TUDOR SORIN

MANUAL DE
INFORMATICĂ

Clasa a IX-a, Profilul real-intensiv

Filiera teoretică, profil real,
Specializările: Matematică-Informatică, Intensiv Informatică

Filiera vocațională, profil militar,
Specializările: Matematică-Informatică, Intensiv Informatică

Clasa a X-a, Profilul real

Filiera teoretică, profil real,
Specializările: Matematică-Informatică, Științe ale naturii

Filiera vocațională, profil militar,
Specializarea: Matematică-Informatică

VARIANTA C++

Respectă noua programă valabilă începând cu anul 2009.
Conținutul este aprobat MEC prin Ordinul nr. 4188.

L&S Soft

CUPRINS

CAPITOLUL 1

Algoritmi

1.1. Noțiuni generale

Noțiunea de algoritm este primară, nu se definește, întocmai ca și noțiunea de mulțime în matematică. Cu toate acestea, ca și mulțimea, algoritmul poate fi descris. *În esență, este vorba de o succesiune de etape care se pot aplica mecanic în vederea obținerii unui anumit rezultat.*

În activitatea cotidiană întâlnim la tot pasul algoritmi. Vom da câteva exemple.

1. *Algoritmul prin care o persoană dă un telefon.* Persoana ridică receptorul, așteaptă tonul. Dacă nu vine tonul, închide telefonul, apoi îl redeschide. După apariția tonului formează numărul, etc. Nu îmi propun să descriu amănunțit algoritmul prin care se dă un telefon pentru că este mult prea cunoscut, dar atrag atenția asupra faptului că se poate descrie foarte bine, astfel încât persoana poate executa mecanic operațiile necesare.

2. *Algoritmul prin care o persoană poate găti un anumit fel de mâncare.* Persoana are la dispoziție făină, zahăr, ouă etc. pe care le combină în anumite proporții, după care amestecul obținut se fierbe (se prăjește sau se pune în cuptor) un anumit interval de timp. Produsului obținut i se mai poate adăuga câte ceva, după care este din nou fiert (prăjit) un interval de timp, după care se obține produsul finit, adică mâncarea dorită.

3. *Algoritmul prin care se adună două fracții.* Cele două fracții se aduc la același numitor, se fac înmulțirile, adunările, apoi fracția este simplificată.

Oricare din algoritmii de mai sus poate fi descris în termeni precisi, astfel încât cel care-l execută poate efectua operațiile fără să fie nevoit să gândească ce are de făcut la un anumit moment.

O analiză sumară a exemplelor ne conduce la următoarele observații:

➢ *În orice algoritm se pornește de la ceva și se dorește obținerea unui anumit rezultat.*

- Dăm un telefon pentru ca un anumit mesaj să ajungă la destinație. Se pornește de la un mesaj și se dorește ca acesta să ajungă la o anumită persoană.
- Dacă dorim să gătim un anumit fel de mâncare, pornim de la anumite produse și obținem mâncarea solicitată.
- Dacă adunăm două fracții, pornim de la cele două fracții și obținem suma lor.

- *În orice algoritm se operează cu anumite "obiecte" asupra cărora sunt permise anumite operații.*
  - Algoritmul prin care dăm un telefon operează cu telefonul. Operațiile permise sunt deschiderea, închiderea telefonului, formarea numărului etc.
  - Algoritmul prin care se obține un anumit fel de mâncare operează cu farfurii, tăvi, aragaz, alimente, etc. Operațiile permise sunt amestecarea, fierberea, prăjirea alimentelor.
  - Algoritmul prin care se adună două fracții operează cu valori numerice. Operațiile sunt cele descrise de regulile matematice.
- *În cele mai multe cazuri cel care elaborează algoritmul este diferit de executant.*
  - Pentru a putea fi aplicat algoritmul prin care se dă un telefon a fost necesară inventarea telefonului, gândirea modului în care cineva dă ușor un telefon. O mulțime de personalități au gândit toate acestea, dar algoritmul poate fi aplicat de orice persoană care știe cifrele între 0 și 9.
  - Pentru a putea fi aplicat algoritmul prin care se obține un anumit fel de mâncare au fost făcute numeroase experimente care au condus până la urmă la o rețetă. Poate găti orice persoană care știe să citească și să folosească aragazul.
  - Pentru a putea aduna două fracții a fost necesar ca matematica să se dezvolte suficient de mult. Să nu uităm că oamenii au lucrat mult timp doar cu numere naturale...

