `1`$\leq$`i<k`.

```
program GenerareaFunctiilorInjective;
uses Crt;
type vector = array[1..10] of integer;
var NrSol: Integer;

procedure Scrie(m: integer; x: vector);
var i: Integer;
begin
     Inc(NrSol);
     WriteLn('Solutia nr. ',NrSol);
     Write('  x │');
     for i := 1 to m do Write(i:3);
     WriteLn; Write('----│');
     for i:=1 to m do Write('---');
     WriteLn; Write('f(x)│');
     for i := 1 to m do Write(x[i]:3);
     WriteLn; ReadLn
end;

function PotContinua(x: vector; k: integer): Boolean;
var atac: Boolean; i: Integer;
begin
    atac := false; i := 1;
    while (i < k) and (not atac) do
          if x[i] = x[k] then atac := True
          else i := i+1;
    PotContinua := not atac
end;

procedure FunctiiInjective(m, n: Integer; var x: vector);
var k: Integer; cont: Boolean;
begin
     k := 1; x[k] := 0;
```

```
     while k > 0 do
           begin
             cont := False;
             while (x[k] < n) and (not cont) do
                   begin
                     x[k] := x[k] + 1;
                     if PotContinua(x,k) then cont := True
                   end;
             if not cont then k := k - 1
             else
                 if k = m then Scrie(m,x)
                 else
                     begin
                       k := k + 1; x[k] := 0
                     end
           end
end;

var x: vector;
    m,n: integer;
begin
     ClrScr;
     WriteLn(' Generarea functiilor injective ');
     WriteLn(' ****************************** ');
     WriteLn; NrSol:=0;
     Write('Dati cardinalul multimii A: '); ReadLn(m);
     Write('Dati cardinalul multimii B: '); ReadLn(n);
     FunctiiInjective(m,n,x)
end.
```

☞ **Observație**

Dacă m=n atunci se obține generarea funcțiilor bijective. Deci singura restricție ce există între elementele componente ale vectorului x este ca acestea să fie distince. Practic am obținut soluționarea şi a problemei generării permutărilor de n elemente, precum şi a aşezării unor ture pe tabla de şah cu n×n pătrățele, care să nu se atace între ele.

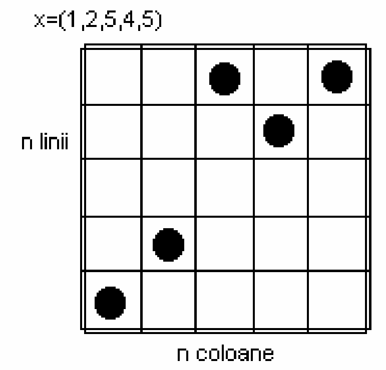
? Întrebări şi exerciții

1 ☺ Enunțați problema generării funcțiilor injective şi descrieți modul de reprezentare a unei soluții rezultat.
2 😐 Scrieți programe pentru generarea permutărilor de n elemente ale unei mulțimi, precum şi pentru aşezarea a n ture pe o tablă de şah cu n×n căsuțe.
3 ☹ Aşezați pe o tablă de şah cu n coloane şi m linii n ture care să nu se atace între ele. Ce problemă este echivalentă cu aceasta?

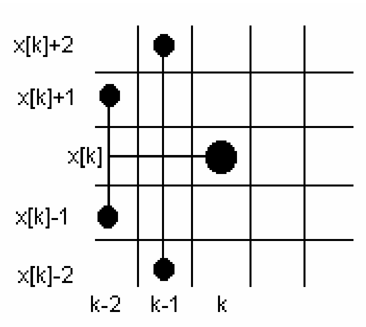
6.2.3. Aşezarea cailor

Să considerăm, acum, că se cere să se aşeze n cai pe o tablă de şah cu n×n pătrățele, care să nu se atace între ei. Problema este destul de complexă, dar dacă se pune condiția ca aceşti cai să fie aşezați câte unul pe fiecare coloană, problema are o rezolvare prin tehnica Back-tracking, asemănătoare cu problema celor 8 regine.

Iată o soluție pentru n=5:



Calul de pe coloana `k` se poate ataca cu caii `k-1` sau `k-2`, după cum se vede din figura de mai jos:



Aşadar, condiţiile de continuare sunt acestea:
• pentru `k=1` nu avem nici o restricţie, deoarece primul cal poate fi pus pe orice linie;
• pentru `k=2` trebuie avut în vedere ca acest al doilea cal să nu se atace cu primul, deci ca diferenţa dintre linia sa şi linia calului `k-1=1` să nu fie `2`;
• pentru `k=3,4` etc, trebuie să avem în vedere că acest cal nou aşezat se poate ataca fie cu calul `k-2` (la o diferenţă a liniilor lor de o unitate), fie cu calcul `k-1` (la o diferenţă de două unităţi a liniilor lor).

Aşadar, putem scrie funcţia de continuare ca mai jos:

```
function PotContinua(x: vector; k: Integer): Boolean;
begin
     if k=1 then PotContinua:=true
     else if k=2 then
             PotContinua:=Abs(x[k]-x[k-1])<>2
          else
             PotContinua:=(Abs(x[k]-x[k-1])<>2)
                 and Abs(x[k]-x[k-2])<>1)
end;
```

Pe baza acestei funcţii, programul de la problema damelor se poate rescrie, astfel încât să se obţină rezolvarea problemei aşezării cailor.

? Întrebări şi exerciţii

1 ☹ Scrieţi programul care să aşeze pe o tablă de şah cu n×n căsuţe n cai, fiecare cal pe o altă coloană, caii neatacându-se între ei.

2 💣 Pe o tablă de şah cu n coloane şi m linii se cere să se aşeze n piese, fiecare pe câte o coloană, care să nu se atace între ele. Piesele atacă precum o tură, un nebun şi un cal simultan.

3 ☹ Realizaţi implementări grafice pentru problema reginelor şi a cailor.

6.2.4. Generarea partițiilor unui număr natural

Să se afişeze toate partițiile unui număr natural nenul n. Printr-o partiție a lui n se înțelege o descompunere a lui n ca sumă de numere naturale nenule.

De data aceasta, vom reprezenta o soluție printr-un vector x de lungime maximă n, având lungimea efectiv folosită k. De acest lucru va ține cont şi procedura Scrie.

Fireşte, condițiile interne ale tabloului x sunt ca suma celor k componente să fie n, deci acest vector reprezintă, de fapt, descompunerea lui n ca sumă de k numere.

De exemplu, n=4 se poate scrie astfel:

```
4 = 1+1+1+1 (k=4)
4 = 1+1+2 (k=3)
4 = 1+3 (k=2)
4 = 2+2 (k=2)
4 = 4 (k=1)
```

☞ Observație

Evident, fiecare x[k] va lua valori între 1 şi n, asta fiindcă nu ne interesează şi cazurile cu descompuneri când apare şi 0 ca termen. Aşadar, bazându-ne pe algoritmul Back-tracking, inițializarea lui x[k] se va face cu 0, pentru orice k. Valoarea maximă pe care o poate lua x[k] poate fi considerată n-s[k-1], unde s[k-1] este suma componentelor deja alese din tabloul x, deci s[k-1]=x[1]+x[2]+...+x[k-1]. Pentru ca relația să fie valabilă şi pentru k=1, s-a considerat vectorul s cu indici de la 0, iar s[0]=0.

Această limitare determină şi realizarea funcției PotContinua, deci condițiile de continuare în Back-tracking. Practic, am făcut o mică optimizare a algoritmului pentru acest caz.

O altă chestiune ce trebuie semnalată este inițializarea lui x[k] cu valoarea x[k-1]-1. Acest lucru va determina ca elementele partiției să fie ordonate crescător, deci se vor evita repetițiile.

De asemenea, trecerea la procedura de afişare se face când s-a ajuns la suma dată, deci s[k]=n. Se vor afişa doar primele k componente din vectorul x.

Exemplu: pentru n=4 şi vectorul x=(1,1,...), în care k=3, avem:
x[k-1]≤x[k]≤n-s[k-1], deci 1≤x[3]≤4-s[2]=4-2=2.

- ♦ Pentru x[3]=1 se va continua cu 1≤x[4]≤4-x[4]=1, care conduce la soluția: 4=1+1+1+1 (cu k=4).
- ♦ Pentru x[3]=2, se va obține soluția 4=1+1+2 (cu k=3).

```
program GenerarePartitiiNumar_Varianta;
uses Crt;
type vector = array[0..10] of integer;
var NrSol: Integer;

procedure Scrie(n,k: integer; x: vector);
var i: Integer;
begin
     Inc(NrSol);
     WriteLn('Solutia nr. ',NrSol);
     Write(n,'=');
     for i := 1 to k-1 do
         Write(x[i],'+');
     WriteLn(x[k]);
     ReadLn
```

```
end;

function PotContinua(x: vector; k,n: integer; var s: vector): Boolean;
var i: Integer;
begin
     s[k]:=s[k-1]+x[k]; PotContinua:=s[k]<=n
end;

procedure PartitiiNumar(n: Integer; var x: vector);
var k: Integer; s: vector;
    cont: Boolean;
begin
     k := 1; x[k] := 0; s[0]:=0;
     while k > 0 do
           begin
             cont := False;
             while (x[k] < n-s[k-1]) and (not cont) do
                   begin
                     x[k] := x[k] + 1;
                     if PotContinua(x,k,n,s) then
                        cont := True
                   end;
             if not cont then
                k := k - 1
             else
                 if s[k] = n then
                    Scrie(n,k,x)
                 else
                    begin
                      k := k + 1;
                      x[k] := x[k-1] - 1
                    end
           end
end;

var x: vector;
    n: integer;

begin
     ClrScr;
     WriteLn(' Generarea partitiilor unui numar intreg ');
     WriteLn(' ************************************** ');
     WriteLn; NrSol:=0;
     Write('Dati numarul n='); ReadLn(n);
     PartitiiNumar(n,x)
end.
```

? Întrebări şi exerciţii

1 😐 Ce noutăţi apar în acest program?
2 ☹ De ce am făcut mărginirea valorilor lui `x[k]`?
3 ☹ Cum am putea economisi memorie, pentru a nu mai păstra suma celorlalte elemente dinaintea lui `x[k]`?
3 💣 Scrieţi o variantă a algoritmului, care să genereze

6.2.5. Plata unei sume cu bancnote de valori date

O generalizare a problemei descrise anterior este următoarea.

Să se afişeze toate modalităţile de a plăti o sumă `n` cu bancnote de valori $b_1, b_2, ..., b_m$. Se presupune că există un număr suficient de bancnote de fiecare fel.

Să scriem programul pe baza celui dinainte, în care o soluție va fi sub forma unui vector `x`, cu `x[k]` = numărul de bancnote de tipul `b[k]` care se vor lua.

Astfel, suma `n` se partiționează în mai multe sume, de forma x[k]×b[k], lucru care apare evidențiat în algoritmul de mai jos.

Atenție

Deoarece numărul de tipuri de bancnote este limitat la `m`, va trebui ca să se țină cont de acest lucru atunci când se trece la atribuirea unei noi valori pentru fiecare `x[k]`.

Asemănător programului din 2.2.4, avem:

$$s[k]=x[1]\times b[1]+x[2]\times b[2]+...+x[k]\times b[k].$$

Limitarea lui `x[k]` se va face astfel:

`(x[k]*b[k]<n-s[k-1])`

```
{$B-}
program PlataUneiSumeCuBancnote;
uses Crt;
const suma_max=100;
type vector = array[0..suma_max] of integer;
var NrSol: Integer;

procedure Scrie(k: integer; b,x: vector);
var i: Integer;
begin
     Inc(NrSol);
     WriteLn('Solutia nr. ',NrSol);
     for i:=1 to k do
         WriteLn('Se iau ',x[i],' bancnote de ',b[i]);
     ReadLn
end;

function PotContinua(x: vector; k,n,m: integer;
                     b: vector; var s: vector): Boolean;
begin
     s[k] := s[k-1] + x[k]*b[k];
     PotContinua := s[k]<=n
end;

procedure PlataBancnote(n,m: Integer; b: vector; var x: vector);
var k: Integer; s: vector;
    cont: Boolean;
begin
     k := 1; x[k] := -1; s[0]:=0;
     while k > 0 do
           begin
             cont := False;
             while (k<=m) and (not cont)
                  and (x[k]*b[k]<n-s[k-1]) do
                   begin
                     x[k] := x[k] + 1;
                     if PotContinua(x,k,n,m,b,s) then
                        cont := True
                   end;
             if not cont then k := k - 1
             else
                 if s[k] = n then Scrie(k,b,x)
                 else
                    begin
                      k := k + 1; x[k] := -1
                    end
           end
```

```
end;

var x,b: vector; i,n,m: integer;
begin
     ClrScr;
     WriteLn(' Plata unei sume prin bancnote date ');
     WriteLn(' ********************************* ');
     WriteLn; NrSol:=0;
     Write('Dati suma n='); ReadLn(n);
     Write('Dati numarul de bancnote: '); ReadLn(m);
     for i:=1 to m do
         begin
           Write('Dati valoarea bancnotei ',i,': ');
           ReadLn(b[i])
         end;
     PlataBancnote(n,m,b,x)
end.
```

☞ Observație

Dacă vectorul `b` este `(1,1,...,1)` și `m=n` atunci se obține problema partițiilor unui număr.

Iată un exemplu de executare a acestui program:

```
 Plata unei sume prin bancnote date
 *********************************
Dati suma n=15
Dati numarul de bancnote: 4
Dati valoarea bancnotei 1: 2
Dati valoarea bancnotei 2: 3
Dati valoarea bancnotei 3: 5
Dati valoarea bancnotei 4: 7
Solutia nr. 1
Se iau 1 bancnote de 2
Se iau 1 bancnote de 3
Se iau 2 bancnote de 5
Solutia nr. 2
Se iau 2 bancnote de 2
Se iau 2 bancnote de 3
Se iau 1 bancnote de 5
Solutia nr. 3
Se iau 3 bancnote de 2
Se iau 3 bancnote de 3
Solutia nr. 4
Se iau 6 bancnote de 2
Se iau 1 bancnote de 3
```

? Întrebări și exerciții

1. ☺ Rescrieți programul anterior, astfel încât să se folosească variabile globale pentru comunicarea între diferite părți ale acestuia.
2. 💣 De ce este nevoie de condiția ca `k≤m`?
3. ☺ Ce se întâmplă dacă elementele lui `b` sunt inițial ordonate descrescător?

6.2.6. Generarea produsului cartezian a mai multor mulţimi

Să considerăm că avem `n` mulţimi cu `m[1]`, `m[2]`, ..., respectiv `m[n]` elemente. Se cere să se afişeze elementele produsului cartezian a acestor mulţimi.

Un exemplu concret ar fi următorul: `n=3`; prima mulţime ar putea cuprinde `m[1]=3` tipuri de ciorbe, a doua muţime ar avea `m[2]=5` tipuri de felul doi, iar a treia `m[3]=4` tipuri de deserturi din catalogul de meniuri al unui restaurant. Ne punem problema determinării tuturor meniurilor ce pot fi servite la acel restaurant.

☞ **Observaţie**

Remarcăm că a genera produsul cartezian al celor n mulţimi se poate face pornind de la generarea produsului cartezian al mulţimilor `{1,2,...,m[1]}`, `{1,2,...,m[2]}` ş.a.m.d..

Trecerea de la mulţimile mai sus enumerate la mulţimile iniţiale se poate face prin simple bijecţii între cele două tipuri de mulţimi.

Să notăm cu `x=(x[1], x[2], ..., x[n])` un element oarecare al produsului cartezian al mulţimilor `{1,2,...,m[1]}`, `{1,2,...,m[2]}` ş.a.m.d.. Se observă că fiecare element nu depinde în nici un fel de celelalte, deci funcţia de continuare va fi adevărată întotdeauna. De fapt, chiar şi condiţiile interne nu există!

Problema este că fiecare element `x[k]` este cuprins între `1` şi numărul total al elementelor din mulţimea a `k`-a, deci `m[k]`. De acest lucru se va ţine cont când se va alege următoarea valoare pentru `x[k]`.

Exemplu: fie cele trei mulţimi: A_1 = {1,2} (m_1=2), A_2 = {1} (m_2=1), A_3={1,2,3} (m_3=3). Se obţin vectorii:

```
(1,1,1)
(1,1,2)
(1,1,3)
(2,1,1)
(2,1,2)
(2,1,3)
```

În total $m_1 \times m_2 \times m_3 = 6$ elemente.

O primă variantă a programului de rezolvare este dată mai jos. El foloseşte doar variabile globale.

```
program ProdusCartezian;
uses crt;
type vector=array [1..20] of Integer;
var i,k,n: Integer; m,x:vector; cont: Boolean;

procedure Scrie;
begin
     for i:=1 to n do
         Write(x[i],',');
     WriteLn; ReadLn
end;

begin
     Write('Dati nr de multimi: '); Readln(n);
     for i:=1 to n do
         begin
           Write('Dati nr de elemente al multimii ',i,': ');
           ReadLn(m[i])
         end;
     k:=1;
     x[k]:=0;
```

```
     while k>0 do
           begin
             cont:=false;
             while (x[k]<m[k]) and (not cont) do
                   begin
                     x[k]:=x[k]+1;
                     cont:=True
                   end;
             if cont=true then
                if k=n then
                   Scrie
                else
                   begin
                     k:=k+1;
                     x[k]:=0;
                   end
             else
                k:=k-1
           end;
     ReadLn
end.
```

☞ **Observație**

Se observă că ciclul *while* evidențiat prin caractere italice, dacă se execută atunci se execută doar o singură data, iar la ieşirea din el, valoarea variabile de control `cont` este `True`. Ceea ce ne permite să transformăm ciclul într-un simplu test cu instrucțiunea if, obținând programul de mai jos:

```
program ProdusCartezian_VariantaSimplificata;
uses crt;
type vector=array [1..20] of Integer;
var i,k,n: Integer; m,x:vector;
    cont: Boolean;

procedure Scrie;
begin
     for i:=1 to n do
         Write(x[i],',');
     WriteLn; ReadLn
end;

begin
     Write('Dati nr de multimi: '); Readln(n);
     for i:=1 to n do
         begin
          Write('Dati nr. de elemente al multimii ',i,': ');
          ReadLn(m[i])
         end;
     k:=1; x[k]:=0;
     while k>0 do
           begin
             if x[k]<m[k] then
                begin
                  x[k]:=x[k]+1;
                  if k=n then
                     Scrie
                  else
                     begin
                       k:=k+1;
                       x[k]:=0;
                     end
```

```
               end
            else
               k:=k-1
          end;
     ReadLn
end.
```

O altă rezolvare a acestei probleme, ca şi a următoarei vor fi învățate la disciplina *Bazele informaticii* (la *Elemente de combinatorică*).

? Întrebări şi exerciții

1 ☺ De ce este de ajuns a rezolva problema produsului cartezian al mai multor submulțimi de numere naturale, de forma `{1,2...,m[k]}`?
2. ☹ Pornind de la programul prezentat, scrieți un program care să rezolve problema cazului general al unor mulțimi de cuvinte
3. ☹ Explicați de ce s-a transformat ciclul *while* într-o decizie *if* în cadrul variantei a doua a programului prezentat.

6.2.7. Generarea submulțimilor unei multimi

Să considerăm că avem o mulțime `A` cu `n` elemente oarecare şi ne punem problema generării tuturor submulțimilor sale. Pentru aceasta, vom considera următorul mod de reprezentare a unei submulțimi a mulțimii inițiale: se va folosi un vector `x=(x[1],` ..., `x[n])`, cu elemente doar `1` şi `2`.

Semnificația acestui vector este: `x[k]=1`, dacă elementul `A[k]` aparține submulțimii curente generate, respectiv `x[k]=2`, dacă nu.

De acest lucru va ține cont procedura de afişare a soluțiilor rezultat `Scrie`.

Să observăm că pentru a genera vectorii aceia cu elemente doar de `1` şi `2` este necesar să considerăm produsul cartezian al `n` mulțimi de forma `{1,2}`. Prin urmare, vom rescrie programul din secțiunea 2.2.6 pentru a obține rezolvarea problemei submulțimilor.

Exemplu: A={a,b,c}. Soluții:

vector	submulțime
`(1,1,1)`	`{a,b,c}`
`(1,1,2)`	`{a,b}`
`(1,2,1)`	`{a,c}`
`(1,2,2)`	`{a}`
`(2,1,1)`	`{b,c}`
`(2,2,1)`	`{c}`
`(2,2,2)`	`{}` (mulțimea vidă)

În total 2^3 mulțimi.

```
program GenerareSubmultimi;
uses crt;
const max=10;
var i,k,n: Integer;
    x: array[1..max] of Integer;
    a: array[1..max] of String;
    cont: Boolean;
```

```
procedure Scrie;
begin
     WriteLn('O solutie este:');
     Write('{ ');
     for i:=1 to n do
         if x[i]=1 then
            Write(a[i],' ');
     WriteLn('}');
     ReadLn
end;

begin
     WriteLn('Generarea submultimilor unei multimi');
     WriteLn('***********************************');
     Write('Dati nr de elemente: '); Readln(n);
     for i:=1 to n do
         begin
           Write('Dati elementul al ',i,
                  '-lea al multimii: ');
           ReadLn(a[i])
         end;
     k:=1; x[k]:=0;
     while k>0 do
           begin
             cont:=false;
             while (x[k]<2) and (not cont) do
                   begin
                     x[k]:=x[k]+1;
                     cont:=True
                   end;
             if cont=true then
                if k=n then Scrie
                else
                   begin
                     k:=k+1; x[k]:=0;
                   end
             else
                k:=k-1
           end
end.
```

? Întrebări şi exerciţii

1 😐 Transformaţi ciclul while evidenţiat cu caractere italice în cadrul programului anterior într-o instrucţiune if, pentru a obţine o variantă simplificată a programului.
2 😐 Rescrieţi programul generării submulţimilor, astfel încât să se folosească vectori cu elemente `0` şi `1` în loc de `1` şi `2`.
3 ☺ De ce nu există condiţii de continuare la problema generării submulţimilor?

6.2.8. Generarea combinărilor

Într-un liceu există doar doi profesori de informatică care trebuie să-şi împartă cele `n` clase care studiază matematica. Ştiind că primul profesor va trebui să aleagă `m` (`m`≤`n`) din cele `n` clase pentru a preda matematica la ele, să se afişeze toate combinaţiile posibile pentru ambii profesori.

Practic, dacă primul profesor îşi alege `m` din cele `n` clase, cel de al doilea le va lua pe celelalte `n-m`, deoarece nu se poate ca doi profesori să predea la aceeaşi clasă. De asemenea, să observăm că nu contează ordinea în care primul profesor alege clasele, de aceea problema se reduce la a genera combinările de `n` elemente luate câte `m`.

Vom folosi vectorul `x` cu următoarea semnificație: `x[k]`=care este cea de a `k`-a clasă pe care a ales-o profesorul 1. Evident, vectorul `x` va avea m elemente, deoarece profesorul 1 poate alege cel mult m clase pentru predare.

Cum ordinea elementelor din vectorul `x` nu contează, înseamnă că acestea pot fi considerate în ordine crescătoare, astfel evitându-se şi repetițiile inutile. Prin urmare, fiecare `x[k]` va lua valori începând cu `x[k-1]+1`, de aceea inițializarea sa se va face cu `x[k-1]`.

Atenție

Acest lucru determină şi o altă modificare în cadrul algoritmului clasic. Ultima valoare pe care o va lua elementul al `k`-lea va fi `n-m+k`. Dacă nu ar fi aşa, atunci, trecându-se de această limită atât pentru `x[k]`, cât şi pentru celelalte elemente de după el, s-ar ajunge ca x[m] să depăşească chiar pe `n`.

Să considerăm un exemplu: `n=5`, `m=3`. Observăm că se pot genera următoarele combinări ca valori pentru vectorul `x`: `(1,2,3)`, `(1,2,4)`, `(1,2,5)`, `(1,3,4)`, `(1,3,5)`, `(2,3,4)`, `(2,3,5)`, `(2,4,5)`, `(3,4,5)`. Se observă de aici că fiecare `x[k]` poate ajunge cel mult la valoarea `5-3+k`.

Programul de mai jos rezolvă problema împărțirii claselor între cei doi profesori:

```
program Combinari;
uses crt;
var x:array[1..10] of integer;
    i,m,k,n:integer;

procedure Scrie;
var i,j:integer;g:boolean;
begin
     WriteLn;
     Write('Profesorul 1 are clasele: ');
     for i:=1 to m do
         Write(x[i],',');
     WriteLn;
     Write('Profesorul 2 are clasele: ');
     for i:= 1 to n do
         begin
           g:=false;
           for j:=1 to m do
               if x[j] = i then g:=true;
               if not g then Write(i,',');
         end;
     WriteLn;
     if ReadKey=#27 then Halt
end;

begin
     ClrScr;
     Write('Dati numarul de clase total n='); ReadLn(n);
     Write('Cate clase are primul profesor? m='); ReadLn(m);
     k:=1; x[k]:=0;
     while k>0 do
           begin
             if x[k] < n-m+k then
                begin
                  x[k]:=x[k]+1 ;
                  if k=m then Scrie
                  else
                     begin
                       k:=k+1;
```

```
                    x[k]:=x[k-1]
                  end
             end
          else
             k:=k-1
        end
end.
```

☞ **Observație**

Procedura `Scrie` s-a modificat substanțial, pentru a depista ce clase va rămâne să ia profesorul 2, după ce a luat `m` clase profesorul 1.

În legătură cu această problemă, trebuie să știți că se va reveni asupra ei şi la disciplina *Bazele informaticii*, unde se va prezenta o metodă de rezolvare asemănătoare.

? Întrebări şi exerciții

1 ☺ Unde şi de ce a dispărut funcția de continuare din programul generării combinărilor.
2 ☹ Rescrieți programul anterior pentru a da nume celor n clase şi a afişa soluția prin numele lor.

6.2.9. Problema discretă a rucsacului

Ne vom referi în continuare la una dintre variantele unei probleme clasice de programare: cu ajutorul unui rucsac de greutate maximă admisibilă `GG` se cer a fi tranportate nişte obiecte, din `n` disponibile, astfel încât încărcătura rucsacului să fie cât mai profitabilă cu putință. Cele `n` obiecte sunt caracterizate prin greutățile lor (memorate într-un vector `G`), precum şi de câştigurile (profiturile) pe care acestea le aduc (memorate într-un vector `C`).

Dacă se consideră că obiectele pot fi secționate (de exemplu sunt fructe sau legume), atunci avem de a face cu *problema continuă a rucsacului*, variantă ce va fi studiată mai târziu, în capitolul referitor la tehnica Greedy (dacă nu ați studiat-o deja în clasa a IX-a, la disciplina *Algoritmi şi limbaje de programare*).

Dacă nu este permisă secționarea obiectelor, se spune că avem de a face cu *problema discretă* sau *problema 0/1 a rucsacului*.

Exemplu: `n=5` obiecte, cu câştigurile `C=(2,3,3,4,3)` şi greutățile `G=(4,2,5,3,4)`. Fie greutatea admisibilă maximă a rucsacului `GG=12`. Se obține soluția formată din obiectele `2`, `3` şi `4` care, deşi nu umplu perfect rucsacul, aduc câştigul maxim de `10`.

Problema se poate soluționa şi pe alte căi mai eficiente, însă vom prezenta rezolvarea sa prin metoda Back-tracking, din considerente de ordin didactic.

Astfel, problema în cauză se reduce la a genera toți vectorii $x=(x_1,x_2,...,x_n)$, cu $x_i\in\{0,1\}$. Fiecare vector reprezintă o modalitate de a umple rucsacul ($x_i=1 \Leftrightarrow$ obiectul `i` s-ar lua în rucsac), deci va avea un câştig curent asociat (o utilitate) (`CC`) şi o greutate curentă (`Greut`), care nu trebuie să depăşească valoarea maximă admisibilă: `GG`.

🔔 **Atenție**

Dintre toți aceşti vectori - care ar putea fi generați şi pe alte căi, de exemplu ca elementele unui produs cartezian, sau prin transformări în baza 2 a primelor `2n-1` numere naturale - se va alege vectorul care are utilitatea maximă printre toți vectorii.

Acest vector - cel mai bun - se va copia într-un vector de acelaşi fel, numit în program `Iau`, cu convenția că `Iau[i]=1` dacă obiectul `i` se pune în rucsac, respectiv `Iau[i]=0` dacă nu.

Iată, aşadar, o modificare a algoritmului de Back-tracking pentru a obține o soluție optimă.

Condițiile de continuare în cazul de față sunt exprimate de faptul că greutatea curentă `Greut` nu o depăşeşte pe cea maximă admisibilă, GG.

```
program Rucsac01;
const max=10;
var CMax,CC,GG:Integer; C,G,X,Iau: array[1..max] of Integer;
      { CMax,CC = cistig maxim, curent,
        GG = greutatea maxima posibila }
    n,k,i: Integer; cont: Boolean;
function PotContinua(k: Integer): Boolean;
{ pot continua daca nu se depaseste GG }
var i: 1..max; Greut: Integer;
begin
  Greut:=0;
  for i:=1 to k do if X[i]=1 then Greut:=Greut+G[i];
  PotContinua:=Greut<=GG
end;
procedure BackTrack;
begin
  k:=1; X[k]:=-1; CMax:=0;
  while k>0 do
    begin
      cont:=False;
      while (X[k]<1) and (not cont) do
        begin X[k]:=X[k]+1; cont:=PotContinua(k) end;
      if cont then
         if k=n then
            begin
              CC:=0;
              for i:=1 to n do if X[i]=1 then CC:=CC+C[i];
              if CC>=CMax then
                 begin
                  CMax:=CC; for i:=1 to n do Iau[i]:=X[i]
                 end
            end
         else begin k:=k+1; X[k]:=-1 end
      else k:=k-1
    end
end;
begin
  Write('n='); ReadLn(n);
  for i:=1 to n do begin
        Write('C[',i,']='); ReadLn(C[i]);
        Write('G[',i,']='); ReadLn(G[i])
      end;
  Write('GG='); ReadLn(GG); BackTrack;
  WriteLn('O sol. cu cistig maxim:');
  for i:=1 to n do
      if  Iau[i]=1 then  WriteLn('Se ia obiectul ',i);
  WriteLn('Câstig = ',Cmax); ReadLn
end.
```

? Întrebări şi exerciții

1 ☺ De ce a fost necesară şi utilizarea vectorului Iau, în plus față de alți algoritmi?
2 😐 Scrieți funcția de continuare în cazul problemei discrete a rucsacului.
3 💣 Generează algoritmul Greedy varianta optimă în cazul problemei discrete a rucsacului ?(Vezi *Algoritmi şi limbaje de programare*, manual pentru clasa a IX-a,).

6.2.10. Generarea funcțiilor surjective

Dacă generarea funcțiilor injective nu ridica problme deosebite, generarea funcțiilor surjective definite de la o mulțime `A` (de exemplu `{1,2,...,m}`) la o mulțime `B` (să zicem `B={1,2,...,n}`) prezintă următoarele particularități:

- fireşte, trebuie ca m≤n, altfel nu putem avea funcții surjective;
- trebuie, în plus, ca să ne asigurăm, la final, că mulțimea `B` coincide cu imaginea mulțimii `A` prin funcția respectivă.

Atenție

Acest ultim lucru este evidențiat în cadrul funcției de continuare.

Semnificația lui `x[i]∈B` este valoarea funcției curente în punctul `i∈A`.

```
program GenerareaFunctiilorSurjective;
uses Crt;
type vector = array[1..10] of Integer;
var NrSol: Integer;
    x: vector;
    n,m,k,i: Integer;



procedure Scrie;
begin
     Inc(NrSol);
     WriteLn('Solutia nr. ',NrSol);
     Write('  x |');
     for i := 1 to m do
         Write(i:3);
     WriteLn;
     Write('----|');
     for i:=1 to m do
         Write('---');
     WriteLn;
     Write('f(x)|');
     for i := 1 to m do
         Write(x[i]:3);
     WriteLn;
     if ReadKey=#27 then Halt
end;


function PotContinua: Boolean;
var codomeniu: set of Byte;
begin
     if k<m then PotContinua:=True
     else
         begin
           codomeniu:=[];
           for i:=1 to m do
               codomeniu:=codomeniu+[x[i]];
           PotContinua:=codomeniu=[1..n]
         end;
end;

procedure FunctiiSurjective;
var cont: Boolean;
begin
     k := 1; x[k] := 0;
```

```
     while k > 0 do
           begin
             cont := False;
             while (x[k] < n) and (not cont) do
                   begin
                     x[k] := x[k] + 1;
                     if PotContinua then cont := True
                   end;
             if not cont then k := k - 1
             else
                 if k = m then
                    Scrie
                 else
                     begin
                       k := k + 1;
                       x[k] := 0
                     end
           end
end;
begin
     ClrScr;
     WriteLn(' Generarea functiilor surjective ');
     WriteLn(' ******************************* ');
     WriteLn; NrSol:=0;
     Write('Dati cardinalul multimii A: '); ReadLn(m);
     Write('Dati cardinalul multimii B: '); ReadLn(n);
     FunctiiSurjective
end.
```

? Întrebări şi exerciţii

1 ☺ Generalizaţi programul prezentat pentru a obţine rezolvarea cazului când A şi B sunt mulţimi cu caractere sau şiruri de caractere.
2 💣 Determinaţi toate funcţiile surjective `f` definite pe mulţimea `{1,2,...,n}` cu valori în mulţimea `{-1,0,1}` astfel încât:

$$f(1)^2+f(2)^2+...f(n)^2=m,$$

`m` fiind dat de la tastatură.

6.2.11. Generarea partiţiilor unei mulţimi

În această problemă se cere să se descompună o mulţime dată sub forma unei reuniuni de mai multe mulţimi disjuncte. Practic, ca şi în alte cazuri prezentate, se poate considera mulţimea primelor `n` numere naturale nenule şi vom determina partiţiile acestei mulţimi.

Soluţia va fi memorată în tabloul `x`, `x[k]` reprezentând cărei mulţimi din partiţie aparţine elementul `k`. De exemplu, dacă `A={1,2,3}`, o soluţie de forma `x=(1,2,2)` înseamnă că am partiţionat mulţimea `A` în submulţimile disjuncte A_1`={1}` şi A_2`={2,3}`.

Atenţie

Pentru a evita repetiţiile, vom scrie astfel programul încât fiecare x[k] să ia valori de la `1` la `x[k-1]+1`.

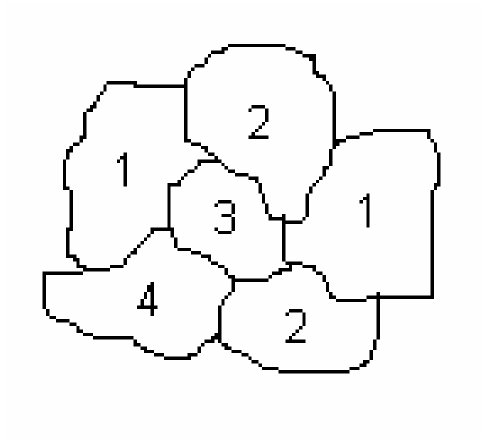
Variabila `max` reprezintă numărul de submulţimi al partiţiei curente şi este determinată şi folosită în procedura de afişare a unei soluţii, `Scrie`.

```
program PartitiileUneiMultimi;
uses Crt;
var x:array [0..20] of Integer;
    NrSol,n,m,k: Integer;

procedure Scrie;
var i,j:integer;
    max: Integer;
begin
     max:=x[1];
     for i:=2 to n do
         if x[i]>max then max:=x[i];
     Inc(NrSol);
     for i:=1 to max do
         begin
           Write('{ ');
           for j:=1 to n do
               if x[j]=i then
                  Write(j,' ');
           Write('} ');
         end;
     WriteLn;
     if ReadKey=#27 then Halt
end;

begin
     ClrScr;
     WriteLn('Generarea partitiilor multimii {1..n}');
     WriteLn('*************************************');
     Write('Dati n='); ReadLn(n);
     NrSol:=0;
     WriteLn('Solutii:'); WriteLn;
     k:=1;x[0]:=0;x[k]:=0;
     while k>0 do
           begin
             if x[k] < x[k-1]+1 then
                begin
                  x[k]:=x[k]+1;
                  if k=n then Scrie
                  else
                     begin
                       k:=k+1; x[k]:=0
                     end
                end
             else k:=k-1
         end;
     WriteLn('Total: ',NrSol,' partitii.'); ReadLn
end.
```

6.2.12. Colorarea hărților

Se dă harta administrativă a unei țări, în care sunt puse în evidență județele (în număr de `n`). Se pune problema colorării județelor, astfel încât, dacă două județe sunt vecine, ele să aibe culori diferite. Sunt disponibile `m` culori distincte.

Exemplu:



Este o reformulare a problemei a colorării nodurilor unui graf, astefl încât două noduri adiacente să aibe culori distincte. Aici graful este planar. (Detalii se pot obține de la disciplina *Bazele informaticii*.).

Matricea `Vecin` are semnificația unei matrice de adiacență. Astfel, dacă județul `i` se învecinează pe hartă cu județul `j`, atunci se pune `Vecin[i,j]=1`, altfel `Vecin[i,j]=0`.

Procedura `Scrie` din programul de mai jos ține cont şi de numele culorilor.

```
program ColorareHarta;
uses Crt;

const nume_cul: array[1..5] of String =
      ('rosu','galben','albastru','verde','violet');

var n,m,k,i,j: Integer;
    x: array[1..20] of Integer;
    Vecin: array[1..20,1..20] of Integer;
    cont: Boolean;


function PotContinua: Boolean;
var i:integer;
    atac:boolean;
begin
     atac:=False;
     for i:=1 to k-1 do
         if (Vecin[i,k]=1) and (x[i]=x[k]) then atac:=True;
     PotContinua:=not atac
end;

procedure Scrie;
var i:integer;
begin
     for i:=1 to n do
         WriteLn('Tara ',i,
                  ' se coloreaza in ',nume_cul[x[i]]);
     ReadLn
end;

begin
     Write('Dati nr. de tari: '); ReadLn(n);
     Write('Dati nr. de culori: '); ReadLn(m);
     for i:=1 to n-1 do
         for j:=i+1 to n do
             begin
               Write('Este vecina tara ',i,' cu tara ',j,
                        ' ? [da=1] ');
```

```
            ReadLn(Vecin[i,j]);
            Vecin[j,i]:=Vecin[i,j]
          end;
    for i:=1 to n do Vecin[i,i]:=0;
    WriteLn; k:=1; x[k]:=0;
    while k>0 do
          begin
               cont:= false;
               while (x[k]<m) and (not cont) do
                   begin
                        x[k]:=x[k]+1;
                        cont:=PotContinua
                   end;
               if cont=true then
                  if k=n then Scrie
                  else
                     begin k:=k+1; x[k]:=0 end
               else k:=k-1;
          end
end.
```

Observație

Deoarece dacă județul `i` este vecin pe hartă cu județul `j`, înseamnă că şi județul `j` este vecin cu `i`, ceea ce a determinat ca citirea matricei `Vecin` să se facă doar pentru elementele de deasupra diagonalei principale.

Asupra acestei probleme se va reveni şi când se va prezenta tehnica Greedy.

? Întrebări şi exerciții

1 😐 Scrieți funcția de continuare pentru problema colorării hărții.
2 ☺ Ce se întâmplă dacă numărul de culori este insuficient?
3 ☹ Scrieți şi testați pe calculator o variantă grafică a acestui program.

6.2.13. Circuitul hamiltonian

Se dâ harta rutieră a unei țări. Se cere să se determine toate posibilitățile de a efectua o excursie prin toate oraşele de pe hartă, trecând prin fiecare oraş exact o singură dată şi întorcându-ne în oraşul de plecare.

Practic avem de a face cu o problemă clasică de teoria grafurilor, numită problema determinării circuitului hamiltonian într-un graf. Se va reveni asupra ei la *Bazele informaticii*, dar şi atunci când, la metoda Greedy, se va prezenta problema comis-voiajorului.

Nodurile grafului sunt oraşele, iar legăturile directe dintre oraşe sunt reprezentate prin muchii în acest graf.

Atenție

Programul prezentat mai jos se bazează tot pe Back-tracking, dar are o anumită particularitate. Pentru a nu se genera de mai multe ori acelaşi circuit, se consideră un nod de plecare (de exemplu 1) ca fixat. Astfel, algoritmul Back-tracking se va rescrie, funcționând de la nodul 2 încolo.

Semnificația lui `x` este următoarea: pentru fiecare `k=1,n`, `x[k]` reprezintă nodul prin care se trece la pasul `k` pe circuitul (drumul închis) care începe cu nodul `1` şi se termină tot cu `1`.

În funcția de continuare se va ține cont că fiecare nod `x[k]` va trebui să fie vecin (în graf) cu nodul `x[k-1]`, adică să fie un drum direct de la orașul prin care se trece la pasul `k` şi orașul prin care s-a trecut la pasul anterior. În plus, ultimul nod al traseului este nodul de plecare (`1`), deci acesta va trebui să fie vecin cu `x[n]`.

