Subprograme

Capitolul

7

- **Noţiunea de modul**
- **Declararea şi apelul unui subprogram**
- **Funcţii şi proceduri în Pascal**
- **Dezvoltarea programelor**
- **Implementări sugerate**
- **Probleme propuse**
- **Soluţiile problemelor**

7.1. Modul

Noţiunea de subprogram a apărut din necesitatea de a diminua complexitatea unui program prin descompunerea acestuia în subprograme. Aplicarea acestui concept reflectă la un anumit nivel *principiul programării modulare*. Conform acestui principiu al ingineriei programării, algoritmii se proiectează astfel încât aceştia să folosească module, unde prin ***modul*** înţelegem o unitate structurală de sine stătătoare (program, subprogram sau unitate de program). Fiecare modul trebuie să aibă un rol bine determinat şi să fie conceput astfel încât să poată fi folosit la nevoie şi în cadrul unui alt program, fără a fi modificat. Orice modul poate fi la rândul său construit din alte module.

Printre *avantajele programării modulare* de care beneficiază şi conceptul de subprogram amintim:

- simplitatea şi claritatea programului;
- modulele sunt mai uşor de descris, de testat, de corectat şi eventual de modificat; optimizarea unui program se poate realiza adesea doar prin înlocuirea unui modul ineficient cu altul mai performant;
- se evită rescrierea unui algoritm de câte ori este nevoie de acesta;
- un modul o dată pus la punct va funcţiona la fel de bine şi într-un alt program care are nevoie de el;
- se creează premisele realizării unor aplicaţii mari prin munca în echipă a mai multor programatori.

Subliniem că este important să se urmărească creşterea independenţei subprogramelor pentru a mări şansele reutilizării lor în cât mai multe programe.

Aplicarea în practică a principiului modularității implică stabilirea unor detalii privind:

- alegerea variantei optime de descompunere;
- verificarea corectitudinii modului în care s-a realizat descompunerea;
- asigurarea comunicării adecvate între subprograme.

Datele care asigură comunicarea între subprogram şi modulul apelant sunt definite într-o interfață a subprogramului cu exteriorul (numită ***lista parametrilor formali***). Restul datelor (*date locale*) sunt ascunse pentru exterior.

Datele definite în subprogram se numesc „locale" şi vor trăi cât timp subprogramul respectiv este activ, din momentul activării (apelării) până la execuția ultimei instrucțiuni a subprogramului.

Datele definite în programul principal se numesc „globale" şi vor fi „văzute" (cu excepția celor redeclarate în subprogramul respectiv) în toate subprogramele, deoarece programul principal este primul bloc care se activează şi ultimul care se dezactivează.

Lista parametrilor formali este declarată în antetul subprogramului şi conține *parametri de „intrare"* şi/sau parametri *de „ieşire"*.

În principiu, un parametru care desemnează date strict de intrare pentru subprogram se va transmite *prin valoare*, iar cel care este de ieşire din subprogram, se va transmite *prin referință*.

7.2. Declararea şi apelul unui subprogram

Structura unui subprogram seamănă cu structura unui program:

Tip_Subprogram Nume(*lista parametrilor formali*) { *antetul subprogramului* }
declarațiile datelor locale
instrucțiune compusă { *descrierea algoritmului de rezolvare a subproblemei* }

unde prin *Tip_Subprogram* înțelegem precizarea tipului subprogramului (în Pascal *funcție* sau *procedură*). *Corpul* subprogramului este format din declarațiile locale ale sale şi o instrucțiune compusă.

Apelul subprogramului se face în partea de acțiuni a modulului apelant. Dacă subprogramul returnează o singură valoare (avem subprogram de tip ***funcție***) apelul poate să apară într-o expresie, oriunde sintaxa limbajului permite prezența unei expresii. Dacă un subprogram calculează mai multe valori (în Pascal avem subprogram de tip ***procedură***), apelul subprogramului are loc în *instrucțiunea procedurală* (va fi în sine o acțiune şi va apărea oriunde poate să apară o instrucțiune).

În principiu, un ***apel*** se scrie precizând numele subprogramului, urmat, între paranteze, de lista *parametrilor actuali*. Efectul apelului constă în activarea subprogramului şi transferul execuției programului la prima instrucțiune executabilă din subprogram.

În prealabil valorile parametrilor actuali transmişi prin valoare se copiază în variabilele care sunt parametri formali (aceştia se păstrează în timpul executării subprogramului în *stiva de execuție*), iar corespunzător parametrilor transmişi prin referință se comunică adresele parametrilor actuali. *Orice modificare de valoare asupra parametrului formal de tip referință (de tip adresă) se va efectua de fapt asupra parametrului actual corespunzător*. Pentru ca acest lucru să fie posibil, parametrul actual aferent unui parametru formal de tip referință trebuie să fie o variabilă.

Lista parametrilor actuali trebuie să aibă tot atâția parametri ca şi lista parametrilor formali şi perechile corespondente de parametri trebuie să fie compatibile ca tip.

Să pătrundem acum mai adânc în procesul apelării unui subprogram.

Pentru gestionarea simplă şi naturală a execuției subprogramelor, se utilizează o zonă specială a memoriei, numită *stivă de execuție* (*Stack*).