Din cele de mai sus, rezultă că noțiunea de algoritm este extrem de generală, cu ea ne întâlnim tot timpul. *În această carte ne ocupăm numai de elaborarea algoritmilor pentru programarea calculatoarelor.* Aici "executantul" este calculatorul, iar cel care elaborează algoritmul poartă numele de **"programator"**. Calculatorul doar execută instrucțiuni, nu gândește, dar viteza de executare a instrucțiunilor este foarte mare, imposibil de atins de om. Frecvent, mai intervine o persoană, de cele mai multe ori diferită de programator, numită **"utilizator"**. Ea este cea care utilizează programul obținut și beneficiază de avantajele lui. De cele mai multe ori, o astfel de persoană are o pregătire minimă de specialitate în informatică. Din păcate, se face deseori confuzia între programator și utilizator.

1.2. Enunțul unei probleme, date de intrare și de ieșire, etapele rezolvării unei probleme

Se consideră funcția:

$$f:\Re\to\Re,\quad f(x)=\begin{cases}x^2-2, & x<0;\\ 3, & x=0;\\ x+2, & x>0.\end{cases}$$

Se cere să se calculeze `f(x)` pentru `10` valori reale ale lui `x`. De exemplu: `-3.1, 7, 0, 2.23`, ș.a.m.d.

Considerăm prima valoare a lui `x`, și anume `-3.1`. Observăm că această valoare este mai mică decât `0`. Calculăm `(-3.1)^2-2`. A doua valoare pentru care trebuie calculată funcția este `7`. Pentru că ea este mai mare decât `0`, calculăm `x+2`, adică `7+2`. A treia valoare este `0`. Funcția ia valoarea `3`. Repetăm calculul pentru cele `7` valori rămase. Această modalitate de calcul este plicticoasă și cere mult timp. Pentru a rezolva astfel de probleme - și nu numai de tipul acesteia - a fost inventat calculatorul. Acesta va efectua calculele în locul nostru. Observați faptul că *pentru efectuarea calculelor, calculatorul trebuie să ia anumite decizii automat*. Astfel, în funcție de valoarea lui `x` (negativă, zero, sau mai mare decât 0) acesta va decide ce expresie calculează pentru a obține `f(x)`. Dacă calculele elementare pot fi făcute de un simplu calculator de buzunar, neprogramabil, deciziile automate le poate lua numai un calculator programabil. Evident, deciziile se iau în urma aplicării algoritmului.

Calculatorul rezolvă o problemă atunci când execută un anumit program, corespunzător problemei. În esență, programul este alcătuit din comenzi (instrucțiuni) pe care calculatorul le execută. *Programul se obține prin codificarea algoritmilor într-un limbaj de programare.*

În continuare, prezentăm în linii mari etapele obținerii unui program.

1. Identificarea datelor de intrare și a celor de ieșire

Am văzut faptul că în orice algoritm se pornește de la ceva și se urmărește un anumit rezultat. Cu alte cuvinte, trebuie să ne fie clar de la ce plecăm și ce vrem să obținem. Aceasta înseamnă că trebuie să cunoaștem **datele de intrare** - de la ele plecăm - și **datele de ieșire** - pe ele trebuie să le obținem. Pentru exemplul considerat, datele de intrare sunt `10` valori reale x_1, x_2, ..., x_{10}. Datele de ieșire sunt cele `10` valori calculate $f(x_1)$, $f(x_2)$, ..., $f(x_{10})$.

Observație. Algoritmii operează cu date de intrare și de ieșire, chiar și atunci când acest fapt nu este atât de evident. Să presupunem că jucăm un joc pe calculator. Aceasta are loc sub controlul unui anumit program, care a fost obținut în urma codificării unui algoritm. O dată de intrare poate fi apăsarea unei taste, un clic al *mouse*-ului, etc. O dată de ieșire poate fi o anumită imagine sau o succesiune de imagini care creează impresia că un obiect se deplasează, un anumit sunet (și acestea sunt codificate numeric), etc.