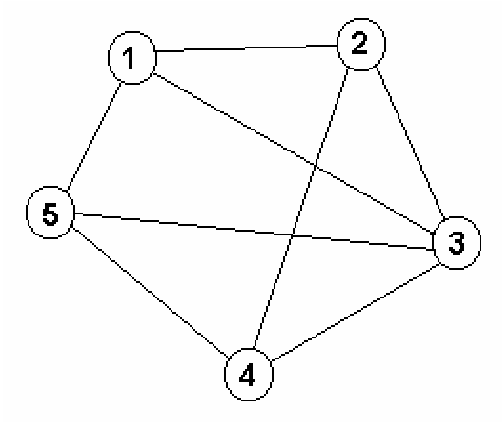
O altă condiție ce trebuie respectată este ca să nu se treacă de două ori prin același oraș. Formal, acest lucru se traduce prin `x[i]≠x[j]`, pentru orice `i` şi `j` distincți. În cadrul funcției `PotContinua`, se va avea în vedere ca până la pasul `k` să nu mai fi trecut prin orașul `x[k]`.

```
program CircuitulHamiltonian;
uses Crt;
const NrSol: Integer = 0;
var n,k,i,j: Integer;
    x: array[1..20] of Integer;
    Vecin: array[1..20,1..20] of Integer;
    cont: Boolean;

procedure Scrie;
begin
     NrSol:=NrSol+1;
     WriteLn('Circuitul ',NrSol,': ');
     for i:=1 to n do
         Write(x[i],' -> ');
     WriteLn(x[1]);
     if ReadKey=#27 then Halt
end;

function PotContinua: Boolean;
var pc: Boolean;
begin
    pc:=True;
    { nodul curent (x[k]) trebuie sa fie
      un nod vecin cu anteriorul }
    if Vecin[x[k],x[k-1]]=0 then pc:=False
    else
        begin
          { ultimul nod trebuie sa fie vecin cu primul }
          if k=n then
             if Vecin[x[n],x[1]]=0 then
                pc:=False
             else
                pc:=True;
          if pc then
          { trebuie sa nu mai fi trecut prin acest nod }
            begin
              for i:=1 to k-1 do
                if x[i]=x[k] then
                   pc:=False
            end
        end;
    PotContinua:=pc
end;

begin
     Write('Dati nr. de orase: '); ReadLn(n);
     for i:=1 to n-1 do
         for j:=i+1 to n do
             begin
               Write('Este drum de la orasul ',i,
                     ' la orasul ',j,' ? [da=1] ');
               ReadLn(Vecin[i,j]);
               Vecin[j,i]:=Vecin[i,j]
             end;
```

```
      for i:=1 to n do
          Vecin[i,i]:=0;
      WriteLn;
      k:=2;
      x[1]:=1; {se considera nodul 1 fixat = 1}
      while k>1 do
            begin
                 cont:= false;
                 while (x[k]<n) and (not cont) do
                     begin
                          x[k]:=x[k]+1;
                          cont:=PotContinua;
                     end;
                 if cont=true  then
                    if k=n then Scrie
                    else begin k:=k+1; x[k]:=1 end
                 else k:=k-1;
            end;
      WriteLn('Total: ',NrSol,' solutii.'); ReadLn
end.
```

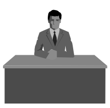
Exemplu: pentru graful din figura de mai jos se obțin 8 soluții:

```
1,2,3,4,5,1;   1,2,4,3,5,1;
1,2,4,5,3,1;   1,3,2,4,5,1;
1,3,5,4,2,1;   1,5,3,4,2,1;
1,5,4,2,3,1;   1,5,4,3,2,1.
```



? Întrebări şi exerciţii

1 ☺ De ce s-a stabilit ca fix nodul `x[1]=1`? Ce s-ar fi întâmplat dacă nu s-ar fi luat aceasă măsură în cadrul programului?
2 ☹ Scrieți funcția de continuare pentru problema circuitului hamiltonian.
3 😐 De ce în cadrul funcției `PotContinua` nu e necesară testarea indicelui `k-1` pentru matricea `Vecin`?
4 💣 Să considerăm că fiecare muchie are un anumit cost pozitiv asociat (adică drumul direct de la un oraş `i` la alt oraş `j` costă `Cost[i,j]` lei). Determinați circuitul hamiltonian de cost minim, adică cel cu suma costurilor muchiilor (drumurilor directe) care compun circuitul minimă.



În atenția profesorului

Se va urmări dezvoltarea la studenți a abilităților de a depista posibilități de optimizare în algoritmi care se bazează pe metoda backtracking.

Probleme

1 ☺ Să se determine toate aranjamentele de `n` elemente luate câte `m`.

2 ☺ Să se scrie un program care, citind un cuvânt şi un număr natural cuprins între 1 şi lungimea acelui cuvânt, să afişeze toate anagramările obţinute din cuvânt, după eliminarea literei de pe poziţia citită.

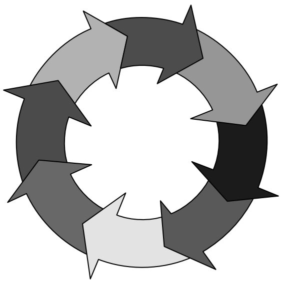
3 💣 Problema căsătoriilor stabile. Se consideră `n` fete care urmează să se căsătorească cu `n` băieţi. Fetele şi băieţii îşi exprimă preferinţele unul faţă de altul prin numere reale din intervalul `[0,1]`. Preferinţa fetei `i` pentru băiatul `j` este dată de `fb[i,j]`, iar preferinţa băiatului `i` pentru fata `j` este dată de `bf[i,j]`. Băiatul ales de fata `i` are numărul `x[i]`. Costul căsătoriei fetei `i` cu băiatul `x[i]` este `fb[i,x[i]]×bf[x[i],i]]`, iar costul general, care trebuie minimizat, este suma tuturor acestor valori. Se cere, în plus, ca cele `n` căsătorii să fie stabile, adică să nu existe `(i,j)` cu `i≠j` astfel încât fata `i` să prefere băiatul `x[j]` băiatului `x[i]`, iar băiatul `x[j]` să prefere fata `i` fetei `j`.

4 ☺ Se dă numărul natural `n>0`. Să se determine toate şirurile de `n` paranteze care se închid corect. De exemplu, pentru n=6 avem: `((()))`, `()()()`, `(()())`, `()(())`, `(())()`.

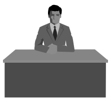
Rezumat

1. Una dintre cele mai cunoscute tehnici generale de elaborare a algoritmilor este metoda Back-tracking. Ea încearcă să elimine generarea tuturor posibilităţilor, pentru a ajunge la rezultat.

2. Metoda Back-tracking se poate aplica acelor probleme pentru care soluţia se poate reprezenta sub forma unui vector `x` ale cărui elemente iau valori din nişte mulţimi finite (de exemplu $\{1,2,...m_k\}$) şi care îndeplinesc anumite condiţii interne.
3. În metoda Back-tracking, elementele vectorului iau valori pe rând, atribuirea unei valori pentru o componentă `x[k]` făcându-se abia după ce s-au atribuit valori pentru toate componentele anterioare ei (`x[1]`, ..., `x[k-1]`), iar între aceste valori nu există incompatibilităţi. Adică se respectă nişte condiţii de continuare.
4. Nerespectarea condiţiilor de continuare implică automat nerespectarea condiţiilor interne, deoarece orice valori am luat pentru x[k+1], ..., x[n] (`n` este numărul de elemente al vectorului), condiţiile interne nu vor fi respectate.
5. Dacă spaţiul valorilor pentru `x[k]` se epuizează, atunci se revine la componenta `k-1`, pentru care se va încerca altă valoare ş.a.m.d..
6. Metoda Back-tracking generează mai multe soluţii.
7. Probleme clasice care pot fi soluţionate prin această metodă sunt: problema damelor, generarea produsului cartezian, generarea combinărilor, problema discretă a rucsacului.



Capitolul 7. Recursivitate

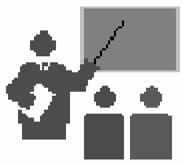


În atenția profesorului

Pe parcursul acestui capitol se vor urmări: a) formarea la studenți a deprinderilor de a utiliza funcțiile recursive; b) formarea la studenți a deprinderilor de a identifica situațiile în care varianta recursivă este preferabilă celei nerecursive sau invers; c) deprinderea studenților cu utilizarea variabilelor locale în cadrul subprogramelor recursive; d) deprinderea studenților cu identificarea problemelelor care pot fi rezolvate utilizând recursivitatea indirectă.

7.1. Prezentare generală

7.1.1. Mecanismul recursivității



Să considerăm că vrem să calculăm factorialul unui număr întreg dat `n`. O modalitate (numită repetitivă sau iterativă) este de a scrie o secvență de forma:

```
fact:=1; for i:=1 to n do fact:=fact*i
```

O altă modalitate este de a ne folosi de următoarea proprietate a factorialului:

```
0!=1, iar n!=n*(n-1)!
```

Astfel, pentru a calcula factorialul unui număr întreg `n` am putea scrie o funcție de genul:

```
function fact(n: LongInt): LongInt;
begin
      if n=0 then fact:=1
      else  fact:=n*fact(n-1)
end;
```

Se observă că în cadrul desrierii funcției `fact` avem un apel al chiar acestei funcții, pentru parametrul efectiv (actual) `n-1`. Este vorba despre o autoapelare a funcției `fact`, ceea ce înseamnă că funcția `fact` este **recursivă**.

Astfel, autoapelul funcției fact generează o nouă activare a acestei funcții, care presupune o eventuală autoapelare ş.a.m.d.. Spunem "eventual", deoarece la un moment dat, autoapelarea se va face cu parametrul efectiv `0`, ceea ce înseamnă că ultimul apel va returna valoarea `1`. Această valoare se va trimite funcției care a apelat fact pentru `n=0`, adică tot lui fact, înapoi, care va înmulți pe `0!=1` cu `1`. Apoi rezultatul (`1`) va fi trimis înapoi funcției care a apelat `fact(1)`, adică tot lui fact, care va obține `1!×2=2!` ş.a.m.d. până la primul apel al lui `fact`.

Putem reprezenta schematic autoapelurile succesive ale funcției fact astfel:

• apeluri succesive:

`fact(n)→fact(n-1)→...fact(2)→fact(1)→fact(0)=1` (oprire) →

• reîntoarceri succesive:

`→fact(1)=1×fact(0)=1×1→fact(2)=2×fact(1)=2×1→fact(3)=3×fact(2)=3×2×1→...fact(n)=n×fact(n-1)=n×(n-1)×(n-2)×...×3×2×1`

Are loc, după cum se vede un calcul al aceluaşi produs, dar în ordine inversă.

Acest caz, când un subprogram se autoapelează, se numeşte **recursivitate directă**.

Există şi posibilitatea unei **recursivităţi indirecte sau încrucişate**, despre care vom vorbi mai târziu. Deocamdată precizăm că aceasta are loc între două sau mai multe subprograme diferite, care se apelează reciproc.

? Întrebări şi exerciţii

1 😐 Ce credeţi că se întâmplă dacă ar lipsi condiţia şi instrucţiunea pentru `n=0` din definiţia funcţiei recursive de calculat factorialul?
2 😐 Descrieţi schematic apelurile recursive pentru `n=3`, ale funcţiei `fact`.
3 ☹ Scrieţi o funcţie recursivă pentru a calcula suma `1+2+...+n`, pentru un `n` dat de la tastatură.

7.1.2. Condiţia de consistenţă a unei definiţii recursive

Să considerăm definiţia recursivă de mai jos (*funcţia lui Ackermann*):

$$A(m,n)=\begin{cases} n+1, & daca \quad m=0, \\ A(m-1,1), & daca \quad n=0, \\ A(m-1,A(m,n-1)), & altfel. \end{cases}$$

În limbajul Pascal, această funcţie se va scrie:

```
function A(m,n: Integer);
begin
  if m=0 then A:=n+1
  else
    if n=0 then
       A:=A(m-1,1)
    else
       A:=A(m-1,A(m,n-1))
end;
```

Să determinăm `A(2,2)`. Vom aplica succesiv definiţia până la identificarea unor argumente pentru care valoarea funcţiei `A` se poate calcula. Apoi vom substitui valoarea calculată în locul celui mai interior argument de pe poziţia a doua, după care vom relua aplicarea definiţiei, dacă numele funcţiei nu mai apare. Avem succesiv:

```
A(2,2)=A(1,A(2,1))=A(1,A(1,A(2,0)))=A(1,A(1,A(1,1)))=A(1,A(1,A(0,A(1,0))))=A(1,A(1,A(0,A(
0,1))))=A(1,A(1,A(0,2)))=A(1,A(1,3))=A(1,A(0,A(1,2)))=A(1,A(0,A(0,A(1,1))))=A(1,A(0,A(0,A
(0,A(1,0)))))=A(1,A(0,A(0,A(0,A(0,1)))))=A(1,A(0,A(0,A(0,2))))=A(1,A(0,A(0,3)))=A(1,A(0,4
))=A(1,5)=A(0,A(1,4))=A(0,A(0,A(1,3)))=A(0,A(0,A(0,A(1,2))))=A(0,A(0,A(0,A(0,A(1,1)))))=A
(0,A(0,A(0,A(0,A(0,A(1,0))))))=A(0,A(0,A(0,A(0,A(0,A(0,1))))))=A(0,A(0,A(0,A(0,A(0,2)))))
=A(0,A(0,A(0,A(0,3))))=A(0,A(0,A(0,4)))=A(0,A(0,5))=A(0,6)=7.
```

Observăm tendinţa de identificare a unor valori direct calculabile, urmată de utilizarea lor în vederea obţinerii valorilor căutate.

Astfel, o definiţie recursivă trebuie să satisfacă următoarea condiţie: *valoarea funcţiei trebuie să fie ori direct calculabilă, ori calculabilă cu ajutorul unei valori direct calculabile*. Această condiţie se numeşte **condiţia de consistenţă**, iar ea este verificată atât de funcţia lui *Ackermann*, cât şi de funcţia recursivă cu ajutorul căreia calculasem factorialul unui număr.

Un exemplu de funcţie inconsistentă este următoarea:

```
function Inconsistent(n: Integer);
begin
   if n=0 then Inconsistent:=1
   else
      Inconsistent:=n*Inconsistent(n+1)
end;
```

? Întrebări şi exerciţii

1 😐 Care este rolul condiţiei de consistenţă într-o definiţie recursivă.
2 ☹ Ce se înţelege prin *condiţie de consistenţă*?
3 😐 Daţi un exemplu de definiţie recursivă inconsistentă.
4 😐 Este următoarea definiţie consistentă sau nu?

```
function F(n: Integer);
begin
   if n=0 then F:=0
   else F:=n+F(n+1)
end;
```

7.1.3. Utilizarea stivelor în recursivitate

Mai întâi trebuie să precizăm că, în informatică, prin stivă se înţelege ceea ce se înţelege şi în viaţa de zi cu zi. Deocamdată vom considera o **stivă** (alocată static) ca fiind un vector asupra căruia se pot face operaţiile: adăugarea unui element, după ultimul element (indicat de un număr `n`), şi eliminarea ultimului element (al `n`-lea) din vector. Vectorul ar putea fi asemuit unei stive de cărţi: putem pune o carte nouă doar peste ultima din celelalte deja existente şi putem să luăm doar cartea de deasupra.

Atunci când, de exemplu, se doreşte calcularea valoarii `3!`, se memorează într-o stivă, iniţial vidă, acest număr `3`, necesar pentru a fi înmulţit cu `2!`. Procesul continuă până la `0!`, când are loc eliminarea, pe rând, a elementelor din stivă, simultan cu câte o înmulţire.

```
function Fact(n: LongInt): LongInt;
begin
     if n=0 then Fact:=1 else Fact:=Fact(n-1)*n
end;
```

Mediul Turbo-Pascal dispune de o stivă proprie, cu o anumită dimensiune. Depăşirea acesteia se face cu opţiunea de compilare `{$S+}`. Opţiunea aceasta este implicită. Când lucrăm cu apeluri recursive ce încarcă foarte mult stiva, dar avem condiţia de consistenţă a recursiei îndeplinită, este bine să folosim opţiunea de compilare `{$S-}`, altfel se poate folosi opţiunea `{$S+}`, care controlează dimensiunea stivei şi nu ne lasă să o supraîncărcăm.

Noi înşine putem simula recursivitatea, implementând o structură proprie de stivă. Următorul program calculează factorialului unui număr, folosind o iteraţie asupra unei stive, iteraţie ce simulează recursivitatea.

```
program SimulareRecursivitate;

const max=20;
type stiva=record
             x: array[1..max] of Integer; n: Integer
           end;

procedure InitStiva(var S: stiva);
begin S.n:=0 end;

function EsteGoalaStiva(var S: stiva): Boolean;
{ folosim "var" pentru economie }
begin EsteGoalaStiva:=S.n=0 end;

procedure PuneInStiva(var S: stiva; elem: Integer);
begin
```

```
      with S do
           if n=max then WriteLn('Stiva plina !')
           else begin Inc(n); x[n]:=elem end
end;

procedure ScoateDinStiva(var S: stiva; var elem: Integer);
begin
      with S do
           if n=0 then WriteLn('Stiva goala !')
           else begin elem:=x[n]; Dec(n) end
end;

function Factorial(n: Integer): Integer;
var S: stiva; F: Integer;
begin
      InitStiva(S); F:=1;

      repeat
       PuneInStiva(S,n); Dec(n)
      until n=0;
      F:=1;
      repeat
       ScoateDinStiva(S,n); F:=F*n
      until EsteGoalaStiva(S);
      Factorial:=F
end;
var n: Integer;
begin
  WriteLn('Calcul factorial - simulare recursivitate');
  Write('Dati n = '); ReadLn(n);
  WriteLn(n,'! = ',Factorial(n)); ReadLn
end.
```

Asupra stivelor vom reveni şi în lecţiile următoare, odată cu reluarea tehnicii de programare Back-tracking în varianta sa recursivă, precum şi atunci când vom discuta despre tehnica Divide-et-impera.

7.2. Funcţii recursive

În continuare vom da unele exemple la recursivitatea directă prin comparare cu metoda iterativă.

7.2.1. Inversarea recursivă a unui cuvânt

- Dacă dorim să inversăm un cuvânt pe măsura introducerii caracterelor sale, atunci vom scrie ceva de genul:

```
program InversareRecursivaCuvant;
uses Crt;

function Invers: String;
var c: Char;
begin
     if EoLn then Invers:=''
     else
```

```
       begin Read(c); Invers:=Invers+c end
end;

begin
     Write('Dati cuvantul: ');
     Write('Cuvantul inversat este: ', Invers);
     ReadKey
end.
```

Funcția `Invers` se termină când se tastează `Enter`.

- Dacă, însă, dorim să scriem o funcție care să returneze inversul (reversul) unui şir dat s, vom proceda ca mai jos:

```
program InverseazaSir;
var s: String;
function Revers(s: String): String;
begin
  if s='' then Revers := ''
  else
     Revers := s[Length(s)] + Revers(Copy(s,1,Length(s)-1))
end;
begin
     Write('Dati sirul de caractere: '); ReadLn(s);
     WriteLn('Inversul sau este: ',Revers(s)); ReadLn
end.
```

Astfel, inversul unui şir `s` dat este format din ultimul caracter al lui `s`, care se pune în fața inversului a ceea ce rămâne din s după eliminarea acestui ultim caracter. Fireşte, dacă `s` este şirul vid, atunci şi inversul său va fi tot şirul vid.

Aceeaşi funcție, în varianta iterativă, s-ar scrie astfel:

```
function Reverse(s: String): String;
var t: String;
    i: Integer;
begin
      t:='';
      for i:=Length(s) downto 1 do t:=t+s[i];
      Reverse:=t
end;
```

7.2.2. Şirul lui Fibonacci

Şirul lui Fibonacci este un şir de numere celebru în matematică. El este definit astfel: primii doi termeni sunt 1 şi 1, iar oricare alt termen se obține din însumarea celor doi imediat dinaintea sa.

Astfel, primii 10 termeni ai şirului sunt: `1, 1, 2, 3, 5, 8, 13, 21, 34, 55`.

Următorul program conține o funcție `F`, care pentru argumentul întreg `n` returnează o valoare ce reprezintă cel de al `n`-lea termen din şirul lui Fibonacci.

```
program SirulLuiFibonacci;
var n: Integer;
function F(n: LongInt): LongInt;
begin
     if (n=1) or (n=2) then F:=1
     else F := F(n-2) + F(n-1)
end;
begin
     Write('Dati n: '); ReadLn(n);
     WriteLn('Al ',n,'-lea termen ',
```

```
        'din sirul lui Fibonacci este: ',F(n));
    ReadLn
end.
```

Observație

O variantă repetitivă a determinării termenilor şirului lui Fibonacci a fost studiată în clasa a IX-a şi o propunem ca exercițiu recapitulativ.

7.2.3. Cel mai mare divizor comun

Cel mai mare divizor comun al două numere întregi se bucură de următoarele proprietăți:

- `cmmdc(a,b)=cmmdc(b,a mod b)`, dacă `a≠0`, respectiv `b`, dacă `a=0`;
- `cmmdc(a,b)=cmmdc(a div b, a mod b)`, dacă `a mod b ≠ 0`, respectiv `b`, dacă `a mod b = 0`;
- `cmmdc(a,b)=cmmdc(a-b,b)`, dacă `a<b`, sau `cmmdc(b-a,b)`, dacă `a>b`, respectiv `a`, dacă `a=b`.

Pornind de la aceste lucruri se pot scrie funcții recursive. Mai jos sunt date două funcții, pentru a doua şi a treia relație.

```
• program CelMaiMareDivizorComunEuclid;
var a,b: Integer;
function Cmmdc(a,b: Integer): Integer;
begin
     if a mod b=0 then Cmmdc := b
     else Cmmdc := Cmmdc(a div b, a mod b)
end;
begin
     WriteLn('Alg. lui Euclid pentru c.m.m.d.c.');
     WriteLn('*********************************');
     Write('Dati a si b: '); ReadLn(a,b);
     WriteLn('C.m.m.d.c. este: ',Cmmdc(a,b));
     ReadLn
end.
```

```
• program CelMaiMareDivizorComunScaderi;
var a,b: Integer;
function Cmmdc(a,b: Integer): Integer;
begin
     if a=0 then Cmmdc:=b
     else
         if a=b then Cmmdc := a
         else
             if a>b then Cmmdc := Cmmdc(a-b,b)
             else Cmmdc := Cmmdc(a,b-a)
end;
begin
  WriteLn('Alg. lui Euclid pentru c.m.m.d.c.');
  WriteLn('*********************************');
  Write('Dati a si b: '); ReadLn(a,b);
  WriteLn('C.m.m.d.c. este: ',Cmmdc(a,b));
  ReadLn
end.
```

În continuare vom prezenta un exemplu ceva mai complex. Vom scrie o funcție pentru determinarea celui mai mare divizor comun al `n` numere întregi. Funcția va fi recursivă, conform relației:

$$\mathrm{CMMDC}(x_1,\dots,x_{n-1},x_n) = \mathrm{CMMDC}(\mathrm{CMMDC}(x_1,\dots,x_{n-1}),x_n).$$

Astfel, vom scrie două funcții. Prima (recursivă) va determina cel mai mare divizor comun al două numere întregi `a` şi `b`, conform primei proprietăți din cele trei prezentate. Cu cea de a doua (tot recursivă) se va determina cel mai mare divizor comun al `n` numere întregi, astfel: se va determina `d`=cel mai mare divizor comun al primelor `n-1` numere, apoi se va determina cel mai mare divizor comun al numerelor `d` şi x_n. Vom memora numerele într-un vector.

```
program Calcul_CMMDC_n_numere;
type vector = array[1..10] of Integer;
var a: vector; x, n, i: Integer;
function Cmmdc2(a,b: Integer): Integer;
begin
     if a = 0 then Cmmdc2:=b
     else Cmmdc2:=Cmmdc2(b, a mod b)
end;
function CMMDC(x: vector; n: Integer): Integer;
begin
   if n=2 then CMMDC:=CMMDC2(x[1],x[2])
   else CMMDC:=CMMDC2(CMMDC(x,n-1),x[n])
end;
begin
   Write('Dati numarul de elemente: '); ReadLn(n);
   for i:=1 to n do
     begin
      Write(' - dati numarul al ',i,
            '-lea: '); ReadLn(a[i])
     end;
   x:=CMMDC(a,n);
   WriteLn('Cel mai mare divizor comun este = ',x);
   ReadLn
end.
```

☞ **Observație**

Deoarece funcția `CMMDC` apelează funcția `CMMDC2`, aceasta din urmă a fost scrisă înaintea primeia. În blocul principal, `a` este un vector, iar `x` un număr întreg. În cadrul funcției `CMMDC2`, `a` este un număr întreg, iar în cadrul funcției `CMMDC` `x` este un vector. Pentru datele de intrare `n=3`, `a=(10,14,6)` se va afişa `2`.

? Întrebări şi exerciții

1 ☹ Puteți scrie o funcție recursivă pentru a determina cel mai mic multiplu comun al două numere?
2 ☹ Dar pentru a determina cel mai mic multiplu comun al `n` numere dintr-un vector?
3 💣 Scrieți o variantă de funcție recursivă pentru a determina cel mai mare divizor comun al `n` numere, pe măsura citirii lor de la tastatură. Ce avantaje şi ce dezavantaje apar în acest nou program?

7.2.4. Funcția lui Ackermann

Funcția lui Ackerman a fost prezentată o dată cu definirea condiției de consistență a unui subprogram recursiv. Ea este dată prin relațiile:

$$A(m,n)=\begin{cases} n+1, & daca \quad m=0,\\ A(m-1,1), & daca \quad n=0,\\ A(m-1,A(m,n-1)), & altfel. \end{cases}$$

Următorul program conține o funcție `Ack` pentru calculul ei.

```
program FunctiaLuiAckermann;
var m,n: LongInt;
function Ack(m,n: LongInt): LongInt;
begin
     if m=0 then Ack:=n+1
     else
         if n=0 then Ack:=Ack(m-1,1)
         else Ack:=Ack(m-1,Ack(m,n-1))
end;
begin
     WriteLn('Functia lui Ackermann. Dati m si n: ');
     ReadLn(m,n);
     WriteLn('Ack(',m,',',n,') = ',Ack(m,n));
     ReadLn
end.
```

Această funcție este, prin definiție, recursivă. Exprimarea repetitivă a unei funcții presupune derecursivarea sa în prealabil.

7.2.5. Suma cifrelor unui număr întreg

Problema determinării sumei cifrelor unui număr întreg s-a prezentat la disciplina *Algoritmi şi limbaje de programare* din clasa a IX-a, o dată cu prezentarea instrucțiunii `while`. Ideea este de a lua ultima cifră a numărului, de a o adăuga sumei (inițial vide), apoi de a proceda la fel cu restul cifrelor. Pentru aceasta, va trebui ca, după fiecare pas, să se elimine din număr cifra din coadă. Această ultimă cifră a unui număr întreg `n` este `n mod 10`, iar eliminarea sa se face prin `n:=n div 10`.

```
program SumaCifrelor;
var n: Integer;
function SumaCifre(n: LongInt): LongInt;
begin
     if n=0 then SumaCifre := 0
     else SumaCifre := n mod 10 + SumaCifre(n div 10)
end;
begin
     Write('Dati un numar intreg: '); ReadLn(n);
     WriteLn('Suma cifrelor sale este: ', SumaCifre(n));
     ReadLn
end.
```

Iată cum funcționează programul anterior (şi funcția `SumaCifre`) pentru numărul `n=74`. Din programul principal se apelează `SumaCifre(74)`. Cum `74` nu este zero, se apelează recursiv funcția pentru numărul `74 div 10`, deci `SumaCifre(7)`. Aici, se va apela SumaCifre(0) (căci `7 div 10 = 0`), care returnează 0. Acest număr se adaugă lui `7 mod 10`, adică lui `7`, obținându-se `7`. (Acest lucru se petrece pentru `SumaCifre(7)`.). Se revine apoi în apelul `SumaCifre(74)`, unde `7` se adaugă lui `74 mod 10`, adică lui `4`, obținânduse `11`, care este suma cifrelor numărului `74`.

? Întrebări şi exerciţii

1 ☺ Scrieţi o funcţie repetitivă pentru a calcula suma cifrelor unui număr. Comparaţi această funcţie cu funcţia din varianta recursivă.
2 😐 Scrieţi o funcţie recursivă care să calculeze produsul elementelor unui vector.
3 ☹ Scrieţi o funcţie recursivă care să determine inversul (oglinditul) unui număr natural dat. Scrieţi şi varianta iterativă a acestei funcţii.

7.2.6. Suma elementelor unui vector

Să scriem un program care, pe baza unei funcţii recursive. să calculeze suma `sum` a componentelor unui vector `a` cu `ne` numere întregi.

Ideea care stă la baza calculării recursive a sumei componentelor vectorului este asemănătoare celei de la calculul factorialului unui număr. Astfel, dacă vectorul nu ar avea elemente, suma ar fi nulă. Dacă are cel puţin un element, atunci suma este dată de suma celor dinainte plus ultimul element.

```
program SumaElementelorUnuiVector_FunctieRec;

type vector=array[1..10] of Integer;

function Suma(x: vector; n: Integer):Integer;
begin
     if n=0 then Suma:=0
     else Suma:=x[n]+Suma(x,n-1)
end;

var a: vector; i,ne,sum: Integer;

begin
     WriteLn('Suma elementelor unui vector - recursiv');
     WriteLn('***************************************');
     Write('Dati nr. de elemente: '); ReadLn(ne);
     for i:=1 to ne do
         begin
           Write('Dati a[',i,']: ');
           ReadLn(a[i])
         end;
     sum := Suma(a,ne);
     WriteLn('Suma elementelor vectorului: ',sum); ReadLn
end.
```

? Întrebări şi exerciţii

1 ☺ Scrieţi un program asemănător celui de mai înainte pentru a calcula produsul elementelor dintr-un vector.
2 ☺ Ce calculează următoarea funcţie recursivă?

```
function NuStiuCe(x: vector; n: Integer):Integer;
begin
     if n=1 then NuStiuCe:=x[1]
     else NuStiuCe:=x[n]+NuStiuCe(x,n-1)
end;
```

Ce diferenţa există faţă de funcţia Suma din programul prezentat?
3 ☹ Scrieţi o funcţie recursivă care să calculeze suma elementelor pare dintr-un vector de numere reale.

4 💣 Scrieți o funcție recursivă care să dea produsul elementelor pare de pe poziții impare dintr-un vector de întregi.

7.2.7. Existența unui element într-un vector

Un exemplu foarte interesant de funcție recursivă este următorul, care face o căutare secvențială a unui element `a` într-un vector `x` în cadrul primelor sale `n` componente.

Fie, așadar, `type vector=array[1..10] of Integer`. Putem scrie funcția logică:

```
function Exista(x: vector; a: Integer): Boolean;
begin
      if n=0 then Exista := False
      else Exista := (x[n]=a) or Exista(x,n-1)
end;
```

Astfel, funcția exprimă formal următorul lucru:

- dacă vectorul nu are elemente, atunci evident `a` nu se află în `x`;
- în schimb, dacă `x` are elemente, atunci a se află în `x` printre primele sale `n` elemente fie dacă este ultimul, fie dacă se află printre cele `n-1` anterioare.

? Întrebări și exerciții

1 ☺ Ce realizează următoarea funcție logică?

```
function NuStiuCe(x: vector; a: Integer): Boolean;
begin
      if n=1 then NuStiuCe := x[1]=a
      else NuStiuCe := (x[n]=a) or NuStiuCe(x,n-1)
end;
```

2 ☹ Scrieți o funcție logică recursivă pentru a determina dacă un vector conține un element negativ sau nu.

3 💣 Scrieți o funcție logică recursivă pentru a determina dacă un vector conține sau nu un element negativ pe o poziție pară din vector.

7.3. Proceduri recursive

7.3.1. Suma componentelor unui vector

Până acum am prezentat recursivitatea prin intermediul subprogramelor de tip funcție. Firește, putem avea și proceduri recursive, iar aici comunicarea de valori între diferitele apeluri succesive se realizează prin intermediul unor parametri variabili (referință). Astel, de exemplu, vom rescrie programul din secțiunea 3.2.7 care calcula (pe baza unei funcții recursive) suma componentelor unui vector cu numere întregi, înlocuind funcția `Suma` cu o procedură `PSuma`, care se bazează pe aceeași idee.

```
program SumaElementelorUnuiVector_ProcRec;

type vector=array[1..10] of Integer;
```

```
procedure PSuma(x: vector; n: Integer; var s: Integer);
begin
     if n=0 then s:=0
     else
       begin
         PSuma(x, n-1,s);
         s:=s+x[n]
       end
end;

var a: vector; i,ne,sum: Integer;

begin
     WriteLn('Suma elementelor unui vector - recursiv');
     WriteLn('*************************************');
     Write('Dati nr. de elemente: '); ReadLn(ne);
     for i:=1 to ne do
         begin
           Write('Dati a[',i,']: ');
           ReadLn(a[i])
         end;
     Suma(a,ne,sum);
     WriteLn('Suma elementelor vectorului: ',sum);
     ReadLn
end.
```

7.3.2. Inversarea unui cuvânt

Să se scrie o procedură recursivă care citeşte caractere (numere) şi le afişează în ordinea inversă citirii (fară a lucra cu şiruri; nu se cunoaşte apriori numărul de caractere).

Rezolvarea se bazează pe procedura `Inverseaza` din programul de mai jos:

Dacă s-a tastat `Enter` (deci funcţia `EoLn` returnează `True`, atunci procesul apelurilor recursive se opreşte, se afişează ultimul caracter şi, la întoarcerile din apelurile recursive are loc o afişare a celorlalte caractere, în ordine inversă.

```
{$X+}
program InversareCuvint;
uses Crt;
var n: Integer;

procedure Inverseaza;
var c: Char;
begin
     Read(c);
     if not EoLn then Inverseaza;
     Write(c)
end;



begin
     ClrScr; Write('Dati cuvantul: ');
     Inverseaza; WriteLn(' este cuvantul inversat.');
     ReadKey
end.
```

7.3.3. inversarea elementelor dintr-un şir

Vom considera următoarea problemă: se dă un şir de cuvinte citite de la tastatură, numărul lor iniţial (`n`) cunoscându-se. Se cere să se prezinte şirul acestor cuvinte în ordinea inversă citirii lor.

Fireşte, programul se poate rezolva repetitiv, folosind un vector care să păstreze cele `n` cuvinte introduse de la tastatură. O rezolvare mai elegantă se bazează pe recursivitate. Astfel, inversarea şirului de cuvinte se poate face pe măsura citirii lor.

Astfel, în cadrul procedurii recursive `Inverseaza` din programul de mai jos, dacă s-a ajuns la ultimul cuvânt, acesta se afişează, iar dacă nu (i<n), atunci se autoapelează această procedură pentru următorul cuvânt.

Cuvintele vor fi afişate invers, datorită revenirilor din apelurile recursive. Până la ultimul cuvânt, celelalte sunt păstrate în stivă.

```
program InversareRecursiva;
var n: Integer;

procedure Inverseaza(i: Integer);
var cuv: String;
begin
     Write('Dati cuvantul ',i,': '); ReadLn(cuv);
     if i<n then
        Inverseaza(i+1)
     else
        WriteLn('Cuvintele in ordine inversa:');
     WriteLn(cuv)
end;

begin
     Write('Dati numarul de cuvinte: '); ReadLn(n);
     Inverseaza(1); ReadLn
end.
```

7.3.4. Transformarea din baza 10 în altă bază

Problema cere să se scrie o procedură recursivă pentru a transforma un număr natural `n` din baza `10` într-o bază `k` (`1 < k < 10`).

Transformarea se bazează pe împărţiri succesive, iar acestea pot fi realizate recursiv pe următoarea idee: a transforma pe `n10` (număr scris în baza `10`) în numărul `nk` (din baza `k`) înseamnă:

- dacă `n10` este nul, atunci şi `nk` va fi tot nul;
- dacă `n10` nu este nul, atunci se transformă din baza `10` în baza `k` acel număr obţinut prin împărţirea lui `n10` la `k`, apoi rezultatul (fie el `nkk`) se înmulţeşte cu `10` şi se adaugă restul împărţirii lui `n10` la `k`.

Atenţie

Înmulţirea cu `10` a lui `nkk` trebuie înţeleasă ca fiind o trecere la un ordin superior a cifrei obţinute, pentru a putea reprezenta numerele din altă bază sub o formă zecimală!

```
program Transformare10k;

var a,b: LongInt; k: Byte;

procedure Transf(n10: LongInt; var nk: LongInt);
var nkk: longInt;
begin
     if n10=0 then nk:=0
     else
         begin
           Transf(n10 div k,nkk);
           nk:=10*nkk + n10 mod k
         end
end;

begin
     Write('Dati numarul in baza 10: '); ReadLn(a);
     Write('Dati baza: '); ReadLn(k); Transf(a,b);
     WriteLn('Numarul in baza ',k,' este: ',b); ReadLn
end.
```

? Întrebări şi exerciţii

Exerciţiile următoare se cer a fi rezolvate folosind subprograme recursive:

1 ☺ Să se calculeze suma S(n)=1+3+5+...+(2n-1).

2 ☺ Să se calculeze produsele:

$$P_1(n)=1\times4\times7\times...\times(3n-2) \text{ şi } P_2(n)=2\times4\times6\times...\times(2n).$$

3 ☹ Să se determine produsul componentelor unui vector.

4 ☹ Să se inverseze un şir de caractere (folosind o procedură recursivă).

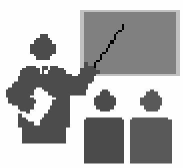
5 ☹ Se consideră declaraţia de tip: `type vector=array[1..20] of Integer`. Să se verifice apartenenţa unui element `a` la un vector `x`, pe baza unei proceduri recursive.

6 ☹ Să se verifice dacă un vector conţine cel puţin un număr negativ în primele `n` poziţii.

7 💣 Să se afişeze conţinutul unui vector.

8 💣 Să se inverseze un vector.

7.4. Varianta recursivă a metodei Back-tracking



Se ştie faptul că în mecanismul tehnici Back-tracking, elementele vectorului soluţie `x` primesc valori pe rând. Dar e posibil ca la un moment dat să se epuizeze valorile pentru o componentă `x[k]` din vector, fără ca restricţiile de continuare să fie îndeplinite. În acel moment are loc o revenire la componenta `k-1`, pentru care se încearcă o nouă valoare.

Practic, este simulată iterativ o recursie, aceea că atribuirea de valori lui `x[k]` face apel la atribuirea de valori lui `x[k+1]`. Astfel toate programele de la Back-tracking se pot rescrie acum sub o formă recursivă.

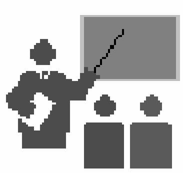
Ne punem problema dacă noile proceduri sunt şi consistente (ca recursivitate), mai ales că nu suntem obişnuiţi cu apeluri de genul `k` → `k+1`.

Procedurile (după cum vom vedea) sunt într-adevăr consistente, deoarece la un moment dat `k` devine egal cu `n` (numărul de componente ale lui `x`). În acest moment se afişează soluţia găsită şi are loc o întoarcere din recursie.

De asemenea, dacă se epuizează toate variantele pentru x[kî, apelul recursiv se încheie, revenindu-se la funcţia apelantă.

În continuare vom prezenta câteva exemple.

7.4.1. Problema celor opt regine



Să se afişeze toate posibilităţile de aşezare a 8 regine pe o tablă de şah în aşa fel încât să nu se atace. Fireşte, numărul de 8 poate fi modificat, aşa cum s-a arătat şi în varianta repetitivă (din capitolul 2).

Soluţia este dată de programul următor:

```
program ProblemaDamelorRecursiv;
uses Crt;
type vector = array[1..8] of integer;
var NrSol: Integer;

procedure Scrie(n: integer; x: vector);
var i: Integer;
begin
     Inc(NrSol);
     WriteLn('Solutia nr. ',NrSol);
     for i := 1 to n do
         WriteLn('Dama de pe coloana ',i,' e pe linia ',x[i]);
     WriteLn;
     ReadLn
end;

function PotContinua(x: vector; k: integer): Boolean;
var atac: Boolean; i: Integer;
begin
    atac := false;
    i := 1;
    while (i < k) and (not atac) do
          if (x[i] = x[k]) or (abs(x[i]-x[k]) = k-i) then
              atac := True
          else
              i := i+1;
          PotContinua := not atac
end;

procedure Dama(k,n: Integer; var x: vector);
var alfa: Integer;
begin
     v := 1;
     while alfa <= n do begin
           x[i] := alfa;
           if PotContinua(x,i) then
              if i = n then Scrie(n,x)
              else Dama(i+1,n,x);
           alfa := alfa + 1
     end
end;

var AsezareDame: vector;
    NrDame: integer;
begin
     ClrScr;
     WriteLn(' Problema damelor - recursiv ');
     WriteLn(' *************************** ');
```

```
     WriteLn; NrSol:=0;
     NrDame := 8;
     Dama(1,NrDame,AsezareDame)
end.
```

Să analizăm funcția principală a programului:

```
procedure Dama(k,n: Integer; var x: vector);
var alfa: Integer;
    cont: Boolean;
begin
     alfa := 1;
     while alfa <= n do begin
           x[k] := alfa;
           if PotContinua(x,k) then
              if k = n then Scrie(n,x)
              else Dama(k+1,n,x);
           alfa := alfa + 1
     end
end;
```

Ce ascunde fiecare din rândurile acestui subprogram? Practic, alfa este valoarea care se dă lui `x[k]`. Ea variază între 1 şi n, aşa cum se arată în ciclul while. Se observă că dacă condițiile de continuare sunt îndeplinite, atunci se poate trece la:

- afişarea soluției (dacă k=n), care va fi urmată de întoarcerea din apelul recursiv;
- trecerea la aşezarea damei următoare (`k+1`).

După ce se revine din afişare sau din apelul recursiv (pentru `k+1`) are loc o creştere a lui `x[k]` la valoarea imediat următoare (`alfa:=alfa+1`), ceea ce înseamnă încercarea de a obține o (nouă) soluție.

Revenirea din apelul recursiv (deci de la `k` la `k-1`) se face după epuizarea tuturor variantelor lui alfa.

☞ **Observație**

Procedura `Dama` se poate scrie şi mai simplu astfel:

```
procedure Dama(k,n: Integer; var x: vector);
var alfa: Integer;
begin
      for alfa:=1 to n do
          begin
            x[k]:=alfa;
            if PotContinua(x,k) then
              if k = n then Scrie(n,x)
              else Dama(k+1,n,x);
          end
end;
```

? Întrebări şi exerciții

1 😐 Scrieți o rezolvare recursivă pentru problema aşezării unor cai pe o tablă de şah `n×n` (câte unul pe fiecare coloană), care să nu se atace între ei.
2 ☹ Realizați o variantă grafică a programelor (cu dame şi cu cai).