Amintim, că în general, se numeşte *stivă* o structură de date cu două capete *bază* şi *vârf*. Introducerea, cât şi extragerea de date în/dintr-o stivă este posibilă doar la unul dintre capete, numit *vârful* stivei. Stiva de execuție este o zonă de memorie în care se depun (şi din care se extrag) date conform acestui principiu.

În urma apelului unui subprogram:

- se întrerupe executarea modulului apelant
- se salvează pe stiva de execuție următoarele informații:
  - adresa instrucțiunii la care se va reveni după execuția subprogramului apelat;
  - valorile parametrilor transmişi prin valoare;
  - adresele parametrilor de tip referință;
- se alocă spațiu pe stivă variabilelor locale;
- se activează subprogramul (se predă controlul primei instrucțiuni a acestuia);
- se execută instrucțiunile subprogramului.

În momentul terminării execuției subprogramului se elimină de pe stivă toate informațiile care s-au depus după apelul acestuia şi controlul revine primei instrucțiuni de după apel din modulul apelant.

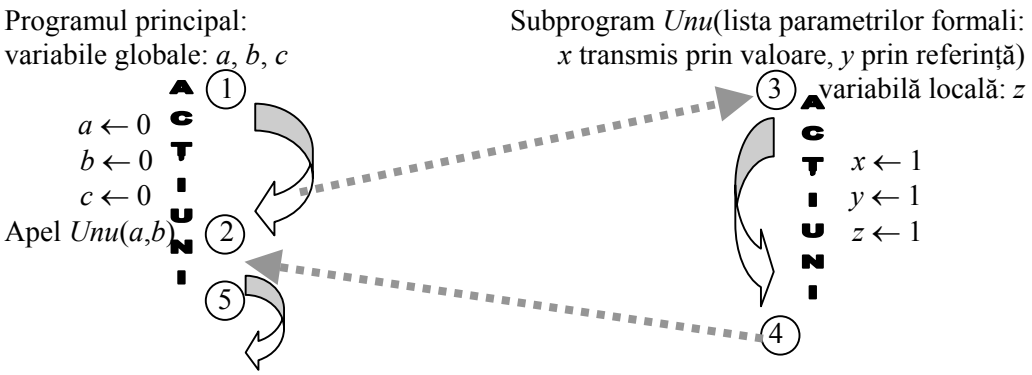
Exemplu

Fie trei variabile *a*, *b* şi *c* declarate în programul principal. Aceste variabile sunt inițializate cu 0 şi se apelează subprogramul *Unu*. După revenirea din subprogram se afişează valorile *a*, *b* şi *c*.

Subprogramul *Unu* are doi parametri formali: *x* este parametru transmis prin valoare, iar *y* este parametru de tip referință. În subprogram se declară variabila locală *z*. Acest subprogram atribuie celor trei variabile *x*, *y* şi *z* valoarea 1.

În următoarea figură programul principal este simbolizat prin segmentul vertical de

la 1 la 6, punctul 2 corespunzând apelului de subprogram, iar punctul 5 primei instrucțiuni de după apel. Subprogramul este reprezentat de segmentul vertical având la capete punctele 3 (prima instrucțiune a subprogramului) şi 4 (ultima instrucțiune). Liniile cu săgeți arată ordinea de execuție a instrucțiunilor din cele două unități de program.



Pentru a prezenta modul de lucru cu stiva de execuție, vom urmări conținutul acesteia în cele 5 puncte (momente) marcate. Variabilele globale *a*, *b* şi *c* nu se alocă pe stivă, ci în secțiunea de date existent la dispoziția programului principal, dar le vom preciza valorile în cele 5 momente.

Moment	**Variabile globale**	**Conținutul stivei**		
1	nedeterminate	vid		
2	$a = 0$; $b = 0$; $c = 0$;	vid		
3	$a = 0$; $b = 0$; $c = 0$;	$x = 0$	adresa lui y	z nedeterminat
4	$a = 0$; $\mathbf{b = 1}$; $c = 0$;	$x = 1$	adresa lui y	$z = 1$
5	$a = 0$; $b = 1$; $c = 0$;	vid		

Observăm din tabel că instrucțiunea de atribuire $x \leftarrow 1$ nu are efect asupra parametrului actual *a*, deoarece a fost transmis prin valoare, în schimb *b* devine 1, deoarece acțiunea asupra parametrului formal *y* transmis prin referință s-a efectuat asupra parametrului actual *b*.

În limbajul Pascal într-un program principal sau subprogram vom avea:

O parte de definiții şi declarații:

```
type...                    { definiții de tip }
const...                   { definiții de constante }
var...                     { declarații de variabile }
procedure...               { declarații de subprograme }
function...
```

O parte de acțiuni (o instrucțiune compusă):
Begin...
instrucțiuni
End.

Observație
Oricare secțiune de definiție sau declarație poate să lipsească, în schimb instrucțiunea compusă trebuie să fie prezentă (dar, poate fi vidă).