2. Elaborarea algoritmului de rezolvare a problemei.

În linii mari, algoritmul specifică operatiile pe care le "are de făcut" calculatorul pentru ca, pornind de la datele de intrare, să obțină datele de ieșire. Iată, de exemplu, cum arată algoritmul simplificat de rezolvare a problemei propuse:

Se parcurg următoarele etape, de `10` ori:

- se citește valoarea lui `x`;
- în funcție de valoarea proprie a lui `x`, se procedează astfel:
  - dacă este negativă, se calculează `f=x`^{2}`-2`;
  - dacă este `0`, valoarea funcției este `3`;
  - dacă este pozitivă, se calculează `f=x+2`;
- se scrie valoarea calculată pentru `f`.

Observați faptul că algoritmul a fost descris în limbaj natural. În practică se folosesc **limbaje de tip pseudocod** sau, din ce în ce mai rar, **scheme logice.**

Ce este un limbaj de tip pseudocod? După cum știm, programele pentru calculator sunt scrise în anumite limbaje: `Pascal`, `C++`, `Fortran`, `Cobol`, etc. Pentru a scrie programele într-un limbaj de programare este necesar să respectăm anumite reguli specifice limbajului. Atunci când elaborăm algoritmul care stă la baza programului este greu să avem în vedere și regulile specifice limbajului. Trebuie să ne concentrăm asupra problemei, nu asupra unor detalii. Atunci va trebui să folosim un limbaj de tip pseudocod. *Adică un limbaj care nu are multe reguli, dar care seamănă cu orice limbaj de programare.* Un algoritm redactat în pseudocod nu poate fi rulat pe calculator, este necesară conversia sa în limbajul de programare dorit. Însă conversia este aproape mecanică, pentru că atunci nu suntem concentrați asupra algoritmului.

Atragem atenția asupra faptului că un adevărat limbaj de tip pseudocod permite *conversia cu ușurință a algoritmului în* ***orice*** *limbaj de programare*. Există multe limbaje de tip pseudocod, aproape orice persoană poate să creeze unul. *În acest capitol folosim un limbaj de tip pseudocod în limba română, limbaj utilizat în prezent în cadrul examenului de bacalaureat.*

Ca și limbajele de tip pseudocod, schemele logice permit redactarea algoritmilor. Ele au fost mult folosite în trecut, dar astăzi se utilizează din ce în ce mai rar. Motivul? Ocupă mult spațiu pe hârtie și creează neplăceri privind reprezentarea grafică. Cu toate acestea sunt considerate intuitive și mulți profesori le utilizează.

3. Transpunerea algoritmului într-un limbaj de programare (`Pascal, C++`)

În această etapă se operează aproape mecanic. Odată cunoscut un limbaj de programare, transpunerea algoritmului nu este o problemă dificilă. **Rețineți:** *nu este greu să înveți un anumit limbaj, este greu să știi să elaborezi algoritmi.*

De fapt, algoritmul poate fi codificat direct în limbajul de programare dorit. Cu timpul, când veți căpăta suficientă experiență și dacă problema nu este dificilă chiar așa veți proceda.

4. Testarea programului și corectarea sa până când "funcționează" corect.

Atunci când programul este complex, este aproape imposibil să fie scris corect de la început. Din acest motiv este necesară faza de mai sus. Nu uitați, calculatorul este cel mai bun corector, hârtia suportă orice greșeală.

1.3 Noțiunea de algoritm, caracteristici

Noțiunea de algoritm a mai fost prezentată în primul paragraf al acestui capitol. Aici o reamintim, o particularizăm pentru programarea calculatoarelor și evidențiem caracteristicile sale. *Prin algoritm înțelegem o succesiune de etape care se pot aplica mecanic pentru ca, pornind de la datele de intrare, să se obțină datele de ieșire.* Rețineți: **pentru orice algoritm trebuie precizat în mod clar care sunt datele de intrare și care sunt cele de ieșire.** Dacă această precizare nu a fost făcută, nu se mai poate vorbi de algoritm. Iată câteva caracteristici ale algoritmilor.

1. Finititudine - *este proprietatea algoritmilor de a furniza rezultatele într-un timp finit.* Nu trebuie înțeles de aici că dacă un algoritm furnizează rezultatele în timp finit este neapărat bun. El trebuie să fie și eficient, adică să alegem calea cea mai simplă de rezolvare, înțelegând prin aceasta, cea care presupune un efort de calcul cât mai mic.