7.4.2. Generarea funcțiilor injective

Revenim asupra unei probleme prezentate în capitolul anterior: să se afişeze toate funcţiile injective `f:A→B`, unde `A` şi `B` sunt două mulţimi cu `m` respectiv `n` elemente. (Se va afişa tabelul de variaţie al funcţiei `f`.).

```
program FunctiiInjectiveRecursiv;
const NrSol: Integer=0;
var x: array[1..20] of Integer;
    m,n: Integer;

procedure Scrie;
var i: Integer;
begin
     NrSol:=NrSol+1;
     WriteLn('Functia nr. ',NrSol);
     for i:=1 to m do
         Write(i:3);
     WriteLn;
     for i:=1 to m do
         Write('---');
     WriteLn('---');
     for i:=1 to m do
         Write(x[i]:3);
     WriteLn; ReadLn
end;

function PotContinua(k: Integer): Boolean;
var i: Integer; atac: Boolean;
begin
     atac:=false; i:=1;
     while (i<k) and (not atac) do
           if x[i]=x[k] then atac:=True
           else i:=i+1;
     PotContinua:=not atac
end;

procedure FunInj(k: Integer);
var alfa: Integer;
begin
     for alfa:=1 to n do
         begin
           x[k]:=alfa;
           if PotContinua(k) then
              if k=m then
                 Scrie
              else
                  FunInj(k+1)
         end
end;

begin
     WriteLn('Generare functiilor injective (recursiv)');
     WriteLn('****************************************');
     Write('Dati m='); ReadLn(m);
     Write('Dati n='); ReadLn(n);
     FunInj(1);
     ReadLn
end.
```

7.4.3. Generarea partiţiilor unui număr natural

O altă problemă pe care ne propunem să o rezolvăm prin Back-tracking recursiv este cea a obținerii tuturor partițiilor unui număr natural nenul `n`. Reamintim că printr-o partiție a lui `n` se înțelege o descompunere a lui `n` ca sumă de numere naturale nenule.

```
program PartitiiNumarRecursiv;
var x: array[1..20] of Integer; n: Integer;

procedure Scrie(k: Integer);
var i: Integer;
begin
     Write('n=');
     for i:=1 to k-1 do Write(x[i],'+');
     WriteLn(x[k])
end;
procedure Partitie(k,n: Integer);
var alfa: Integer;
begin
     for alfa:=1 to n do
         begin
           x[k]:=alfa;
           if n-x[k]>0 then Partitie(k+1,n-x[k])
           else Scrie(k)
         end
end;
begin
     WriteLn('Generare partitii numar (recursiv)');
     WriteLn('**********************************');
     Write('Dati n='); ReadLn(n); Partitie(1,n); ReadLn
end.
```

7.4.4. Plata unei sume cu bancnote de valori date

Reamintim problema aceasta, prezentată în capitolul anterior.

Să se afişeze toate modalitățile de a plăti o sumă `S` cu bancnote de valori $b_1>b_2>...>b_n$.

Se presupune că există un număr suficient de bancnote din fiecare tip.

Propunem cititorului să rezolve singur această problemă bazându-se pe soluțiile din secțiunile 3.4.3 şi 2.2.5. Succes!

7.5. Backtracking în plan

7.5.1. Problema labirintului

Există şi situații când soluțiile unor probleme, rezolvabile prin metoda Back-tracking se pot da sub forma unei matrice, care să conțină, în nişte "căsuțe", numerele `1,2,...,p`, unde `p` este lungimea vectorului asociat soluției (în varianta de până acum). Numărul `p` poate varia de la caz la caz. Un bun exemplu îl constituie următoare problemă.

Faptul că se utilizează o matrice pentru memorarea soluție a dus la denumirea metodei de **Back-tracking în plan**.

Se dă un labirint memorat sub forma unei matrice `Labirint` de elemente `0` şi `1`, în care unitățile corespund spațiilor pe unde se poate trece, iar zerourile zidurilor. Un şoricel pus într-o anumită căsuță a labirintului (`(i_initial,j_initial)`) va trebui să ajungă într-o altă casuță a labirintului, unde se află o bucățică de cașcaval ((i_final,j_final)). El se poate mișca doar ortogonal, nu și diagonal.

Pentru a determina toate posibilitățile de a ajunge la cașcaval, fără a trece de mai multe ori prin același loc, vom folosi metoda Back-tracking, într-o variantă recursivă, cu unele modificări. Astfel, nu vom memora drumurile parcurse de șoricel sub forma unui vector x, deoarece nu știm cât de mare poate ajunge acest vector la un moment dat (E drept că uneori am folosit și o astfel de memorare, ca în cazul partițiilor unui număr, însă acum această modalitate de reprezentare a soluției problemei este destul de dificilă.).

Vom folosi, în schimb, o matrice Traseu asociată tablei. Convenim ca Traseu[i,j] să fie egală cu o valoare numită pas, dacă șoricelul trece la pasul pas pe acolo, în drumul său căre cașcaval, respectiv 0, dacă șoricelul nu trece pe acolo.

De asemenea, observăm că șoricelul, dacă se află în poziția (i,j), nu se poate deplasa decât în patru direcții, sus, jos, stânga și dreapta, în toate aceste direcții doar cu o căsuță (dacă nu este zid!). Astfel, din poziția (i,j), unde a ajuns la pasul pas, va trece într-o poziție nouă (i_nou,j_nou), la pasul următor, pas+1.

Noile coordonate se obțin din cele vechi prin adăugarea valorilor -1, 0, sau 1, în conformitate cu cele patru direcții. Acest lucru se va realiza folosind două șiruri speciale de numere, notate prin oriz și vert.

Programul următor folosește o matrice constantă pentru un labirint de dimensiune 8×10. Una din soluțiile programului pentru datale de intrare (4,4) și (1,6) este dată de matricea de mai jos:

0	0	0	0	0	6	0	0	0	0
0	0	0	0	0	5	0	0	0	0
0	0	0	2	3	4	0	0	0	0
0	0	0	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0

```
program ProblemaLabirintului;
const m=8; n=10;
type sir=array[1..4] of ShortInt;
     matrice=array[1..m,1..n] of Byte;
const Labirint: matrice
 =((0,0,0,0,0,1,0,0,0,0), (0,0,0,1,0,1,0,0,0,0),
   (0,0,0,1,1,1,0,0,0,0), (1,1,1,1,0,1,0,0,0,0),
   (0,0,0,1,0,1,0,0,0,0), (0,1,1,1,1,1,1,1,0,0),
   (1,1,0,0,1,0,0,0,0,0), (0,0,0,0,1,0,0,0,0,0));
const oriz: sir = (-1,0,1,0); vert: sir = (0,1,0,-1);
var Traseu:matrice; i,j,i_initial, j_initial, i_final,j_final: Byte;
procedure Scrie;
var i,j: Integer;
begin
  for i:=1 to m do
      begin for j:=1 to n do Write(Traseu[i,j]:3); WriteLn end;
  WriteLn
end;
procedure Drum(i,j,pas: Byte);
{procedura recursiva de back-tracking}
var i_nou,j_nou: ShortInt; varianta: Byte;
begin
  for varianta:=1 to 4 do
    begin
      i_nou:=i+oriz[varianta]; j_nou:=j+vert[varianta];
      if (i_nou in [1..m]) and (j_nou in [1..n]) then
         if (Labirint[i_nou,j_nou]=1)
```

```
              and (Traseu[i_nou,j_nou]=0) then
              begin
                Traseu[i_nou,j_nou]:=pas;
                if (i_nou=i_final) and (j_nou=j_final) then Scrie
                else Drum(i_nou,j_nou,pas+1);
                Traseu[i_nou,j_nou]:=0
              end
      end
  end;
  begin
   { se initializeaza matricea Traseu }
   for i:=1 to m do for j:=1 to n do Traseu[i,j]:=0;
   Write('Dati pozitia initiala -> '); ReadLn(i_initial, j_initial);
   Write('Dati pozitia finala   -> '); ReadLn(i_final, j_final);
   Traseu[i_initial, j_initial]:=1; WriteLn('Solutii : ');
   Drum(i_initial, j_initial, 2); ReadLn
  end.
```

7.5.2. Acoperirea unei table de şah prin săritura calului

Se consideră o tablă de şah de dimensiune nxn şi un cal plasat în colţul de stânga sus. Se cere să se afişeze toate posibilităţile de mutare a acestei piese de şah astfel încât să treacă o singură dată prin fiecare pătrat al tablei.

Rezolvarea problemei se poate face pe baza unui algoritm de Back-tracking în plan, ca şi la problema şoricelului din labirint. De această dată, îmsă, avem o deplasare care ţine cont de cum mută calul la şah (două căsuţe pe o direcţie şi una pe cealaltă), ceea ce conduce la modificarea celor două şiruri:

```
const di: sir = (-1,1,2,2,-1,1,-2,-2);
            dj: sir = (-2,-2,-1,1,2,2,-1,1);
```

În plus, va trebui nu să ajungem într-un loc anume, ci, dimpotrivă, să parcurgem toate locurile, ceea ce se exprimă prin apelul condiţionat al lui `Scrie` din cadrul procedurii `Pas`:

```
if p=m*n then Scrie
```

Aici `p` este pasul curent, iar `m` şi `n` sunt dimensiunile tablei de şah considerate).

```
program DrumulCalului;
var m,n,p: Integer;
    t: array[1..5,1..5] of Integer;

procedure Scrie;
var i,j: Integer;
begin
     WriteLn;
     for i:=1 to m do
         begin
           for j:=1 to n do
               Write(T[i,j]:3);
           WriteLn
         end;
     ReadLn
end;

procedure Pas(i,j,p: Integer);
type sir=array[1..8] of Integer;
const di: sir = (-1,1,2,2,-1,1,-2,-2);
      dj: sir = (-2,-2,-1,1,2,2,-1,1);
var k,i_n,j_n: Integer;
begin
     for k:=1 to 8 do
         begin
           i_n:=i+di[k]; j_n:=j+dj[k];
```

```
           if (1<=i_n) and (i_n<=m) and
              (1<=j_n) and (j_n<=n) then
              begin
                if T[i_n,j_n]=0 then
                   begin
                     T[i_n,j_n]:=p;
                     if p=m*n then Scrie
                     else Pas(i_n,j_n,p+1);
                     T[i_n,j_n]:=0
                   end
              end
         end
end;

begin
     Write('Dati m si n : '); ReadLn(m,n);
     T[1,1]:=1; Pas(1,1,2)
end.
```

Iată prima soluție care se obține dacă se rulează programul pentru `m=5` şi `n=5`:

1	14	9	20	23
10	19	22	15	8
5	2	13	24	21
18	11	4	7	16
3	6	17	12	25

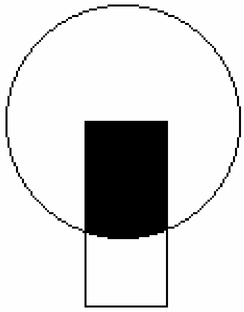
7.5.3. Algoritmul de acoperire a unei suprafețe delimitate de un contur închis

Algoritmul (numit şi "Fill") va fi prezentat într-o variantă grafică prin programul de mai jos, pe care ar fi bine să-l scrieți pe calculator. Încercați, de asemenea, o generalizare a acestei probleme, trecând la diferite tipuri de haşuri (texturi), citite dintr-o anumită matrice.

Atenție

Nu încercați să umpleți cu acest procedeu suprafețe prea mari, deoarece stiva se încarcă extraordinar de mult, ceea ce poate avea efecte neplăcute!

Exemplu:



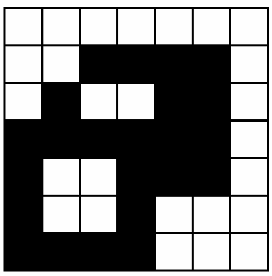
```
program Umplere;
uses Graph;
procedure Fill(x,y: Word);
begin
     if GetPixel(x,y)<>White then
        begin
          PutPixel(x,y,White);
          Fill(x-1,y);
          Fill(x+1,y);
          Fill(x,y-1);
          Fill(x,y+1)
        end
end;
var gd,gm: Integer;
begin
     gd:=0; InitGraph(gd,gm,'C:\BP\BGI');
     Rectangle(300,200,340,250);
     Circle(320,200,30);
     Fill(320,220);
     ReadLn;
     CloseGraph
end.
```

7.5.4. Problema fotografiei

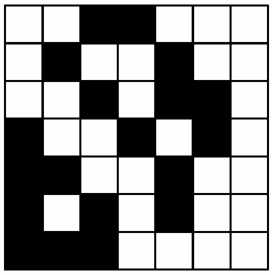
Fotografia alb negru a unui obiect este reprezentată sub forma unei matrice cu `n` linii şi `m` coloane, ale cărei elemente sunt `0` sau `1`. Elementele notate cu `1` reprezintă punctele ce aparţin obiectului. Două elemente de valoare `1` fac parte din acelaşi obiect dacă sunt adiacente pe linie, coloană sau diagonale. Se cere să se determine numărul obiectelor din fotografie.

Rezolvarea se bazează pe algoritmul de umplere descris în secţiunea anterioară. Dacă ar fi un singur obiect, atunci prin umplerea acestuia s-ar obţine matricile `A` şi `T` identice (vezi programul). Dacă `A` şi `T` diferă prin cel puţin un element, atunci înseamnă că există mai multe obiecte în fotografie.

Exemplu: În figura de mai jos (corespunzătoare matricii din program) avem un singur obiect:



Iar în următoarea figură avem două obiecte:



```
program Fotografie;
const n=7;
type matrice=array[1..n,1..n] of 0..1;
     sir=array[1..8] of ShortInt;
const x: sir = (-1,-1,0,1,1, 1, 0,-1);
      y: sir = ( 0, 1,1,1,0,-1,-1,-1);
      A: matrice =
         ((0,0,0,0,0,0,0),
          (0,0,1,1,1,1,0),
          (0,1,0,0,1,1,0),
          (1,1,1,1,1,1,0),
          (1,0,0,1,1,1,0),
          (1,0,0,1,1,0,0),
          (1,1,1,1,0,0,0));
var T: matrice;
    i,j: Byte;
    unu: Boolean;

procedure Fill(i,j: Byte);
var ii,jj: ShortInt;
    k: Integer;
begin
     for k:=1 to 8 do
         begin
           ii:=i+x[k]; jj:=j+y[k];
           if (ii in [1..n]) and (jj in [1..n]) then
              if (A[ii,jj]=1) and (T[ii,jj]=0) then
                 begin
                   T[ii,jj]:=1;
                   Fill(ii,jj)
                 end
         end
end;

begin
     for i:=1 to n do
         for j:=1 to n do
             T[i,j]:=0;
     for i:=1 to n do
         for j:=1 to n do
             if A[i,j]=1 then
                begin
                  T[i,j]:=1;
                  Fill(i,j); i:=n; j:=n
                end;
     unu:=True;
     for i:=1 to n do
         for j:=1 to n do
             if A[i,j]<>T[i,j] then
                begin
                  unu:=False;
                  i:=n;
```

```
                j:=n
              end;
      if unu then
         WriteLn('Un singur obiect.')
      else
         WriteLn('Mai multe obiecte.');
      ReadLn
end.
```

Probleme

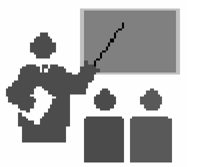
1 Problema “Attila şi regele”: Un cal (pe care stă Attila) şi un rege se află pe o tablă de şah. Unele câmpuri sunt “arse”, poziţiile lor fiind cunoscute. Calul nu poate călca pe câmpuri “arse”, iar orice mişcare a calului “arde” câmpul pe care se duce. Să se afle dacă există o succesiune de mutări permise (cu restricţiile de mai sus) prin care calul să ajungă la rege şi să revină la poziţia iniţială. Poziţia iniţială a calului, precum şi poziţia regelui sunt considerate nearse.

2 Un ţăran primeşte o bucată dreptunghiulară de pământ pe care doreşte să planteze o livadă. Pentru aceasta, el va împărţi bucata de pământ în m×n pătrate, având dimensiunile egale, iar în fiecare pătrat va planta un singur pom din cele patru soiuri pe care le are la dispoziţie. Să se afişeze toate variantele de a alcătui livada respectând următoarele condiţii:
a) Nu trebuie să existe doi pomi de acelaşi soi în două căsuţe învecinate ortogonal sau diagonal. b) Fiecare pom va fi înconjurat de cel puţin un pom din toate celelalte trei soiuri.(æăranul are la dispoziţie suficienţi pomi de fiecare soi.).

3 Un teren muntos are forma unei matrice cu m×n zone, fiecare zonă având o înălţime. Un alpinist pleacă dintr-o anumită zonă şi trebuie să ajungă într-o zonă maximă în altitudine. Dintr-o zonă, alpinistul se poate deplasa diagonal sau ortogonal, într-una din zonele (căsuţele) alăturate, doar urcând sau mergând la acelaşi nivel. Poate el ajunge într-unul din vârfuri? Dacă da, arătaţi toate soluţiile problemei.

7.6. Metoda Divide-et-impera

7.6.1. Prezentare generală



Divide-et-impera este o tehnică (recursivă) ce constă în următoarele:

- dacă problema este rezolvabilă direct, atunci ea se rezolvă
- altfel se descompune în două sau mai multe probleme mai simple, de aceeaşi natură cu problema iniţială (numite subprobleme), care se rezolvă prin aceeaşi metodă; soluţia problemei iniţiale se obţine prin combinarea soluţiilor subproblemelor.

De exemplu, fie $A=(a_1,a_2,...,a_n)$ şi trebuie efectuată o prelucrare oarecare asupra elementelor sale. Mai mult, presupunem că pentru orice p, q naturale, cu $1\le p<q\le n$, există $m\in\{p,...,q-1\}$ astfel încât prelucrarea secvenţei $\{a_p,...,a_k\}$ se poate face prelucrând secvenţele $\{a_p,...,a_m\}$ şi $\{a_{m+1},...,a_q\}$.

Se apelează procedura descrisă mai jos, astfel:

DivideEtImpera(1,n,α), în α obţinându-se rezultatul final.

Am notat prin ε lungimea maximă a unei secvenţe $\{a_p,...,a_q\}$ pentru care prelucrarea se poate face direct (prin procedura Prelucreaza).

Procedura Combina realizează combinarea rezultatelor a două secvenţe vecine. m este obţinut prin apelul procedurii Divide.

Iată deci varianta recursivă a metodei Divide-et-impera:

```
procedure DivideEtImpera(p, q: Integer;  var α: ...);
begin
  if q - p < ε then Prelucreaza(p, q, α)
  else
    begin
      Divide(p, q, m);
      DivideEtImpera(p, m, β);
      DivideEtImpera(m+1, q, γ);
      Combina(β, γ, α)
    end
end;
```

Printre exemplele clasice care folosesc această metodă se numără: *sortarea prin interclasare; sortarea rapidă* (“*quick-sort*”)*; turnurile din Hanoi*.

7.6.2. Determinarea maximului si minimului unui şir

Să se determine cel mai mare şi cel mai mic element dintr-un sir.
Să rezolvăm problema pentru minim.

În acest caz, vom considera problema rezolvabilă direct dacă numărul de elemente este 2 (deci q-p=1). În acest caz, se compară elementele de pe pozițiile p şi q şi se determină minimul corespunzător.

Dacă problema nu este rezolvabilă direct, atunci ea se descompune în următoarele subprobleme:

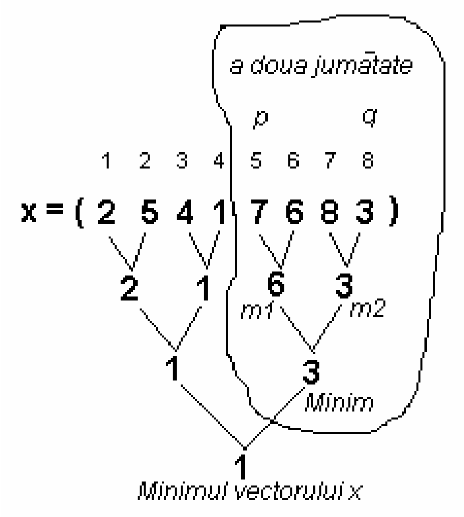
- se determină minimul jumătății din stânga, fie acesta m1;
- se determină minimul jumătății din dreapta, fie acesta m2.

Minimul final va fi cel mai mic element dintre m1 şi m2. Fireşte, determinările lui m1 şi m2 se fac recursiv.

De exemplu, fie vectorul x=(2,5,4,1,7,6,8,3). Acesta se va împărți în doi vectori: (2,5,4,1) şi (7,6,8,3). Primul se va împărți, la rându-i, în (2,5) şi (4,1). Rezultă două elemente minime, corespunzătoare acestor două părțil: 2, respectiv 1. Minimul din (2,5,4,1) va fi, aşadar, numărul 1.

Cu cea de a doua jumătate se procedează la fel, obținându-se minimul 3. Dintre 1 şi 3 cel mai mic este 1, care va fi minimul întregului vector x.

Aşadar, avem o rezolvare după schema următoare:



(Conform funcției `Minim` din programul de mai jos, avem pentru cea de a doua jumătate a vectorului nostru: `p=5`, `q=8`, `m1=6`, `m2=3`, rezultând `Minim=3`).

```
program DeterminareMinim;
type vector=array[1..10] of Integer;
function Minim(x: vector; p,q: Integer): Integer;
var m: Integer; m1,m2: Integer;
begin
     if q-p=1 then
        if x[p]<x[q] then
           Minim:=x[p]
        else
           Minim:=x[q]
     else
        begin
          m:=(ic+sf) div 2;
          m1:=Minim(x,ic,m);
          m2:=Minim(x,m,sf);
          if m1<m2 then Minim:=m1
          else Minim:=m2
        end
end;

var x: vector;
    n,i: Integer;

begin
     Write('Dati nr de elemente, apoi elementele: ');
     ReadLn(n);
     for i:=1 to n do
         ReadLn(x[i]);
     WriteLn('Cel mai mic este: ',Minim(x,1,n)); ReadLn
end.
```

Propunem cititorului să scrie o funcție şi un program asemănătoare pentru a determina maximul unui vector.

7.6.3. Metoda căutarii binare

Problema cere să se verifice dacă un număr se află printre elementele unui şir ordonat crescător.

În acest caz particular, căutarea secvenţială a unui număr nu e prea eficientă, deoarece dacă numărul se află în a doua jumătate a secvenţei, deci ar fi de preferat să nu-l căutăm în prima jumătate. De aceea, îl vom căuta în acea jumătate în care, în mod logic, *s-ar putea găsi*.

Algoritmul căutării binare este:

- ♦ dacă numărul din mijloc este mai mic decât numărul căutat, atunci căutăm în a doua jumătate;
- ♦ dacă numărul din mijloc este mai mare ca numărul căutat, atunci căutăm în prima jumătate;
- ♦ dacă numărul din mijloc este egal cu numărul căutat, înseamnă că am găsit numărul în cauză si trebuie să oprim căutarea.

Căutarea în jumătatea aleasă se face tot la fel, deci se va înjumătăţi si această zonă etc..

Atenţie

Acest procedeu este mai rapid decât cel al căutării secvenţiale, dar nu se poate aplica decât dacă vectorul este deja ordonat.

• *Varianta recursivă:*

```
program CautareBinara_Recursiv;
var a: array[1..20] of Integer;
    e,n,i: Integer; g: Boolean;

procedure CB(ic,sf: Integer; var g: Boolean);
var m: Integer;
begin
     if ic<=sf then
        begin
          m:=(ic+sf) div 2;
          if e=a[m] then
             g:=True
          else
             if e<a[m] then
                CB(ic,m-1,g)
             else
                CB(m+1,sf,g)
        end
     else g:=False
end;
begin
     Write('n='); ReadLn(n);
     for i:=1 to n do
         begin
           Write('a[',i,']=');
           ReadLn(a[i])
         end;
     Write('elem. cautat='); ReadLn(e);
     CB(1,n,g);
     if g then WriteLn('Exista') else WriteLn('Nu exista');
     ReadLn
end.
```

• *Varianta repetitivă:*

```
program CautareBinara_Iterativa;
var a: array[1..20] of Integer;
    m,e,n,i,ic,sf: Integer; g: Boolean;
```

```
begin
     Write('n='); ReadLn(n);
     for i:=1 to n do
         begin
           Write('a[',i,']=');
           ReadLn(a[i])
         end;
     Write('elem. cautat='); ReadLn(e);
     ic:=1; sf:=n; g:=False;
     while (ic<=sf) and (g=False) do
        begin
          m:=(ic+sf) div 2;
          if e=a[m] then
             g:=True
          else
             if e<a[m] then sf:=m-1
             else ic:=m+1
        end;
     if g then
        WriteLn('Exista') else WriteLn('Nu exista');
     ReadLn
end.
```

? Întrebări şi exerciţii

1 😐 Rescrieți programul recursiv astfel încât să folosiți o funcție de căutare binară în locul procedurii `CB`.
2 😐 Modificaţi programele astfel încât, în cazul în care elementul se găseşte pe o anumită poziție în cadrul vectorului, aceasta să se afişeze.
3 💣 Scrieți o variantă iterativă pentru căutarea binară. (Ați învățat în clasa a IX-a la *Algoritmi și limbaje de programare*!).

7.6.4. Căutarea prin interpolare

Căutarea prin interpolare este o ameliorare a metodei căutării binare, care se bazează pe strategia adoptată de o persoană când caută un cuvânt într-un dicționar. Astfel, dacă cuvântul căutat începe cu litera *C*, deschidem dicționarul undeva mai la început, iar când cuvântul începe cu litera *V*, deschidem dicționarul mai pe la sfârşit.

Dacă `e` este valoarea căutată în vectorul `a[ic..sf]`, atunci partiționăm spațiul de căutare pe poziția `m=ic+(e-a[ic])*(sf-ic)/(a[sf]-a[ic])`. Această partiționare permite o estimare mai bună în cazul în care elementele lui a sunt numere distribuite uniform.

Propunem cititorului realizarea programului corespunzător metodei descrise.

7.6.5. Turnurile din Hanoi

Se spune că în Vietnamul antic, în Hanoi, erau trei turnuri, pe unul din ele fiind puse, în ordinea descrescătoare a diametrelor lor, mai multe (opt) discuri de aur. Din motive obiective, nişte călugări, care le aveau în grijă, trebuiau să aşeze discurile pe cel de-al doilea turn, în aceeaşi ordine. Ei puteau să folosească, eventual, turnul al treilea, deoarece, altfel discurile nu ar fi avut stabilitate. Discurile puteau fi mutate unul câte unul. De asemenea, nu era permis a aşeza un disc mai mare peste unul mai mic, pentru ca cel de deasupra să nu-l strice pe cel de dedesupt.

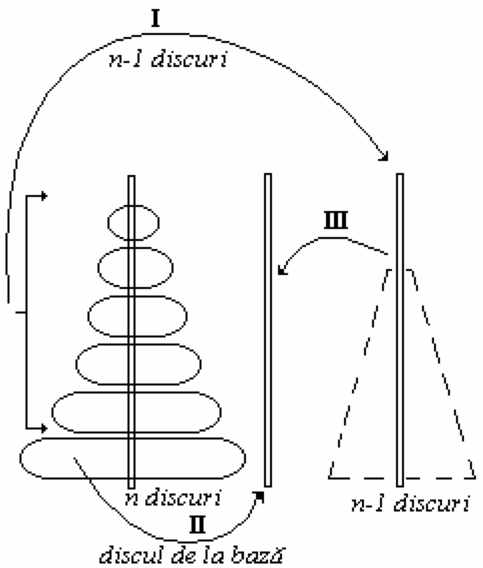
Deşi, aparent simplă, după câteva încercări, folosind un prototip în miniatură, cititorul va constata că problema nu e banală. Însă este posibilă o rezolvare optimă a ei (cu numai 2^n-1 mutări, deci 255, pentru n=8), folosind tehnica recursivă Divide-et-impera.

Problema este de a muta n discuri de la turnul 1 la turnul 2. Pentru a o rezolva, să vedem cum se mută, în general, n discuri de la un turn p la un turn q. Se mută primele n-1 discuri de pe p pe r, r fiind turnul auxiliar, apoi singurul disc rămas pe p (discul cel mai mare) se mută de pe p pe q, după care cele m-1 discuri sunt mutate de pe r pe q.

Fireşte, mutarea celor n-1 discuri de la p la r şi de la r la q se realizează la fel, deci printr apeluri recursive.

Mutarea primelor n-1 discuri este corectă, deoarece existența discurilor de diametre mai mari, la bazele celor trei turnuri nu afectează cu nimic mutările discurilor mai mici.

În cazul limită n=1, avem doar o mutare a discului din vârful turnului p spre q, adică problema se rezolvă direct. Putem spune, aşadar, că avem o descompunere în trei probleme a problemei mari.



Programul de mai jos soluționează (cu animație) problema descrisă, pentru n=8. Procedura de bază este Han, iar celelalte proceduri sunt pentru mişcarea discului curent. Există şi două proceduri cu structură inedită. Ele sunt scrise în limbaj de asamblare. Folosind întreruperea 10h, acestea realizează ascunderea, respectiv reafişarea cursorului din modul text.

```
program TurnurileDinHanoi;
uses Crt;
const Pauza=10; forma= #219; Virf: array [1..3 ] of Byte=(13,22,22);
procedure HideCursor; assembler;
{ ascunde cursorul pilpâitor, in modul text }
asm MOV AX,$0100; MOV CX,$2607; INT $10 end;
procedure ShowCursor; assembler;
{ reafiseaza cursorul }
asm MOV AX,$0100; MOV CX,$0506; INT $10 end;
function ColTija (tija : Byte) : Byte; {stabileste coloana unei tije}
begin ColTija := 24*tija-8 end;
procedure MutaDreapta (disc, tija1, tija2 : Byte);
var i,k: Byte;
begin
     for i := ColTija(tija1)-disc
         to Pred(ColTija(tija2)-disc) do
         begin
           Delay(Pauza);
```

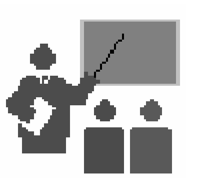
```
            if KeyPressed then Halt(1);
            GoToXY(i,3);
            for k:=0 to 2*disc do Write(' ');
            GoToXY(i+1,3);
            for k:=0 to 2*disc do Write(forma)
          end
end;
procedure MutaStanga (disc, tija1, tija2 : Byte);
var i,k: Byte;
begin
     for i := ColTija(tija1)-disc
         downto Succ(ColTija(tija2)-disc) do
         begin
           Delay(Pauza); if KeyPressed then Halt(1);
           GoToXY(i,3);
           for k:=0 to 2*disc do Write(' ');
           GoToXY(i-1,3);
           for k:=0 to 2*disc do Write(forma)
         end
end;
procedure Coboara (disc, tija : Byte);
var i,k:  Byte;
begin
  for i := 3 to Pred(Virf[tija]-1) do
    begin
      Delay(Pauza); if KeyPressed then Halt(1);
      GoToXY(ColTija(tija)-disc,i);
      for k:=0 to 2*disc do Write(' ');
      GoToXY(ColTija(tija)-disc,i+1);
      for k:=0 to 2*disc do Write(forma)
    end;
  Dec(Virf[tija])
end;
procedure Ridica (disc, tija : Byte);
var i,k: Byte;
begin
  for i := Virf[tija] downto 4 do
    begin
      Delay(Pauza); if KeyPressed then Halt(1);
      GoToXY(ColTija(tija)-disc,i);
      for k:=0 to 2*disc do Write(' ');
      GoToXY(ColTija(tija)-disc,i-1);
      for k:=0 to 2*disc do Write(forma)
    end;
  Inc(virf[tija])
end;
procedure Muta(disc, tija1, tija2 : Byte);
begin
  Ridica(disc,tija1);
  if (tija1 < tija2) then
     MutaDreapta(disc,tija1,tija2)
  else  MutaStanga(disc,tija1,tija2);
  Coboara(disc,tija2)
end;
procedure Han(n, tija1, tija2, tija3 : Byte);
begin
  if (n = 1) then Muta(1,tija1,tija2)
  else
    begin
      Han(n-1,tija1,tija3,tija2);
      Muta(n,tija1,tija2);
      Han(n-1,tija3,tija2,tija1)
  end
end;
```

```
procedure Initializari;
var k,disc: Byte;
begin
  HideCursor; ClrScr;
  for disc:=1 to 9 do
      begin
        GoToXY(ColTija(1)-disc,Virf[1]+disc-1);
        for k:=0 to 2*disc do
        Write(forma);
      end
end;
begin { PROGRAM }
  Initializari;
  GoToXY(28,1); WriteLn(' ~Turnurile din Hanoi~ ');
  Han(8,1,2,3); ShowCursor
end.
```

☞ **Observaţie**

Citorii care nu cunosc elemente de limbaj de asamblare vor considera procedurile `ShowCursor` şi `HideCursor` ca atare. Mai puţin importantă este înţelegerea modului cum sunt ele realizate şi mai mult ce execută ele.

7.6.6. Sortare rapidă prin partiţionare



Un alt exemplu de utilizare a tehnicii Divide-et-impera îl constituie acest algoritm avansat de sortare (numit **quick-sort**), datorat profesorului *C. Hoare*, care foloseşte o procedură `Pozitioneaza`. Această procedură se ocupă de o anumită parte din vectorul de sortat `A`, cuprinsă între indicii `start` şi `finis`. Ea poziţionează componenta de pe poziţia `start` pe o anumită poziţie `k`, între `start` şi `finis`, poziţie pe care respectivul element va rămâne până la final, astfel încât toate elementele de pe poziţii între `start` şi `k-1` să fie mai mici sau egale cu `A[k]`, iar toate elementele de pe poziţii între `k+1` şi `finis` să fie mai mari sau egale cu `A[k]`.

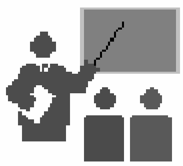
În procedura de sortare `Quick`, după ce s-a realizat poziţionarea, în conformitate cu tehnica Divide-et-impera, se va autoapela această procedură pentru cele două părţi rămase nesortate, dinainte şi de după elementul poziţionat `A[k]`.

În procedura `Pozitioneaza` se compară, în mod repetat, două elemente, cele de pe poziţiile `i` şi respectiv `j` din vector. Iniţial `i` este `start`, iar `j` este `finis`. La fiecare pas, dacă `A[i]>A[j]` se interschimbă `A[i]` cu `A[j]`, lucru urmat fie de mărirea lui `i` cu o unitate, fie la micşorarea lui `j` cu o unitate. Astfel, până când `i` devine egal cu `j`, poziţiile `i` şi `j` se apropie, una de alta, odată cu eventualele interschimbări necesitate de relaţia existentă între elementele de pe cele două poziţii. Modificările lui `i` şi `j` se realizează cu ajutorul variabilei `d`, care ia una din valorile `0` şi `1`, astfel încât asupra lui `i` se poate executa o incrementare cu `d`, iar asupra lui `j` o decrementare cu `1-d`.

```
program QuickSort;
const max=10;
type vector=array[1..max] of Integer; var A: vector;  i,n: Integer;
procedure Pozitioneaza(start,finis:Integer;
                       var k:Integer; var A:vector);
  procedure Schimba(var x,y: Integer);
  var aux: Integer; begin aux:=x; x:=y; y:=aux end;
var i,j,d: Integer;
begin
     d:=0; i:=start; j:= finis;
     while i<j do
```

```
            begin
              if A[i]>A[j] then
                 begin Schimba(A[i],A[j]); d:=1-d end;
              Inc(i,d); Dec(j,1-d)
            end;
     k:=i
end;
procedure Quick(inceput, sfarsit: Integer; var A: vector);
var k: Integer;
begin
     if inceput<sfarsit then
        begin
          Pozitioneaza(inceput, sfarsit,k,A);
          Quick(inceput,k-1,A); Quick(k+1,sfarsit,A)
        end
end;
begin
     WriteLn('Quick - sort');
     Write('Dati n = '); ReadLn(n);
     for i:=1 to n do
         begin
           Write('. A[',i,'] = ');
           ReadLn(A[i])
         end;
     Quick(1,n,A);
     WriteLn('Vectorul sortat este: ');
     for i:=1 to n do Write(A[i],', ');
     ReadLn
end.
```

7.6.7. Sortare prin interclasare



Algoritmul de **sortare prin interclasare** (numit şi merge-sort) constituie un exemplu reprezentativ pentru folosirea metodei Divide-et-imperaîn programare. Astfel, dacă avem de sortat un vector, atunci îl împărţim în două, sortăm - la fel - cele două părţi ale vectorului, apoi le interclasăm. Dacă şi vectorii rezultaţi după împărţire sunt destul de mari (mai mult decât un singur element), atunci procedăm la împărţirea şi a acestor vectori şi tot aşa.

Astfel, vom scrie o procedură

`SortInterclas(inceput,sfarsit:Integer)`

care va sorta vectorul `A` între poziţia `inceput` şi poziţia `sfarsit`.

Procedura va determina poziţia din mijloc şi se va autoapela pentru `inceput` şi `mijloc-1`, apoi pentru `mijloc+1` şi `sfarsit`, după care se vor interclasa cele două părţi ale vectorului.

```
program SortarePrinInterclasare;
const max=10;
var A: array[1..max] of Integer; i,n: 1..max;

procedure Interclaseaza(start,mijloc,finis: Integer);
var B: array[1..max] of Integer; i,j,k: Integer;
begin
  k:=start; i:=start; j:=mijloc+1;
  while (i<=mijloc) and (j<=finis) do
        if A[i]<A[j] then
           begin
             B[k]:=A[i];
             i:=i+1;
             k:=k+1
           end
```

```
         else
            begin
              B[k]:=A[j];
              j:=j+1;
              k:=k+1
            end;
  if i<=mijloc then
     for j:=i to mijloc do
         begin
           B[k]:=A[j];
           k:=k+1
         end
  else
     for i:=j to finis do
         begin
           B[k]:=A[i];
           k:=k+1
         end;
  for i:=start to finis do
      A[i]:=B[i]
end;
```

```
procedure SortInterclas(inceput,sfarsit: Integer);
var centru: Integer;
begin
  if inceput < sfarsit then
   begin
     centru:=(inceput+sfarsit) div 2;
     SortInterclas(inceput,centru);
     SortInterclas(centru+1,sfarsit);
     Interclaseaza(inceput,centru, sfarsit)
   end
end;
```

```
begin
  Write('n='); ReadLn(n);
  for i:=1 to n do
      begin
        Write('A[',i,']='); ReadLn(A[i])
      end;
  SortInterclas(1,n);
  for i:=1 to n do Write(A[i],',');
  ReadLn
end.
```

7.7. Alte probleme ale căror rezolvări se pot defini în termeni recursivi

7.7.1. Generarea partițiilor unei mulțimi

Se consideră o mulțime cu `n` elemente. Se cere să se determine toate partițiile mulțimii `A`. (A_1, A_2, ..., A_p este o partiție a mulțimii `A` dacă şi numai dacă reuniunea acestor mulțimi este `A` şi intersecția lor este vidă.).

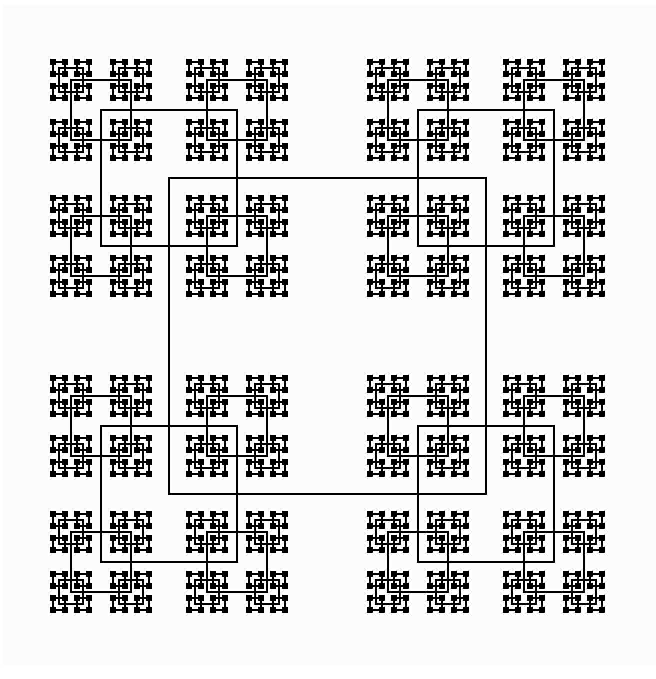
Programul de mai jos este o rescriere recursivă a programului prezentat în 2.2.11.

```
program PartitiiMultimeRecursiv;
const NrSol: Integer=0;
var x: array[1..20] of Integer;
    n: Integer;
```

```
procedure Scrie(max: Integer);
var i,j: Integer;
begin
     NrSol:=NrSol+1;
     WriteLn('Partitia nr. ',NrSol);
     for i:=1 to max do
         begin
           Write('{ ');
           for j:=1 to n do
               if x[j]=i then Write(j,' ');
           WriteLn('}')
         end;
     ReadLn
end;

procedure PartMult(k: Integer);
var alfa, max, i: Integer;
begin
     max:=0;
     for i:=1 to k-1 do
         if x[i]>max then max:=x[i];
     if k=n+1 then Scrie(max)
     else
        for alfa:=1 to max+1 do
            begin x[k]:=alfa; PartMult(k+1) end
end;

begin
     WriteLn('Generare partitiilor unei multimi (recursiv)');
     WriteLn('********************************************');
     Write('Dati n='); ReadLn(n);
     PartMult(1)
end.
```

7.7.2. Figuri recursive

Să încercăm să desenăm următoarea figură, pe care o vom numi “diamant”.