7.3. Funcții şi proceduri (în Pascal)

În limbajul Pascal subprogramele pot fi:
- ***funcții***: calculează mai multe valori; una se returnează prin numele funcției;
- ***proceduri***: calculează mai multe valori (sau nici una).

Apelul unei funcții face parte dintr-o instrucțiune (unde apare într-o expresie), pe când apelul de procedură este definit ca fiind o instrucțiune.

Declararea unei funcții în limbajul Pascal
function *Nume_funcție* (*lista parametrilor formali*) **:** *tip_rezultat;*
declarațiile şi definițiile locale funcției
instrucțiune compusă

Observație
Printre instrucțiunile din cadrul instrucțiunii compuse a funcției (corpul funcției) trebuie să existe cel puțin o instrucțiune de atribuire prin care numelui funcției i se atribuie o valoare.

Declararea unei proceduri în limbajul Pascal
procedure *Nume_Procedură* (*listă parametrilor formali*) ;
declarații şi definiții locale procedurii
instrucțiune compusă

Observații (valabile în limbajul Pascal)
1) În fața oricărui parametru formal transmis prin referință se scrie cuvântul cheie **var**. Parametrii formali care nu au cuvântul **var** în față se consideră a fi parametri transmişi prin valoare.
2) Tipul parametrilor formali se poate preciza *doar cu identificator de tip*;
3) Declarațiile în lista parametrilor formali sunt separate prin **;**. Atunci când avem mai mulți parametri de acelaşi tip, aceştia se separă prin virgulă. În următorul exemplu *a*, *d*, *e* şi *f* sunt parametri de ieşire (transmişi prin referință), *b* este parametru transmis prin valoare.
```
procedure Suma(var a:Longint; b:Char; var d,e,f:Real);
```
3) Parametrii actuali, menționați în apel sunt separați prin virgulă.

4) Valoarea returnată de o funcție (ținând cont de faptul că aceasta returnează prin identificatorul ei *o singură* valoare) poate fi doar de următoarele tipuri:[*)]
   - orice tip întreg;
   - orice tip real;
   - caracter (`Char`);
   - `Boolean`;
   - enumerare (având definit identificator de tip);
   - subdomeniu (având definit identificator de tip);
   - **`string`**.

Exemplul 1

Să se calculeze următoarea expresie: $C(n,k)=\dfrac{1\cdot 2\cdot ...\cdot n}{(1\cdot 2\cdot ...\cdot k)[1\cdot 2\cdot ...\cdot (n-k)]}$.

Se observă că în numărător trebuie să calculăm factorialul lui *n*, în numitor factorialul lui *k* şi factorialul lui (*n* – *k*). Vom scrie un subprogram pentru calcularea factorialului lui *x* şi îl vom apela pentru *n*, pentru *k* şi pentru (*n* – *k*). Deoarece acest subprogram, corespunzător unui apel va calcula o singură valoare, ea va fi atribuită identificatorului funcției cu care o calculăm.

În programul principal declarăm variabilele *n* şi *k*, ele reprezintă datele de intrare. Rezultatului nu-i planificăm nici o variabilă, deoarece acesta va fi accesat pentru afişare prin identificatorul funcției, în momentul apelului.

Funcția `fact` va avea un parametru formal *x* care va fi pentru acest subprogram un parametru de intrare (deci, se va transmite prin valoare). Funcția va returna valoarea factorialului lui *x*. Variabila locală *p* este necesară, deoarece în absența ei, ar trebui să scriem în instrucțiunea **`for`** instrucțiunea de atribuire: `fact:=fact*i` care ar însemna apelarea funcției `fact` în momentul în care aceasta încă este activă. (De fapt, dacă am scrie instrucțiunea de atribuire exact în această formă, compilatorul ne-ar cere parametru actual în membrul drept după identificatorul `fact`, deoarece doar în membrul stâng putem scrie identificator de funcție fără parametri, apariția în membrul drept al acestuia fiind considerată apel de funcție[**)]).

```
Program Expresie;
var n,k:Longint;                                  { variabile globale }

function fact(x:Integer):Longint;                 { antetul funcției }
var i:Integer;                                    { variabile locale }
    p:Longint;
```

[*)] Există funcții care returnează un pointer (vom învăța în clasa a X-a).

[**)] Astfel am avea un apel recursiv (vezi capitolul 17).

```
begin
  p:=1;                                   { iniţializarea factorialului }
  for i:=1 to x do p:=p*i;                { calcularea factorialului }
  fact:=p                                 { atribuire identificatorului funcţiei }
end;

Begin
  ReadLn(n,k);
  WriteLn(fact(n)/(fact(k)*fact(n-k)))
End.
```

Exemplul 2

Se consideră următorul program. Ce afişează acesta?

```
Program Exemplu2;
var i:Integer;

function Suma(n:Integer):Integer;
var S:Integer;
begin
  S:=0;
  for i:=1 to 5 do
    S:=S+n;
  suma:=S
end;

Begin
  for i:=1 to 5 do begin
    Write(i,' ');
    Write(suma(i),' ');
    WriteLn(i)
  end
End.
```

Să urmărim variaţia variabilelor:

- programul principal începe cu ciclul **for**, prima valoare a variabilei *i* este 1;
- se afişează 1;
- se apelează subprogramul cu parametrul actual *i*, a cărui valoare se copiază în *n*, deoarece este transmis prin valoare;
- în subprogram se calculează *S*, care va avea valoarea 0 + 1 + 1 + 1 + 1 + 1 = 5;
- se atribuie valoarea 5 identificatorului de funcţie;
- se afişează valoarea 5;
- urmează o altă afişare a variabilei *i* şi (surpriză!) se afişează 5 (ultima valoare a lui *i* conform ciclului **for** din subprogram), ceea ce conduce şi la terminarea instrucţiunii **for** din programul principal.