Exemplu: *Se cere să se elaboreze algoritmul prin care se listează primele 100 pătrate perfecte.* O primă idee ar fi să testăm care număr (începând cu `1`, continuând cu `2`, `3`, etc.) este pătrat perfect. Dacă acesta îndeplinește condiția este tipărit, altfel se trece mai departe. Algoritmul se termină atunci când am listat `100` de pătrate perfecte. Altfel: se tipăresc valorile 1^2, 2^2, ..., 100^2. E mai simplu, nu? Să vedem de ce. În primul caz se analizează primele `10000` numere naturale (pentru că $100^2=10000$) și se tipăresc cele care sunt pătrate perfecte. În al doilea caz se

efectuează doar `100` de înmulțiri. Veți spune că nu contează, calculatorul face calculele extrem de rapid. Nu-i așa! Pentru probleme complexe, un algoritm performant rulează cu mult mai repede decât unul care nu îndeplinește această condiție. *Aceasta înseamnă că, atunci când avem de elaborat un algoritm, nu ne vom opri la prima soluție găsită.* Referitor la această proprietate, se impun anumite precizări. Există probleme pentru care nu se cunosc algoritmi de rezolvare rapizi. *Pentru anumite seturi de date de intrare, este necesar un timp de calcul de ordinul sutelor sau chiar miilor de ani și aceasta pe calculatoarele ultramoderne.* Din acest motiv, în practică se consideră că pentru aceste probleme nu se cunosc algoritmi de rezolvare, chiar dacă timpul de lucru este finit.

2. **Claritatea** - *este proprietatea algoritmilor prin care procesul de calcul este descris precis, fără ambiguități.*

3. **Generalitatea** - *este proprietatea algoritmilor de a rezolva o întreagă clasă de probleme.* Să considerăm problema pusă în acest paragraf. De fapt, funcția se calculează pentru ***oricare*** `10` valori reale. Mai general ar fi fost să calculăm valorile pe care le ia funcția pentru oricare `n` valori reale, cu `n` număr natural citit. Mai mult, se poate obține un algoritm care să primească drept date de intrare atât numărul de valori, valorile propriu-zise, cât și funcția ale cărei valori trebuie calculate.

1.4. Obiectele cu care lucrează algoritmii și operații permise

În linii mari, **algoritmii lucrează cu două tipuri de obiecte: date și variabile.** Cu acestea sunt permise anumite operații. **Precizăm că operațiile care vor fi efectuate sunt exemplificate în limbaj de tip pseudocod.**

1.4.1. Date

Așa cum am văzut, orice algoritm pornește de la anumite date de intrare, le prelucrează, iar în final obține date de ieșire. În exemplul prezentat, datele de intrare sunt reprezentate de cele `10` valori ale lui `x`, iar datele de ieșire sunt cele `10` valori ale lui `f(x)`.

Datele pot fi clasificate după tipul lor:

- **întregi;**
- **reale;**
- **logice;**
- **șir de caractere.**

Datele din primele două tipuri poartă numele de **date numerice**.

Datele întregi sunt numere aparținând mulțimii numerelor întregi. **Exemple:** `100, -6700, +122`.

Datele reale sunt numere cu zecimale. La o astfel de dată, în loc de virgulă se folosește punctul. Exemplu: în loc de `3,25` se scrie `3.25`. După cum știm din matematică, mulțimea numerelor reale este o reuniune între mulțimea numerelor raționale și mulțimea numerelor iraționale. Nici un calculator din lume nu poate reține un număr irațional, întrucît acesta are o infinitate de zecimale. Din acest motiv, în informatică, printr-o **dată reală** înțelegem o valoare cu un număr finit de zecimale.

Datele logice au numai două valori: `TRUE` (adevărat) și `FALSE` (fals).

Datele de tip șir de caractere sunt reprezentate de șiruri de caractere cuprinse între apostrof. Exemple: `'un text'`, `'facem reforma'`.

Aici este momentul să vorbim despre o categorie aparte de date și anume **constantele. Constantele sunt date care nu se modifică pe parcursul aplicării algoritmului.** Acestea pot fi date de oricare dintre tipurile precizate. *Ele se caracterizează prin faptul că sunt utilizate în algoritm, fără a fi citite sau obținute din calcule.* Se folosesc, de exemplu, în calculele care se fac sau sunt mesaje care trebuie să apară la fiecare rulare a programului rezultat în urma algoritmului, etc.