Un “diamant” este perfect caracterizat de coordonatele centrului său, precum şi de latura pătratului iniţial. Alte elemente nu mai sunt necesare, deoarece pătratele imediat următoare, au laturile de `c` ori mai mici ş.a.m.d., unde `c` este o constantă reală oarecare, `c≥1`.

De fapt, din ce constă un diamant? Dintr-un pătrat de centru `(x,y)` şi latură `l`, care are patru diamante în colţurile acestui pătrat.

Aceasta este o definiţie recursivă a diamantului. Ea ne va permite să scriem cu uşurinţă o procedură care să-l deseneze. Aceasta va desena un pătrat, apoi se va autoapela de patru ori, pentru cele patru diamante din colţuri. Însă nu putem merge aşa la infinit! De aceea, procedura va face efectiv desenări doar dacă latura pătratului este mai mare decât o anumită valoare minimă, de pildă `0`, `1` sau `2`.

```
program FiguraRecursiva;
uses Graph;
procedure OpenGraph;
var gd,gm: Integer;
begin
     gd:=0; InitGraph(gd,gm,'c:\bp\bgi')
end;
procedure Patrat(x,y,l: Integer);
var l2: Integer;
begin
  l2 := l div 2;
  Rectangle(x - l2, y - l2, x + l2, y + l2)
end;
{$S-}
procedure Diamant(x,y,l: Integer);
const c = 2.3;
var l2,l3: Integer;
begin
  if l>0 then
     begin
       Patrat(x,y,l);
       l2 := l div 2;
       l3 := Round(l/c);
       Diamant(x-l2,y-l2,l3);
       Diamant(x-l2,y+l2,l3);
       Diamant(x+l2,y-l2,l3);
       Diamant(x+l2,y+l2,l3)
     end
end;
{$S+}
begin
  OpenGraph;
  Diamant(GetMaxX div 2, GetMaxY div 2, GetMaxY div 3);
  ReadLn;
  CloseGraph
end.
```

7.7.3. Explorarea grafurilor în adâncime

La revederea după 10 ani de la terminarea liceului, mai mulţi foşti colegi să gândesc să sărbătorească evenimentul la un restaurant. organizatorii vorbesc cu ospătarii să unească mesele din salonul restaurantului în grupe. Astfel se vor forma nişte mese mai mari. La fiecare masă va sta câte un grup, constituit după următorul criteriu: cei care au stat, pe durata celor patru ani de liceu, în bancă unul cu altul, vor sta acum la aceeaşi masă. De câte mese mari va fi nevoie şi care vor fi grupurile de persoane ce vor lua loc la fiecare din aceste mese?

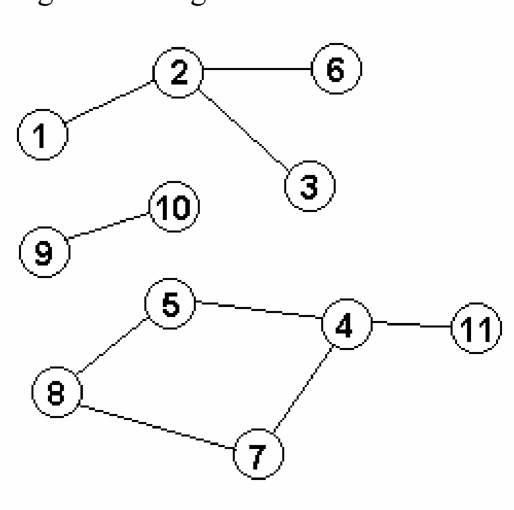
Practic, se poate defini (pe mulțimea participanților la revederea de 10 ani) o relație de echivalență (simetrică, reflexivă şi tranzitivă) astfel: persoana `x` este în relație cu persoana `y` dacă ele vor sta la aceeaşi masă în restaurant. Astfel, se formează un graf, în care nodurile sunt persoanele, între `x` şi `y` existând muchie, dacă `x` şi `y` au stat în aceeaşi bancă în liceu. Fireşte, va exista drum de la x la `y` dacă `x` stă la aceeaşi masă cu `y`. (Relația de "a sta la masă" este, de fapt, închiderea tranzitivă a relației de "a fi stat în aceeaşi bancă").

Problema cere, aşadar, să se determine componentele conexe ale grafului dat. Pentru aceasta vom folosi o procedură de explorare în adâncime a grafului, pornind de la un anumit nod. Algoritmul este recursiv.

```
program DFS_ComponenteConexe;
const max=10;
type Graf=array[1..max,1..max] of 0..1;
var Marcat: array[1..max] of Boolean;

procedure ExploreazaInAdincime(v: Integer;
            G: Graf; n: Integer);
var i: 1..max;
begin
     if not Marcat[v] then
        begin
          Marcat[v]:=True;
          Write(v,',');
          for i:=1 to n do
              if (G[v,i]=1) and (not Marcat[i]) then
                 ExploreazaInAdincime(i,G,n)
        end
end;

procedure DeterminaComponenteConexe(G: Graf; n: Integer);
var v,i: 1..max; NrCompConexe: Integer;
    ToateMarcate: Boolean;
begin
     for i:=1 to n do Marcat[i]:=False;
     v:=1; NrCompConexe:=0;
     repeat
           ExploreazaInAdincime(v,G,n);
           WriteLn;
           NrCompConexe:=NrCompConexe+1;
           ToateMarcate:=True;
           for i:=1 to n do
               if not Marcat[i] then
                  begin
                    v:=i; ToateMarcate:=False
                  end
     until ToateMarcate;
     WriteLn('Nr. componente conexe = ',NrCompConexe)
end;

var n,i,j: 1..max;
    G: Graf;

begin
     Write('n='); ReadLn(n);
     for i:=1 to n-1 do
         for j:=i+1 to n do
             begin
               Write('G[',i,',',j,']=');
               ReadLn(G[i,j]);
               G[j,i]:=G[i,j]
             end;
```

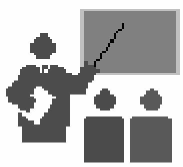
```
    DeterminaComponenteConexe(G,n);
    ReadLn
end.
```

Exemplu: Să considerăm graful din figura următoare.



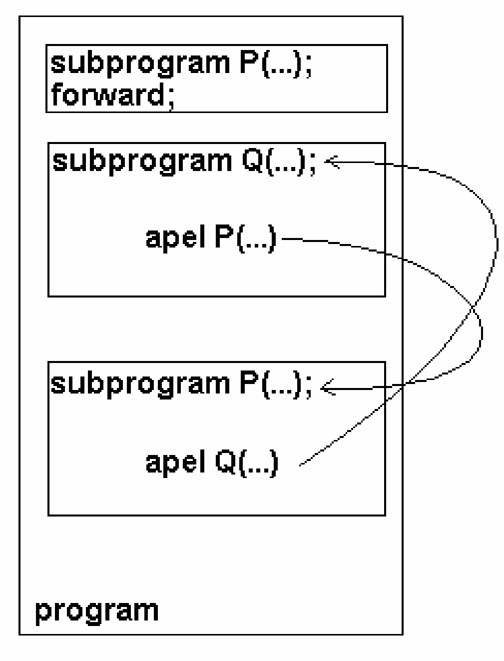
Mai întâi se va marca nodul `1` şi se va explora în adâncime (`1`, `2`, `3`, `6`) componenta conexă din care face parte acest nod. După aceasta, se va determina primul nod (ca număr de ordine) nemarcat încă. Acesta este `4`. Se va parcuge în adâncime graful din acest nod, obținându-se componenta conexă din care face parte `4`: `4`, `5`, `8`, `7`, `11`. Apoi se va obține şi componenta conexă `9`,`10`, după acelaşi procedeu. În acest moment, nu vor mai exista noduri nemarcate, prin urmare procedura `DeterminaComponenteConexe` se termină.

7.8. Recursivitate indirectă. Directiva forward



- Toate subprogramele recursive prezentate până acum se autoapelau în mod *direct*, motiv pentru care se spune că s-a folosit **recursivitatea directă**.
- Există, însă, şi situații când un subprogram `P` va apela la el însuşi din cadrul altui subprogram `Q`, apelat de `P`. Asfel, cele două subprograme se apelează reciproc. Conform regulilor de sintaxă şi topică ale limbajului, cum `P` apelează pe `Q`, înseamnă că `Q` va trebui să apară înaintea lui `P`. Or şi `Q` apelează pe `P`, iar pentru a soluționa acest impas, se declară unul din subprograme înaintea celuilalt: se scrie doar antetul său, urmat de cuvântul rezervat **forward**. Apoi, când se defineşte subprogramul declarat înainte ("forward"), în scrierea antetului se poate renunța la lista parametrilor.

Spunem că avem de a face cu o **recursivitate indirectă** sau **încrucişată**.



Iată un exemplu:

```
program RecursivitateIndirecta;
function F(x: Integer): Integer; forward;
function G(x: Integer): Integer;
begin
  if x=0 then
     G:=1
  else
     G:=G(x-1)+F(x-1)
end;
function F;
begin
  if x=0 then
     F:=1
  else
     F:=F(x-1)+G(x-1)
end;
begin
  WriteLn(F(5));
  ReadLn
end.
```

Programul va afişa valoarea 32.

În atenția profesorului

Exemplul prezentat este pur teoretic. Propunem să se încerce evitarea acestui gen de recursivitate, deoarece de multe ori duce la programe mai greu de urmărit.

În continuare vom prezenta niște exemple de utilizare a rrecursivității indirecte.

7.8.1. Şirul mediilor aritmetico-geometrice al lui Gauss.

Se cere să se calculeze, pentru un n dat, a_n şi b_n definite astfel:

$a_1 = a;\ b_1 = b;$

$a_n = (a_{n-1} + b_{n-1}) / 2$

$$b_n = Sqrt(a_{n-1}*b_{n-1})$$

Soluția este dată de următorul program:

```
program SiruriDefiniteRecursiv;
var a,b: Real; n: Integer;

function b_n(n: Integer): Real; forward;

function a_n(n: Integer): Real;
begin
     if n=0 then a_n:=a
     else a_n:=(a_n(n-1)+b_n(n-1))/2
end;

function b_n(n: Integer): Real;
begin
     if n=0 then
        b_n:=b
     else b_n:=Sqrt(a_n(n-1)*b_n(n-1))
end;
begin
     Write('Dati a: '); ReadLn(a);
     Write('Dati b: '); ReadLn(b);
     Write('Dati n: '); ReadLn(n);
     WriteLn('a(',n,')=',a_n(n):0:10);
     WriteLn('b(',n,')=',b_n(n):0:10);
     ReadLn
end.
```

7.8.2. Deplasarea pe ecran a unui text.

Propunem următoarea problemă pentru lo lecție desfăşurată în laboratorul de informatică. Se va folosi recursivitatea indirectă în rezolvarea acestei probleme.

Să se simuleze deplasarea pe ecran a unei linii de text între partea de sus şi cea de jos a ecranului, folosind o procedură pentru deplasarea în jos şi una pentru deplasarea în sus. În momentul atingerii uneia dintre margini, linia îşi continuă deplasarea în sens contrar.

7.8.3. Transformarea unei expresii aritmetice în forma poloneză prefixată

Se consideră o expresie aritmetică corectă, cu paranteze rotunde, cuprinzând operatori aritmetici binari din multimea `{+, -, *, /}` şi operanzi reprezentați prin litere mici din alfabet. Să se transforme expresia în forma poloneză postfixată.

De exemplu, forma postfixată pentru expresia `a+b*c-d/e-(f-g)` este: `abc*+de/-fg--`.

Să observăm că o expresie este de forma:

$$t_1 \ oa_1 \ t_2 \ oa_1 \ \ldots \ oa_{n-1} \ t_n,$$

unde oa_i sunt operatori aditivi (+ sau −), iar t_i sunt termeni.

Un termen are forma

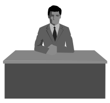
$$f_1 \ om_1 \ f_2 \ om2 \ \ldots \ om_{p-1} \ f_p,$$

unde om_i sunt operatori multiplicativi (`*` sau `/`), iar f_i sunt factori.

Un factor este fie un caracter, fie o expresie intre paranteze.

Din aceste definiții se deduce evident o modalitate de rezolvare a problemei în cauză folosind tehnica recursivității indirecte.

```
{$X+}
program FormaPostfixataExpresie;
uses Crt;
var c: Char;
procedure Citire;
begin
     if not EoLn then Read(c)
end;
procedure Factor; forward;
procedure Termen; forward;
procedure Expresie;
var op: Char; { operator aditiv }
begin
     Termen;
     while c in ['+','-'] do
           begin
             op:=c; Citire;
             Termen; Write(op)
           end
end;
procedure Termen;
var op: Char;
begin
     Factor;
     while c in ['*','/'] do
           begin
             op:=c; Citire;
             Factor; Write(op)
           end
end;
procedure Factor;
begin
     if c='(' then begin Citire; Expresie end
     else Write(c);
     Citire
end;
begin
     WriteLn('Dati expresia'); Citire;
     WriteLn('Expresia in forma postfixata'); Expresie;
     ReadKey
end.
```



În atenția profesorului

Se vor urmări: a) formarea la alevi a deprinderilor necesare utilizarii functiilor recursive; b) dezvoltarea abilitatilor de a identifica situatii în care varianta recursiva este preferabila celei nerecursive sau invers; c) dezvoltarea capacitatii de identificare a problemelor care necesita utilizarea recursivitatii indirecte.

Probleme

1 💣 Să se determine exponentul la care apare numărul prim p în descompunerea în factori primi a numărului $N=1\times 2\times 3\times \ldots \times n$, pentru un număr n natural dat.
2 ☺ Se dau două numere naturale n şi k. Să se calculeze n^k. Se va ține cont de faptul că $n^k = (n^{k/2})^2$, dacă k este par, respectiv $n(n^{(k-1)/2})^2$, dacă k este impar.
3 ☹ Să se determine dacă un şir de caractere este sau nu palindrom.
4 💣 Se dă o bucată de tablă de formă dreptunghiulară cu lungimea L şi înălțimea H, având pe suprafața ei N găuri de coordonate numere întregi. Se cere să se decupeze din ea o bucată de arie maximă care nu prezintă găuri. Sunt permise numai tăieturi orizontale şi verticale.

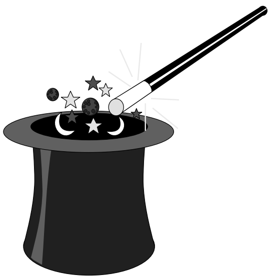
5 😐 Scrieți un program în care calculatorul să ghicească un număr natural ale de dumneavoastră (numărul este cuprins între `1` şi `30000`). Atunci când calculatorul vă propune un număr, îi veți răspunde precizând dacă numărul a fost ghicit, dacă este mai mic sau mai mare.
6 ☹ Se consideră un vector cu `n` componente, numere naturale. Definim plierea vectorului ca fiind suprafața unei jumătăți, numită donatoare, peste o alta, numită receptoare. În cazul în care vectorul are un număr impar de componente, cea din mijloc este eliminată. În acest fel se ajunge la un vector ale cărui elemente au numerotarea jumătății receptoare. De exemplu, vectorul (`1,2,3,4,5`) se poate plia în două moduri: (`1,2`) şi (`4,5`). Plierea se aplică în mod repetat, până se ajunge la un vector cu o singură componentă, numită element final. Să se precizeze care sunt elementele finale şi care este şirul de plieri prin care se poate ajunge la ele.
7 ☹ Se dă un vector cu `n` componente la început nule. O secțiune pe poziția `k` va incrementa toate elementele aflate în zona de secționare anterioară situate între poziția `1` şi `k`.
Exemplu: (`0,0,0,0,0,0,0`), se secționează pe poziția `4`;
(`1,1,1,1,0,0,0`), se secționează pe poziția `1`;
(`2,1,1,1,0,0,0`), se secționează pe poziția 3;
(`3,2,2,1,0,0,0`), etc.

Să se determine o ordine de secționare a unui vector cu `n` elemente astfel încât suma elementelor sale să fie `s`.
8 💣 Se dau $n=2^m$ puncte în plan. Să se găsească distanța minimă între toate perechile de puncte date. Se va folosi metoda Divide-et-impera: se va împărți mulțimea de puncte în două submulțimi cu `n/2` puncte, se va găsi distanța minimă pentru fiecare submulțime şi apoi se va determina soluția finală.

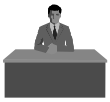
Rezumat

1. O tehnică generală de elaborare a algoritmilor şi foarte elegantă este recursivitatea. Ea este reprezentată de rezolvarea unei probleme mai mari prin autoapelul procedurii de rezolvare a acelei probleme pentru dimensiuni mai mici. Aceasta este recursivitatea directă.

2. O altă modalitate este de a folosi proceduri (funcții) care se apelează reciproc. Avem de a face, în acest caz, cu recursivitatea indirectă. (Se foloseşte directiva `forward`.).
3. Metoda Divide-et-impera este o modalitate de rezolvare a problemelor prin descompunerea lor în probleme mai mici, care fie se pot rezolva direct, fie se descompun, la rândul lor, în alte probleme şi mai mici (subprobleme). Soluția unei probleme se obține din combinarea soluțiilor subproblemelor sale.
4. Metoda Divide-et-impera poate fi reprezentată de cele mai multe ori sub o formă recursivă, dar şi sub formă iterativă.
5. Exemple clasice de folosire a metodei Divide-et-impera în rezolvarea de probleme sunt: căutarea binară, sortarea prin partiționare, sortarea prin interclasare, problema turnurilor din Hanoi.
6. Metoda Back-tracking poate fi exprimată şi sub formă recursivă. Întoarcerea dintr-un apel recursiv reprezentând revenirea la o componentă anterioară a vectorului rezultat. Apelarea recursivă a procedurii se face după ce se verifică condițiile de continuare şi când acestea sunt îndeplinite. Avem de a face cu trecerea la noua componentă sau afişarea soluției (când am ajuns la capăt).
7. Probleme clasice care pot fi soluționate prin această metodă sunt: problema damelor, generarea partițiilor unui număr şi altele.
8. La unele probleme rezolvabile prin metoda Back-tracking clasică (iterativă sau recursivă), soluția se exprimă greoi, ca şi algoritmul, de altfel. De aceea s-a dezvoltat metoda Back-tracking în plan, unde soluția se exprimă sub forma unei matrice. Probleme clasice în a căror rezolvare se foloseşte această metodă sunt: problema ieşirii dintr-un labirint, problema umplerii unei suprafețe închise.



Capitolul 8. Metoda Greedy



În atenția profesorului

Pe parcursul acestui capitol, se vor urmări: a) formarea la studenți a deprinderilor de a identifica problemele care necesită utilizarea acestei metode; b) formarea deprinderilor de a rezolva comparativ unele probleme folosind metoda Greedy şi alte metode, punând în evidență eficacitatea acestei metode.

8.1. Prezentare generală

Această metodă se foloseşte în problemele în care, dată fiind o mulțime `A` cu `n` elemente, se cere să se determine o submulțime `B` a sa, care să îndeplinească anumite condiții (eventual un anumit criteriu de optim).

Metoda Greedy este următoarea:

- se inițializează mulțimea `B` la mulțimea vidă;
- se alege un anumit element din `A`;
- se verifică dacă elementul ales poate fi adăugat mulțimii `B`;
- procedeul continuă aşa, repetitiv, pînă ce au fost determinate toate elementele din `B`.

Cu această tehnică de programare, foarte utilizată dealtfel, v-ați întâlnit şi în clasa a IX-a, la *Algoritmi şi limbaje de programare*, unde au fost prezentate suficiente exemple. Există şi o serie de probleme celebre, în a căror rezolvare se poate folosi această strategie: *problema spectacolelor, problema continuă a rucsacului, problema comisului-voiajor, algoritmul lui Dijkstra pentru drumuri minime*.

8.2. Probleme pentru care metoda Greedy determină soluția optimă

8.2.1. Maximizarea/minimizarea valorii unei expresii

Se dau n numere întregi nenule $b_1, b_2, ..., b_n$ şi m numere întregi nenule $a_1, a_2, ..., a_m$. Să se determine un subşir al şirului $b_1, b_2, ..., b_n$ care să maximizeze/minimizeze valoare expresiei:

$$E = a_1 * x_1 + a_2 * x_2 + ... + a_m * x_m$$

ştiind că $n > m$ şi că $x_i \in \{b_1, b_2, ..., b_n\}$, $b_i>0$, $\forall i=1,n$.

Algoritmul de rezolvare este următorul: se ordonează cei doi vectori (`a` şi `b`) (cele două mulțimi de numere) crescător `a` şi `b`. Apoi se grupează cele mai mari elemente pozitive din `a` cu elementele cele mai mari din `b` şi elementele cele mai mici din `a` cu elementele cele mai mici din `b`.

```
program MaximizareExpresie;
type vector=array[1..10] of Integer;
var a,b,c: vector;
    m,n,i,j,k: Integer;
    expr: Integer;
procedure Schimba(var a,b: Integer);
var aux: Integer;
begin
     aux:=a; a:=b; b:=aux
end;
begin
     WriteLn('Maximizarea unei expresii');
     WriteLn('************************');
     Write('Dati m: '); ReadLn(m);
     for i:=1 to m do
         begin
           Write('Dati a[',i,']=');
           ReadLn(a[i])
         end;
     Write('Dati n: '); ReadLn(n);
     for i:=1 to n do
         begin
           Write('Dati b[',i,']=');
           ReadLn(b[i])
         end;
     { ordonez crescator a si b }
     for i:=1 to m-1 do
         for j:=i+1 to m do
             if a[i]>a[j] then Schimba(a[i],a[j]);
     for i:=1 to n-1 do
         for j:=i+1 to n do
             if b[i]>b[j] then Schimba(b[i],b[j]);
     { acum se iau elementele pozitive din b si se grupeaza
       cu elementele cele mai mari din a }
     i:=n; j:=m; expr:=0;
     while (a[j]>0) and (j>0) do
           begin
             Write(b[i],'*',a[j],' + ');
             expr:=expr+b[i]*a[j];
             i:=i-1; j:=j-1
           end;
     k:=j;
     i:=1; j:=1;
     while k>0 do
           begin
             Write(b[i],'*',a[j],' + ');
             expr:=expr+b[i]*a[j];
             i:=i+1; j:=j+1; k:=k-1
           end;
     WriteLn(' => ',expr);
     ReadLn
end.
```

8.2.2. Problema spectacolelor

Într-o sală, într-o zi, trebuie planificate `n` spectacole. Pentru fiecare spectacol se cunoaşte ora de începere (`start[i]`) şi durata spectacolului (`durata[i]`). Se cere să se planifice un număr maxim de spectacole astfel încât să nu se suprapună.

Să considerăm `A` = mulțimea inițială de spectacole şi `B` = mulțimea spectacolelor ce vor fi alese.

Pentru a rezolva problema prin tehnica Greedy, prelucrarea care se va face asupra mulțimii `A` este o ordonare crescătoare după ora de finalizare.

Apoi se iau spectacolele în ordine, astfel încât fiecare spectacol să înceapă după ce s-a terminat cel anterior lui.

Exemplu: Numărul total de spectacole `n=6`, cu cei doi vectori: `start=(2,4,1,3,6,8)` şi `durata=(1,2,2,2,1,3)`. Se vor obține orele de terminare a spectacolelor: `(4,6,3,5,7,11)`.

În urma sortării după criteriul orelor de terminare a spectacolelor, se va obține ordinea: `3,1,4,2,5,6`.

Se ia spectacolul `3` (inițial şi nu după ordonare!), care se termină la ora `3`. Urmează spectacolul `1`, care începe, însă, la ora `2`, deci se sare peste el. Următorul este spectacolul `4`, care începe la ora `3` şi se termină la ora `5`. Urmează spectacolul `2`, care nu se ia. În schimb, se ia spectacolul `5`, care începe la ora `6`, oar apoi spectacolul `6`. Ora de terminare a tututor spectacolelor va fi, aşadar, `11`.

```
program ProblemaSpectacolelor;
var n: Integer; { numarul de spectacole }
    start,durata: array[1..10] of Integer; { caracteristicile }
    i,j: Integer;
    a: array[1..10] of Integer; { permutarea spectacolelor }
    ora_sf: Integer; { ora ultimului spectacol }

procedure Schimba(var x,y: Integer);
var aux: Integer;
begin
     aux:=x; x:=y; y:=aux
end;

begin
     WriteLn('Problema spectacolelor');
     WriteLn('**********************');
     Write('Dati numarul de spectacole: '); ReadLn(n);
     for i:=1 to n do
         begin
           WriteLn('Spectacolul nr. ',i,':');
           Write(' - ora de incepere: '); ReadLn(start[i]);
           Write(' - durata lui     : '); ReadLn(durata[i]);
           a[i]:=i {permutarea identica}
         end;
     { se ordoneaza spectacolele
       crescator dupa ora de terminare }
     for i:=1 to n-1 do
         for j:=i+1 to n do
             if start[i]+durata[i] > start[j]+durata[j] then
                begin
                  Schimba(start[i],start[j]);
                  Schimba(durata[i],durata[j]);
                  Schimba(a[i],a[j])
                end;
     { se iau spectacolele in ordine,
       astfel incit fiecare spectacol sa
       inceapa dupa ce s-a terminat cel anterior lui }
     WriteLn('Solutie:');
     ora_sf:=start[1]+durata[1];
     WriteLn('Spectacolul ',a[1]);
     i:=2;
     while i<=n do
           begin
```

```
            if ora_sf<=start[i] then
               begin
                 WriteLn('Spectacolul ',a[i]);
                 ora_sf:=start[i]+durata[i];
               end;
            i:=i+1
          end;
     ReadLn
end.
```

8.2.3. Problema continuă a rucsacului

Cu ajutorul unui rucsac de greutate maximă admisibilă `GG` trebuie să se transporte o serie de obiecte din `n` disponibile, având greutățile G_1, G_2, ..., G_n, aceste obiecte fiind de utilitățile C_1, C_2, ..., C_n. Dacă pentru orice obiect `i` putem să luăm doar o parte $x_i \in [0,1]$ din el, atunci spunem că avem **problema continuă a rucsacului**, pe care o vom prezenta aici, iar dacă obiectul poate fi luat doar în întregime sau nu, spunem că avem **problema discretă a rucsacului**, pe care am prezentat-o la metoda Back-tracking.

În problema continuă a rucsacului, prin raportarea utilităților la greutăți obținem utilitățile pe unitate de greutate, astfel încât va trebui să ordonăm obiectele în funcție de aceste raporturi și să le încărcăm, pe cât posibil, în întregime în rucsac, până când acesta se umple. Astfel, reprezentând soluția în vectorul `x`, vom pune la început `x[i]:=1`, până când greutatea rămasă disponibilă, notată `GGr`, nu mai permite punerea unui obiect în întregime. Atunci, vom face `x[i]:=GGr/G[i]`, ultimul obiect fiind "tăiat".

Programul de mai jos rezolvă această problemă; mai întâi ordonează obiectele, apoi le afişează ordonate, după care aplică metoda descrisă, la fiecare pas actualizând greutatea rămasă disponibilă `GGr`, după formula: `GGr:=GG-G[i]`.

```
program Rucsac;
const max=5;
var n,i,j: Integer;
    C,G,X: array[1..max] of Real;
    GG,GGr, aux: Real;
    a: array[1..max] of Integer;
    aux2: Integer;

begin
     Write('Nr. obiecte = '); ReadLn(n);
     for i:=1 to n do
         begin
           Write('C[',i,']='); ReadLn(C[i]);
           Write('G[',i,']='); ReadLn(G[i]);
           a[i]:=i
         end;
     Write('Greut. max. = '); ReadLn(GG);
     for i:=1 to n-1 do
         for j:=i+1 to n do
             if C[j]/G[j] > C[i]/G[i] then
                begin
                  aux:=C[j]; C[j]:=C[i]; C[i]:=aux;
                  aux:=G[j]; G[j]:=G[i]; G[i]:=aux;
                  aux2:=a[i]; a[i]:=a[j]; a[j]:=aux2
                end;
     WriteLn('Am ordonat ...');
     for i:=1 to n do
         WriteLn('C[',a[i],']=',C[i]:5:2,
                '  G[',a[i],']=', G[i]:5:2,
                ' -> ',C[i]/G[i]:5:2);
```

```
     GGr:=GG; i:=1;
     while (i<=n) do
           if GGr > G[i] then
              begin
                X[i]:=1; Gr:=GGr-G[i]; i:=i+1
              end
           else
              begin
                X[i]:=GGr/G[i];
                for j:=i+1 to n do X[j]:=0;
                i:=n+1
              end;
     for i:=1 to n do
         WriteLn('X[',a[i],']=',X[i]:5:2);
     ReadLn
end.
```

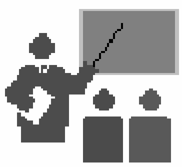
Exemplu:

```
Nr. obiecte = 4
C[1]=3
G[1]=4
C[2]=2
G[2]=7
C[3]=4
G[3]=2
C[4]=1
G[4]=2
Greut. max. = 7
Am ordonat ...
C[3]= 4.00  G[3]= 2.00 ->  2.00
C[1]= 3.00  G[1]= 4.00 ->  0.75
C[4]= 1.00  G[4]= 2.00 ->  0.50
C[2]= 2.00  G[2]= 7.00 ->  0.29
X[3]= 1.00
X[1]= 1.00
X[4]= 0.50
X[2]= 0.00
```

Aşadar, în urma ordonării după câştig pe unitatea de greutate, se obţine permutarea: `(3,1,4,2)`. Obiectul `3` şi obiectul `1` se iau în întregime, din cel de al patrulea se ia jumătate, iar obiectul al doilea nu se ia deloc.

Propunem ca exerciţiu să modificaţi programul astfel încât să se determine şi câştigul maxim.

8.2.4. Algoritmul lui Dijkstra pentru drumuri de cost minim în grafuri



Dându-se un (di)graf `G=(V,E)` şi o funcţie de cost $C:E\rightarrow R$ ataşată muchiilor, se cere să se determine drumurile minime (de cost minim) de la un nod i_0 la toate nodurile din graf, precum şi costurile acestor drumuri.

Pentru rezolvarea problemei se foloseşte metoda Greedy: se selectează nodurile grafului, unul câte unul, în `n-1` paşi, în ordinea crescătoare a costului drumului de la nodul de start la ele, într-o mulţime care iniţial conţine doar nodul de start.

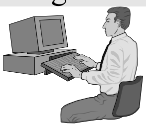
În implementarea algoritmului se foloseşte un vector `Anterior` (cu legături de tip "tata") cu semnificaţia: dacă `Anterior[i]=k` atunci `k` este nodul anterior nodului `i` pe drumul minim de la i_0 (nodul de start) la nodul `i`.

De asemenea, se utilizează doi vectori cu `n` componente, `d` şi `Selectat`:

- `d[i]`=costul minim al drumului de la i_0 la `i`;
- `Selectat[i]=True` ⇔ nodul `i` este selectat.

Algoritmul este:

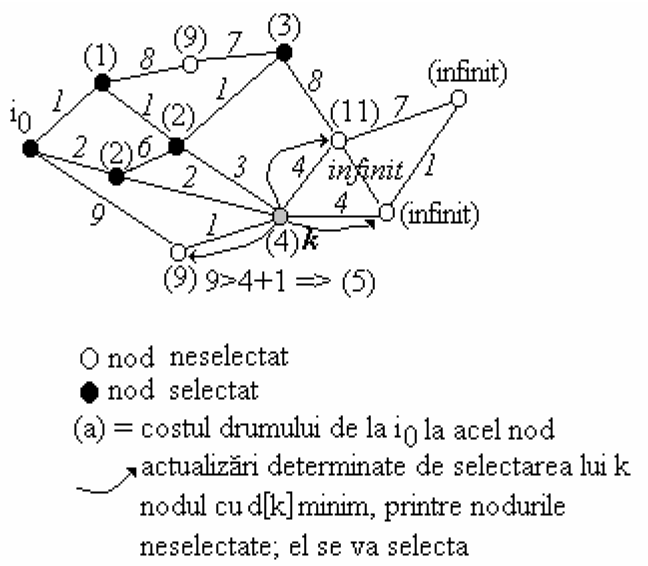
- La început, se selectează i_0.
- La fiecare pas p din cei `n-1` paşi:
  * se caută nodul `i` neselectat cu `d[i]` minim, fie acesta `k` şi se selectează;
  * se actualizează vectorul `d` pentru acele noduri `i` neselectate, dar pentru care fostul `d[i]` este mai mare decât `d[k]`+costul muchiei `(i,k)`; astfel `d[i]` devine `d[k]+C[k,i]`, iar `Anterior[i]:=k`. Aşadar, drumul de la i_0 la `i` are ca nod intermediar (exact înainte de `i`) pe nodul `k`.
- La sfârşit, folosind vectorul `Anterior`, se afişează drumurile de la i_0 la fiecare nod `i` al grafului, precum şi costurile acestor drumuri.



```
program Dijkstra;
const max=10; infinit = 1000;
var C: array[1..max,1..max] of Integer; { matricea costurilor }
    i0: Integer; { nodul de start }
    d: array[1..max] of Integer;
    { d[i] = costul drumului de la i0 la i, la un moment dat }
    Selectat: array[1..max] of Boolean;
    { Selectat[i]=True <=> nodul i este
      in multimea nodurilor parcurse }
    Anterior: array[1..max] of Integer;
    { Anterior[i] = nodul anterior nodului i,
                    pe drumul de la i0 la i }
    min: Integer; pas,i,j,k,n: 1..max; gata: Boolean;

{$S-}
procedure Drum(i: Integer);
{ afiseaza drumul de la i0 la i, recursiv }
begin
     if Anterior[i]<>0 then
        begin
          Drum(Anterior[i]);  Write(',',i)
        end
     else Write(i)
end;
{$S+}

begin { citirea grafului }
  Write('Nr. virfuri = '); ReadLn(n);
  for i:=1 to n-1 do
      begin
        C[i,i]:=0;
        for j:=i+1 to n do
            {pentru digrafuri se va pune "for j:=1 to n do"}
            begin
              Write('C[',i,',',j,']='); ReadLn(C[i,j]);
              if C[i,j]=0 then C[i,j]:=infinit;
              C[j,i]:=C[i,j] { nu e valabila la digrafuri ! }
            end
      end;
  Write('i0 ='); ReadLn(i0); { citirea nodului de start }
  for i:=1 to n do
      begin
       { la inceput nici un nod nu e selectat }
        Selectat[i]:=False;
        { se considera costul drumului de la i0 la i
          ca fiind costul muchiei de la i0 la i }
        d[i]:=C[i0,i];
        { daca acesta e un numar finit, atunci nodul
          anterior lui i se considera i0, altfel 0 }
        if d[i]<infinit then Anterior[i]:=i0
```

```
        else Anterior[i]:=0
      end;
      { se selecteaza nodul de start i0, care nu are
        anterior (0), iar costul drumului de la i0 la el
        insusi este 0 }
  Selectat[i0]:=True; Anterior[i0]:=0; d[i0]:=0;
  for pas:=1 to n-1 do
      begin
        { printre nodurile neselectate,
          se cauta cel aflat la
          distanta minima fata de i0 si se selecteaza }
        min:=infinit;
        for i:=1 to n do
            if (not Selectat[i]) and (d[i]<min) then
               begin min:=d[i]; k:=i end;
        {k este nodul selectat, deci se adauga multimii}
        Selectat[k]:=True;

        for i:=1 to n do
            if (not Selectat[i])
                and (d[k]+C[k,i]<d[i]) then
            begin
              d[i]:=d[k]+C[k,i];
              Anterior[i]:=k
            end
      end;
  { se afiseaza drumurile de la i0
    la fiecare nod i si costurile lor }
  for i:=1 to n do
      begin
        WriteLn('Drumul minim de la ',i0,' la ',i,':');
        Drum(i); WriteLn;
        WriteLn('Costul sau este: ',d[i]); WriteLn
      end;
  ReadLn
end.
```

Exemplu:



8.2.5. Arborele parțial de cost minim

Problema determinării arborelui parțial de cost minim dintr-un graf poate fi formulată sub forma unei probleme practice astfel:

Pentru construirea unei rețele interne de comunicație între secțiile unei întreprinderi s-a întocmit un proiect în care au fost trecute toate legăturile ce se pot realiza între secțiile întreprinderi. În vederea definitivării proiectului și întocmirea necesarului de materiale etc., se cere să se determine un sistem de legături ce trebuie construit, astfel încât orice secție a întreprinderii să fie racordată la această rețea de comunicație, iar cheltuielile de construcție să fie minime.

Algoritmul de rezolvare pe care îl folosim se bazează pe tehnica Greedy. La început se ia nodul 1 şi se pune în arbore. Apoi, la fiecare din cei `n-1` paşi (arborele va avea `n-1` muchii) se ia muchia de cost minim printre muchiile cu o extremitate în arborele deja creat şi cu alta în afara acestuia, apoi această muchie se adaugă arborelui.

Pentru o alegere simplă a celei mai mici muchii cu respectiva proprietate, se ordonează mai întâi muchiile crescător după costuri. De aceea, chiar şi graful va fi dat sub forma unui şir de muchii, fiecare fiind caracterizată de extremitățile şi de costul ei.

Verificarea sau stabilirea faptului că un anumit nod se află sau nu în arbore se va face cu ajutorul unui vector numit `Marcat`, de valori booleene.

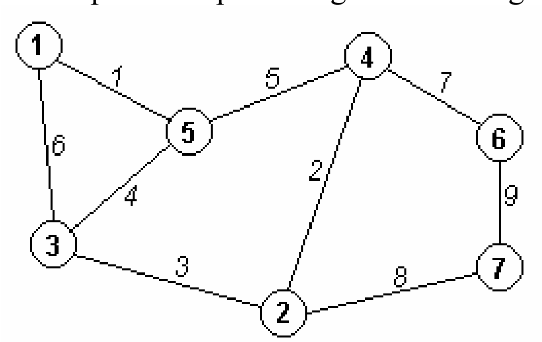
```
program ArborelePartialDeCostMinim;
type muchie=record
                  u,v: Integer; { extremitatile muchiei }
                  c: Integer; { costul muchiei }
            end;
var graf: array[1..20] of muchie;
    n,m: Integer; { nr. de noduri/muchii }
    Marcat: array[1..20] of Boolean;
    i,j: Integer;
    cost_min: Integer;
procedure Schimba(var a,b: muchie);
var aux: muchie;
begin
     aux:=a; a:=b; b:=aux
end;

begin
     WriteLn('Arborele partial de cost minim');
     WriteLn('******************************');
     Write('Nr. de noduri: '); ReadLn(n);
     Write('Nr. de muchii: '); ReadLn(m);
     for i:=1 to m do
         with graf[i] do
              begin
                WriteLn('Dati datele muchiei a ',i,'-a !');
                Write('primul nod: '); ReadLn(u);
                Write('al doilea : '); ReadLn(v);
                Write('costul    : '); ReadLn(c)
              end;
     WriteLn('Arborele este: ');
     cost_min:=0;
     { se ordoneaza muchiile crescator dupa cost }
     for i:=1 to m-1 do
         for j:=i+1 to m do
             if graf[i].c > graf[j].c then
                Schimba(graf[i],graf[j]);
     { la inceput nici un nod nu e marcat }
     for i:=1 to m do
         Marcat[i]:=False;
     { marcam nodul 1 }
     Marcat[1]:=True;
     { arborele va avea n-1 muchii }
     for i:=1 to n-1 do
         begin
           { se determina prima muchie
             cu o extremitate marcata si alta nu }
           j:=1;
           while not (Marcat[graf[j].u]
```

```
                        xor Marcat[graf[j].v]) do
                    j:=j+1;
              { nodul j exista sigur, 1<=j<=n ! }
              Marcat[graf[j].u]:=True;
              Marcat[graf[j].v]:=True;
              WriteLn('Muchia ',graf[j].u,'-',graf[j].v,
                      ' de cost ',graf[j].c);
              cost_min:=cost_min+graf[j].c
            end;
      WriteLn('Cost minim = ',cost_min);
      ReadLn
end.
```

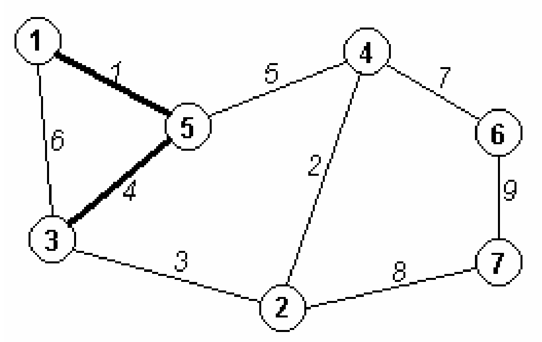
Observație

Algoritmul prezentat este o variantă a algoritmului lui Prim, în care graful este dat prin lista de muchii şi nu prin matricea de adiacență.

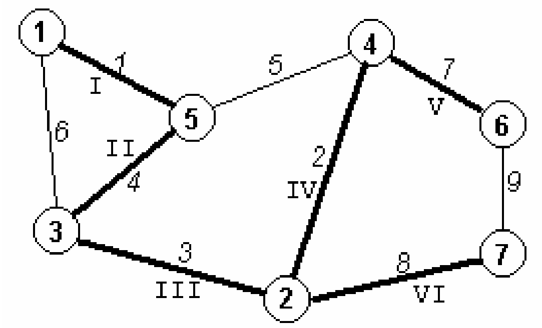
Să exemplificăm algoritmul prezentat pe cazul grafului din figură:



Mai întâi se va marca nodul `1` şi apoi se va lua muchia `(1,5)`. Următoarea muchie care se va lua va fi muchia `(5,3)`, deoarece ea este cea mai ieftină printre cele care au un nod marcat (luat în arbore) şi altul nu (nodul `5` este deja în arbore, iar nodul `3` nu).