Se observă că valoarea variabilei *globale i* s-a modificat „din greşeală” în subprogram. Pentru a evita astfel de greşeli se va urmări ca variabilele folosite de subprogram să fie întotdeauna definite local subprogramului în care se folosesc.

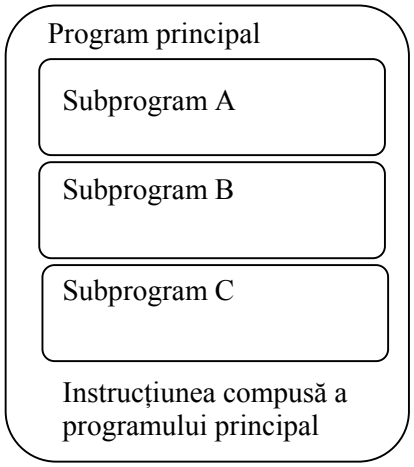
Pentru a obține rezultatele dorite, în programul de mai sus se va adăuga în partea de declarații locale a funcției `Suma` declarația: **`var`** `i:Integer`.

7.4. Dezvoltarea programelor

Dacă, în procesul proiectării algoritmilor, o problemă se descompune în subprobleme şi acestea se descompun la rândul lor în alte subprobleme până când se obțin subprobleme care nu necesită continuarea procesului de divizare spunem că avem un program dezvoltat descendent (***top down***). Această tehnică corespunde modului în care uzual se proiectează algoritmii, folosind subalgoritmi. Tehnica de proiectare în care se porneşte de la unele programe inițiale, acestea fiind folosite pentru a construi module din ce în ce mai sofisticate care, asamblate să rezolve problema, se numeşte ***bottom up***.

7.4.1. Dezvoltarea ascendentă (*bottom up*)

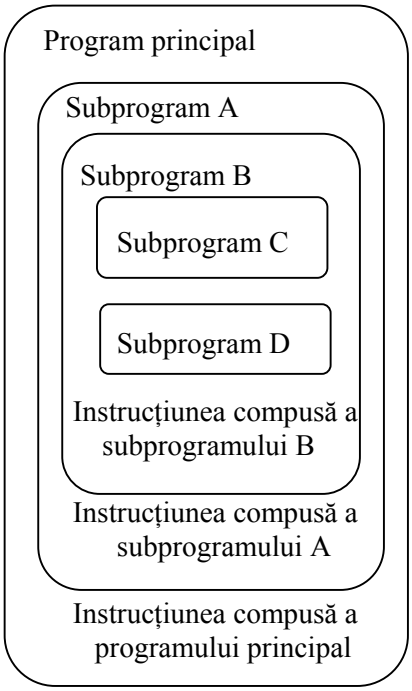
Un program dezvoltat ascendent va avea toate subprogramele declarate unul după altul în programul principal. O astfel de abordare are ca avantaj faptul că subprogramele sunt mai uşor de testat şi de controlat.



Trebuie menționat faptul că fiecare subprogram va putea apela numai subprograme declarate anterior propriei declarații. Din această restricție de vizibilitate decurge şi dezavantajul acestui stil de lucru care constă în dependența subprogramelor de unele subprograme definite anterior.

7.4.2. Dezvoltarea descendentă (*top down*)

Într-un program dezvoltat descendent fiecare subprogram va avea în propria zonă de definiţii şi declaraţii subprogramele pe care le foloseşte. În cazul subprogramelor imbricate unul în celălalt funcţionează aceleaşi reguli ca între unitatea de program principal şi subprogram, adică entităţile declarate, de exemplu, conform figurii următoare, în subprogramul A se „văd" în subprogramele B, C şi D. Astfel, de regulă, lista parametrilor se scurtează.



7.5. Implementări sugerate

În scopul formării deprinderilor de a lucra cu subprograme vă recomandăm să:

1. rezolvaţi majoritatea problemelor tratate până acum, în care problema iniţială se descompune în subprobleme folosind subprograme;
2. rezolvaţi probleme în care un acelaşi subprogram se apelează în mod repetat pentru date diferite;
3. realizaţi un *unit* propriu care conţine toate problemele rezolvate cu subprograme şi scrierea unor programe care folosesc acest unit.

7.6. Probleme propuse

7.6.1. Cifre comune

Să se afişeze cifrele comune a două numere naturale date.

Date de intrare
Cele două numere se citesc de pe primele două linii ale fişierului **COMUN.IN**.

Date de ieşire
Cifrele comune ale celor două numere se vor scrie în fişierul **COMUN.OUT**, despărțite prin câte un spațiu. În cazul în care nu există nici o cifră comună, în fişierul de ieşire se va scrie `'Nu exista cifre comune.'`.