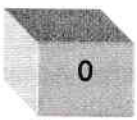
1.4.2. Variabile

În problema analizată aveam de citit `10` valori de intrare. În matematică acestea ar fi notate x_1, x_2, ..., x_{10}. În prelucrare, pentru orice dată de intrare de acest tip se fac aceleași operații. Este lipsit de sens să explicăm ce facem cu x_1 pentru ca apoi să explicăm ce facem cu x_2 etc. Din acest motiv, în algoritm se prezintă ce se face cu un anume `x`, oricare ar fi el dintre cele `10`. De asemenea, nu are rost să precizăm că se tipărește `f(`x_1`)`, apoi `f(`x_2`)` ș.a.m.d. Se notează pur și simplu că se tipărește `f`.

Până în acest moment avem o formalizare matematică comună. De aici s-a și pornit atunci când s-a ajuns la variabile. Pentru algoritmul nostru `x` și `f` sunt variabile. *Ne imaginăm variabilele ca pe niște "cutiuțe" care rețin date.* Fiecare variabilă are un nume.

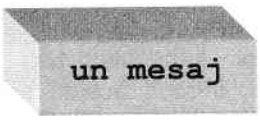
Exemple:

- Variabila `a` reține data întreagă `0`.



- Variabila `a` reține data întreagă `6`.

- Variabila **b** reține data de tip șir: **'un mesaj'**.



b

O variabilă poate reține date ***numai*** *de un tip anume.* Astfel avem variabile care pot reține date întregi, variabile care pot reține date reale, variabile care pot reține date logice și variabile care pot reține date de tip șir de caractere.

De aici rezultă o caracteristică fundamentală a variabilelor: **tipul lor**. Acesta determină natura datelor care pot fi reținute de variabila respectivă. Astfel avem:

- variabile de tip întreg - le vom nota în pseudocod cu **întreg**;
- variabile de tip real - notate **real**;
- variabile de tip logic - notate **logic**;
- variabile de tip șir - notate **șir**.

Facem următoarea convenție: pentru ca un algoritm să poată folosi o variabilă, aceasta trebuie declarată - adică anunțată. Iată cum arată declarațiile variabilelor **a** și **b**:

```
întreg a
șir b;
```

Dacă trebuie să declarăm mai multe variabile de același tip procedăm ca în exemplul următor:

```
întreg c, d
logic e, f.
```

✓ Cu toate că o variabilă are un nume unic, conținutul ei poate fi diferit de la un moment la altul, pe parcursul executării programului. De aici provine denumirea de variabilă.

Dar dacă, în mod abstract, este posibil să notăm o intrare cu **x** și o ieșire cu **f**, cum este posibil să folosim această abstractizare în momentul în care algoritmul este transpus într-un limbaj de programare? Practic, în memoria calculatorului se rezervă pentru fiecare variabilă un spațiu. Aceasta înseamnă că am rezervat un singur spațiu pentru **x** și un singur spațiu pentru **f**.

1.4.3. Expresii

În scopul efectuării calculelor, sau a luării anumitor decizii algoritmii folosesc ***expresii***. **O expresie este alcătuită din unul sau mai mulți operanzi legați între ei prin operatori.**

- **Operanzii** pot fi constante și variabile.
- **Operatorii** au rolul de a preciza operațiile care se efectuează.

În linii mari, expresiile sunt de două feluri:

A) Expresii aritmetice

B) Expresii logice.

A) **Expresii aritmetice**. Operanzii sunt constante sau variabile de tip întreg sau real. Mai des întâlniți sunt operatorii:

- pentru adunare folosim operatorul `'+'`;
- pentru scădere folosim operatorul `'-'`;
- pentru înmulțire folosim operatorul `'*'`;
- pentru împărțire folosim operatorul `'/'`.

Exemplu: dacă `a` și `b` sunt două variabile de tip întreg care reține `3`, respectiv `7`, atunci expresia `a+3*b` ia valoarea `3+3x7`, adică `24`.

✓ În pseudocod putem utiliza ca operator orice simbol cu semnificație matematică.
Exemple: $\sqrt{}$ pentru radical, $[\ ]$ pentru parte întreagă, etc.