Se procedează aşa mai departe, până se obține arborele parțial evidențiat în figura următoare, cu costul minim = `25`. (Am notat cu cifre romane ordinea în care se iau muchiile, conform algoritmului descris).



? Întrebări şi exerciţii

1 ☹ Având un număr nelimitat din fiecare dintre monezil de 1, 5 şi 25 de unităţi, să se dea rest unui client folosind un număr cât mai mic de monezi.

2 💣 Având un număr nelimitat din fiecare dintre monezile k_1, k_2, ..., k_p unităţi, să se dea restul unui client folosind un număr cât mai mic de monezi.

3 ☺ Se dă o mulţime $X=\{x_1, x_2, ..., x_n\}$ cu elemente reale. Se cere să se determine o submulţime Y a sa astfel încât suma elementelor acestei submulţimi să fie maximă.

8.3. Probleme pentru care metoda Greedy nu determină soluția optimă

8.3.1. Problema comis-voiajorului

Un comis-voiajor trebuie să treacă prin `n` oraşe. Se cere să se determine un traseu care trece prin toate oraşele o singură dată şi revine în oraşul de plecare. De asemenea, se cere ca şi costul călătoriei sale să fie minim.

Este, de fapt, vorba despre problema unui circuit hamiltonian de cost minim într-un graf dat prin matricea costurilor muchiilor sale.

Algoritmul Greedy pe care l-am implementat în programul următor nu conduce la o soluție optimă, ci la una cu un cost destul de redus al circuitului determinat.

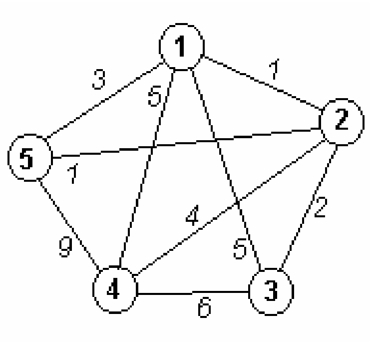
La început se pleacă dintr-un anumit nod `np`. Fie nodul curent numit `nod`. Apoi se alege muchia de cost minim cu o extremitate în `nod` şi cealaltă într-un alt nod din graf, prin care nu s-a mai trecut. Acesta devine noul nod curent.

Faptul că sa trecut sau nu printr-un anumit nod este dat de un vector de valori logice, numit `Marcat`.

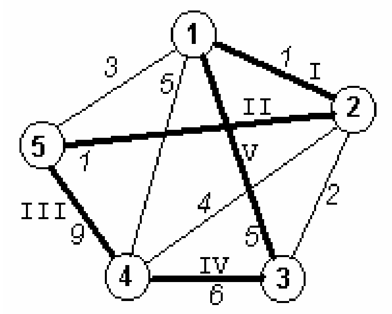
```
program ComisVoiajor;
{ circuit hamiltonian de cost minim }
const infinit=MaxInt;
var Cost: array[1..20,1..20] of Integer;
    n, np, nod: Integer; { nr. de noduri, nodul de plecare, nodul curent }
    v,i,j: Integer; { v = nodul ales}
    Marcat: array[1..20] of Boolean;
    min,cost_total: Integer;
begin
     Write('Dati nr. de orase: '); ReadLn(n);
     for i:=1 to n-1 do
         for j:=i+1 to n do
             begin
               Write('Dati costul de la ',i,
                     ' la ',j,' [infinit=0] ');
               ReadLn(Cost[i,j]);
               if Cost[i,j]=0 then Cost[i,j]:=infinit;
               Cost[j,i]:=Cost[i,j]
             end;
     for i:=1 to n do
         begin Cost[i,i]:=0; Marcat[i]:=False end;
     Write('Dati nodul de plecare: '); ReadLn(np);
     WriteLn('Un circuit hamiltonian cu cost redus este: ');
     Marcat[np]:=True;
     { se alege nodul de start si se afiseaza }
     Write(np,'->');
     cost_total:=0; nod:=np;
     for i:=2 to n do
         begin
           { se alege o muchie
             cu o extremitate in nod si cealalta v,
             v nemarcat, astfel incit nod-v are cost minim }
          min:=infinit;
          for j:=1 to n do
              if (not Marcat[j]) and (Cost[nod,j]<min) then
                 begin
                   v:=j; min:=Cost[nod,j]
                 end;
          Write(v,'->'); nod:=v; Marcat[v]:=True;
          cost_total:=cost_total+min
```

```
            end;
        if Cost[nod,np]<infinit then
           begin
             WriteLn(np);
             cost_total:=cost_total+Cost[nod,np];
             WriteLn('Cost_total: ',cost_total);
           end
        else
           begin
             WriteLn(np);
             WriteLn('Cost total: infinit...')
           end;
        ReadLn
end.
```

Exemplu: Să considerăm graful din figura următoare:



Aplicând algoritmul Greedy din programul anterior, se obține circuitul `1-2-5-4-3-1` (evidențiat mai jos), cu costul de `22` unități, pe când, dacă se opta pentru circuitul `1-2-3-4-5-1`, se obținea costul de `21` unități.



În figură am notat cu cifre arabe numerele care indică ordinea în care se iau muchiile pe traseu.

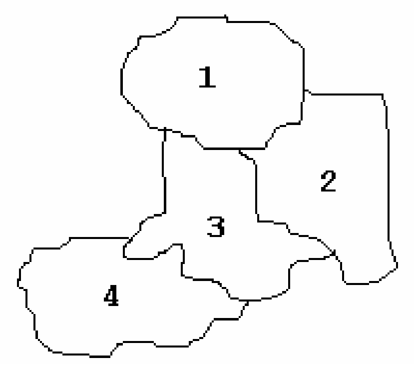
8.3.2. Problema colorării hărților

`N` țări sunt date precizându-se relațiile de vecinatate. Se cere să se determine o posibilitate de colorare a hărții (cu cele `n` țări), astfel încât să nu existe țări vecine colorate la fel.

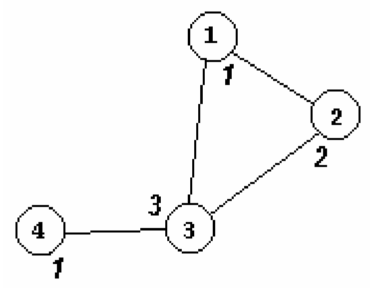
Vom prezenta un algoritm Greedy care va colora, pe rând, fiecare nod în cea mai mică (ca indice) culoare posibilă. Modul în care se iau nodurile grafului este uneori esențială. Această ordine este memorată în vectorul `a`.

De exemplu, harta din figura următoare are `N=4`:

```
end.
```



Ea se poate reprezenta sub forma grafului următor, în care am prezentat şi culorile asociate nodurilor, conform algoritmului descris, aplicat pentru permutarea identică `a=(1,2,3,4)`.




```
program ColorareGrafGreedy;
uses Crt;
var n,i,j: Integer;
    a,x: array[1..20] of Integer;
    Vecin: array[1..20,1..20] of Integer;
begin
     Write('Dati nr. de noduri: '); ReadLn(n);
     for i:=1 to n-1 do
         for j:=i+1 to n do
             begin
               Write('Este vecin nodul ',i,
                     ' cu nodul ',j,' ? [da=1] ');
               ReadLn(Vecin[i,j]);
               Vecin[j,i]:=Vecin[i,j]
             end;
     for i:=1 to n do Vecin[i,i]:=0;
     WriteLn('Dati permutarea: ');
     for i:=1 to n do
         begin
           Write('a[',i,']='); ReadLn(a[i])
         end;
     WriteLn('O colorare Greedy a nodurilor grafului este:');
     for i:=1 to n do
       begin
         x[a[i]]:=1;
         for j:=1 to i-1 do
           if (Vecin[a[i],a[j]]=1)
              and (x[a[j]]=x[a[i]]) then x[a[i]]:=x[a[i]]+1
         end;
     for i:=1 to n do
         WriteLn('- nodul ',i,' in culoarea ',x[i]);
     ReadLn
end.
```

Probleme

1 💣 Memorarea optimă a textelor pe benzi. Fiind date `n` texte de lungimi L_1, L_2, ..., L_n şi `m` benzi magnetice, se cere poziționarea optimă a textelor pe aceste benzi, astfel încât timpul de citire a unui text oarecare de pe benzi să fie minim (ori de câte ori este nevoie de un text, sunt citite toate textele aflate înaintea lui pe bandă şi, bineînțeles, textul respectiv). Se presupune că frecvența de citire a textelor este aceeaşi.

2 💣 Interclasarea optimă a mai multor şiruri ordonate. Se dau `n` şiruri S_1, S_2, .., S_n de lungimi L_1, L_2, ..., L_n. În cadrul fiecărui şir elementele sunt ordonate crescător. Se cere obținerea unui şir `S` cu $L_1+L_2+...+L_n$ elemente ordonate crescător, şirul `S` conținând exact elementele din cele `n` şiruri. Acest lucru se va realiza făcând succesiv interclasări de câte două şiruri ordonate, ceea ce necesită un anumit timp `t`. Se cere determinarea ordinei de realizare a interclasărilor astfel încât `t` să fie minim.

3 😐 Stația de servire. O stație de servire trebuie să satisfacă cererile a `n` clienți. Timpul de servire necesar clientului `i` este t_i. Se cere să se minimizeze timpul total de aşteptare `T` = suma timpilor de aşteptare pentru clienții `i`, `i=1,n`.

De exemplu, dacă avem `n=3` cu $t_1=5$, $t_2=10$ şi $t_3=3$, sunt posibile şase ordini de servire:

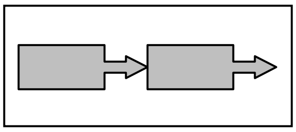
ordinea	T
1,2,3	5+(5+10)+(5+10+3) = 38
1,3,2	5+(5+3)+(5+3+10)=31
2,1,3	10+(10+5)+(10+5+3)=43
2,3,1	10+(10+3)+(10+3+5)=41
3,1,2	3+(3+5)+(3+5+10)=29
3,2,1	3+(3+10)+(3+10+5)=34

Aşadar, timpul minim (29) se obține pentru ordinea 3,1,2.

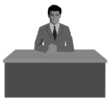
4 ☹ Fie `a` un număr dat în baza `10`. Reprezentarea sa în baza `2` are `m` cifre de `1` şi `n` cifre (semnificative) de `0`. Să se genereze şi să se afişeze atât în baza `2`, cât şi în baza `10`, toate numerele ale căror reprezentări au `m` cifre de `1` şi `n` cifre de `0` (semnificative).

Rezumat

1. Metoda Greedy, cu care v-ați întâlnit în clasa a IX-a, este o metodă generală de rezolvare a unor probleme. Ea se aplică acelor probleme în care, dându-se `A` - o mulțime finită de elemente se cere să se determine o submulțime `B` a sa, ale cărei elemente să îndeplinească anumite condiții.
2. În metoda Greedy, mai întâi are loc o prelucrare a mulțimii `A`. Apoi, se consideră pe rând, elementele lui `A` şi, dacă îndeplinesc condiția cerută de problemă, se adaugă mulțimii `B`.
3. Metoda Greedy se poate aplica cu succes în rezolvarea unor probleme de teoria grafurilor. Alte exemple de algoritmi bazați pe metoda Greedy: problema continuă a rucsacului, maximizarea valorii unei expresii, problema spectacolelor.
4. Nu întotdeauna aplicarea metodei Greedy conduce la o soluție optimă, dar ea poate fi una foarte apropiată de optim.



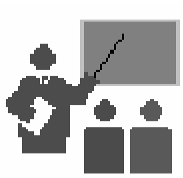
Capitolul 9. Structuri dinamice de date



În atenția profesorului

Pe parcursul acestui capitol, se vor urmări: a) înțelegerea de către studenți a diferenței dintre variabilele statice şi dinamice, a rolului variabilelor pointer; b) evidențierea avantajelor alocării dinamice; c) prezentarea diferitelor tipuri de structuri de date înlănțuite dinamice; d) identificarea problemelor rezolvabile optim utilizând diferitele tipuri de structuri dinamice.

9.1. Tipul referință. Noțiunea de variabilă dinamică



Am văzut în lecția anterioară că un graf se poate memora sub forma matricei sale de adiacență. Dacă, însă, graful are foarte multe noduri, folosirea matricei de adiacență este ineficientă.

De asemenea, dacă vrem să facem un program pentru un concurs de admitere în liceu, stocarea datelor despre elevii candidați nu o vom putea realiza folosind vectori, deoarece nu putem declara vectori de lungimi apropiate de situații reale (500 de candidați, de pildă). Astfel, ar trebui să dispunem de o structură nouă de date, un fel de *listă*, care să păstreze mai multe informații înlănțuite între ele, într-o zonă de memorie mai largă.

Sunt, de asemenea, unele cazuri în care avem nevoie de structuri speciale de date, de exemplu, pentru a memora o ierarhie oarecare, ca de pildă un arbore genealogic. Pentru asemenea cazuri şi nu doar, structurile de date învățate în clasa a IX-a, numite *statice*, nu ne ajută prea mult. Toate inconvenientele pot dispare, însă, odată cu utilizarea structurilor *dinamice* de date.

9.1.1. Variabile statice şi variabile dinamice

Variabilele pot fi *statice* sau *dinamice*. Cele **statice** sunt alocate în timpul compilării, în zone bine definite de memorie. Structura, tipul şi locul lor din memorie nu se pot modifica în timpul execuției programului. Toate variabilele pe care le-am folosit până acum erau statice.

Limbajul Pascal oferă posibilitatea alocării memoriei în timpul executării programului, pentru unele variabile, numite *dinamice*, în funcție de necesitate şi, de asemenea, există posibilitatea eliberării memoriei ocupate de ele. Variabilele **dinamice** trebuie să aibă tipul bine definit, însă nu se vor declara în secțiunea "`var`". De asemenea, accesul la astfel de variabile nu se face direct. Lor li se pune în corespondență un tip **referință,** în mod biunivoc, despre care se spune că referă sau indică spre respectiva variabilă dinamică.

Variabila de tip referință poate conține referiri numai la variabila dinamică care i-a fost pusă în coresponență. Referirea se realizează prin memorarea în variabila de tip referință a adresei unde este stocată variabila dinamică. Corespondența între variabila dinamică şi tipul de referință permite cunoaşterea structurii variabilei dinamice.

Pentru variabilele de tip referință se va aloca, în timpul compilării, un spațiu de memorie de 4 octeți, care va conține adresa de memorie a variabilei dinamice referite.

Variabilele dinamice se alocă într-o zonă de memorie numită **heap**, diferită de zona fixă a programului, unde se memorează variabilele statice.

O variabilă dinamică referită va ocupa un spaţiu de memorie corespunzător tipului ei: 2 octeţi pentru `Integer`, 6 octeţi pentru `Real`, 10 octeţi pentru un şir de 9 caractere (`String[9]`) etc..

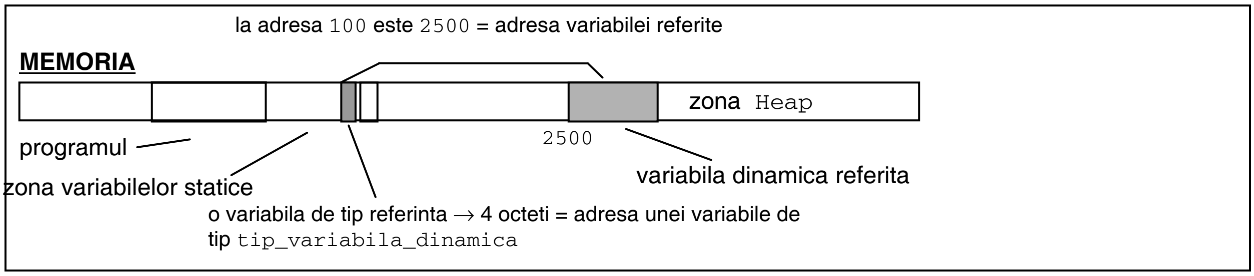
9.1.2. Definirea unui tip referinţă

Definirea unui tip referinţă se poate face în secţiunea `type` astfel:

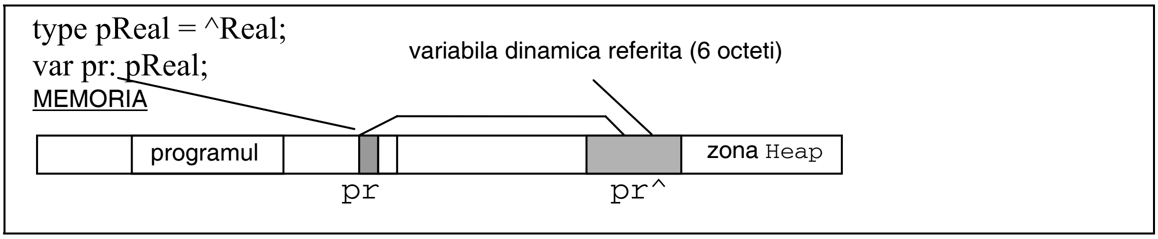
```
type tip_referinta = ^tip_variabila_dinamica;
```

(simbolul "^" se citeşte "pointer la...").

Mulţimea valorilor de tip `tip_referinta` constă într-un număr nelimitat de adrese. Fiecare adresă identifică o variabilă de tip `tip_variabila_dinamica`. La această mulţime de valori se mai adaugă o valoare specială, numită **`Nil`**, care nu identifică nici o variabilă.



Exemplu:



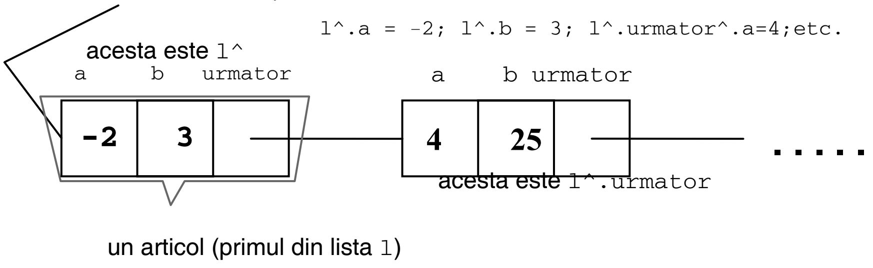
Este permis ca, în momentul întâlnirii tipului variabilei dinamice, acesta să nu fie cunoscut încă (referire înainte); acest tip trebuie declarat mai târziu, în *aceeaşi* declaraţie de tip.

Exemplu:

```
type p_complex = ^complex;
       complex = record re, im: Real end;
var r1, r2: p_complex; r3: ^complex; r4: ^Char;
```

O altă facilitate a limbajului constă în posibilitatea utilizării tipurilor care se autoreferă (sunt definite recursiv) sau înlănţuite:

```
type lista = ^articol;
     articol = record
                a, b: Integer;
               urmator: lista
           end;
var l: lista;
```



9.1.3. Utilizarea variabilelor dinamice. Avantaje

♦ Memorarea unei variabile dinamice se realizează în două faze:

- alocarea zonei de memorie pentru variabila dinamică, cu procedura **`New`**;
- memorarea efectivă, adică depunerea, la adresa pregătită, a datelor corespunzătoare variabilei dinamice.

♦ Eliberarea zonei de memorie corespunzătoare unei variabile dinamice, ocupată cu procedura `New`, se realizează cu procedura **`Dispose`**.

Exemplu:

```
program TestPointer;
type complex = record
               re, im: Real
              end;
     pComplex = ^complex;
var pz: pComplex;
begin
   New(pz); pz^.re:=1.2; pz^.im:=5.6;
   WriteLn('Nr. complex: ',pz^.re, ' ', pz^.im);
   Dispose(pz); ReadLn
end.
```

Cu variabilele dinamice se pot face face toate operațiile care se pot executa cu datele de respectivul tip.

Printre avantajele utilizării pointerilor se numără:

* folosirea unui spațiu de memorie redus, în cazul utilizării unor structuri complexe de date;
* folosirea mai eficientă a spațiului de memorie, prin eventuala reutilizare a sa (după `Dispose`).

Exemplu:

```
type persoana = record
                 nume, prenume: String;
                  {2*256 octeti = 512 octeti}
                 virsta: Integer
                   {2 octeti, deci 514 octeti, in total}
                end;
```

```
       pPersoana = ^persoana; { 4 octeti }
       mult_pers_1 = array[1..50] of persoana;
       mul_pers_2 = array[1..50] of pPersoana;
  var M1: mult_pers_1; {50*514=25700 octeti}
     M2: mult_pers_2  {50*4=200 octeti, restul e in heap !}
```

? Întrebări şi exerciţii

1 ☺ Câte tipuri de varibile cunoaşteţi în limbajul Pascal?
2 😐 Ce se înţelege prin variabilă dinamică?
3 ☹ Unde sunt memorate variabilele dinamice? Apar ele declarate în secţiunea “`var`”?
3 ☹ Ce se înţelege prin tip referinţă? Cum se asociază acesta unui tip de bază?
4 ☹ Cum se alocă memorie unei variabile dinamice? La ce foloseşte procedura `Dispose`?
5 ☹ Ce avantaje prezintă utilizarea datelor alocate dinamic?
6 💣 Pentru declaraţiile din finalul paragrafului, scrieţi un program care să opereze asupra unor mulţimi de persoane: citire, adăugare, eliminare, căutare de persoane.

9.2. Liste

9.2.1. Operaţii elementare: inserare, căutare şi eliminare element

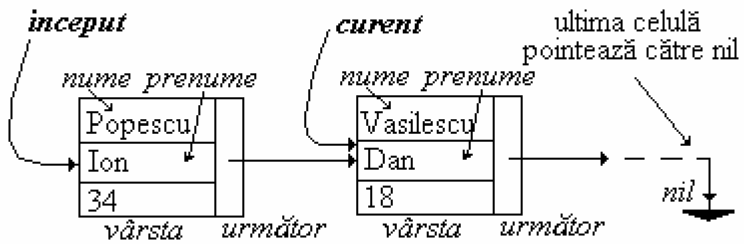
Să considerăm ultimul exemplu din secţiunea anterioară (6.1.3). Pentru a înlătura dezavantajul dimensiunii foarte mari a vectorilor (50 de elemente), vom realiza o listă înlănţuită, în care fiecare articol (nod) al listei va conţine informaţiile despre o persoană:

```
type pPersoana = ^persoana;
     persoana = record
                  nume, prenume: String[20];
                 virsta: Integer;
                  urmator: pPersoana
           end;
var inceput, curent: pPersoana;
{elementul de la inceput si cel curent din lista}
```

Pe baza acestor declaraţii, vom crea o listă de persoane şi vom căuta apoi o anumită persoană în listă.

```
program CuListaDinamica;
type pPersoana = ^persoana;
     persoana = record
                  nume, prenume: String[20];
                  virsta: Integer; urmator: pPersoana
                 end;
var n,p: String[20]; v: Integer;
     gasit: Boolean; raspuns: Char;
    inceput, curent: pPersoana;
begin
  inceput:=Nil;
  repeat
    Write('Nume='); ReadLn(n);
```

```
      Write('Prenume='); ReadLn(p);
      Write('Virsta='); ReadLn(v);
      New(curent);
      with curent^ do
           begin
             nume:=n;
             prenume:=p;
             virsta:=v;
            urmator:=inceput
           end;
      inceput:=curent;
      Write('Continuati [D/N] ? '); ReadLn(raspuns)
    until raspuns in ['n','N'];
    Write('Nume cautat = '); ReadLn(n);
    curent:=inceput; gasit:=False;
    while (curent<>Nil) and (not gasit) do
      if curent^.nume=n then
         gasit:=True
      else
         curent:=curent^.urmator;
    if gasit then
      WriteLn('Prenume=',curent^.prenume,
              ', virsta=',curent^.virsta)
    else
      WriteLn('Persoana inxexistenta in lista...');
    ReadLn
  end.
```



În cadrul programului am încadrat secțiunea care adaugă un element în fața listei deja create.
Astfel, avem:
Se alocă memorie pentru un nou element.

```
New(curent);
```

Se scriu datele pentru noul element:

```
with curent^ do
     begin
       nume:=n;
       prenume:=p;
       virsta:=v;
```

... şi se pune în fața listei deja create:

```
        urmator:=inceput
     end;
```

Apoi, noul început al listei va fi chiar acest element curent:

```
inceput:=curent;
```

În continuare, să rescriem programul astfel încât să eliminăm un anumit element, dat de la tastatură. Pentru a putea să ştergem chiar şi primul element, vom adăuga în fața listei, după ce aceasta s-a creat, un element de formă, deoarece parcurgerea listei se va face altfel, cu un element "în urmă".

Astfel, elementul de şters nu va fi `curent`, ci `curent^.urmator`, pentru a se permite legarea lui `curent^` de `curent^.urmator^.urmator^`.

```
program StergereDinListaDinamica;
type pPersoana = ^persoana;
     persoana = record
                 nume, prenume: String[20];
                 virsta: Integer; urmator: pPersoana
                end;

var n,p: String[20]; v: Integer;
    gasit: Boolean; raspuns: Char;
    inceput, curent, de_sters: pPersoana;

begin
  inceput:=Nil;
  repeat
    Write('Nume='); ReadLn(n);
    Write('Prenume='); ReadLn(p);
    Write('Virsta='); ReadLn(v);
    New(curent);
    with curent^ do
         begin
           nume:=n;
           prenume:=p;
           virsta:=v;
           urmator:=inceput
         end;
    inceput:=curent;
    Write('Continuati [D/N] ? '); ReadLn(raspuns)
  until raspuns in ['n','N'];
  Write('Nume de sters = '); ReadLn(n);
  New(curent);
  curent^.urmator:=inceput;
  gasit:=False;
  while (curent^.urmator<>Nil) and (not gasit) do
    if curent^.urmator^.nume=n then
       gasit:=True
    else
       curent:=curent^.urmator;
  if gasit then
     begin
       de_sters:=curent^.urmator;
       Write('Se sterge ');
       Write(de_sters^.nume,' ',de_sters^.prenume);
       WriteLn(' cu varsta de ',
               de_sters^.virsta,' ani.');
       curent:=curent^.urmator^.urmator;
       Dispose(de_sters)
     end
  else
    WriteLn('Persoana inxexistenta in lista...');
  ReadLn
end.
```

? Întrebări şi exerciţii

1 😐 Scrieţi un program care să citească datele unor persoane dintr-un fişier text (datele sunt: nume, prenume şi vârsta) şi să afişeze numele şi prenumele tuturor persoanelelor care sunt majore (peste 18 ani, inclusiv).

2 ☹ Să se construiască o listă de numere reale, citite de la tastatură, până la întâlnirea numărului `0`. Să se determine câte dintre ele sunt şi întregi.

3 ☹ Se citesc numere întregi de la tastatură, până se întâlneşte `0`. Să se construiască o listă cu aceste numere. Să se determine câte perechi consecutive de numere sunt prime între ele, în listă.

4 💣 Să se construiască o listă `L` de caractere, citite de la tastatură, apoi să se construiască o listă `M` a caracterelor din `L`, aşezate în ordine inversă.

9.2.2. Stive şi cozi. Operaţii specifice

Listele, implementate dinamic, sunt foarte utile atunci când se lucrează cu multe informaţii, pe care vectorii se dovedesc incapabili a le stoca, sau ineficienţi.

Cazuri particulare de liste simplu înlănţuite sunt *stivele* şi *cozile*. Acestea implementează două mecanisme diferite de intrare şi ieşire a elementelor din listă. La ambele feluri de liste, un nod al listei este o înregistrare ce conţine o informaţie (`info`), precum şi un pointer (indicator) către precedentul (următorul) element al listei (`prec` sau `urm`).

Prezentare generală

- **Stiva** este o structură dinamică de date reprezentată de o listă simplu înlănţuită în care mecanismul de intrare - ieşire a elementelor este de tip LIFO - ultimul intrat este primul ieşit (*last in - first out*). Astfel, vom memora o stivă în felul următor:

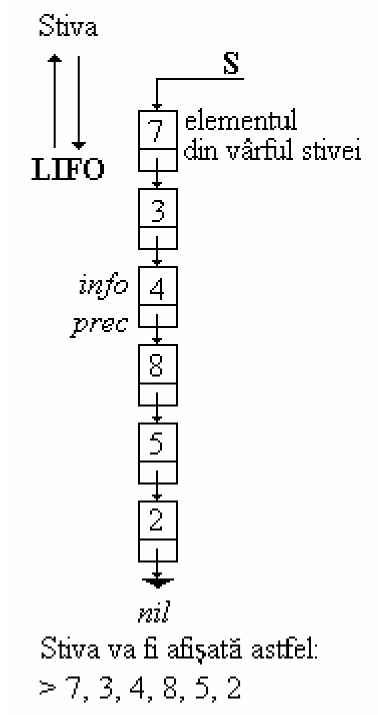
```
type Stiva = ^Celula;
     Celula = record
                info: Integer; { informatia }
                prec: Stiva
             { precedentul element din stiva }
              end;
var S: Stiva;
```

Aşadar, este de ajuns un pointer către primul element al stivei, pentru a realiza atât operaţia de adăugare a unui element (numită adesea *Push*), cât şi cea de eliminare (*Pop*), deoarece ambele operaţii se realizează prin partea superioară a listei.

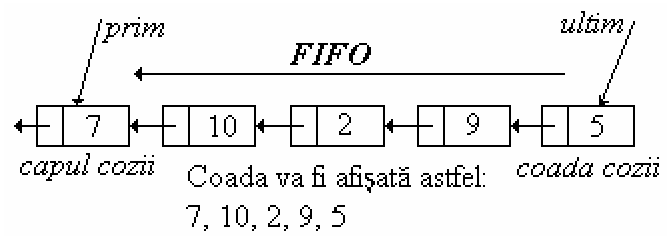
- **Coada** este o structură dinamică de date reprezentată de o listă simplu înlănţuită în care mecanismul de intrare - ieşire a elementelor este de tip FIFO - primul intrat este primul ieşit (*first in - first out*). Astfel, vom memora o coadă în felul următor:

```
type  PCelula = ^ Celula;
      Celula = record
               info: Integer; {informatia }
               urm: PCelula {urmatorul la coada}
               end;
      Coada = record prim, ultim: Pcelula end;
var C: Coada;
```

Aşadar, în cazul cozii, avem nevoie de doi pointeri, unul către primul element al cozii (*capul cozii*), iar altul către ultimul său element (*coada cozii*), deoarece introducerea în listă se face prin spate, iar eliminarea prin faţă.



Coada



Programe demonstrative

Prezentăm două programe demonstrative care utilizează aceste structuri de date. Elementele din listă sunt numere întregi. Introducerea unui element se face prin simpla scriere a sa, dar trebuie ca elementul să nu fie nici `0`, nici `-1`. Eliminarea unui element (conform mecanismului implementat (LIFO sau FIFO) se realizează prin introducerea lui `-1`. Oprirea programului se face introducând `0`.

1. Programul cu stive. Prin apelul procedurii `Init(S)` se inițializează o stivă `S`, prin `Push(S,elem)` se adaugă elementul întreg `elem` la stiva `S`, iar prin `Pop(S,elem)` se scoate elementul `elem` din stiva `S`.

```
program CuStiva;
type Stiva = ^Celula;
     Celula = record
                info: Integer;
                prec: Stiva
              end;
procedure Init(var S: Stiva);
begin S := Nil end;

procedure Push(var S: Stiva; elem: Integer); { pune }
var C: Stiva;
begin
```

```
  New(C); C^.info := elem;
  C^.prec := S; S := C
end;

procedure Pop(var S: Stiva; var elem: Integer); { scoate }
var C: Stiva;
begin
  if S = Nil then
     begin
       Write('Stiva goala ...');
       elem := -1
     end
  else
     begin
       C:=S; elem:=C^.info;
       S:=C^.prec; Dispose(C)
     end
end;

procedure Afis(S: Stiva);
var C: Stiva;
begin
  C := S; Write('Stiva este: ',#16);
  while C <> Nil do
        begin
          Write(C^.info,',');
          C := C^.prec
        end;
  WriteLn
end;

var S: Stiva; elem: Integer;
begin
  Init(S); Afis(S);
  repeat
    Write('Dati elementul (-1 = scoate, 0 = stop): ');
    ReadLn(elem);
    if elem <> 0 then
     if elem <> -1 then
        begin
          Push(S,elem);
          Afis(S)
        end
     else
      begin
        Pop(S,elem);
        WriteLn('Am scos ', elem);
        Afis(S)
      end
  until elem = 0;
end.
```

2. Programul cu coada. Acesta funcţionează la fel ca şi cel anterior, doar că procedurile de adăugare şi eliminare se numesc `Pune`, respectiv `Scoate`, iar mecanismul prin care au loc aceste operaţii este de tip FIFO.

```
program CuCoada;
type PCelula = ^Celula;
     Celula = record
                info: Integer;
                urm: PCelula
              end;
     Coada = record
```

```
              prim,ultim:PCelula
            end;

procedure Init(var C: Coada);
begin
  New(C.prim); New(C.ultim);
  C.prim^.urm:=C.ltim;
  C.ultim^.urm:=C.prim
end;
procedure Pune(var C: Coada; elem: Integer);
var P: PCelula;
begin
  New(P); P^.info:=elem;
  P^.urm:=Nil;
  if C.prim=nil then
     {nici un element}
     begin
       C.prim:=P; C.ultim:=P
     end
  else
     begin
       C.ultim^.urm:=P;C.ultim:=P
     end
end;

procedure Scoate(var C: Coada; var elem: Integer);
var P: PCelula;
begin
 P := C.prim;
 if C.prim = Nil then
  begin
    Write('Coada este vida ...');
    elem := -1
  end
 else
  begin
    elem := C.prim^.info;
    C.prim := C.prim^.urm;
    Dispose(P)
  end
end;

procedure Afis(C: Coada);
var P: PCelula;
begin
  Write('Coada este: ');
  P := C.prim;
  while P <> Nil do
        begin
          Write(P^.info,','); P := P^.urm
        end;
  WriteLn
end;
var C: Coada; elem: Integer;
begin
  Init(C);
  Afis(C);
  repeat
    Write('Dati elementul (-1=scoate, 0=stop): ');
    ReadLn(elem);
    if elem <> 0 then
      if elem <> -1 then
        begin
          Pune(C,elem); Afis(C)
```

```
        end
      else
        begin
          Scoate(C,elem); WriteLn('Am scos ', elem);
          Afis(C)
        end
  until elem = 0;
end.
```

9.2.3. Liste dublu înlănţuite. Operaţii specifice

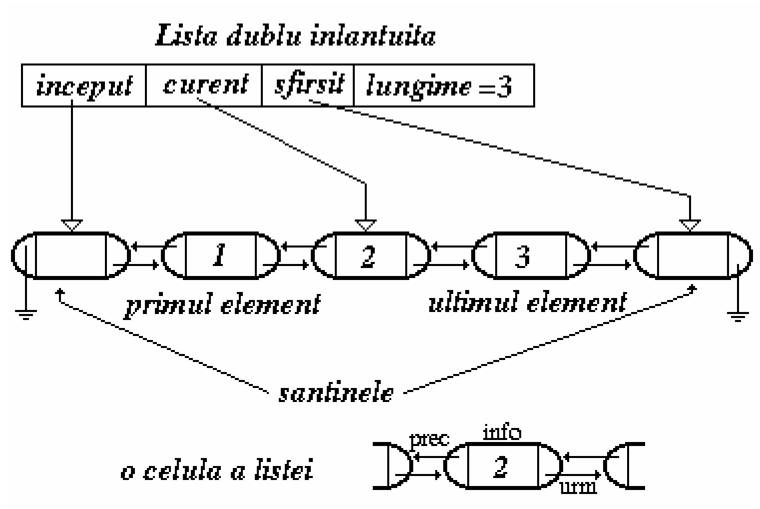
În cazul listelor dublu înlănţuite avem, spre deosebire de stive şi cozi, următoarele noi elemente:

- există doi pointeri speciali: `inceput` şi `sfirsit` care indică spre două celule extreme ale listei, dar care *nu* fac parte din listă; ei se numesc *santinele*;
- există un pointer numit `curent` care indică întotdeauna elementul curent din listă;
- fiecare element a listei este legată prin doi pointeri (`prec` şi `urm`) de elementele dinaintea şi de după el din cadrul listei;
- avem un câmp `lungime`, care va indica lungimea listei.

Astfel, lista de numere 1, 2, 3 va fi memorată ca în figura de mai jos.
Operaţiile ce se cer a se efectua cu o astfel de listă sunt:

- iniţializarea listei;
- adăugarea unui element la sfârşitul listei;
- inserarea unui element înaintea elementului curent din listă;
- ştergerea elementului curent din listă.
- afişarea listei.

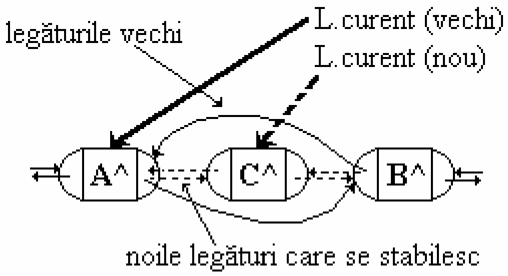
Când se adaugă sau se inserează un nou element în listă, acel element devine cel curent. Când elementul curent se şterge din listă, locul său este preluat de elementul care îl succeda, în cadrul listei.



În momentul în care se introduce sau se elimină un element din listă, dispar unele legături şi apar altele.

De pildă, să presupunem că avem o listă `L` şi, înaintea elementului curent se inserează o informaţie `elem`. Atunci vom crea o nouă celulă, cu numele `C`, în care vom pune această informaţie. Să numim celula curentă `A`, iar cea de după ea `B`. Atunci, va trebui să rupem legăturile dintre `A` şi `B`, stabilind, de asemenea, legături între `A` şi `C` şi între `C` şi `B`. Totodată, lungimea listei va creşte cu o unitate. Aceste lucruri sunt evidenţiate în procedura de mai jos şi în figura asociată.

```
procedure Insereaza(var L: Lista; elem: tip_info);
var A,B,C: PCelula;
begin
     New(C);
     A:=L.curent; B:=L.curent^.urm;
     C^.info:=elem;
     C^.prec:=A; C^.urm:=B; A^.urm:=C;
     B^.prec:=C; L.lung:=L.lung+1;
     L.curent:=C
end;
```



Procese aproape inverse au loc la eliminarea elementul curent din listă. Fireşte, acest lucru se poate realiza doar dacă lista nu este vidă. Pentru a elibera efectiv zona de memorie ocupată de elementul eliminat, se apelează procedura `Dispose`. Eliminarea elementului curent va presupune legarea elementului ce-l precede cu cel ce îl succede.

```
procedure Sterge(var L: Lista);
var C: PCelula;
begin
  with L do
    if lung>0 then
       begin
         C:=curent; C^.prec^.urm:=C^.urm;
         C^.urm^.prec:=C^.prec;
         curent:=C^.urm; Dispose(C);
         lung:=lung-1
       end
end;
```

Succesorul elementului eliminat devine element curent. De asemenea, lungimea listei scade cu o unitate.

În continuare prezentăm un program pentru admiterea în liceu. Candidaţii sunt puşi într-o listă dinamică, dublu înlănţuită. Astfel, numărul de candidaţi va fi limitat doar de memoria rămasă disponibilă în calculator, deci poate fi foarte mare, comparativ cu cazul memorării elevilor într-un vector, pe care l-aţi studiat în clasa a IX-a.