Restricții şi precizări
- $1 \leq număr_1, număr_2 \leq 1000000000$.

Exemple

COMUN.IN
```
12345
9377722
```
COMUN.OUT
```
2 3
```

COMUN.IN
```
12345
9877788
```
COMUN.OUT
```
Nu exista cifre comune.
```

7.6.2. Prime şi „inversele" lor prime

Să se genereze toate perechile de numere prime formate din trei cifre care au proprietatea de a fi, unul inversul celuilalt. Aici prin „invers" înțelegem numărul format din cifrele numărului dat, în ordine inversă.

Date de ieşire
În fişierul de ieşire **PRIME.OUT** se vor scrie perechile de numere, câte una pe fiecare linie, despărțite prin câte un spațiu.

Exemplu
PRIME.OUT
```
101 101
107 701
113 311
131 131
...
```

7.6.3. Eliminare de numere perfecte şi prietene

Se consideră un şir de numere naturale. Să se elimine din acest şir toate numerele perfecte, precum şi perechile de numere prietene aflate pe poziţii consecutive.

Date de intrare
Pe prima linie a fişierului **SIR.IN** se află elementele şirului, separate prin spaţii.

Date de ieşire
Elementele rămase în şir după eliminare, se vor scrie în fişierul **SIR.OUT**, separate prin câte un spaţiu.

Restricţii şi precizări
- $1 \le nr \le 1000000000$.

Exemple

SIR.IN	SIR.OUT
11 28 6 133 220 284 7 99	11 133 7 99

7.6.4. Generarea cuvântului de lungime maximă

Să se determine cuvântul de lungime maximă care poate fi construit din literele unui text, astfel încât caracterele cuvântului să fie în ordine alfabetică şi să fie culese din text exact în ordinea în care se află în cuvânt (nu neapărat pe poziţii succesive).

Date de intrare
Pe prima linie a fişierului de intrare **CUV.IN** se află textul dat.

Date de ieşire
Cuvântul determinat, care are lungime maximă se va scrie în fişierul **CUV.OUT**.

Restricţii şi precizări
- textul are cel mult 255 de caractere;
- în text există cuvinte formate din litere mici ale alfabetului englez şi spaţii.

Exemplu

CUV.IN	CUV.OUT
iarba si iasca	abis

7.7. Soluțiile problemelor propuse

7.7.1. Cifre comune

Versiunea de algoritm prezentată în continuare se bazează pe operațiile cu mulțimi. În acest caz este necesar un subprogram care să construiască o mulțime din cifrele unui număr. Subproblemele care vor fi implementate cu subprograme în cazul acestei probleme, vor fi:

- citirea numerelor;
- crearea mulțimii de cifre, (mulțimile le implementăm cu ajutorul şirurilor, dar după ce se va învăța tipul **`set`** (în Pascal), se recomandă, ca exercițiu, implementarea programului, folosind acest tip de structură de date);
- afişarea cifrelor aparținând intersecției de mulțimi.

În Pascal declarația şirului care reprezintă mulțimea va fi:

```
type multime_de_cifre=array[0..9] of Boolean;
var A,B,inters:multime_de_cifre;
```

Dacă, de exemplu, valoarea elementului `A[i]` este *adevărat*, înseamnă că cifra `i` face parte din mulțimea `A`.

```
Subalgoritm Creare_mulțime(număr,M):        { subprogram tip procedură }
  pentru cifra=0,9 execută:                 { deocamdată mulțimea M este vidă }
    M[cifra] ← fals
  sfârşit pentru
  cât timp număr ≠ 0 execută:
    cifra ← rest[număr/10]
    M[cifra] ← adevărat                     { cifra cifra există în număr }
    număr ← [număr/10]
  sfârşit cât timp
sfârşit subalgoritm
```

Intersecția mulțimii `A` cu `B` este un şir de valori logice, unde un element are valoarea *adevărat* dacă, cifra corespunzătoare se regăseşte şi în `A` şi în `B`.

```
Subalgoritm Intersecție(A,B,inters):        { subprogram tip procedură }
  pentru cifra=0,9 execută:                 { deocamdată intersecția este vidă }
    inters[cifra] ← fals
  sfârşit pentru
```

```
  pentru cifra=0,9 execută:
    dacă A[cifra] şi B[cifra] atunci { dacă cifra se află şi în A şi în B }
      inters[cifra] ← adevărat
    sfârşit dacă
  sfârşit pentru
sfârşit subalgoritm
```

Pentru afişare, verificăm apartenenţa cifrei *cifra*, (*cifra* = 0, 1, ..., 9) la mulţimea intersecţie `inters`. Dacă mulţimea este vidă, afişăm mesajul cerut în enunţ.

```
Subalgoritm Afişare(inters):                    { subprogram tip procedură }
  există ← fals
  pentru cifra=0,9 execută:
    dacă inters[cifra] atunci
      scrie cifra
      există ← adevărat                         { am găsit o cifră comună }
    sfârşit dacă
  sfârşit pentru
  dacă nu există atunci
    scrie 'Nu exista cifre comune.'
  sfârşit dacă
sfârşit subalgoritm
```

Programul principal va apela subprogramele în ordinea lor firească: citire, creare mulţime *A*, creare mulţime *B*, determinare intersecţie şi afişarea intersecţiei.