Exemplu: dacă `y` este o valoare întreagă care reține `9`, expresia $\sqrt{y}+\left[\frac{y}{2}\right]$ ia valoarea $\sqrt{9}+\left[\frac{9}{2}\right]$, adică $3+[4.5]$, adică `7`.

B) **Expresii logice.**

Anumiți operatori, deși au ca operanzi variabile, constante sau expresii aritmetice produc ca rezultat valori logice: `true`, `false`.

Din prima categorie fac parte operatorii de comparare:

- pentru egalitate folosim operatorul '=';
- pentru inegalitate folosim operatorul '≠';
- pentru mai mare folosim operatorul '>';
- pentru mai mare sau egal folosim operatorul '≥';
- pentru mai mic folosim operatorul '<';
- pentru mai mic sau egal folosim operatorul '≤'.

Exemple: Variabila **a** reține **2**, iar variabila **b** reține **5**. Atunci:

a<b ia valoarea **true**;
a>b ia valoarea **false**;
a=b ia valoarea **false**;
a≠b ia valoarea **true.**
a≥2 ia valoarea **true;**
b≤6 ia valoarea **true.**

Alți operatori au ca operanzi variabile, constante, expresii logice și produc ca rezultat valori logice: **true**, **false**.

- operatorul **SAU** acționează asupra a doi operanzi logici și dacă *cel puțin unul din ei este* **true**, rezultatul este **true**, contrar rezultatul este **false**.
- operatorul **SI** acționează asupra a doi operanzi logici și dacă *ambii* sunt **true**, rezultatul este **true**, contrar rezultatul este false.
- operatorul **NOT** acționează asupra unui operand logic și dacă operandul este **true**, rezultatul este **false**, contrar rezulltatul este **false**.

Exemple:

false SAU true ia valoarea **true;**
false SAU false ia valoarea **false;**
false SI true ia valoarea **false;**
true SI true ia valoarea **true**;
NOT false ia valoarea **true;**
NOT true ia valoarea **false.**

Relații foarte importante:

NOT(a>b) ⇔ a≤b. Se poate gândi așa: dacă **a** nu este mai mare ca **b**, atunci **a** este mai mic sau egal cu **b** și este corect. În realitate, calculatorul lucrează cu expresii logice.

Între **a** și **b** pot exista 3 relații:

1. **a<b.** Atunci **a>b** ia valoarea **false**; **NOT(false)=true**. Pe de altă parte, în aceleași condiții, **a≤b** ia valoarea **true.** Avem egalitate.

2. **a>b.** Atunci **a>b** ia valoarea **true**; **NOT(true)=false**. Pe de altă parte, în aceleași condiții, **a≤b** ia valoarea **false.** Avem egalitate.

3. **a=b.** Atunci **a>b** ia valoarea **false**; **NOT(false)=true**. Pe de altă parte, în aceleași condiții, **a≤b** ia valoarea **true.** Avem egalitate.

De asemenea, mai avem relațiile de mai jos, care se demonstrează în mod asemănător:

NOT(a<b) ⇔ a≥b
NOT(a≤b) ⇔ a>b
NOT(a≥b) ⇔ a<b

Mai avem relațiile lui *de Morgan* (**a** și **b** sunt cele două valori logice).

1. **NOT(a SI b) =NOT a SAU NOT b**;
2. **NOT(a SAU b) =NOT a SI NOT b**;

Pentru demonstrarea acestor relații se iau în considerare toate valorile pe care le pot lua valorile logice **a** și **b**. Demonstrăm prima relație.

a) a=true, b=true. a SI b =true. NOT (a SI b)=false. Pe de altă parte: **NOT a= false**, **NOT b= false**; **false SAU false =false.**

b) a=true, b=false. a SI b =false. NOT (a SI b)=true. Pe de altă parte: **NOT a= false**, **NOT b= true**; **false SAU true =true.**

c) a=false, b=false. a SI b =false. NOT (a SI b)=true. Pe de altă parte: **NOT a= true**, **NOT b= true**; **true SAU true =true.**

d) a=false, b=true. a SI b =false. NOT (a SI b)=true. Pe de altă parte: **NOT a=true**, **NOT b= false**; **true SAU false =true.**