Pentru a realiza ordonarea elevilor după un anumit criteriu (alfabetic sau după medii, descrescător), am implementat o variantă a algoritmului de sortare "*bubble-sort*", care foloseşte o listă `L` în locul unui vector. În esenţă, algoritmul este acelaşi. Deosebiri apar în parcurgerea listei, precum şi în cazul elementelor care se compară: în loc de `x[i]` şi `x[i+1]`, acum se compară elementele `L.curent^.info` şi `L.curent^.urm^.info`, de fapt acele câmpuri ale lor care sunt cheie în ordonarea efectuată. În program, secvenţa care implementează algoritmul "*bubble-sort*" este încadrată.

```
{$X+}
program CuListe;
uses Crt;
type elev = record nume: String; nota1, nota2, media: Real end;
     tip_info=elev;
     PCelula = ^Celula;
     Celula = record
                info: tip_info;
                prec, urm: PCelula
              end;
     Lista = record
                inceput, curent,  sfirsit: PCelula;
                lung: Integer
             end;

procedure Init(var L: Lista);
begin
  with L do
     begin
       lung:=0;
       New(inceput); New(sfirsit);
       inceput^.urm:=sfirsit;
       inceput^.prec:=nil; sfirsit^.prec:=inceput;
       sfirsit^.urm:=nil; curent:=inceput
     end
end;

procedure PozitioneazaLaInceput(var L: Lista);
begin
  while not (L.curent=L.inceput^.urm) do
        L.curent:=L.curent^.prec
end;

procedure Adauga(var L: Lista; elem: tip_info);
var C: PCelula;
begin
  New(C);
  with C^ do
    begin
      info:=elem;
      prec:=L.sfirsit^.prec;
      urm:=L.sfirsit^.prec^.urm
      { sau L.sfirsit}
    end;
  with L do
    begin
      lung:=lung+1; sfirsit^.prec^.urm:=C;
      sfirsit^.prec:=C; curent:=C
    end
end;

procedure Sterge(var L: Lista);
var C: PCelula;
begin
  with L do
    if lung>0 then
       begin
         C:=curent;
         C^.prec^.urm:=C^.urm;
         C^.urm^.prec:=C^.prec;
         curent:=C^.urm; Dispose(C); lung:=lung-1
       end
end;
```

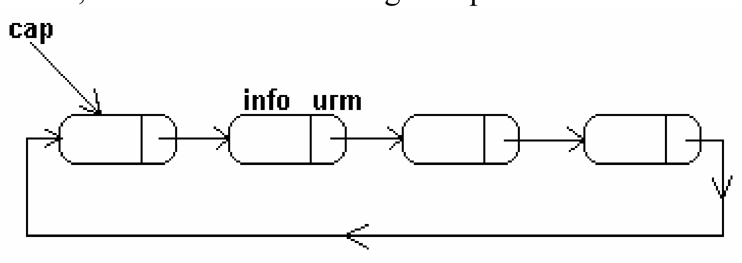
```
procedure Insereaza(var L: Lista; elem: tip_info);
var A,B,C: PCelula;
begin
  New(C);  A:=L.curent; B:=L.curent^.urm;
  C^.info:=elem; C^.prec:=A;
  C^.urm:=B; A^.urm:=C;
  B^.prec:=C; L.lung:=L.lung+1;
  L.curent:=C
end;

procedure Afis(L: Lista);
var i: Integer;
begin
  PozitioneazaLaInceput(L); i:=1;
  while not (L.curent = L.sfirsit) do
    begin
      with L.curent^.info do
           WriteLn(i,'. ',nume:20,
                   nota1:6:2,nota2:6:2, media:6:2);
      L.curent:=L.curent^.urm; i:=i+1
    end;
  WriteLn('Total : ',i-1,' elevi.')
end;
procedure AdaugareElev(L: Lista);
var E: elev;
begin
  WriteLn('Adaugare elev');
  with E do
    begin
      Write('Dati numele : ');
      ReadLn(nume);
      Write('Dati notele : ');
      ReadLn(nota1,nota2);
      media:=(nota1+nota2)/2
    end;
  Adauga(L,E)
end;
procedure EliminareElev(var L: Lista);
var numele: String; gasit: Boolean;
begin
  WriteLn('Eliminare elev'); PozitioneazaLaInceput(L);
  WriteLn('Dati numele elevului : ');
  ReadLn(numele); gasit:=False;
  while (not (L.curent = L.sfirsit)) and (not gasit) do
    if L.curent^.info.nume = numele then
       gasit:=True
    else
       L.curent:=L.curent^.urm;
  if gasit then
     Sterge(L)
  else
     WriteLn('Nu exista acest elev...')
end;
procedure CautareElev(L: Lista);
var numele: String; gasit: Boolean;
begin
  WriteLn('Cautare elev'); PozitioneazaLaInceput(L);
  WriteLn('Dati numele elevului : '); ReadLn(numele);
  gasit:=False;
  while (not (L.curent = L.sfirsit)) and (not gasit) do
    if L.curent^.info.nume = numele then
       gasit:=True
    else
       L.curent:=L.curent^.urm;
```

```
  if gasit then
     with L.curent^.info do
          WriteLn(nume,' ',nota1:6:2,nota2:6:2,media:6:2)
  else
     WriteLn('Elev inexistent...')
end;
procedure ListareEleviDupaNume(L: Lista);
var gata: Boolean; aux: elev;
begin
  WriteLn('Listare elevi dupa nume');
  {ordonare elevi dupa nume:bubble-sort}
  repeat
    gata:=True;
    {ma pozitionez la inceput si parcurg lista}
    PozitioneazaLaInceput(L);
    while not (L.curent^.urm=L.sfirsit) do
      begin
        if L.curent^.info.nume>L.curent^.urm^.info.nume then
           begin
             gata := False; aux := L.curent^.info;
             L.curent^.info:=L.curent^.urm^.info;
             L.curent^.urm^.info:=aux
           end;
        L.curent:=L.curent^.urm
     end
  until gata;
  Afis(L) { afisare lista }
end;

procedure ListareEleviDupaMedii(L: Lista);
var gata: Boolean; aux: elev;
begin
  WriteLn('Listare elevi dupa medii');
  {ordonare elevi: metoda bubble-sort}
  repeat
    gata:=True; {ma pozitionez la inceput, parcurg lista}
    PozitioneazaLaInceput(L);
    while not (L.curent^.urm=L.sfirsit)do
      begin
        if L.curent^.info.media < L.curent^.urm^.info.media
           then
           begin
             gata := False; aux := L.curent^.info;
             L.curent^.info:=L.curent^.urm^.info;
             L.curent^.urm^.info:=aux
           end;
        L.curent:=L.curent^.urm
      end
  until gata;
  Afis(L)
end;
var L: Lista; optiune: Char;
begin
  Init(L);
  repeat
    ClrScr;WriteLn('Meniu:');WriteLn;
    WriteLn('1 = adaugare elev');
    WriteLn('2 = stergere elev');
    WriteLn('3 = cautare elev');
    WriteLn('4 = listare dupa nume');
    WriteLn('5 = listare dupa medii');
    WriteLn('0 = oprire program');
    WriteLn; Write('Optiune=');
    ReadLn(optiune); ClrScr;
```

```
    case optiune of
         '1': AdaugareElev(L); '2': EliminareElev(L);
         '3': CautareElev(L); '4': ListareEleviDupaNume(L);
         '5': ListareEleviDupaMedii(L)
    end;
    if optiune in ['1'..'5'] then ReadKey
 until optiune='0'
end.
```

9.2.4. Liste circulare

Într-o listă circulară, ultimul element este legat de primul:

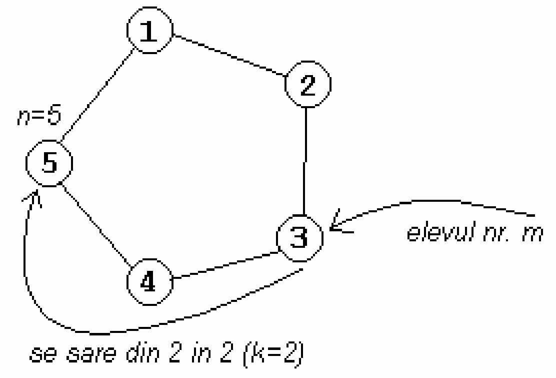


Următorul program creează o listă circulară şi o afişează.

```
program ListaCirculara;
type lista = ^celula;
     celula = record
                info: Char;
                urm: lista
              end;
var cap, p, q: lista; { cap = capul listei }
    i,n: Integer; { n = nr. de elemente curente din lista }
begin
     WriteLn('Lista circulara');
     WriteLn('***************');
     Write('Dati numarul de elemente: '); ReadLn(n);
     Write('Dati prima informatie: ');
     New(cap); ReadLn(cap^.info);
     cap^.urm:=cap;
     for i:=2 to n do
         begin
           Write('Dati a ',i,'-a informatie: ');
           New(p); ReadLn(p^.info);
           q:=cap^.urm; p^.urm:=q;
           cap^.urm:=p;
           cap:=p
         end;
     cap:=cap^.urm;
     WriteLn('Afisam lista: ');
     p:=cap;
     repeat
       Write(p^.info,'->');
       p:=p^.urm
     until p=cap;
     Write(p^.info);
     ReadLn
end.
```

În continuare vom prezenta un program cu liste circulare dublu înlănţuite.

Un număr de `n` copii stau într-un cerc. La un moment dat, începând cu copilul m se elimină din cerc în cerc al `k`-lea copil, cercul apoi strângându-se. Se cere să se afişeze ordinea în care sunt eliminați copii din cerc, în funcție de n, m, k şi de direcția de parcurgere a cercului. De exemplu, pentru n=5, m=3, k=2 şi direcția stânga, se va obține ordinea: 3, 5, 2, 1, 4.



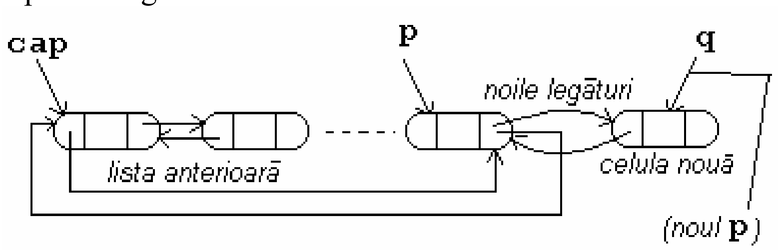

```
program ListeCircularedubluInlantuite;
type lista=^celula;
     celula=record
              info: Integer;
              prec,urm: lista
            end;
var cap,p,q: lista;
    dir: Char;
    i,j,n,m,k: Integer;

begin
     Write('Liste circulare dublu inlantuite. ');
     WriteLn('Exemplu');
     Write('Dati nr. de copii: '); ReadLn(n);
     Write('Dati copilul de start: '); ReadLn(m);
     Write('Dati ratia: '); ReadLn(k);
     { punem elementul 1 in lista }
     New(p); p^.info:=1; p^.urm:=p; p^.prec:=p;
     cap:=p; { lista are capul cap si coada p }
     { punem celelalte elemente in lista }
     for i:=2 to n do
         begin
           New(q);
           q^.info:=i;
           p^.urm:=q;
           q^.prec:=p;
           q^.urm:=cap;
           p:=q
         end;
     while cap^.info<>m do
           cap:=cap^.urm;
     Write('Dati directia de deplasare [s,d]: ');
     ReadLn(dir);
     for i:=1 to n do
         begin
           WriteLn('Se elimina copilul nr. ',cap^.info);
           p:=cap;
           p^.prec^.urm:=p^.urm;
           p^.urm^.prec:=p^.prec;
           case dir of
                's': cap:=cap^.prec;
                'd': cap:=cap^.urm
           end;
           Dispose(p);
           { sar peste ceilalti k copii }
           for j:=1 to k do
               case dir of
```

```
                    's': cap:=cap^.urm;
                    'd': cap:=cap^.prec
               end
          end;
      ReadLn
end.
```

Instrucţiunea compusă care adaugă un nou element în listă este:

```
            begin
              New(q);
              q^.info:=i;
              p^.urm:=q;
              q^.prec:=p;
              q^.urm:=cap;
              p:=q
            end;
```

Ea se poate reprezenta grafic astfel:



Realizaţi proceduri pentru diferite operaţii cu listele circulare: creare, afişare, ştergerea unui element, adăugarea unui element etc.

9.2.5. Sortare topologică

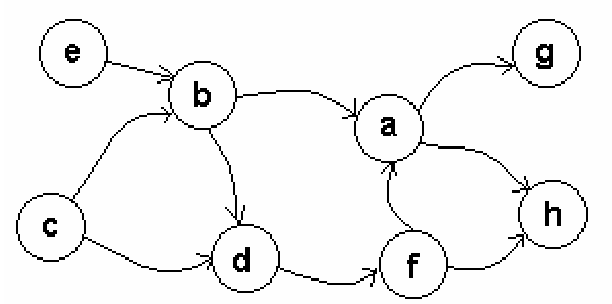
Să considerăm T = o mulţime de termeni (cuvinte) dintr-un anumit domeniu. Dorim să-i cuprindem într-un dicţionar în care, în momentul definirii oricărui termen, termenii din T care intră în această definiţie să fi apărut anteriori. Astfel, avem de a face cu o relaţie de ordine parţială: spunem că a<b (unde a şi b sunt din T) dacă în definiţia lui b apare termenul a.

Problema care se pune este ca termenii să fie aranjaţi în dicţionar astfel încât dacă a apare înaintea lui b, atunci fie a<b, fie a şi b nu sunt comparabili (nici a nu apare în definiţia lui b, nici b în definiţia lui a). Aceasta reprezintă o **sortare topologică** a elementelor din T.

Problema nu are întotdeauna soluţie. Astfel, dacă T={a,b,c} şi avem a<b<c<a, nu putem sorta topologic pe T.

Putem reprezenta relaţiile de ordine stabilite între două elemente a şi b ale lui T printr-o săgeată (un arc orientat) de la un vârf etichetat cu a la un vârf etichetat cu b.

Astfel, de pildă, pentru T={a,b,c,d,e,f,g,h} şi relaţiile: e<b, c<b, c<d, b<d, b<a, a<f, d<f, f<h, a<h, a<g vom construi un digraf (graf orientat) de genul:



Despre noțiunea de graf orientat (sau digraf) se va studia la *Bazele informaticii*.

Sortarea topologică a celor `n` elemente se realizează în `n` pași: la fiecare pas `i` se afișează și se elimină din digraf un termen `j` cu proprietatea că în vârful corespunzător lui nu intră nici un arc.

Pentru a putea realiza acest lucru, va trebui să avem, pentru fiecare termen `j` o listă a succesorilor săi, adică a acelor elemente `k` cu proprietatea că `termen[j]<termen[k]`.

Să notăm cu `lista_succ[j]` lista succesorilor termenului `j`.

Pentru a testa dacă un nod `j` are sau nu arce care intră în el, va trebui să păstrăm un număr (`nr_pred[j]`) care să indice numărul de predecesori ai termenului din acel nod.

Eliminarea unui termen `j` din digraf va presupune atât eliminarea vârfului care îl conține, cât și a tuturor arcelor ce pleacă din acel vârf. De aceea, aceste operații se pot exprima astfel:

- eliminarea vârfului:

```
nr_pred[j]:=-1
```

- pentru orice element din lista succesorilor lui `j` numărul predecesorilor scade cu o unitate:

```
p:=lista_succ[j];
while p<>Nil do
  begin
      nr_pred[p^.info]:=nr_pred[p^.info]-1;
      p:=p^.urm
  end
```

Adăugarea unei relații `a<b` în digraf presupune următoarele:

- se adaugă `b` la lista succesorilor lui `a`:

```
New(p); p^.info:=k2;p^.urm:=lista_succ[k1];
lista_succ[k1]:=p;
```

- se crește numărul de predecesori ai lui `b`:

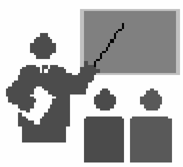
```
nr_pred[k2]:=nr_pred[k2]+1
```

Programul următor rezolvă problema ordonării topologice. Pentru exemplul din figură se obține: `e, c, b, d, f, a, g, h.`

```
program SortareTopologica;
const max=10;
type pVarf=^Varf; { referinta la Varf }
     Varf=record { un element al listelor de succesori }
             info: Integer; { indicele termenului de definit }
             urm: pvarf { legatura la urmatorul element al listei}
          end;
var termen: array[1..max] of String;
    { termen[i] = al i-lea termen }
    nr_pred: array[1..max] of ShortInt;
    { nr_pred[i] = numarul predecesorilor celui de al i-lea termen }
    lista_succ: array[1..max] of pVarf;
    { lista_succ[i] = lista succesorilor celui de al i-lea termen }
    n: Byte; { numarul de termeni }
    m: Byte; { numarul de relatii }
    a,b: String; { doi termeni aflati in relatie de ordine a<b }
    i,j,k: Byte; gata: Boolean; p: pVarf; { variabile de lucru }
    k1,k2: Byte; { k1 = pozitia lui a in termen, k2 = pozitia lui b}
begin
     WriteLn('Sortare topologica');
     WriteLn('******************');
     Write('Dati numarul de termeni: '); ReadLn(n);

     for i:=1 to n do
         begin
           Write('Dati termenul nr. ',i,': ');
           ReadLn(termen[i]);
           nr_pred[i]:=0;
           lista_succ[i]:=Nil
         end;
     Write('Dati numarul de relatii: '); ReadLn(m);
     for i:=1 to m do
         begin
```

```
          WriteLn('Relatia nr. ',i,': a<b:');
          Write('a='); ReadLn(a); { a si b trebuie sa fie in }
          Write('b='); ReadLn(b); { cadrul vectorului termen }
    { se determina pe ce pozitie este a si pe ce pozitie este b }
          for j:=1 to n do
              begin
                if termen[j]=a then k1:=j;
                if termen[j]=b then k2:=j
              end;
    { se adauga b la lista succesorilor lui a }
          New(p);
          p^.info:=k2;
          p^.urm:=lista_succ[k1];
          lista_succ[k1]:=p;
          { se creste numarul de predecesori ai lu b }
          nr_pred[k2]:=nr_pred[k2]+1
        end;
  { la fiecare pas se afiseaza si se elimina din retea (digraf)
    primul element intilnit care nu are predecesori: termen[j] }
    for i:=1 to n do
        begin
          j:=1; gata:=False;
          while (j<=n) and (not gata) do
                if nr_pred[j]=0 then
                   begin
                     { se afiseaza termenul j }
                     Write(termen[j],', ');
                     gata:=True;
                     { se elimina }
                     nr_pred[j]:=-1;
               { pentru orice element din lista succesorilor lui j
                 { numarul predecesorilor scade cu o unitate }
                     p:=lista_succ[j];
                     while p<>Nil do
                           begin
                             nr_pred[p^.info]:=nr_pred[p^.info]-1;
                             p:=p^.urm
                           end
                   end
                else j:=j+1
          end;
end.
```

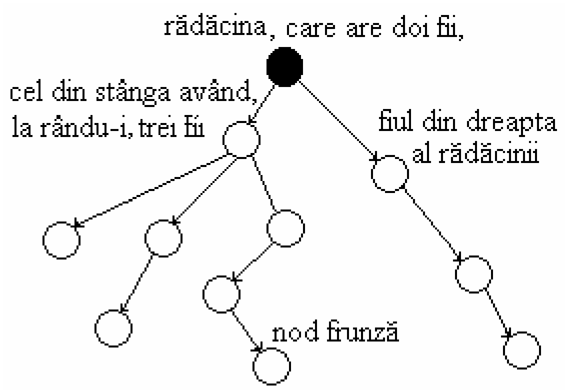
9.3. Arbori



Fie $H=(V,E)$ un digraf (graf orientat). Se numeşte *rădăcină* a lui G un vârf $v_0 \in V$, astfel încât oricare ar fi un vârf $v \in V$, există cel puțin un drum de la v_0 la v. Dacă $H=(V,E)$ este un digraf, prin *graful suport* al lui H vom înțelege graful obținut din H, prin renunțarea la orientarea arcelor. H se numeşte **arborescență** dacă are o rădăcină şi graful său suport este arbore.

În informatică, arborescențele sunt numite, prin abuz de limbaj, **arbori**, specificându-se rădăcina şi considerând implicită orientarea muchiilor corespunzător parcurgerii drumului unic de la rădăcină la fiecare vârf. Fiecare vârf are astfel, nişte *fii*, adică vecinii imediat următori pe drumul de la rădăcină în jos (către *frunze*, adicâ noduri fără fii). Un arbore cu cel mult doi fii se mai numeşte şi **arbore binar**.

De exemplu:



Privit invers, un arbore seamănă, într-adevăr, cu un arbore din natură (un copac), astfel încât denumirea de *arbore*, ca şi cele de *frunză* sau *rădăcină* se justifică. Observăm că putem considera arborele ca fiind organizat pe mai multe *nivele*. Primul nivel este cel al rădăcinii. Urmează nivelul fiilor acesteia ş.a.m.d., până la ultimul nivel, cel al ultimelor frunze.

În continuare ne vom ocupa, în mod deosebit, de arborii binari, deoarece, aşa cum vom arăta mai târziu, studiul arborilor oarecare se reduce la studiul arborilor binari.

9.3.1. Arbori binari

Referirea unui arbore binar şi, implicit, definirea sa, va fi făcută printr-un pointer către nodul său rădăcină. Fiecare nod din arbore este o înregistrare cu următoarele elemente:

- o informaţie `info` de tip întreg;
- doi pointeri către cei doi fii (subarborii stâng şi drept) ai nodului: `stg` şi `dr`.

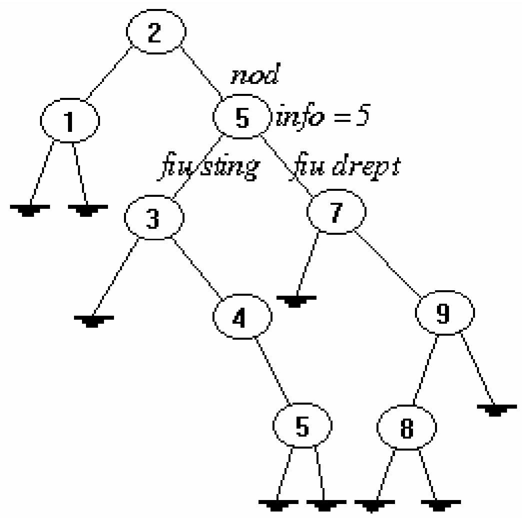
În programul care urmează, vom construi astfel de arbori, care au, în plus, următoarea proprietate: fiecare nod din arbore este mai mare sau egal cu nodurile din fiul stâng şi mai mic decât nodurile din fiul drept (din punct de vedere al câmpului info). Vom numi un astfel de arbore: **arbore de căutare - sortare**.

- O *căutare* a unui element în astfel de arbori este, într-adevăr, uşor de realizat: dacă elementul căutat este identic cu informaţia din nod, atunci căutarea se încheie cu succes, dacă nu, atunci se pleacă pe una din cele două direcţii: dacă elementul căutat este mau mic, atunci se merge pe fiul stâng, altfel pe fiul drept. Dacă se încearcă o trecere dincolo de un nod terminal, deci la `Nil`, atunci căutarea eşuează. Procedura `Cautare` din următorul program demonstrativ returnează subarborele având ca rădăcină elementul găsit în arborele în care s-a căutat.
- Un arbore binar poate fi *parcurs* în trei feluri:
  - în *inordine*: se parcurge mai întâi, recursiv, în inordine, fiul stâng, apoi rădăcina, apoi fiul drept;
  - în *postordine*: se parcurge fiul stâng, apoi cel drept, în final rădăcina;
  - în *preordine*: se parcurge rădăcina, fiul stâng şi apoi fiul drept.

Observaţie

Parcurgerea în inordine a unui astfel de arbore duce la afişarea în ordine crescătoare a elementelor din nodurile arborelui, motiv pentru care astfel de arbori pot fi consideraţi şi de *sortare*.

Dacă se inserează, pe rând, elementele: 2, 5, 7, 3, 4, 9, 1, 5, 8, 0=stop,atunci se obține arborele din figura următoare.



Programul de mai jos implementează structura de arbore discutată, precum şi principalele operații realizabile cu astfel de structuri de date.

```
program ArboriBinari;
type arbore = ^nod;
     nod = record info: Integer; stg, dr: arbore end;
procedure Init(var A: arbore);
begin A := Nil end;
{$S-}
```

```
procedure Inserare(var A: arbore; elem: Integer);
var UnNod: nod;
begin
  if A <> Nil then
   if elem <= A^.info then
       Inserare(A^.stg, elem)
   else
       Inserare(A^.dr, elem)
  else
   begin
     New(A);
     UnNod.info:=elem;
     UnNod.stg:=Nil; UnNod.dr:=Nil;
     A^:=UnNod
   end
end;
```

```
procedure Cautare(A: arbore; elem: Integer; var nodul: arbore);
begin
 if A = Nil then nodul := Nil
 else
   if A^.info = elem then nodul := A
   else
     if elem <= A^.info then Cautare(A^.stg, elem, nodul)
     else Cautare(A^.dr, elem, nodul)
end;
```

```
procedure PreOrdine(A: arbore);
begin
  if A <> Nil then
     begin Write(A^.info,','); PreOrdine(A^.stg); PreOrdine(A^.dr) end
end;
procedure PostOrdine(A: arbore);
begin
  if A <> Nil then
     begin PostOrdine(A^.stg); PostOrdine(A^.dr); Write(A^.info,',') end
end;
```

```
procedure InOrdine(A: arbore);
begin
  if A <> Nil then
     begin Inordine(A^.stg); Write(A^.Info,','); InOrdine(A^.dr) end
end;
{$S+}
var A, B: arbore; elem: Integer;
begin
  Init(A);
  repeat
    Write('Dati elementul de inserat (0 = stop): '); ReadLn(elem);
    if elem <> 0 then Inserare(A,elem)
  until elem = 0;
  WriteLn('Arborele in preordine : '); PreOrdine(A); WriteLn;
  WriteLn('Arborele in postordine : '); PostOrdine(A); WriteLn;
  WriteLn('Arborele in inordine', ' => elementele sortate: ');
  InOrdine(A); WriteLn;
  repeat
    Write('Dati elementul cautat'); Write(' (0 = stop): ');
    ReadLn(elem);


    if elem <> 0 then
       begin
         Cautare(A, elem, B);
         if B <> Nil then
            begin
              WriteLn('L-am gasit,', 'arborele in inordine este:');
              InOrdine(B); WriteLn
            end
         else WriteLn('Nu l-am gasit.')
       end
  until elem = 0;
end.
```

Cea mai complicată operaţie într-un arbore binar de căutare este ştergerea unui nod (astfel încât arborele să-şi păstreze proprietăţile de arbore binar).

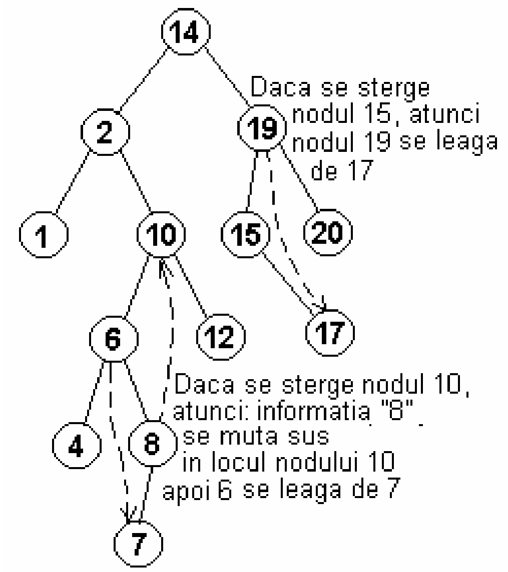
Nodul care se şterge poate fi în una din următoarele patru ipostaze:

1. nu are nici un fiu - caz în care nodul tată va indica către Nil;
2. are doar fiul stâng - nodul tată se leagă de fiul stâng al nodului care se şterge;
3. are doar fiul drept - nodul tată se leagă de fiul drept al nodului care se şterge;
4. are ambii fii - se aduce în nodul care trebuie şters informaţia nodului cu cheia cea mai mare din subarborele său stâng şi se leagă în dreapta tatălui nodului mutat fiul său stâng.

Exemplu:

Pentru arborele din figura de mai jos, nodul `20` se şterge conform regulii 1, fiind cazul cel mai simplu. Dacă ar trebui şters nodul `15`, atunci sar lega nodul `19` direct de nodul `17`, care ar deveni fiul din stânga al nodului `19`.

Dacă, însă, trebuie şters nodul `10`, atunci în locul lui `10` se aduce nodul cu informaţia cea mai mare din subarborele său stâng, deci se aduce informaţia “`8`”, apoi se leagă nodul `6` în dreapta cu nodul `7`.



Procedura care realizează ştergerea unui element elem din arborele A este recursivă şi este prezentată mai jos. Ea are în cuprinsul ei o procedură `St`, care rezolvă cazul 4:

```
procedure Sterge(var A: arbore; elem: Integer);
var q: arbore;
    procedure St(var A:arbore);
    begin
         if A^.dr=Nil then
            begin
              q^.info:=A^.info;
              q:=A;
              A:=A^.stg;
            end
         else
            St(A^.dr)
    end;
begin
     if A=Nil then
        WriteLn('Elementul ',elem,' nu exista...')
     else
       if elem=A^.info then
          begin
            q:=A;
            if q^.stg=Nil then
               A:=q^.dr
            else
               if q^.dr=Nil then
                  A:=q^.stg
               else
                 St(q^.stg);
            Dispose(q)
          end
      else
          if elem<A^.info then Sterge(A^.stg,elem)
          else Sterge(A^.dr,elem)
end;
```

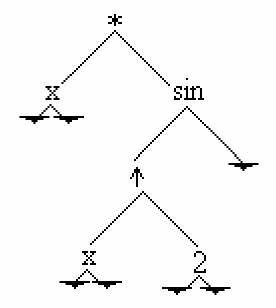
Propunem ca exerciţiu modificarea programului prezentat astfel încât să conţină şi procedura `Sterge`, pe care să o apeleze pentru ştergerea diferitelor noduri din arbore,

9.3.2. Arborele binar asociat unei expresii algebrice

Orice expresie aritmetică sau algebrică poate fi memorată sub forma unui arbore binar. Fiecare nod din arbore va fi fie un operator, fie un operand. Operatorii vor avea doi fii, care sunt operanzii lor. Un caz special îl constituie funcțiile, care vor avea doar un fiu (de exemplu, fiul din stânga). Operanzii nu vor avea nici un fiu.

Prin urmare, dacă avem o expresie aritmetică oarecare, putem obține din ea arborele binar asociat ei.

De exemplu pentru expresia `x*sin(x^2)` vom avea arborele binar:



- funcțiile unare, precum `sin` aici, vor avea fiul din dreapta `nil`, iar numerele şi `x` vor avea ambii fii `nil`;
- forma postixată asociată acestui arbore este: `x x 2 ^ sin *`.

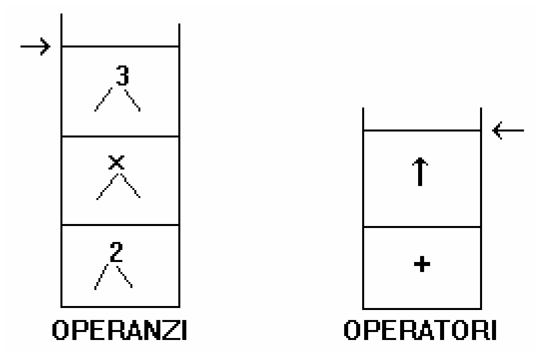
Astfel, o expresie va fi reprezentată prin arborele binar corespunzător formei postfixate a expresiei.

Vom utiliza o structură arborescentă binară. Pentru a transforma o expresie obişnuită într-un arbore vom folosi două stive, una a operatorilor şi funcțiilor (care pot fi privite ca fiind operatori unari), cealaltă a operanzilor, care vor fi arbori mai mici, din care se vor forma arbori mai mari, folosind drept rădăcină ceea ce se afla în vârful stivei operatorilor, la respectivul moment.

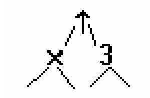
Să vedem cum vom proceda pentru expresia `2+x^3`.

Vom introduce în stiva operatorilor arborele micuț: $/^{2}\backslash$ (şi nu numărul `2`), apoi în stiva operatorilor `+`, apoi arborele $/^{x}\backslash$ în stiva operanzilor.

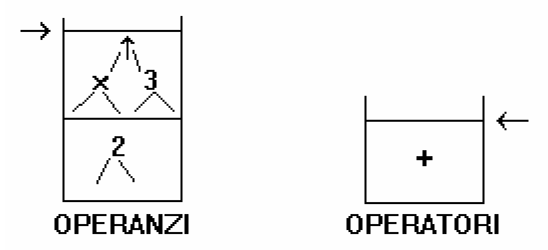
Observăm că `^` (ridicarea la putere) este de prioritate mai mare ca `+`, deci nu vom face `2 + x`, ci îl vom introduce şi pe `^` în stiva operatorilor, apoi pe $/^{3}\backslash$ în cea a operanzilor.



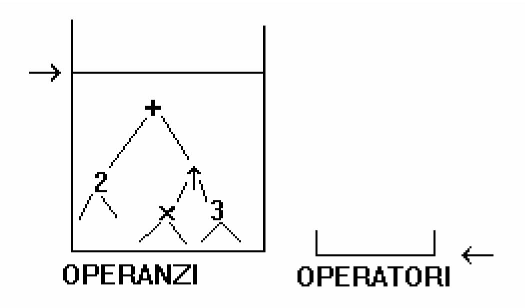
Acum va trebui să scoatem cei doi arbori mici din stivă şi, împreună cu operatorul `^` (scos din stiva operatorilor) drept rădacină, să formăm un nou arbore:



pe care îl vom introduce în stiva operanzilor:



Singura operație rămasă este + şi, procedând ca mai înainte, obținem:



În acest moment, stiva operatorilor este goală, iar stiva operanzilor conține exact arborele binar asociat expresiei date.

Atenție

Probleme deosebite apar atunci cind în vârful stivei operațiilor este - sau /, iar operația succesoare (în expresie) este tot -, respectiv /. (Aceste simboluri reprezintâ operații necomutative.)

Programul următor realizează aceste transformări

Afişarea arborelui binar rezultat se face în mod grafic. (Trebuie cunoscută calea DOS către fişierul EGAVGA.BGI, din mediul *Turbo-Pascal*. Noi am considerat că aceasta este “C:\BP\BGI”.)

```
program ArboreleAsociatUneiExpresiiAlgebrice;
uses Graph,Crt;
const lmax = 3; vmax = 50;
type expresie  =  string; informatie  =  string[lmax];
     vector  =  record
                  info:array[1..vmax] of informatie; nr:1..vmax
                end;
     arbore  =  ^nod;
     nod   =  record info:informatie; stg,dr:arbore end;
var expresia, expr_fpo: expresie; vectorul: vector;
    arborele: arbore;

procedure OpenGraph;
var gd,gm: Integer;
begin
  gd:= Detect; InitGraph(gd,gm,'C:\BP\BGI'); {calea fisierului EGAVGA.BGI}
end;
procedure Vectorizeaza(expr: expresie; var vec: vector);
    procedure Schimba(var x, y: informatie);
    var a: informatie;
    begin a := x; x := y; y := a end;
```

```
var i,j,k: Integer;
begin
  i := 0; k := 0;
  while i<Length(expr) do
    begin
      Inc(i);
      if expr[i] in ['s','c','l','e'] then
         begin
           Inc(k);
           if expr[i]<>'l' then
              vec.info[k]:=expr[i]+expr[i+1]+expr[i+2]
           else
              vec.info[k] := expr[i]+expr[i+1]+' ';
           if expr[i] = 'l' then Inc(i)
           else
             begin
               vec.info[k] := vec.info[k]+expr[i+2];
               Inc(i,2)
             end
         end
      else
        if expr[i] in ['(',')','+','-','*','/','^','x'] then
           begin Inc(k); vec.info[k] := expr[i] end
        else
         begin { cifre };
           Inc(k); vec.info[k] := '        '; j := 1;
           while not (expr[i] in ['(',')','+','-','*',
                    '/','^','s','c','l','e']) do
           begin
             vec.info[k][j] := expr[i];
             Inc(i); Inc(j)
           end;
           Dec(i)
        end
    end;
  vec.nr := k
end;
```

procedure Arborizeaza(vec: vector; var arb: arbore);

```
const topmax = 25;
var i,top1,top2: Integer;
    operator: array[1..topmax] of String[lmax];
    operand: array[1..topmax] of arbore;
  function prioritate(op:char):byte;
  begin
    case op of
         '(',')':prioritate := 0; '+','-':prioritate := 1;
         '*','/':prioritate := 2; '^':prioritate := 3;
         's','c','l','e':prioritate := 4
    end
  end;
begin { arborizeaza }
  i := 0; top1 := 0; top2 := 1; operator[top2] := '(';
  while (i<=vec.nr) and (top2>0) do
    begin { 1 }
      Inc(i);
      if vec.info[i][1] in ['x','0'..'9'] then
         begin
           Inc(top1); New(arb); arb^.info := vec.info[i];
           arb^.stg := nil; arb^.dr := nil; operand[top1] := arb;
         end
      else
        if vec.info[i][1]='(' then
          begin Inc(top2);operator[top2]:='(' end
        else
         begin { 2 }
           while (top2>0) and
                (not (operator[top2][1] in ['(',')'])) and
                (prioritate(operator[top2][1])>=
                 prioritate(vec.info[i][1]))
             do begin
                 if operator[top2][1] in ['l','c','s','e'] then
```

```
                     begin
                       New(arb); arb^.info := operator[top2];
                       arb^.stg := operand[top1]; arb^.dr := nil;
                       operand[top1] := arb
                     end
                  else begin { + - * / ^ }
                         New(arb); arb^.info := operator[top2];
                         arb^.stg:=operand[top1-1];
                         arb^.dr:=operand[top1];
                         operand[top1-1]:=arb; Dec(top1)
                       end;
                 Dec(top2)
               end; { while - 2}
          if top2>0 then
             if (operator[top2] <> '(') or
                (vec.info[i][1] <> ')') then
                begin
                  Inc(top2);
                  operator[top2]:=vec.info[i]
                end
             else Dec(top2)
      end; { else ...}
  end; { while }
  if (i=vec.nr) and (top2=0) then
     begin
       New(arb);
       arb := operand[1]
     end
  else
     begin
       New(arb);
       arb^.info:='old';
       arb^.stg:=nil;
       arb^.dr:=nil
     end
end;
{$S-}
procedure Tipareste(arb: arbore; nivel,x0: Integer);
const dy=30;
var i, dx: Integer;
begin
  dx := GetMaxX; for i := 1 to nivel do dx := dx div 2;
  if arb=nil then OutTextXY(x0+dx, nivel*dy, #207)
  else begin
         OutTextXY(x0+dx, nivel*dy, arb^.info);
         Line(x0+dx, nivel*dy+5, x0+dx div 2, (nivel+1)*dy-5);
         Line(x0+dx, nivel*dy+5, x0+dx+dx div 2, (nivel+1)*dy-5);
         Tipareste(arb^.stg,nivel+1,x0);
         Tipareste(arb^.dr,nivel+1,x0+dx)
       end
end;
{$S+}

{$S-}
procedure FormaPostfixata(arb: arbore; var expr: expresie);
var expr1, expr2: expresie;
begin
  expr := '';
  if arb<>nil then
     begin
       FormaPostfixata(arb^.stg, expr1);
       FormaPostfixata(arb^.dr, expr2);
       expr := expr1 + ' ' + expr2 + ' '+arb^.info
     end
end;
{$S+}
begin { programul principal }
  ClrScr; WriteLn(' Dati expresia ! ');
  ReadLn(expresia); expresia := expresia+')';
  vectorizeaza(expresia,vectorul); New(arborele);
  Arborizeaza(vectorul,arborele);
```

```
  OpenGraph; SetTextJustify(CenterText, CenterText);
  OutTextXY(GetMaxX div 2, 10,Copy(expresia,1,Length(expresia)-1));
  FormaPostfixata(arborele, expr_fpo);
  OutTextXY(GetMaxX div 2, GetMaxY-30, expr_fpo);
  Tipareste(arborele,1,0); Dispose(arborele);
  ReadLn; CloseGraph
end.
```

Atenție

După cum se observă şi din corpul programului principal anterior, paşii algoritmului sunt:

* se citeşte expresia în forma normală: `expresia`;
* se memorează expresia într-un tablou (`vector`), cu care se va lucra în continuare;
* se construieşte arborele asociat expresiei, din acest vector;

Se obține şi forma postfixată a arborelui, apoi arborele se afişează sub formă grafică.

9.3.3. Arbori oarecare

La începutul paragrafului am definit noțiunea de arbore (oarecare) apoi am lucrat cu arbori binari (în secțiunea 6.3.1).

Un arbore oarecare are mai mulți fii. Pentru aceasta, fiecare nod va fi memorat prin informația sa şi printr-un set de legături către nodurile fii. De aceea, va trebui să ştim câți fii are fiecare nod şi să păstrăm legăturile către aceşti fii într-un vector de pointeri către alte noduri.

```
type arbore = ^nod;
     nod = record
             NrFii:Byte;
             info:Integer;
             fiu: array[1..10] of arbore
           end;
```

Crearea şi căutarea într-un arbore oarecare

Un arbore oarecare se poate crea recursiv, o dată cu citirea informațiilor nodurilor sale, de la tastatură, ca în procedura de mai jos:

```
procedure Citeste(var A: arbore; nr, parinte: Integer);
var i: Byte;
begin
  New(A); Write('Dati info pt. fiul ',nr,' al lui ', parinte,': ');
  ReadLn(A^.info);
  if A^.info<>0 then
     begin
       Write(‘Dati numarul de fii’, ‘pentru nodul ‘,A^.info,’ : ‘);
       ReadLn(A^.NrFii);
       for i:=1 to A^.NrFii do
           Citeste(A^.fiu[i],i,A^.info);
     end
  else
     A:=nil
end;
```

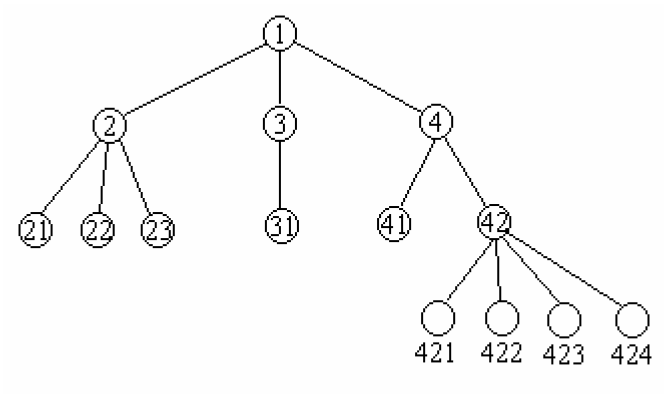
La început, procedura `Citeste` se va apela pentru nodul rădăcină al întregului arbore, astfel: `Citeste(A,0,0)`. Acest lucru va duce la crearea unei legături `Nil`.

Argumentele procedurii sunt:

- `A`: pointer către nodul curent din subarborele ce urmează a fi creat;
- `parinte`: informația din nodul părinte al nodului curent
- `nr`: numărul nodului curent ca fiu al nodului părinte.

De pildă, pentru figura de mai jos, dacă se apelează această procedură pentru subarborele cu rădăcina în nodul 4, atunci vom avea:

- `A` = pointer către nodul `4`;
- `parinte` = nodul `1`;
- `nr = 3`, deoarece nodul `4` este al treilea fiu al nodului părinte `1`.

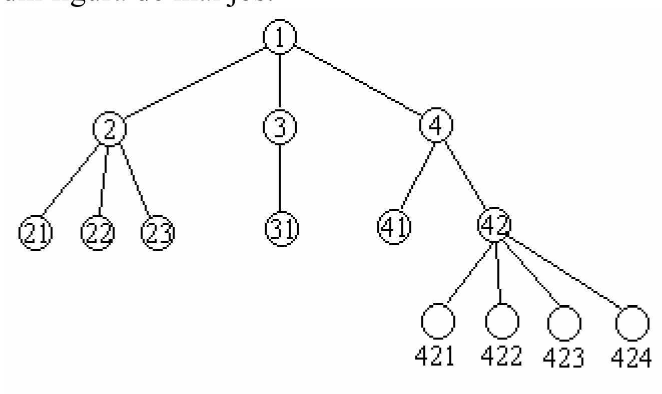


Propunem cititorului să încerce singur realizarea unei proceduri care să parcurgă în adâncime şi să caute un element într-un astfel de arbore.