7.7.2. Prime şi „inversele” lor prime

O primă idee de rezolvare ar fi de a genera pe rând toate numerele de trei cifre (de la 100 la 999, eventual de la 101 la 997) şi de a verifica dacă atât numărul, cât şi acel număr care conţine cifrele numărului dat în ordine inversă sunt sau nu numere prime. În caz afirmativ acestea le-am scrie în fişierul de ieşire.

Observăm că pentru fiecare număr studiat suntem nevoiţi să verificăm de două ori dacă un număr este prim (o dată pentru numărul dat, a doua oară pentru numărul generat din cifrele numărului luate în ordine inversă). Acesta este un motiv suficient de puternic pentru a scrie un subalgoritm pentru verificarea primalităţii unui număr. Subprogramul va fi apelat odată pentru numărul dat şi apoi pentru „inversul” său.

```
Subalgoritm Prim(nr):                           { subprogram tip funcţie }
  limita ← parte întreagă din rădăcina pătrată a lui nr
  divizor ← limita
  cât timp nr nu se împarte exact la divizor execută:
    divizor ← divizor - 1
  sfârşit cât timp
```

```
                    { dacă s-a găsit cel puţin un divizor >1 numărul nr nu este prim }
  dacă divizor = 1 atunci prim ← adevărat
                   altfel prim ← fals
  sfârşit dacă
sfârşit subalgoritm
```

Determinarea numărului format din cifrele numărului dat în ordine inversă o putem realiza cu subalgoritmul `Invers`, care în Pascal va fi o funcție care va returna valoarea numărului nou.

```
Subalgoritm Invers(nr):                          { subprogram tip funcţie }
  inv ← 0
  cât timp nr ≠ 0 execută:
    cifra ← rest[nr/10]
    inv ← inv*10 + cifra
    nr ← [nr/10]
  sfârşit cât timp
  invers ← inv
sfârşit subalgoritm

Algoritm Prime:              { algoritmul corespunzător programului principal }
  pentru număr=101,997 execută:
    dacă Prim(număr) şi Prim(invers(număr)) atunci
      scrie număr,invers
    sfârşit dacă
  sfârşit pentru
sfârşit algoritm
```

În urma executării programului care implementează acest algoritm observăm că fiecare pereche *este afişată de două ori*. Dacă dorim să evităm afişarea dublă, este necesară modificarea algoritmului.

Dacă punem condiția ca prima cifră a primului număr să fie mai mică sau egală cu ultima cifră a primului număr, vom evita afişările duble. Al doilea număr se construieşte întotdeauna din primul şi deci nu este necesar să punem condiții asupra lui. Numerele din perechi astfel construite vor fi generate crescător. Instrucțiunea alternativă din structura de tip **pentru** s-ar putea scrie astfel:

```
dacă [număr/100] ≤ rest[număr/10] atunci...
```

În acest algoritm, instrucțiunea **pentru** se va relua de 997 – 101 + 1 = 897 ori.

Într-o altă abordare a acestei probleme generăm primul număr din cifrele *a*, *b* şi *c* şi al doilea din aceleaşi cifre, dar în ordine inversă, ținând cont totodată şi de condiția ca prima cifră să fie mai mare sau egală cu a treia. Această variantă are şi avantajul de a

evita generarea dublurilor fără a folosi o condiție suplimentară. De asemenea, nu vom lua în considerare ca posibil număr având proprietatea solicitată decât numerele impare. Un număr impar are ultima cifră impară, deci în ciclul **for** care generează a treia cifră se vor construi numere doar cu cele 5 cifre impare (1, 3, 5, 7, 9).

Astfel, numărul executărilor corpului ciclului este $10 \cdot 9 \cdot 5 = 450$ (care este mai mic decât numărul executărilor în cazul primei variante: 897). În plus, această variantă nu necesită apelul subalgoritmului `invers`, deoarece inversul numărului poate fi construit printr-o simplă atribuire. Subalgoritmul de verificare a primalității se va executa doar de 332 de ori, deoarece dacă despre `număr` se constată că nu este prim, numărul `invers` nu se mai verifică.

```
Algoritm Prime_şi_inversate_prime_2:
  pentru a=1,9 execută:                { prima cifră din număr nu poate fi 0 }
    pentru b=0 la 9 execută:
                          { ultima cifră trebuie să fie cel puțin egală cu prima cifră }
      pentru c=a la 9 execută:
        dacă c este număr impar atunci { limităm generarea la numere impare }
          număr ← a*100 + b*10 + c
          invers ← c*100 + b*10 + a
          dacă Prim(număr) şi Prim(invers) atunci
            scrie număr,invers
          sfârşit dacă
        sfârşit dacă
      sfârşit pentru
    sfârşit pentru
  sfârşit pentru
sfârşit algoritm
```

7.7.3. Eliminare de numere perfecte şi prietene

Cerința acestei probleme este de a elimina dintr-un şir numerele perfecte sau perechile de numere prietene aflate pe poziții consecutive în şir. Se observă că acțiunea principală este de a elimina unul sau două elemente dintr-un şir, pe baza unei condiții date.