Memorarea arborilor oarecare prin arbori binari

Un arbore oarecare poate fi memorat printr-un arbore binar.

Într-adevăr, să considerăm un arbore oarecare, cu informațiile din noduri numere întregi, ca arborele din figura de mai jos.

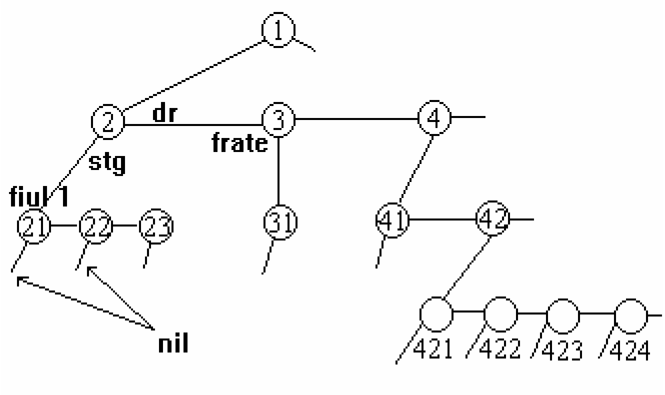


Observăm că putem lega rădăcina `1` de nodul `2`, iar apoi, nodul `2` poate fi legat de primul fiu al său (`21`) şi de următorul fiu al rădăcinii, deci `3`, despre care se spune că este un *frate* a lui `2`.

Procedând astfel pentru toate nodurile din arbore, vom obține un arbore binar, cu două legături:

- cea din stânga este către primul fiu,
- cea din dreapta către primul frate din dreapta al acestui fiu.

Astfel, arborele din figura anterioară se va memora în arborele binar de mai jos:



Vom prezenta mai jos un program demonstrativ care va face următoarele:

- va citi un arbore oarecare `A` de la tastatură;
- va afişa acest arbore, pe vericală, aşa cum este afişată structura de directoare `DOS`, în urma unei comenzi `TREE`;
- va memora arborele `A` sub forma unui arbore binar `A2` (de tip `arbore2`);
- va afişa, în mod asemănător, acest arbore `A2`.

În privinţa procedurilor care execută aceste operaţii ale programului, trebuie să precizăm următoarele:

♦ procedura `Citeste` preia de la tastatură datele despre un arbore oarecare `A` în felul următor: se citeşte informaţia dintr-un nod (cel rădăcină), apoi numărul de fii pe care îi are acest nod; pentru fiecare fiu se va apela recursiv procedura, citindu-se subarborii care au ca rădăcini respectivii fii;

♦ procedura `TiparireInPreordine` (precum cazul ei particular dat de procedura `TiparireInPreordine2`) foloseşte caractere grafice speciale, pentru a desena structura de arbore. Aceste caractere grafice speciale formează diferite *antete*, care se afişează înaintea unui nod. (Aceste antete pot fi spaţii sau antete de verticale, care semnifică că suntem pe un nivel inferior, sau chiar antete depinzând de numărul de ordine al nodului, ca fiu al tatălui său. Steluţa (‘`*`’)simbolizează `Nil`.);

♦ procedura `TransfArb` este cea mai importantă, ea realizând transformarea unui arbore oarecare `A` într-un arbore binar `A2`; procedura este recursivă şi se apelează pentru primul fiu al rădăcinii lui `A` (notat `fiu1`), precum şi pentru toţi ceilalţi fii (între `2` şi `NrFii`), care se înlănţuie la dreapta, în arborele `A2`.

```
program TransformArbOarecareInArbBinar;
uses Crt;
const max=10;
type arbore = ^nod;
     nod=record
           NrFii:Byte;
           info:Integer;
           fiu: array[1..10] of arbore
         end;
     arbore2 = ^nod2;
     nod2 = record
              info: Integer;
              stg,dr: arbore2
            end;
```

```
procedure TransfArb(a: arbore; var a2: arbore2);
var fiu1:arbore; c,b,d: arbore2; i: Byte;
begin
  if a<>nil then
```

```
     begin
       New(a2);
       a2^.info:=a^.info;
       a2^.stg:=nil;
       a2^.dr:=nil;
       if a^.NrFii>0 then
          begin
            fiu1:=a^.fiu[1];
            TransfArb(fiu1,b);
            a2^.stg:=b; d:=b;
            for i:=2 to a^.NrFii do
                begin
                  TransfArb(a^.fiu[i],c);
                  b^.dr:=c;
                  b:=b^.dr
                end
          end
     end
  else
     a2:=nil
end;
```

```
procedure TiparireInPreordine2(A: arbore2);
const stinga=#195#196#196;
      dreapta=#192#196#196;
      vertical=#179#32#32;
      MaxNivele=20;
var antet: array[1..MaxNivele] of String;
    k: Integer;

  procedure TPreordine(curent: arbore2);
  var i: Integer;
  begin
    for i:=0 to k do
        Write(antet[i]);
    if curent<>nil then
       WriteLn(curent^.info)
    else
       WriteLn('*');
    if curent<>nil then
       if (curent^.stg<>nil) or (curent^.dr<>nil) then
          if k<MaxNivele then
             begin
               if antet[k]=stinga then
                  antet[k]:=vertical
               else
                  antet[k]:= ' ';
               k:=k+1; antet[k]:=stinga;
               TPreordine(curent^.stg);
               antet[k]:=dreapta;
               TPreordine(curent^.dr); k:=k-1
             end
  end;
begin
  k:=0;
  antet[k]:= '-->';
  TPreordine(A)
end;
procedure Citeste(var A: arbore; nr, parinte: Integer);
var i: Byte;
begin
  New(A);Write('Dati info pt. fiul ',nr, ' al lui ', parinte, ': ');
  ReadLn(A^.info);
  if A^.info<>0 then
     begin
       Write('Dati numarul de fii', 'pentru nodul ',A^.info,' : ');
       ReadLn(A^.NrFii);
       for i:=1 to A^.NrFii do
           Citeste(A^.fiu[i],i,A^.info);
     end
  else
     A:=nil
```

```
end;
procedure TiparireInPreordine(A: arbore);
const stinga=#195#196#196;
      dreapta=#192#196#196;
      vertical=#179#32#32;
      MaxNivele=20;
var antet: array[0..MaxNivele] of String[3]; k: Integer;
  procedure TPreordine(curent: arbore);
  var j,i: Integer;
  begin
    for i:=0 to k do
        Write(antet[i]);
    if curent<>nil then
       WriteLn(curent^.info)
    else
       WriteLn('*');
    if curent<>nil then
       if curent^.NrFii>0 then
          if k<MaxNivele then
             begin
               if antet[k]=stinga then
                  antet[k]:=vertical
               else
                  antet[k]:='   ';
               k:=k+1;
               for j:=1 to curent^.NrFii-1 do
                   begin
                     antet[k]:=stinga;
                     TPreordine(curent^.fiu[j])
                   end;
               antet[k]:=dreapta;
               with curent^ do
                    TPreordine(fiu[NrFii]);
               k:=k-1
             end
  end;
begin
  k:=0; antet[k]:='-->'; TPreordine(A)
end;
var A: arbore; A2: arbore2;
begin
  ClrScr;
  Citeste(A,0,0);
  TiparireInPreordine(A);
  TransfArb(A,A2);
  TiparireInPreordine2(A2);
  ReadLn
end.
```

Iată un exemplu de funcționare a programului anterior pentru arborele oarecare considerat în figura de mai înainte:

```
Dati info pt. fiul 0 al lui 0: 1
Dati numarul de fii pentru nodul 1 : 3
Dati info pt. fiul 1 al lui 1: 2
Dati numarul de fii pentru nodul 2 : 3
Dati info pt. fiul 1 al lui 2: 21
Dati numarul de fii pentru nodul 21 : 0
Dati info pt. fiul 2 al lui 2: 22
Dati numarul de fii pentru nodul 22 : 0
Dati info pt. fiul 3 al lui 2: 23
Dati numarul de fii pentru nodul 23 : 0
Dati info pt. fiul 2 al lui 1: 3
Dati numarul de fii pentru nodul 3 : 1
Dati info pt. fiul 1 al lui 3: 31
Dati numarul de fii pentru nodul 31 : 0
Dati info pt. fiul 3 al lui 1: 4
Dati numarul de fii pentru nodul 4 : 2
Dati info pt. fiul 1 al lui 4: 41
```

```
Dati numarul de fii pentru nodul 41 : 0
Dati info pt. fiul 2 al lui 4: 42
Dati numarul de fii pentru nodul 42 : 4
Dati info pt. fiul 1 al lui 42: 421
Dati numarul de fii pentru nodul 421 : 0
Dati info pt. fiul 2 al lui 42: 422
Dati numarul de fii pentru nodul 422 : 0
Dati info pt. fiul 3 al lui 42: 423
Dati numarul de fii pentru nodul 423 : 0
Dati info pt. fiul 4 al lui 42: 424
Dati numarul de fii pentru nodul 424 : 0
-->1
   +--2
   ¦  +--21
   ¦  +--22
   ¦  +--23
   +--3
   ¦  +--31
   +--4
      +--41
      +--42
         +--421
         +--422
         +--423
         +--424
-->1
   +--2
   ¦  +--21
   ¦  ¦  +--*
   ¦  ¦  +--22
   ¦  ¦     +--*
   ¦  ¦     +--23
   ¦  +--3
   ¦     +--31
   ¦     +--4
   ¦        +--41
   ¦        ¦  +--*
   ¦        ¦  +--42
   ¦        ¦     +--421
   ¦        ¦     ¦  +--*
   ¦        ¦     ¦  +--422
   ¦        ¦     ¦     +--*
   ¦        ¦     ¦     +--423
   ¦        ¦     ¦        +--*
   ¦        ¦     ¦        +--424
   ¦        ¦     +--*
   ¦        +--*
   +--*
```

9.3.4. Vizualizarea structurii arborescente de directoare

Ca o aplicație practică interesantă a arborilor oarecare, vom realiza împreună un program care va vizualiza structura arborescentă de directoare, cu rădăcina în directorul curent, de pe unitatea de disc curentă. Astfel, programul va funcționa precum comanda `TREE` din sistemul de operare `DOS`.

Vom construi, pe baza programului din lecția anterioarâ, arborele asociat structurii de directoare respective, apoi vom afişa acest arbore în preordine, pe verticală.

Nodurile arborelui vor fi numele de directoare. Pentru a vedea ce subdirectoare are un anumit director (în care ne aflăm la un moment dat), se folosesc două proceduri speciale `FindFirst` şi `FindNext`, care găsesc primul şi respectiv următorul fişier din directorul curent, care au un anumit format sau tip.

Dintre acestea se selectează doar acelea a căror atribut conține identificatorul de director (căci din punctul de vedere al sistemului de operare, un director este tot un fişier, care are un anumit atribut special).

Atenție

Vor face excepție directoarele '`.`' şi '`..`' ce corespund directorului curent, respectiv directorului părinte.

Subdirectoarele curente vor fi fii nodului ce corespunde directorului curent. Apoi, cu procedura `ChDir` vom schimba directorul, trecând din nodul curent într-unul din fii, pentru care se va construi, la fel, subarborele corespunzător (acest fiu va fi rădăcină în noul arbore).

În continuare, Ne întoarcem la tatăl subdirectorului (prin `ChDir('..')`), apoi trecem în următorul fiu, pentru care procedăm la fel, şi tot aşa, până se epuizează fii.

Procedura recursivă care construieşte arborele cu rădăcina într-un director se numeşte `Construieste`. Ea se va apela din programul principal pentru directorul curent. La sfârşit, când arborele întreg este construit, acesta se afişează cu procedura `TiparireInPreordine`, care la rândul său cuprinde o subprocedură recursivă `TPreordine`.

În cele ce urmează prezentăm programul şi comentăm diferite utilizări ale subprogramelor din biblioteca DOS, care au fost folosite în el.

```
program Arb;
{ comanda "Tree" }
uses Dos;
const max=500;
type TipInfo = String[12]; { informatia din nodurile arborelui este un nume de
fisier, cu tot cu extensie }
     arbore = ^nod;
     nod = record
             info: TipInfo; NrFii: Byte;
             fiu: array[1..max] of arbore
            end;
procedure TiparireInPreordine(A: arbore);
const stinga=#195#196#196#196; dreapta=#192#196#196#196;
      vertical=#179#32#32#32; MaxNivele=20;
var antet: array[0..MaxNivele] of String[4];
    k: Integer;
  procedure TPreordine(curent: arbore);
  var j,i: Integer;
  begin
    for i:=0 to k do
        Write(antet[i]);
    if curent<>nil then
       WriteLn(curent^.info)
    else
       WriteLn('*'); { nu se afisa niciodata }
    if curent<>nil then
       if curent^.NrFii>0 then
          if k<MaxNivele then
             begin
               if antet[k]=stinga then
                  antet[k]:=vertical
               else
                  antet[k]:=#32#32#32#32; { patru spatii }
               k:=k+1;
               for j:=1 to curent^.NrFii-1 do
                   begin
                     antet[k]:=stinga; { pentru primii fii }
                     TPreordine(curent^.fiu[j])
                   end;
               antet[k]:=dreapta; { ultimul fiu }
               TPreordine(curent^.fiu[curent^.NrFii]); k:=k-1
```

```
            end {TPreordine }
    end;
  begin
    k:=0;
    antet[k]:='--> ';
    TPreordine(A)
  end;
```

Procedura `Construieste` va căuta, cu `FindFirst` şi `FindNext`, toate fişierele (deci cu şablonul '`*.*`') `FisDirector` din directorul curent, care sunt directoare (au atributul `Directory`).

Directorul curent se află cu apelul `GetDir(0,P)`, în care `P` este o variabilă de tip `PathStr`, adică un şir de caractere corespunzător căii curente, iar `0` este o constantă, asociată unei anumite unităţi de disc: `0` = unitatea curentă, `1=A`, `2=B` etc..

Din tot şirul `P`, reprezentând calea curentă, ne interesează numai numele directorului şi extensia sa (dacă directorul are extensie). Astfel, se desparte `P` în mai multe subşiruri prin: `FSplit(P,D,N,E)`, obţinându-se `D` = calea până la directorul părinte inclusiv, iar `N` şi `E` vor fi numele directorului şi respectiv extensia sa. Acestea din urmă se compun pentru a forma informaţia ce va fi ataşată nodului rădăcină al arborelui: `A^.info:=N+E`.

O dată determinat directorul curent, se caută toţi fii săi prin secvenţa:

- determină primul fiu, dacă există:
```
     FindFirst('*.*', Directory, FisDirector);
```
- cât timp există încă un fiu (fapt semnalat prin valoarea `0` în variabila de sistem `DosError` care e declarată în biblioteca `DOS`):
```
     while DosError = 0 do
           begin
```
- asignează numele fişierul fizic `FisDirector.Name` la variabila fişier `F` şi determină-i atributele (cu procedura `GetFAttr`):
```
             Assign(F, FisDirector.Name);
             GetFAttr(F,Attr);
```
- dacă printre aceste atribute se află şi cel de director (constanta `Directory`, cu valoare `16`, declarată în unit-ul `DOS`), iar directorul nu este nici '`.`' şi nici '`..`', atunci adaugă fişierul director fiilor rădăcinii arborelui:
```
         if (Attr and Directory<>0) and
            (FisDirector.Name<>'.') and
            (FisDirector.Name<>'..') then
             begin
               Inc(A^.NrFii);  New(A^.fiu[A^.NrFii]);
               A^.fiu[A^.NrFii]^.info:=FisDirector.Name
             end;
```
- apoi treci la căutarea următorului fişier care se potriveşte şablonului dorit:
```
             FindNext(FisDirector)
           end;
```

În final, se va merge în fiecare dintre subdirectoarele fii şi se va proceda recursiv pentru continuarea construirii structurii arborescente.

Procedura completă şi programul principal sunt prezentate mai jos.

```
procedure Construieste(var A: arbore);
var FisDirector: SearchRec; F: File;
    Attr: Word; { adica intreg intre 0 si 65535 }
    P: PathStr; { String[80] }
    D: DirStr; { String[67] } N: NameStr; { String[8] }
    E: ExtStr; { String[4] } i: Integer;
begin
     A^.NrFii:=0; GetDir(0,P); FSplit(P,D,N,E); A^.info:=N+E;
     FindFirst('*.*', Directory, FisDirector);
     while DosError = 0 do
       begin
```

```
          Assign(F, FisDirector.Name); GetFAttr(F,Attr);
          if (Attr and Directory<>0) and (FisDirector.Name<>'.')
             and (FisDirector.Name <> '..') then
             begin
               Inc(A^.NrFii); New(A^.fiu[A^.NrFii]);
                A^.fiu[A^.NrFii]^.info:=FisDirector.Name
             end;
          FindNext(FisDirector)
        end;

      for i:=1 to A^.NrFii do
          begin
            ChDir(A^.fiu[i]^.info);
            Construieste(A^.fiu[i]); ChDir('..')
          end
end;
var A: arbore; Cale: PathStr;
begin
     New(A); GetDir(0,Cale);
     WriteLn('Arborele directorului ',Cale);
     Construieste(A); TiparireInPreordine(A); WriteLn
end.
```

În urma executării acestui program, se va afişa o structură arborescentă de genul:

```
Arborele directorului C:\MSOFFICE
--> MSOFFICE
    ├───SETUP
    │   └───MS-BTTNS
    ├───WINWORD
    │   ├───TEMPLATE
    │   ├───MACROS
    │   ├───STARTUP
    │   ├───LETTERS
    │   └───WORDCBT
    ├───EXCEL
    │   └───LIBRARY
    │       └───MSQUERY
    └───CLIPART
        └───PCSFILES
```

Ca exerciţiu, propunem cititorului să realizeze un program similar în care să fie afişate şi fişierele obişnuite conţinute în directoare, eventual cu litere mici şi după subdirectoarele fii.

Probleme

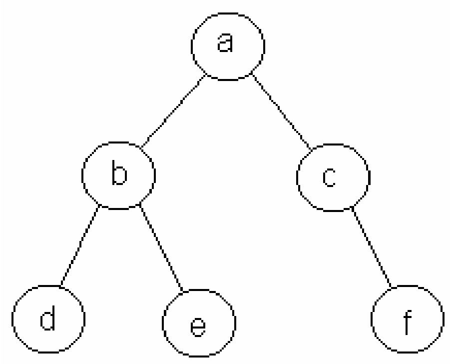
1 ☺ Dată fiind o listă de numere întregi, să se creeze o listă ce să cuprindă doar elementele pare din prima listă. În continuare, să se concateneze cele două liste.

2 ☺ Pentru o stivă dată, să se creeze o coadă, scoţând câte un element din stivă şi adăugându-l la coadă.

3 😐 Să se sorteze un fişier text, alfabetic, preluând liniile fişierului într-o listă dinamică (simplu sau dublu înlănţuită).

4 😐 Scrieţi un program care să creeze listeze în preordine nodurile unui arbore, fiecare nod fiind urmat de lista celor doi fii ai săi cuprinşi între paranteze rotunde. De exemplu, pentru arborele din figura de mai jos avem lista: `a(b (d (#,#),e(#,#)), c(#, f(#,#)).`

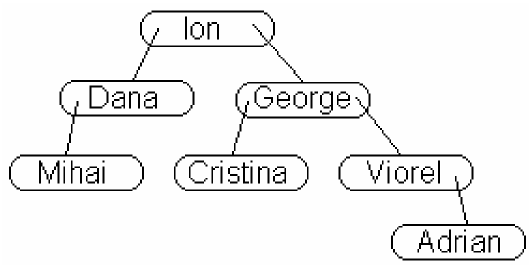
Simbolul # indică pointerul `Nil`.



5 ☹ Pentru problema anterioară, lista să se creeze dinamic, în loc să se doar o afişare a sa.
6 ☹ Se consideră un arbore binar, informaţiile din noduri fiind numele unor persoane. Arborele este creat recursiv în preordine, după regula următoare: dacă se introduce simbolul `#`, atunci este `Nil`; dacă nu, atunci se pune informaţia citită (numele unei persoane) şi se apelează procedura de creare pentru fiul din stânga şi pentru cel din dreapta. Pentru un astfel de arbore scrieţi o funcţie care să verifice dacă două persoane sunt sau nu fraţi.

De exemplu, în figura de mai jos este reprezentat un arbore care a fost creat în preordine după şirul de cuvinte: *Ion*, *Dana*, *Mihai*, #, #, #, *George*, *Cristina*, #, #, *Viorel*, #, *Adrian*, #, #.

Cristina şi Viorel, ca şi Dana şi George sunt fraţi, pe când Viorel şi Mihai nu sunt. (Se consideră nume de persoane distincte.).



7 💣 Scrieţi un program care să evalueze o expresie aritmetică ce conţine doar paranteze, operatorii de adunare, scădere, înmulţire şi împărţire şi numere reale. Se vor folosi două stive, una a operanzilor, alta a operatorilor. Acestea se vor implementa dinamic.
8 💣 Un caz aparte al arborilor oarecare este cel al arborilor multicăi în care fiecare nod are un vectori cu mai multe componente şi o serie de pointeri către alte noduri din arbore. Se mai numesc şi arbori B.

Pentru astfel de noduri avem declaraţia:

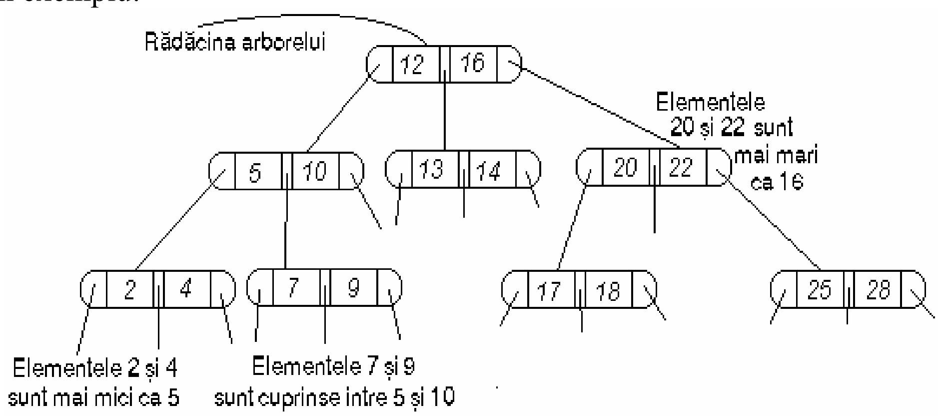
```
const max=10;
type arboreB = ^nod;
     nod = record
            n: Byte;
            fiu: array[0..max] of arboreB;
            info: array[1..max] of Integer
          end;
```

În fiecare nod al arborelui se consideră verificate condiţiile:

- `info[i]<=info[i+1], ∀i=1,n-1;`
- `fiu[i]=Nil, ∀i=n+1,max;`
- dacă `fiu[i-1]≠Nil` şi/sau `fiu[i]≠Nil` atunci:
  `fiu[i-1]^.info[j]≤info[i]≤fiu[i]^.info[k],`
  `∀i=1,n, ∀j=1,fiu[i-1]^.n; ∀k=1,fiu[i]^.n.`

Cu alte cuvinte, elementele cu informaţii din fiecare nod sunt ordonate crescător, iar între oricare două elemente de pe poziţii consecutive se află un pointer către un nod, care, dacă nu este `Nil`, atunci conţine elemente cuprinse între cele două elemente din nodul dat. De asemenea, există un pointer în faţă către elemente mai mici decât primul element din nod, precum şi un pointer la sfârşit către elemente mai mari decât ultimul element din nod.

Iată un exemplu:



Se cere să se scrie proceduri care să creeze un astfel de arbore, să caute un element într-un astfel de arbore, să-l afişeze grafic sau să şteargă.

9 Creaţi o structură dinamică de date corespunzătoare pentru crearea unei matrice, apoi rotiţi această matrice cu 90 de grade în sens trigonometric.

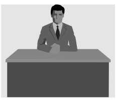
10 Aceeaşi problemă ca cea anterioară, doar că se cere să se oglindească matricea faţă de axa verticală ce trece prin mijlocul matricei.

Rezumat

1. Limbajul Turbo Pascal permite crearea de variabile dinamice, pentru care se alocă memorie în zona heap. Lor li se pune în corespondenţă variabile statice de tip referinţă (numite pointeri).
2. Alocarea de memorie pentru o variabilă dinamică referită prin variabila referinţă `p` se realizează cu `New(p)`. Eliberarea spaţiului de memorie respectiv se face cu `Dispose(p)`.
3. Folosind pointeri se pot defini structuri de date care se autoreferă (recursive). Acestea sunt listele simplu sau dublu înlănţuite şi arborii.
4. Un caz particular de liste simplu înlănţuite sunt stivele, care funcţionează după mecanismul LIFO. Alt caz particular este reprezentat de cozi, care funcţionează după mecanismul FIFO.
5. Operaţiile specifice unei liste sunt: parcurgere şi afişare, ştergerea, inserarea şi adăugarea unui element etc.
6. Arborii binari sunt cazuri particulare de arbori, în care fiecare nod conţine o informaţie şi doi pointeri către doi fii, stâng şi drept. Studiul arborilor oarecare se poate reduce la studiul arborilor binari.
7. Un caz particular de arbori binari sunt arborii de căutare, în care informaţia din fiecare nod este mai mare decât informaţia din nodul fiului stâng şi mai mică sau egală cu cea din nodul fiului drept.

Capitolul 10. Probleme recapitulative

În continuare sunt prezentate mai multe probleme care pot fi rezolvate prin metodele de programare învățate anul acesta sau nu. Fireşte, alegerea sau nealegerea unei anumite metode sau tehnici, a unui anumit algoritm, aplicarea unei anumite scheme de rezolvare, eventual îmbunătățite, revine la latitudinea rezolvitorului.



În atenția profesorului

Problemele propuse au diferite grade de complexitate, iar ordinea în care apar este pur întâmplătoare. Unele probleme pot fi date ca teme pentru acasă, iar altele pot constitui probleme de concursuri şi olimpiade şcolare. Rămâne la latitudinea profesorului să aleagă acele probleme necesare atingerii unui scop didactic pe care şi l-a propus, în funcție de pregătirea studenților, de nivelul grupei, de materia parcursă, de contextul în care se cer a fi rezolvate.

1. Să se aşeze `2n-2` nebuni pe o tablă de şah cu n^2 pătrate astfel încât nici o pereche de nebuni să nu se amenințe.

2. Să se aşeze pe o tablă de şah cu n^2 pătrate cât mai multe dame care să nu se atace între ele.

3. Scrieți o procedură recursivă şi una iterativă pentru a căuta un cuvânt într-un vector de cuvinte, care sunt puse în ordine alfabetică.

4. Pe produsul cartezian $N \times N$ se defineşte operația: `(a,b)(b,c)=(a,c)`, pentru orice a, b şi $c \in \mathbf{N}$, despre care ştim că:
este asociativă: `((a,b)(b,c))(c,d)=(a,b)((b,c)(c,d))`;
efectuarea ei necesită exact $a \times b \times c$ secunde.
Fiind date $x_1, x_2, ..., x_n \in \mathbf{N}$, $n \geq 3$, care este timpul minim şi cel maxim în care se poate efectua produsul $(x_1,x_2)(x_2,x_3)...(x_{n-1},x_n)$?
De exemplu, pentru produsul `(7,1)(1,9)(9,3)` avem rezultatul `(7,3)`, care se poate obține cel puțin în 48 secunde şi cel mult în 4 minute şi 12 secunde.

5. Un țăran primeşte o bucată dreptunghiulară de pământ pe care doreşte să planteze o livadă. Pentru aceasta, el va împărți bucata de pământ în $m \times n$ pătrate, având dimensiunile egale, iar în fiecare pătrat va planta un singur pom din cele patru soiuri pe care le are la dispoziție. Să se afişeze toate variantele de a alcătui livada respectând următoarele condiții:
a) Nu trebuie să existe doi pomi de acelaşi soi în două căsuțe învecinate ortogonal sau diagonal.
b) Fiecare pom va fi înconjurat de cel puțin un pom din toate celelalte trei soiuri.
Observație: ææranul are la dispoziție suficienți pomi de fiecare soi.

6. Un teren muntos are forma unei matrice cu $m \times n$ zone, fiecare zonă având o înălțime. Un alpinist pleacă dintr-o anumită zonă şi trebuie să ajungă într-o zonă maximă în altitudine. Dintr-o zonă, alpinistul se poate deplasa diagonal sau ortogonal, într-una din zonele (căsuțele) alăturate, doar urcând sau mergând la acelaşi nivel. Poate el ajunge într-unul din vârfuri? Dacă da, arătați toate soluțiile problemei.

7. Se citesc numere întregi de la tastatură, până la întâlnirea numărului 0. Se cere să se creeze două liste, una a numerelor negative, iar alta a numerelor pozitive prime.

8. Se dă o listă de numere întregi pozitive. Să se creeze o listă care să conțină doar numerele pare, apoi să se concateneze cele două liste.

9. Se dă un arbore oarecare, informațiile din noduri fiind şiruri de caractere. Să se construiască o listă dublu înlănțuită care să conțină toate şirurile din nodurile arborilor, care au lungimile pare, apoi să se ordoneze această listă.

10. `N` pitici aşezați unul în spatele celuilalt poartă căciuli colorate roşii sau albe. Fiecare pitic spune două numere, primul reprezentând numărul de căciuli albe, respectiv roşii pe care le poartă piticii din fața sa.

a) Çtiind că piticii cu căciulă roşie mint (dau incorect cel puțin unul din cele două numere), iar cei cu căciulă albă spun întotdeauna adevărul, să se determine culoarea căciulii fiecărui pitic.

Se vor citi de la tastatură: numărul `n` de pitici şi cele `n` perechi de numere. Se va tipări pe ecran o succesiune de litere *A* şi *R* reprezentând culorile alb, respectiv roşu ale căciulilor în ordinea în care stau piticii în şir.

b) Çtiind că fiecare pitic îşi păstrează culoarea căciulii determinată la punctul (a), să se afle dacă este posibilă schimbarea ordinii piticilor în şir astfel încât toți piticii să spună adevărul. În caz afirmativ, se vor tipări numerele de ordine inițiale ale piticilor în noua ordine stabilită.

Exemplu: Pentru datele de intrare:

`5, (2,1), (0,1), (1,1),(0,0),(2,2)`

se obțin rezultatele: a) `RAARA` şi b) `DA: 4,2,3,1,5`.

11. Să se tipărească toate permutările circulare ale unui vector de numere reale dat. De exemplu, o permutare circulară a şirului `5,7,25,8,-1,30,2` este şirul: `8,-1,30,2,5,7,25`.

12. Pentru un vector dat, să se determine o secvență de lungime maximă care formează o progresie aritmetică. De exemplu, pentrui vectorul `5,2,15,23,2,4,6,-1,33.5,81,21,-19` avem două soluții: `2,4,6` şi `81,21,-19`.

13. Într-un grup de `n` persoane, se cunosc perechile `(i,j)` cu semnificația că persoana `i` îi comunică persoanei `j` orice bârfă. Să se determine dacă în acest grup se va transmite o bârfă tuturor persoanelor, o dată ce bârfa a fost auzită de una din persoanele din grup.

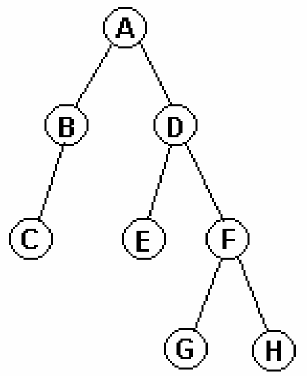
14. Într-un grup de `n` persoane se precizează perechi de persoane care se consideră prietene. Folosind principiul că "prietenul prietenului meu mi-este prieten", să se determine grupurile cu un număr maxim de persoane între care se pot stabili relații de prietenie, directe sau indirecte.

15. Rețeaua de distribuire a apei calde pentru o centrală termică zonală este formată dintr-un sistem de conducte care leagă centrala de blocuri şi blocurile între ele. Centrala se consideră a fi punctul 0 de distribuire, iar fiecare bloc are asociat un număr `i`. Se cunosc distanțele de la centrală la blocuri, precum şi distanțele între oricare două blocuri. Să se afişeze perechile de numere desemnând punctele de distribuire între care trebuie să se monteze conducte astfel încât fiecare bloc să fie alimentat cu apă caldă (nu neapărat direct de la centrală) şi lungimea totală a conductelor necesare să fie minimă.

16. Un elev vrea să călătorească din localitatea `X` în localitatea `Y`. Dacă în țara respectivă există n localități şi ştiind timpul necear pentru a ajunge dintr-o localitate în alta (în cazul în care se poate ajunge direct) se cere să se determine timpul minim în care elevul poate să ajungă din `X` în `Y`.

17. Se pun în memorie (heap) două numere. Să se interschimbe valorile lor utilizând numai adrese.

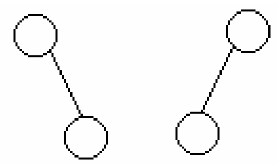
18. Să se determine înălțimea (adică numărul de nivele) a unui arbore binar. De exemplu arborele din figura următoare are înălțimea 3.



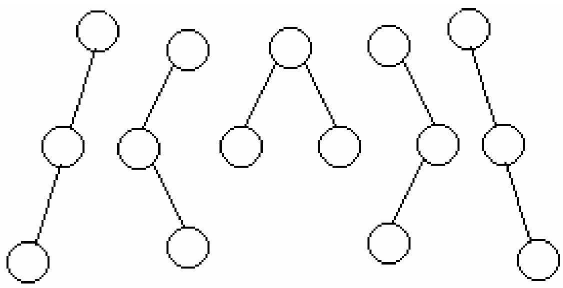
19. Se dau secvențele obținute prin parcurgerile în preordine şi în inordine ale unui arbore binar. Construiți arborele binar corespunzător.

De exemplu, fie `A,B,C,D,E,F,G,H` parcurgerea în preordine şi `C,B,A,E,D,G,F,H` parcurgerea în inordine. Arborele va fi cel din figura din problema anterioară.

20. Să se determine numărul de arbori binari distincți cu `n` noduri, făcând abstracție de numerotarea nodurilor. De exemplu, pentru n=1 există un singur arbore binar, pentru n=2 există doi arbori binari distincți:



iar pentru n=3 există 5:



21. Fie `n` secvențe S_1, S_2, ..., S_n de lungimi respectiv L_1, L_2, ..., L_n, ordonate nedescrescător. Să se interclaseze cele n secvențe.

22. Scrieți o variantă iterativă şi una recursivă pentru determinarea nodului cu informația cea mai mică dintr-un arbore binar. Aceeaşi problemă pentru cazul unui arbore binar de căutare.

23. Se consideră un caroiaj dreptunghiular cu `m` linii şi `n` coloane, în care anumite poziții sunt ocupate (interzise), precum şi o poziție inițială `(i0,j0)`, considerată liberă. Se cere să se determine pentru toate pozițiile la care poate ajunge un mobil ce pleacă din punctul inițial `(i0,j0)`, distanța lor față de acest punct, măsurată în deplasări elementare. Se precizează că o deplasare elementară a mobilului constă în repoziționarea sa:

- cu o poziție la dreapta pe aceeaşi linie;
- cu o poziție la stânga pe aceeaşi linie;
- cu o poziție în jos pe aceeaşi coloană;
- cu o poziție în sus pe aceeaşi coloană,

dacă noua poziție este liberă.

24. Memorarea compactă a numerelor prime. O metodă simplă de memorare a numerelor prime este printr-o secvență de 0 şi 1 astfel încât al `n`-lea termen din secvență este 1 dacă şi numai dacă `n` este prim. Pentru memorarea unui element este suficient un singur bit. Un cuvânt-calculator este o secvență de biți, a cărei lungime depinde de tipul calculatorului. Un element al tipului `Integer` este memorat într-un cuvânt-calculator (16 biți = 2 bytes). Să se scrie un program care să construiască tabela compactă a primelor n numere pentru un n dat. Tabela va fi un tablou unidimensional cu elemente de numere întregi şi se va ține seama de lungimea cuvântului pentru calculatorul pe care va fi testat programul.

25. Ciurul lui Eratostene. Numerele prime mai mici decât sau egale cu `n` pot fi memorate într-un vector `x` cu lungimea `n`, astfel încât `x[i]=i` dacă `i` este prim şi `x[i]=-1` dacă nu. Să se scrie un program care să construiască un astfel de vector pentru un `n` dat, utilizând următoarea strategie:

- inițial se pune `x[i]` pentru orice `i`;
- se parcurge secvențial vectorul `x` de la stânga la dreapta şi pentru fiecare element `x[i]` prim (`t[i]=i`) se determină toate elementele `t[k]` cu `k>i`, `t[k]=k` şi `i` divide `k` şi se elimină din mulțimea numerelor presupuse a fi prime (`t[k]:=-1`).

Care este ordinul de complexitate al programului dumneavoastră?

26. Să se arate că orice număr natural `n>2` se poate scrie ca o sumă de numere prime. Să se scrie un program care, pentru un număr natuiral `n>2` dat, determină o secvență de lungime minimă de numere prime a căror sumă este egală cu `n`. Să se arate corectitudinea programului.

27. Să se scrie un program care să determine toate numerele naturale `n` cu proprietățile:

- `n` are patru cifre distincte;
- singurii factori primi ai lui `n` sunt cifrele care îl compun.

28. Să se scrie un program care determină toate perechile de numere naturale prime `(a,b)`, cu `a`, `b` ≤ `n`, pentru un `n` dat, astfel încât `a-b` sau `a+b` este un număr natural prim.

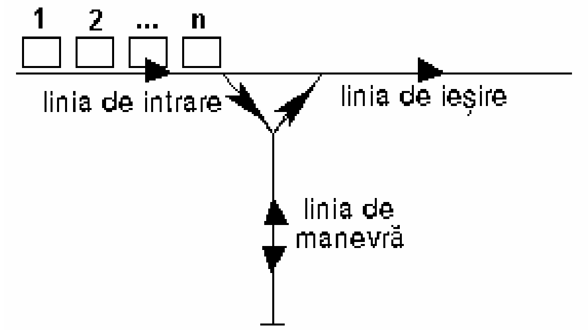
29. Se consideră două numere naturale foarte mari, `a` şi `b`, reprezentate sub forma unor vectori. Să se determine reprezentarea lui a^b.

30. Să se scrie un program care să genereze recursiv permutările unei mulțimi de `n` elemente.

31. Se consideră `n` cuburi de laturi l_i şi culori c_i. Să se determine cel mai mare turn care se poate forma cu aceste cuburi, astfel încât să nu se pună un cub mai mare peste unul mai mic, iar două cuburi vecine să fie de culori diferite. Prin "cel mai mare turn" se va înțelege, pe rând:

a) turnul cu cea mai mare înălțime;
b) turnul cu cele mai multe cuburi componente.

32. Într-un triaj există o linie de cale ferată pentru manevre, ca în figura următoare, pe linia de intrare sunt n vagoane numerotate de la 1 la n. Deplasarea vagoanelor se face numai în sensurile indicate de săgețile din figură. Cunoscând ordinea în care trebuie să fie vagoanele pe linia de ieşire, să se scrie un program care să determine dacă acest lucru este posibil, iar dacă da să se afişeze mutările care trebuie executate pentru a soluționa problema.



De exemplu, pentru cazul în care avem n=3 şi ordinea finală trebuie să fie `3,1,2`, atunci mutările vor fi:

- se mută vagonul 3 de pe linia de intrare pe linia de manevră;
- se mută vagonul 2 de pe linia de intrare pe linia de ieşire;
- se mută vagonul 1 de pe linia de intrare pe linia de ieşire;
- se mută vagonul 3 de pe linia de manevră pe linia de ieşire.

33. Să se interclaseze două liste dublu înlănțuite care sunt ordonate, pentru a obține o altă listă ordonată.

34. Rezolvați problema turnurilor din Hanoi prin metoda Back-tracking, fără a folosi recursivitatea.

35. Se consideră următorul joc de două persoane: Se dau patru grămezi de bete de chibrit care contin respectiv 1,3,5,7 bețe. Fiecare din jucători extrage alternativ unul sau mai multe bețe dntr-o grămadă până când se extrag toate bețele din toate grămezile. Câştigă jucătorul care extrage ultimul. Se cere.
a) Să se realizeze un program care să simuleze jocul între calculator şi un jucător uman implementând o strategie de câştig pentru calculatir, jocul fiind început de jucătorul uman.
b) În aceleaşi condiții de la punctul anterior, să se generalizeze jocul pentru `N` grămezi care conțin fiecare $a_1,a_2,..,a_N$ bețe de chibrit, cu a_i numere impare, `i=1,..,N`.
Observație: Se va evita folosirea unei metode de căutare totală în spațiul soluțiilor.
36. Să se descompună o tablă de şah în numărul maxim de dreptunghiuri disjuncte care satisfac următoarele condiții:
a) fiecare dreptunghi este format din acelaşi număr de pătrate, albe şi negre;
b) nu există două dreptunghiuri cu acelaşi număr de pătrate albe;
Să se determine toate soluțiile posibile.
37. Fanii jocului Scrabble sunt obişnuiți cu anagramele - grupuri de cuvinte cu aceleaşi litere dar în altă ordine (ex: *ACAR, ARAC, CARA*). Există totuşi cuvinte care nu au acest atribut, adică indiferent cum sunt aranjate literele componente, nu se poate forma un alt cuvânt (ex: *MOS*). Asemenea cuvinte sunt numite ananagrame.

Bineînțeles că aceste definiții depind de domeniul în care lucrăm. Un asemenea domeniu poate fi întreg dicționarul limbii române, dar asta ar crea unele probleme. Putem restrânge domeniul, sa spunem de exemplu: domeniul muzical, caz în care *NOTA* devine o ananagrama relativă (pentru că *TONA* nu este în acelaşi domeniu).