Subalgoritmul `Elimin` va elimina elementul având indicele *poziție* din şirul *a* şi va returna şirul modificat, precum şi dimensiunea actualizată a acestuia.

```
Subalgoritm Elimin(a,n,poziție):          { subprogram tip procedură }
  pentru i=poziție,n-1 execută:     { se deplasează elementele tabloului }
    a[i] ← a[i+1]          { cu o poziție la stânga, începând cu poziție + 1 }
  sfârşit pentru
  n ← n - 1                                    { numărul de elemente scade }
sfârşit subalgoritm
```

Proprietățile numerelor care trebuie eliminate:

- număr perfect: este egal cu suma divizorilor mai mici decât el însuşi;
- numere prietene: perechi de numere în care un număr este egal cu suma divizorilor celuilalt (mai mici decât acesta) şi invers (vezi capitolul 4).

Se observă că în ambele situații este nevoie de suma divizorilor proprii (plus 1) ai numărului. Deci, va fi util să scriem un subprogram care calculează această sumă.

```
Subalgoritm Suma_divizori(număr):              { subprogram tip funcție }
  s ← 1                                  { în această sumă se adaugă şi 1 }
  pentru d=2,[număr/2] execută:
    dacă rest[număr/d] = 0 atunci
      s ← s + d
    sfârşit dacă
  sfârşit pentru
  Suma_divizori ← d
sfârşit subalgoritm
```

Cele două subprograme care verifică dacă un număr este perfect, respectiv dacă două numere date sunt prietene vor returna o valoare de adevăr (*adevărat* sau *fals*).

```
Subalgoritm Perfect(număr):                    { subprogram tip funcție }
  dacă Suma_divizori(număr) = număr atunci
    Perfect ← adevărat
  altfel
    Perfect ← fals
  sfârşit dacă
sfârşit subalgoritm

Subalgoritm Prietene(număr1,număr2):           { subprogram tip funcție }
  dacă (Suma_divizori(număr1) = număr2) şi
       (Suma_divizori(număr2) = număr1) atunci
    Prietene ← adevărat
  altfel Prietene ← fals
  sfârşit dacă
sfârşit subalgoritm
```

7.7.4. Generare cuvânt de lungime maximă

În rezolvarea acestei probleme avem nevoie de două tablouri de lucru:

- Elementele şirului L vor fi lungimi de cuvinte. L_i păstrează lungimea cea mai mare posibilă a cuvântului care începe cu litera care în text se află pe poziția i. Aceste cuvinte trebuie să respecte cerința ca literele din componența lor să fie în ordine alfabetică.

- Tabloul *urm* îl construim pentru a asigura posibilitatea de a afişa cel mai lung cuvânt găsit. urm_i va conține pentru a *i*-a literă din cel mai lung cuvânt, indicele (în textul dat) a literei următoare în cuvânt.

Textul dat îl vom parcurge de la sfârşit spre început, ceea ce înseamnă că vom iniția-liza valoarea L_n cu 1, având semnificația că cel mai lung cuvânt care începe cu a *n*-a literă din text are lungimea 1. De asemenea, putem iniția liza şi valoarea elementului urm_n cu 0, deoarece ştim că după ultimul element nu poate urma nici unul.

textul	a	p	a		c	u	r	g	e		l	i	n
Indice	1	2	3	4	5	6	7	8	9	10	11	12	13
L													1
urm													0

În cele ce urmează vom căuta să determinăm acel cel mai lung cuvânt în fața căruia este permis să alipim a *i*-a literă din textul dat. Evident, aceasta, dacă este permis, nu poate fi la acest pas decât ultima literă. În concluzie L_{12} va fi egal cu 2 (cel mai lung cuvânt care începe cu a 12-a literă din text are lungimea 2), iar urm_{12} va fi 13 (după a 12-a literă urmează a 13-a):

textul	a	p	a		c	u	r	g	e		l	i	n
Indice	1	2	3	4	5	6	7	8	9	10	11	12	13
L												2	1
urm												13	0

Studiem litera `l`. Aceasta nu poate fi pusă în fața literei `i`, deoarece încalcă cerința privind ordonarea alfabetică, dar se poate pune în fața literei `n`. Deci, cel mai lung cuvânt care începe cu litera `l` are lungimea 2, iar următoarea literă din cuvânt este din nou a 13-a.

textul	a	p	a		c	u	r	g	e		l	i	n
Indice	1	2	3	4	5	6	7	8	9	10	11	12	13
L											2	2	1
urm											13	13	0

Caracterul următor este spațiu. Acesta nu intră în configurația nici unui cuvânt, deci elementele corespunzătoare din şirul *L* şi *urm* vor fi egale cu 0. Urmează să studiem caracterul `e`. Acesta se poate pune în fața lui `l`, în fața lui `i` şi în fața lui `n`. Alegem dintre cele trei posibilități pe aceea care conduce la un cuvânt mai lung. Dintre literele `l` şi `i` o alegem pe prima, şi astfel cel mai lung cuvânt care începe cu litera `e` are lungimea 3, iar următoarea literă este a 11-a.

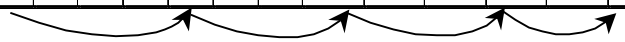
textul	a	p	a		c	u	r	g	e		l	i	n
Indice	1	2	3	4	5	6	7	8	9	10	11	12	13
L									3	0	2	2	1
urm									11	0	13	13	0

Litera `g` nu poate fi pusă în fața lui `e`, în schimb poate fi pusă în fața lui `l`, `i` şi `n`. Procedăm cu litera `g` la fel cum am procedat cu `e`. Literele `r` şi `u` nu pot fi puse în fața nici uneia dintre literele luate în considerare până acum, deci valorile corespunzătoare lor sunt 1 în şirul *L* şi 0 în şirul *urm*. Litera `c` poate fi pusă în fața tuturor caracterelor; alegem acea posibilitate care dă naştere la un cuvânt mai lung. Aceasta este `g`. Continuând acest procedeu până la epuizarea şirului, ajungem la următoarele valori în cele două şiruri *L* şi *urm*:

textul	a	p	a		c	u	r	g	e		l	i	n
indice	1	2	3	4	5	6	7	8	9	10	11	12	13
L	5	2	5	0	4	1	1	3	3	0	2	2	1
urm	5	6	5	0	8	0	0	11	11	0	13	13	0

După ce am construit aceste două şiruri, lungimea celui mai lung cuvânt este egal cu valoarea elementului maxim din şirul *L*. Deoarece noi trebuie să afişăm nu lungimea, ci cuvântul, vom proceda în felul următor. Afişăm acel caracter (din text) care are indicele egal cu indicele elementului maxim din şirul *L* (cu această literă începe cel mai lung cuvânt). Elementul şirului *urm*, având acest indice conţine indicele literei următoare, care se afişează. Urmărind astfel conţinutul şirului *urm* afişăm literele celui mai lung cuvânt până la sfârşitul acestuia (valoarea corespunzătoare ultimei litere în şirul *urm* este 0).

textul	**a**	p	a		**c**	u	r	**g**	e		**l**	i	**n**
Indice	1	2	3	4	5	6	7	8	9	10	11	12	13
L	5	2	5	0	4	1	1	3	3	0	2	2	1
urm	5	6	5	0	8	0	0	11	11	0	13	13	0



Afişăm `a`, deoarece valoarea maximă în şirul *L* se află pe poziția 1. Apoi:
$urm_1 = 5$, afişăm caracterul `c` (poziția 5);
$urm_5 = 8$, afişăm litera `g` (poziția 8);
$urm_8 = 11$, afişăm litera `l` (poziția l1);
$urm_{11} = 13$, afişăm litera `n` (poziția 13);
$urm_{13} = 0$, deci afişarea se termină.

Subalgoritmul care modelează prelucrarea descrisă este următorul:

```
Subalgoritm Caut(n,textul):
  L[n] ← 1      { cel mai lung cuvânt care începe cu a n-a litera are lungimea 1 }
  urm[n] ← 0                    { după ultima literă nu urmează alt caracter }
  pentru i=n-1,1 execută:
    max ← 0
    indice ← 0
    pentru j=i+1,n execută:
                { încercăm să punem litera textul[i] în fața lui textul[j] }
      dacă (textul[i] ≠ ' ') şi (textul[j] ≠ ' ') şi
                                    (textul[i] < textul[j]) atunci
        dacă L[j] > max atunci      { dintre literele în fața cărora se poate }
           { pune a i-a literă, o alegem pe cea cu care începe cel mai lung cuvânt }
          max ← L[j]
          indice ← j
        sfârşit dacă
      sfârşit dacă
    sfârşit pentru
    dacă textul[i] ≠ ' ' atunci
   { cel mai lung cuvânt care începe cu a i-a literă este egală cu lungimea celui mai }
      L[i] ← max + 1 { lung cuvânt în fața căruia punem a i-a literă (max) + 1 }
    altfel
      L[i] ← 0                                  { dacă am avut caracter spațiu }
    sfârşit dacă
    urm[i] ← indice  { în indice am ținut minte în fața cărei litere o punem }
  sfârşit pentru
  Maxim                                        { căutăm lungimea maximă din L }
  Scrie
sfârşit subalgoritm
```

În subalgoritmul de afişare *indice* reprezintă indicele elementului maxim în şirul *L*.

```
Subalgoritm Scrie:
  scrie textul[indice]     { afişăm litera cu care începe cel mai lung cuvânt }
    i ← urm[indice]                              { indicele literei următoare }
    cât timp i > 0 execută:                        { cât timp există următor }
      scrie textul[i]
      i ← urm[i]             { indicele literei care urmează după cea afişată }
    sfârşit cât timp
sfârşit subalgoritm
```

Subprogramele `Maxim` şi `Scrie` fac parte din zona de declarații a subprogramului `Caut`. Astfel toate variabilele declarate în `Caut` sunt accesibile din subprogramele `Maxim` şi `Scrie`, acestea neavând astfel nevoie de parametri.