Scrieți un program care să citească un dicționar dintr-un anumit domeniu şi să determine toate ananagramele relative. De remarcat că cuvintele formate dintr-o singură literă sunt ananagrame deoarece ele nu pot fi "rearanjate". Un dicționar nu conține mai mult de 1000 de cuvinte. Observație: *tieD* şi *EdiT* sunt anagrame!

Intrarea: Fişierul de intrare conține mai multe dicționare. Fiecare dicționar va consta dintr-o succesiune de linii. Nici o linie nu va avea mai mult de 80 de caractere, dar poate conține oricâte cuvinte. Cuvintele pot avea cel mult 20 de litere (mici sau mari) şi nu pot fi separate pe două linii. Sfârşitul unui dicționar este marcat printr-o linie conținând doar caracterul "#".

Ieşirea: Pentru fiecare dicționar se vor tipări ananagramele, fiecare pe câte o linie, în ordine lexicografică. Un set de rezultate pentru un dicționar se va termina cu o linie conținând doar "#".

Exemplu:

Intrare:
```
ladder came tape soon leader acme RIDE lone Dreis peat
ScAlE orb  eye  Rides dealer  NotE derail LaCeS  drIed
noel dire Disk mace Rob dries
#
```
Ieşire:
```
Disk
NotE
derail
drIed
eye
ladder
soon
#
```
38. Băcanul din oraşul dumneavoastră încearcă o nouă metodă de afişare a prețurilor produselor pe care le are în stoc. În loc de a marca, pentru fiecare articol, prețul lui (în lei), etichetele băcanului au prețuri comparative față de alte produse. De exemplu, untul poate fi etichetat `unt=margarina + 100`, margarina poate fi etichetată `cafea + 111`, câteva articole sunt marcate chiar cu prețul respectiv. Scrieți un program care să facă o listă cu prețurile produselor lui, în maniera obişnuită (adică, de exemplu, `unt=225`).

În legătură cu datele de intrare se presupun următoarele:
- toate articolele au nume formate din maxim 10 litere mici

- există cel mult 100 de articole
- liniile de intrare dau fie prețul unui articol direct, fie îl consideră egal cu prețul altui produs `+/-` un număr de lei
- toate liniile de intrare sunt corecte din punct de vedere sintactic
- fiecare articol apare o singura data în stânga semnului '='.

La ieşire, daca prețul unui produs nu poate fi dedus din fişierul de intrare, dati-l ca fiind `blank`.

De exemplu, pentru intrarea:

```
lapte = zahar - 125
faina = 225
zahar = faina + 10
cafea = ceai
```

ieşirea va fi:

```
cafea   = blank
faina   = 225
lapte   = 110
zahar   = 235
ceai    = blank
```

39. Se spune că demult, la marginile unui regat puternic, trăia un vrăjitor care avea o comoară. Se mai spunea că această comoară depăşea cu mult ca valoare chiar şi bogățiile regelui. Într-o vreme, vrăjitorul, plictisindu-se de ocupațiile sale obişnuite şi dorind să-şi găsească o distracție pe măsura puterii sale, dădu sfoara în țară că o bună parte din comoara sa va putea fi luată de cel care va şti să o câştige. Comoara vrăjitorului era formată din `n` grămezi de monezi de aur (fiecare grămadă conținând un număr oarecare de monezi, grămezile nefiind neapărat egale). Cel care dorea să obțină aur din comoara vrăjitorului trebuia să respecte regulile impuse de acesta şi care erau următoarele:

a) aurul trebuia cărat de exact `n` slujitori (tot atâția câte grămezi);

b) cel care vroia aurul putea ca din cele `n` grămezi să aleagă un număr oricât de mare de grămezi (eventual le putea alege pe toate `n`), astfel încât:

- dacă alege o grămadă, trebuie să ia toate monezile din acea grămadă;
- numărul total de monezi rezultat din toate grămezile alese trebuie să se poată împărți exact la cei `n` slujitori care le vor căra;

c) numărul total de monezi din grămezile alese trebuie să fie maxim posibil.

Altfel dacă vrajitorul îi arată celui care a ales că putea să facă o alegere mai bună, acesta nu mai primea nimic.

Dându-se numărul `n` de grămezi şi, în acelaşi timp de slujitori, numărul `nr[i]` de monezi din fiecare grămadă `i` ($1 \le i \le n$) trebuie găsită mulțimea grămezilor care trebuie alese .

40. Se dă un fişier cu `n` ($n \le 1000000$) numere întregi între `0` şi `n` (inclusiv). Să se afişeze numărul care lipseşte. Observație: între `0` şi `n` sunt `n+1` numere întregi, deci într-adevăr unul dintre ele nu apare în fişier. (Nu uitați ! Problema timpului este esențială!).

41. Problema găsirii arborelui parțial minim al unui graf este foarte cunoscută. De data aceasta problema constă în a găsi (eficient) arborele parțial minim al unui graf cu un număr foarte mare de muchii a cărui reprezentare, deci, nu mai poate fi păstrată în memorie ci trebuie păstrată (şi prelucrată) într-un fişier.

42. Se consideră o tablă liniara formată din `2n+1` căsuțe, în care sunt aşezate `n` piese albe şi `n` piese negre, ca mai jos:

A	A	...	A	-	N	...	N	N

`n` piese   `n` piese

unde **A** reprezintă o piesă albă, **N** o piesă neagră, iar '**-**' o căsuță liberă.

Se cere să se treacă cele `n` piese albe în locul celor negre şi cele negre în locul celor albe (adică să se ajungă în configuraţia `NN...N-AA...A`) ştiind că sunt posibile doar următoarele mutări:

1. `...A_...    ->  ..._A...`
2. `...AN_..    ->  ..._NA...`
3. `..._N...    ->  ...N_...`
4. `..._AN..    ->  ...NA_...`

unde semnul `-->` arată că din configuraţia din stânga se trece în cea din dreapta, iar `..` semnifică orice configuraţie de piese albe şi negre inclusiv cea vidă. Se va afişa şirul de mutări care rezolvă problema.

43. Andrei urăşte să urce. El are o bicicletă pe care merge oriunde se poate, alegând bineînţeles drumurile cele mai scurte şi uşoare. Partea bună (pentru el): locuieşte într-un oraş unde toate străzile formează o reţea strict pătratică, fiind orientate sau nord-sud (numite bulevarde) sau est-vest (numite alei). Deci, distanta între orice două intersecţii consecutive este aceeaşi. Partea rea: oraşul este de munte, cu multe străzi în pantă şi cu sens unic. Pentru a ajunge într-un anumit loc, Andrei alege totdeauna traseul pe baza a trei reguli:

1. Evită orice strada care urcă cu mai mult de 10m între două intersecţii consecutive.
2. Nu foloseşte niciodată sensul interzis.
3. Foloseşte cel mai scurt drum posibil.

Ajutaţi-l pe Andrei să foloseasca un drum acceptabil.

Intrare:

Fişierul de intrare conţine datele în următoarea formă:

Pe prima linie, două numere întregi (`n,m`) separate prin cel putin un spaţiu; `n` reprezintă numărul de alei, iar `m`, cel de bulevarde (`1≤n,m≤220`).

Pe următoarele `n` linii se află altitudinile punctelor de intersecţie. Fiecare linie reprezintă o alee şi conţine o secvenţă de `m` numere întregi separate prin cel puţin un spaţiu; ele reprezintă altitudinea în metri a punctelor de intersecţie de pe aleea respectivă.

Urmează una sau mai multe linii care definesc drumurile cu sens unic. Fiecare astfel de drum este reprezentat prin două perechi de numere întregi separate prin cel puţin un spaţiu, sub forma: `bulevard alee bulevard alee`

Drumul cu sens unic porneşte din punctul de intersecţie al primei perechi şi se încheie în punctul unde se intersectează a doua pereche. Dacă cele două puncte nu sunt adiacente, drumul cu sens unic va cuprinde şi alte intersecţii. De exemplu `5 7 5 10` reprezintă drumurile `5-7` spre `5-8`, `5-8` spre `5-9`, şi `5-9` spre `5-10`. Definiţiile drumurilor se termină cu o linie care conţine patru zerouri în formatul anterior.

În final vor urma una sau mai multe linii care conţin perechi de puncte (în aceeaşi reprezentare) între care Andrei vrea să găsească un drum optim. Sfârşitul fişierului de intrare este dat de patru zerouri separate prin cel puţin un spaţiu.

Se presupune că toate bulevardele şi toate aleile sunt în domeniile definite de prima linie a fişierului de intrare şi că toate drumurile sunt construite sau pe direcţia nord-sud, sau est-vest.

Ieşirea:

Pentru fiecare drum solicitat de fişierul de intrare, ieşirea va lista o secvenţă de puncte de la poziţia de pornire la cea finală, formând ruta pe care o poate urma Andrei, conform condiţiilor sale. Două puncte consecutive de forma `bulevard-alee` sunt separate prin cuvântul '`spre`'. Dacă există mai multe drumuri care verifică criteriile lui Andrei, se va lista unul din ele. Daca nu este nici o soluţie sau dacă punctul de început şi cel final coincid, ieşirea va fi un mesaj adecvat.

Două seturi consecutive de ieşiri sunt separate prin câte o linie albă.

Exemplu: Pentru intrarea

```
3 4
```

```
10 15 20 25
19 30 35 30
10 19 26 20
1 1 1 4
2 1 2 4
3 4 3 3
3 3 1 3
1 4 3 4
2 4 2 1
1 1 2 1
0 0 0 0
1 1 2 2
2 3 2 3
2 2 1 1
0 0 0 0
```

o ieşire posibilă este:

```
1-1 spre 1-2 spre 1-3 spre 1-4 spre 2-4 spre 2-3 spre 2-2
Pentru a merge de la 2-3 la 2-3 stai pe loc!
Nu exista drum acceptabil de la 2-2 la 1-1.
```

44. Timp şi mobilitate. Să ne imaginăm un aparat care măsoară minutele scurse prin acumularea unor bile în diverse căsuțe. Să presupunem că dispozitivul are prevăzute 3 căsuțe care măsoară un minut, 5 minute şi respectiv o oră. În decursul unui minut, un braț rotativ mişcă o bilă, o ridică şi o depozitează în una din aceste căsuțe. Dispozitivul este prevăzut pentru a măsura timpul între `1:00` şi `12:59` (fără a indica `a.m.` sau `p.m.`).

De exemplu, două bile în indicatorul minut, 6 bile în indicatorul 5-minute şi 5 bile în indicatorul ora, vor reprezenta timpul `5:32`.

Din păcate acest gen de ceas nu poate indica data, deşi acest lucru se poate deduce. În deplasarea lor, bilele îşi schimbă poziția relativă într-un mod previzibil, ceea ce poate da informații despre timpul scurs între două poziții. Mai mult, începând cu un moment, situațiile încep să se repete.

Se cere să se scrie un program care să determine timpul scurs până la prima repetare a poziției, în funcție de numărul total de bile care se folosesc.

Operațiile pe care le execută ceasul cu bile:

- La fiecare minut, bila aflată într-o stivă este ridicată şi depozitată în căsuța care indică un minut şi care este capabilă să conțină până la patru bile.
- Când aici vine a cincea bilă, greutatea lor face ca fundul cutiei să se desfacă şi cele patru bilele cad înapoi în stivă; bila care a creat această schimbare se deplaseaza însă mai departe până la cutia care indică 5-minute.
- Această a doua cutie poate conține 11 bile; o a 12-a bilă cauzează răsturnarea înapoi în stivă a celor 11 bile şi rostogolirea celei de-a 12-a în cutia care marchează o oră. Çi această a treia cutie poate primi tot 11 bile, dar conține de la început o bilă, astfel încât ora indicată se numără de la 1 la 12.
- O a 12-a bilă intrată în cutia de 5-minute, după ce provoacă golirea acestei cutii, se rostogoloeşte în cutia corespunzatoare orei şi - fiind şi aici depăşită capacitatea, cutia se rastoarnă, cele 12 bile revin în stivă şi în cutie rămâne ultima bilă.

Intrare:

Fişierul de intrare defineşte o succesiune de ceasuri cu bile, fiecare ceas lucrând ca mai sus. Ceasurile diferă numai prin numărul de bile pe care le are în stivă la ora `1:00`, când pornesc toate ceasurile. Acest număr este dat pentru fiecare ceas, câte unul pe fiecare linie şi nu include bila aflată de la început în cutia a treia (pentru ore). Numerele valide sunt în intervalul `[27,127]`.

Sfârşitul fişierului de date este semnalat prin cifra `0` pe o linie.

Ieşirea:

Pentru fiecare ceas, programul trebuie să repetă la ieşire numărul de bile (dat la intrare) urmat de numărul de zile (perioade de 24 ore) scurse până când ceasul ajunge la aceeaşi configurație de la început.

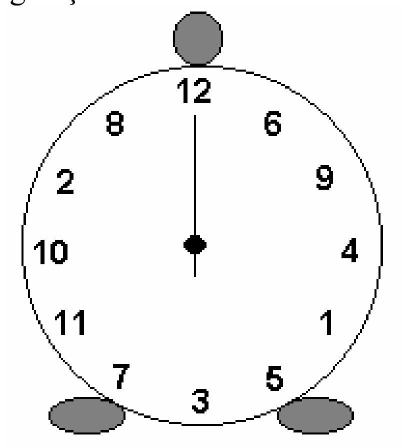
Exemplu: Pentru intrarea

```
30
45
0
```

ieşirea va fi:

```
30 bile cicleaza dupa 15 zile.
45 bile cicleaza dupa 378 zile.
```

45. În vitrina unui anticariat se găseşte următorul ceas:



Despre el se ştiu următoarele informații:

1. Funcționează bine şi arată ora exactă, deşi nu are minutar.
2. O parte din cifre - nu se ştie câte - au fost schimbate.
3. Dacă s-ar şti câte cifre au rămas la locul lor, s-ar putea deduce uşor ora.
4. Este posibil ca discul cadranului să fie rotit spre stânga sau dreapta, deci ceasul să nu fie în poziția lui firească.

Ce oră este?

46. Timbre. Filateliştii colecționează timbre cu mult timp înainte ca oficiile poştale să reglementeze utilizarea lor. Un exces de timbre poate crea dificultăți serviciilor poştale, dar poate bucura pe colecționari. Orice serviciu poştal militează pentru aplicarea pe plic a unui număr cât mai mic de timbre. Pentru aceasta vi se cere să scrieți un program care să ajute serviciul poştal. Mărimea plicului restrictionează numărul de timbre care poate fi lipit pe plic. De exemplu, dacă există numai timbre de 1 leu şi 3 lei şi pe un plic se pot lipi maxim 5 timbre, se pot acoperi astfel toate cheltuielile poştale între 1 şi 13 lei;

Deşi cinci timbre de 3 lei puse pe plic ar aduce poştei 15 lei, nu este posibil să se pună pe plic timbre în valoare de 14 lei. Deoarece serviciul poştal doreşte un interval de costuri poştale fară "găuri", el va considera în acest caz doar un cost poştal maxim de 13 lei.

Intrare:

Prima linie a fiecărui set de date conține un întreg `S` reprezentând numărul maxim de timbre ce pot fi lipite pe un plic. A doua linie conține un numar `N` care arată câte serii de valori de timbre sunt în setul de date. Fiecare din următoarele `N` linii conține câte o serie de valori de timbre.

Primul numar de pe linie dă numărul de valori al seriei; el este urmat de lista valorilor, ordonată crescător, ca în exemplu. Fiecare serie are cel mult `S` valori. Valoarea maximă a lui `S` este `10`, cea mai mare valoare a unui timbru este `100` iar valoarea maximă a lui `N` este `10`. Setul de intrare se termină cu un set de date care începe cu `0` (`S` este `0`).

Ieşirea:

Se scoate câte o linie pentru fiecare set de date, care dă acoperirea maximă fără găuri, urmată de seria de timbre care dă această acoperire. Formatul de scriere este:

```
acoperire maxima = <valoare>: <valorile seriei>
```

Dacă un set de date conține mai multe seturi de valori de timbre care dau aceeaşi acoperire maximă, se va tipări setul cu cel mai mic număr de valori. Dacă şi aici avem egalitate, se selectează setul cu cea mai joasă valoare maximă. De exemplu, dacă pe plic se pot lipi maxim 5 timbre, atunci seriile 1,4,12,21 şi 1,5,12,28 conduc la aceeaşi acoperire maximă de 71 lei. Deoarece ambele serii sunt formate din acelaşi număr de timbre (4), al doilea criteriu duce la alegerea seriei 1,4,12,28. Dacă şi după acest criteriu rămân mai multe soluții posibile, se alege una oarecare.

Exemplu:

Intrare:

```
5
2
4 1 4 12 21
4 1 5 12 28
10
2
5 1 7 16 31 88
5 1 15 52 67 99
6
2
3 1 5 8
4 1 5 7 8
0
```

Ieşire:

```
acoperire maxima =  71 :  1  4 12 21
acoperire maxima = 409 :  1  7 16 31 88
acoperire maxima =  48 :  1  5  7  8
```

47. Trenuri. Societatea de transport urban a planificat un sistem de transport între zona centrală a oraşului şi suburbii. O parte a acestui proiect constă în planificarea trenurilor pe diverse rute între cele mai depărtate stații şi zona comună de oprire a metroului. O bună planificare conține şi o fază de simulare a circulației ternurilor. O astfel de simulare constă dintr-o serie de scenarii în care două trenuri, unul plecând din stația centrală de metrou, iar celălalt din cea mai departată stație din suburbii merg unul spre altul. Scopul este de a afla unde şi când se întâlnesc cele două trenuri. Pentru aceasta se cere să scrieți un program. Modelul oricărui sistem este construit într-o variantă simplificată. Toate scenariile se vor baza pe următoarele ipoteze:

1. Timpul de oprire în stații este acelaşi.
2. Timpii de accelerare şi de frânare sunt aceiaşi, ca şi viteza de rulare.
3. Când un tren pleacă din stație, el accelerează (cu o rată constantă) până ajunge la viteza maximă. Rămâne la această viteză până când începe să frâneze (cu aceeaşi rată constantă) la apropierea stației următoare. Viteza cu care pleacă un tren din stație şi cea cu care ajunge la următoarea stație sunt zero (`0.0`). Stațiile consecutive de pe un traseu sunt suficient de distanțate pentru a permite unui tren să accelereze până la viteza maximă şi apoi sa frâneze.
4. Ambele trenuri din fiecare scenariu pleacă în acelaşi moment din cele două stații.
5. Fiecare traseu are cel mult 30 stații.

Intrare:

Toate valorile de intrare sunt numere reale. Datele pentru fiecare scenariu sunt în formatul următor:

d_1 d_2 ... d_n 0.0

Pentru un traseu, lista distanțelor (în km) de la fiecare stație la stația centrală de metrou. Stațiile sunt listate în ordinea crescătoare a distanțelor, începând cu cea mai apropiată (stația 1). Toate distanțele sunt strict pozitive. Lista se termină cu valoarea `0.0`

`v` Viteza maximă a trenului, în m/minut.

`s` Accelerația constantă a trenului m/minut^2.

`m` Numărul de minute cât stă un tren în stație.

Datele de intrare se termină cu un set de date care începe cu `-1.0`

Ieşirea:

Pentru fiecare scenariu, ieşirea constă din următoarele date:

1. Numărul scenariului (numărarea este consecutivă începând cu scenariul #1)
2. Timpul scurs (în minute) până când cele două trenuri se întâlnesc. Timpii se dau cu o cifră zecimală. În plus, dacă trenurile se întâlnesc într-o stație, se cere numărul stației unde se întâlnesc.
3. Distanța în km între stația centrală de metrou şi locul unde se întâlnesc cele două trenuri. Distanțele se exprimă cu trei cifre zecimale.

Exemplu:

Date de intrare:

```
15.0 0.0
5280.0
10560.0
5.0
3.5 7.0 0.0
5280.0
10560.0
2.0
3.4 7.0 0.0
5280.0
10560.0
2.0
-1.0
```

Răspuns:

```
Scenariul #1:
   Timpul de intalnire: 7.8 minute
   Distanta: 7.500 Km de la statia centrala de metrou
Scenariul #2:
   Timpul de intalnire: 4.0 minute
   Distanta: 3.500 Km de la statia centrala de metrou, in statia 1
Scenariul #3:
   Timpul de intalnire: 4.1 minutes
   Distanta: 3.400 Km de la statia centrala de metrou, in statia 1
```

48. Cea mai lungă subsecvență comună. O subsecvență a unui şir $X_1, X_2, .., X_n$ este un şir care se obține ştergând zero sau mai multe elemente din şirul inițial. Elementele care se şterg nu trebuie să fie neapărat pe poziții consecutive în şir. De exemplu: `2, 3, 2, 1` este o subsecvență a şirului `2, 4, 3, 1, 2, 1` ea obținându-se prin ştergerea lui `4` şi a primei apariții a lui `1` din şirul inițial. Dându-se două şiruri $X_1, X_2, ..., X_n$ şi $Y_1, Y_2, ..., Y_m$ o subsecvență comună a celor două şiruri este un şir care este subsecvență şi pentru primul şir şi pentru al doilea. Problema constă în a găsi o subsecvență de lungime maximă a două şiruri date.

49. Dezarhivarea. O schemă simplă de comprimare a unui fişier text poate fi utilizată pentru fişierele care nu conțin cifre. Schema de comprimare necesită crearea unui liste de cuvinte din fişierul nearhivat. Când este întâlnit un caracter nealfabetic în fişierul care trebuie arhivat, este copiat direct în fişierul comprimat. Un cuvânt este copiat la fel doar dacă este vorba de prima apariție a lui. În acest caz este pus la începutul listei de cuvinte. Dacă nu e prima apariție, atunci în fişierul arhivat este copiată poziția lui din listă, iar cuvântul este mutat la începutul listei. Numerotarea pozițiilor în listă începe de la `1`.

Scrieți un program care reconstituie un fişier arhivat prin metoda precedentă. Deci, având ca intrare un fişier comprimat, are la ieşire fişierul original. Se poate presupune că un cuvânt nu are mai mult de 50 de caractere şi fişierul original nu conține cifre. Se consideră ca fiind cuvânt o secvență maximală de litere mari sau mici. Se face deosebire între literele mari şi cele mici.

De exemplu:

`x-ray` conține 2 cuvinte: `x` şi `ray`

`Mary's` conține 2 cuvinte: `Mary` şi `s`

Nu se dă o limită superioară a numărului de cuvinte distincte din fişierul de intrare. Sfârşitul fişierului de intrare este marcat de o linie ce conține numai caracterul '0'.

Exemplu:

Intrare:

```
Dear Sally,
   Please, please do it--1 would 4
Mary very, 1 much. And 4 6
8 everything in 5's power to make
14 pay off for you.
    --Thank 2 18 18--
0
```

Ieşire:

```
Dear Sally,
   Please, please do it--it would please
Mary very, very much. And Mary would
do everything in Mary's power to make
it pay off for you.
    --Thank you very much--
```

50. Considerăm un depozit care are `n` camere, care conțin cantitățile de marfă c_1, c_2, ..., c_n, care sunt numere naturale distincte. Să se scrie un program care să determine un grup de camere cu proprietatea că suma cantităților de marfă pe care le conțin se poate împărți exact la cele `n` camioane pe care o transportă.

51. În curtea liceului s-au adunat `m×n` (`0<m,n<51`) fete şi băieți aliniați pe `m` linii şi `n` coloane. Directorul sosit la întâlnirea cu elevii solicită profesorului de sport să rămână pe loc acei băieți situați într-un dreptunghi de arie maximă, care nu conține nici o fată. Profesorul de sport cere ajutorul unui informatician care să precizeze colțurile stânga sus şi dreapta jos ale unui astfel de dreptunghi, precum şi numărul total de băieți situați în el. Datele de intrare se citesc dintr-un fişier text sub forma:

```
m n
a[1,1] a[1,2]. . . a[1,n]
. . .
a[m,1] a[m,2]. . . a[m,n]
```

unde `a[i,j]` este `1` pentru băiat şi `0` pentru fată.

Rezultatul va fi afişat pe ecran sub forma:

```
numar maxim baieti =3
coltul stanga sus: (... , ...)
coltul dreapta jos:(... , ...)=20
```

sau mesajul

```
"Problema nu are solutie"
```

Exemplu:

Pentru fişierul de intrare:

```
3 4
1 0 1 1
0 1 1 1
1 1 1 1
```

o soluție posibilă este:

```
numar maxim baieti=6
coltul stanga sus: (2, 2)
coltul dreapta jos:(4, 4)
```

52. Bancherii. Un număr de `n` (`0<n<101`) bancheri, fiecare având o anumită sumă de bani, doresc să formeze o asociație din `k` (`0<k≤n`) membri astfel încât suma totală de bani a acestora să fie exact `s`. Fiecare membru al asociației participă cu toată suma. Se cere, dacă este posibil, să se afişeze numerele de ordine ale membrilor dintr-o astfel de asociație şi sumele cu care participă fiecare. Datele de intrare se citesc dintr-un fişier text sub forma:

```
n  s k
a[1] a[2] ... a[n]
```

unde `a[i]` reprezintă suma bancherului cu numărul de ordine `i`.

Rezultatul va fi afişat pe ecran sub forma:

```
bancherii si sumele sunt:
```

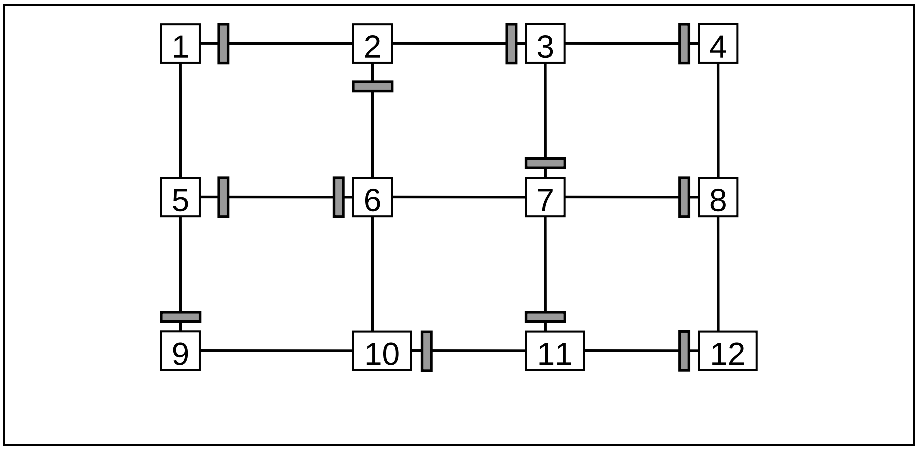
```
        i  a[i]
        ..
```
(pe k linii)
sau mesajul:
"Problema nu are solutie"

De exemplu, pentru fişierul de intrare:
```
5  10  3
4  3  4  2  5
```
o soluţie posibilă este:
```
bancherii si sumele sunt:
2  3
4  2
5  5
```
53. Se dă un şir de n (0<n<501) numere naturale. Spunem că x este mai ghiduş decât y dacă reprezentarea binara a lui x conţine mai puţine cifre 1 decât în reprezentarea binară a lui y. Să se formeze un nou şir cu un număr maxim de elemente din şirul dat, fără a modifica ordinea iniţială, astfel încât orice element al noului şir este mai ghiduş decât următorul.

Datele de intrare se citesc dintr-un fişier text sub forma:
```
n a[1] a[2] . . . a[n]
```
Çirul obţinut va fi afişat pe ecran pe orizontală.

De exemplu, pentru fişierul de intrare:
```
10 15 2 64 12 12 8 7 2 15 62
```
o soluţie posibilă este: `64   12   7   15   62`

54. Bariere. Un şoricel se află situat într-un nod al unei reţele dreptughiulare de dimensiune m×n, având forma şi numerotarea nodurilor conform figurii (în care m=4, n=3):



Fiecare nod v al reţelei are exact o barieră pe o muchie vw, care blochează trecerea şoricelului de la v la w, dar şi de la w la v. (Pentru exemplul din figura anterioară, în nodul 2 avem o barieră către nodul 6, care împiedică trecerea şoricelului de la nodul 2 la nodul 6, dar şi de la nodul 6 la nodul 2.). Çoricelul trebuie să ajungă la o bucăţică de caşcaval, situată într-un alt nod al reţelei, parcurgând reţeaua pe drumul de cost minim, respectând următoarele reguli:

a) Çoricelul, aflat în nodul v, poate trece la nodul w, dacă nu există nici o barieră pe muchia vw; această trecere îl costă 1$.

b) Çoricelul poate schimba poziţia barierei din nodul curent, ceea ce îl costă tot 1$.

Pentru exemplul din figura anterioară, şoricelul (presupus a fi iniţial în nodul 2) poate ajunge în nodul 7, în mai multe moduri, de exemplu: a) mută bariera din 2 (aşezând-o către nodul 1), se deplasează apoi în nodul 6, apoi în 7 (costul: 3$); b) mută bariera din 2 (aşezând-o către nodul 1), se deplasează apoi în nodul 6, apoi în nodul 10, unde pune bariera către nodul 9, apoi se duce în 11, pune bariera de aici către nodul 10 şi, în sfârşit, se deplasează în nodul 7 (costul: 7$). Se cere să se determine un astfel de drum de cost minim al şoricelului către caşscaval.

55. Structura liniilor telefonice instalate în şanţurile ROMTELECOM dispune de n (n≤100) noduri. Se cunosc lungimile cablurilor dintre diverse noduri. Să se determine o reţea de lungime totală minimă care să permită comunicarea între oricare două noduri (direct sau indirect), ştiind că între două noduri date, `a` şi `b`, trebuie să existe minim `m` metri de cablu (direct sau indirect).

Datele de intrare se citesc dintr-un fişier text sub forma:

```
a b  m
n
i j k
..
```

unde `i`, `j` sunt nodurile, iar `k` lungimea cablului direct dintre ele.

Rezultatul va fi într-un fişier text sub forma:

```
k i j ..
```

unde `k` lungimea totală a cablului, iar `i, j` sunt perechile de noduri alese. sau mesajul `"Problema nu are solutie"`

De exemplu, pentru fişierul de intrare:

```
1  4  3
5
1 3 1
1 5 4
2 4 2
3 4 1
3 5 2
4 5 3
```

o soluție posibilă este:

```
8  1 3  2 4  3 5  4 5
```

56. Se consideră un graf neorientat. Să se verifice dacă el are sau nu un circuit de lungime a) 3; b) 4.

57. Să se verifice dacă un graf orientat aciclic conține sau nu un drum hamiltonian. Puteți găsi un algoritm liniar?

58. Să se găsească, folosind un algoritm liniar, dacă există, un circuit într-un graf conex care conține două noduri date `a` şi `b`, dar nu conține nodul `c`.

59. Scrieți un program care să determine (dacă există) un nod al unui graf conex prin dispariția căruia graful rămâne conex. Se consideră că o dată cu dispariția nodului respectiv, dispar şi arcele incidente lui.

60. Talk-show. `2n` parlamentari participă la discuții la un talk-show televizat, care durează `n` ore. Parlamentarii se aşază sub forma unui semicerc, pe mai multe fotolii, la mijloc fiind un moderator. După fiecare oră de discuții, are loc o pauză publicitare, după care parlamentarii îşi schimbă locurile între ei. Să se determine variantele de aşezare astfel încât un parlamentar să nu aibă în două ore diferite acelaşi vecin, inclusiv moderatorul.

61. Pentru o expresie aritmetică conținând paranteze, operatorii `+`, `-`, `/` şi `*` şi operanzi numerici, să se determine valoarea sa. (Se vor folosi două stive, una a operanzilor, iar alta a operatorilor).

62. Se cere să se scrie un program care să deriveze (formal) o expresie. Se va folosi faptul că orice expresie aritmetică poate fi memorată sub forma unui arbore binar. (Observație: Pentru rezolvarea acestei probleme sunt necesare cunoştințe de *Analiză matematică* ce vor fi studiate în clasa a XI-a.).

63. Se dau doi arbori binari. Se cere să se înlocuiască fiecare nod al primului cu cel de al doilea arbore.

64. Se dă un arbore oarecare, informațiile din noduri fiind şiruri de caractere. Să se construiască o listă dublu înlănțuită care să conțină toate şirurile din nodurile arborilor, care au lungimile pare, apoi să se ordoneze această listă.

65. Scrieți o funcție care verifică dacă elementele unei liste simplu înlănțuite cuprinzând date de tip `char` sunt sau nu ordonate.

66. Într-o listă circulară dublu înlănțuită să se înlocuiască fiecare apariție a unui caracter care este vocală cu cea mai apropiată consoană din cadrul listei. Aceeaşi problemă în cazul unei liste circulare simplu înlănțuite.

67. Într-o listă circulară simplu înlănţuită să se înlocuiască fiecare apariţie a unui caracter care este vocală cu cea mai apropiată consoană din alfabet. Aceeaşi problemă când lista este o coadă (necirculară).

68. Scrieţi subprograme pentru a calcula suma, diferenţa, produsul şi pentru a efectua împărţirea cu cât şi rest a două polinoame, ale căror coeficienţi (reali) sunt memoraţi în liste create dinamic.

69. Fie F_k al k-lea termen din şirul lui Fibonacci. Un *arbore Fibonacci* de ordin k are $F_{k-1}-1$ vârfuri interne (notate cu 1, 2, ..., $F_{k+1}-1$) şi F_{k+1} frunze (notate cu 0, -1, -2, ... $-(F_{k+1}+1)$) şi se construieşte după cum urmează:

pentru k=0 şi k=1 arborele este [1].;

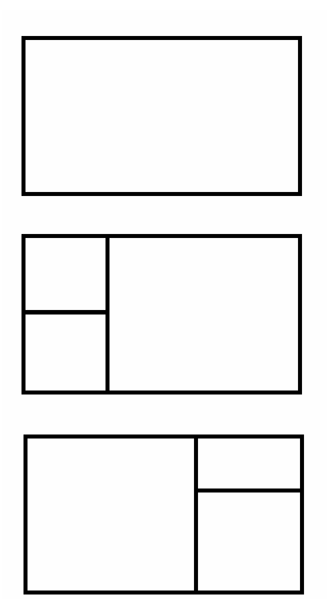
pentru $k\geq2$, rădăcina este notată cu F_k, subarborele stâng este arbore Fibonacci de ordin k-1, iar subarborele drept este arbore Fibonacci de ordin k-2, în care valorile vârfurilor sunt mărite cu F_k.

a) Să se scrie o funcţie (recursivă) pentru a construi un arbore Fibonacci de ordin n.

b) Să se parcurgă arborele creat în ordine, listând doar informaţia din nodurile interne.

70. Se dau trei liste alocate dinamic, fiecare cuprinzând cuvinte ordonate alfabetic. Se cere să se realizeze lista tuturor cuvintelor în ordine alfabetică.

71. Descompunere. Se consideră un dreptunghi de dimensiuni $a\times b$, cu a, b numere întregi pozitive, ce satisfac: $b-a<a<b$. (*). Există mai multe moduri de a descompune un asemenea dreptunghi în două pătrate şi un dreptunghi. În figură sunt date două exemple de descompunere ale aceluiaşi pătrat. Presupunem că s-a realizat o asemenea descompunere. Procesul de descompunere se aplică apoi dreptunghiului rezultat în urma descompunerii anterioare şi continuă în aceeaşi manieră până când se obţine un dreptunghi ce nu mai satisface relaţia (*).



Problema constă în determinarea unui şir de descompuneri în urma cărora să rezulte un număr total minim de figuri componente. Să se scrie un program care citeşte dimensiuni de dreptunghiuri şi afişează lanţurile de descompuneri corespunzătoare sub forma unei secvenţe de numere întregi: p_1, p_2,..., p_k, d_1, d_2, unde p_1, p_2, ..., p_k sunt lungimile pătratelor în ordinea obţinerii acestora, iar d_1 şi d_2 sunt dimensiunile dreptunghiului din ultima descompunere.

72. Decupare. Se dă o suprafaţă dreptunghiulară conţinând pătrăţele elementare albe şi negre şi se cere să se decupeze din ea o subsuprafaţă dreptunghiulară în care diferenţa dintre numărul pătrăţelelor albe şi al celor negre să fie maximă, în valoare absolută. Dacă există mai multe astfel de subsuprafeţe, se cere să se afişeze una de arie minimă.

73. Coodul Booth. Codul Booth este o reprezentare a numerelor în baza este 3, dar cifrele sunt 0, 1 şi, în loc de 2, apare -1. Vom reprezenta cifra -1 prin "!". Cifra 1 intră în calcul cu valoare 1, cifra 0 cu valoarea 0, iar cifra "!" cu valoarea -1.

Exemple:

- numărul 9 se reprezintă în cod Booth ca 100 = $1\times3^2+0\times3^1+0\times3^0$;
- numărul 8 se reprezintă în cod Booth ca 10! = $1\times3^2+0\times3^1+(-1)\times3^0$;
- numărul 10 se reprezintă în cod Booth ca 1!!! = $1\times3^3+(-1)\times3^2+(-1)\times3^1+(-1)\times3^0$.

Aşadar, numerele naturale se scriu ca sume algebrice de puteri ale lui 3. Çtim câ pentru orice numâr natural reprezentarea sa Booth este unicâ.

a) Sâ se scrie o funcţie care, primind reprezentarea Booth a unui număr natural, oferă la ieşire reprezentarea acestui număr în baza 10.

b) Sâ se scrie o funcţie care, primind un numâr în baza 10, oferâ la ieşire reprezentarea Booth a acestuia.

c) Sâ se scrie funcţii pentru adunarea, respectiv înmulţirea, a douâ numere reprezentate în cod Booth, fârâ a face trecerea în altâ bazâ.

74. Scatii. În n copaci dispuşi circular sunt n scatii, câte unul în fiecare copac. La momentele de timp i = 1,2,3,... are loc câte o acţiune descrisă de următoarea schemă: doi scatii zboară pe copacii alăturaţi celor de pe care pleacă, dar în sensuri opuse (unul în sensul acelor de ceasornic, altul în sens invers acelor de ceasornic, ca în figura 1). Se ştie că pentru n impar există secvenţe finite de acţiuni care adună scatii într-un singur copac. O asemenea secvenţă, pentru n=3 este reprezentată grafic în figura 2.

Să se scrie un program care pentru un număr natural impar n dat construieşte o secvenţă de acţiuni care adună toţi scatii într-un singur copac. Se presupune că numerotarea copacilor se face în sensul invers acelor de ceasornic. Se va afişa la fiecare pas ce scatii pleacă, de unde pleacă şi unde ajung.

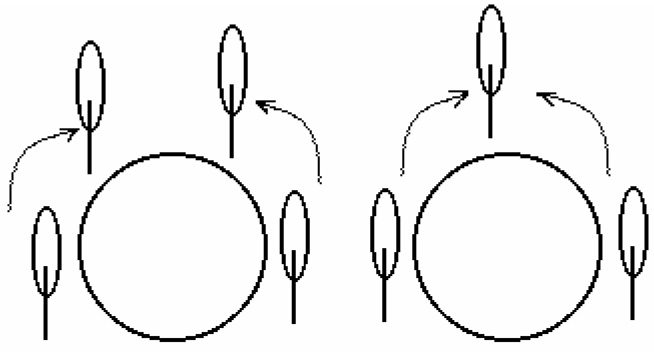


figura 1 figura 2

75. Se citeşte dintr-un fişier text un număr întreg n şi apoi n numere întregi a[1], a[2], ..., a[n-1], a[n].

Se cere să se afişeze o expresie aritmetică astfel încât:

- să aibă valoarea numarului a[n];
- operaţiile folosite de expresie sunt +, -, *, /; diviziunea poate fi folosită numai dacă rezultatul este un întreg;
- pot fi folosite paranteze, fără restricţii;
- operanzii aleşi sunt din numerele a[1], ..., a[n-1], fiecare putând apare de cel mult odată.

Exemple:

a) Intrare: n = 5, a = 1 2 25 75 103
 Ieşire: 1+(2+(25+75))=103

b) Intrare: n = 7, a = 10 10 10 10 25 75 875
 Ieşire: (10*(10-(10-(10+75))))+25=875

c) Intrare: n = 4, a = 6 25 75 101
 Ieşire: Imposibil !

BIBLIOGRAFIE

1. Adrian Atanasiu, Rodica Pintea - *Culegere de probleme Pascal*, Editura Petrion, Bucureşti, 1996.

2. Bogdan Pătruț - *Algoritmi şi limbaje de programare (manual de informatică pentru clasa a IX-a)*, Editura Teora, Bucureşti, 1998.

3. Bogdan Pătruț - *Aplicații în C şi C++*, Editura Teora, Bucureşti, 1998.

4. Bogdan Pătruț - *Învățați limbajul Pascal în 12 lecții*, Editura Teora, Bucureşti, 1997

5. Doina Rancea - *Limbajul Turbo Pascal*, Editura Libris, Cluj-Napoca, 1994.

6. Dorel Lucanu - *Proiectarea algoritmilor. Tehnici elementare*, Editura Universității "Al. I. Cuza", Iaşi, 1993.

7. Emanuela Mateescu, Ioan Maxim - *Arbori*, Editura æara Fagilor, Suceava, 1996.

8. Leon Livovschi, Horia Georgescu - *Sinteza şi analiza algoritmilor*, Editura Ştiiințifică şi Enciclopedică, Bucureşti, 1986

9. Octavian Aspru - *Tehnici de programare*, Editura Adias, Rm. Vâlcea, 1997

10. Tudor Bălănescu - *Corectitudinea algoritmilor*, Editura Tehnică, Bucureşti, 1995.

11. Tudor Sorin - *Tehnici de programare*, Editura Teora, Bucureşti, 1994.

12. Valeriu Iorga, Eugenia Kalisz, Cristian æăpuş - *Concursuri de programare. Probleme şi soluții*, Editura Teora, Bucureşti, 1997.

13. Victor Mitrana - *Provocarea algoritmilor*, Editura Agni, Bucureşti, 1994.