Recapitulare

Capitolul

1

- **Prelucrarea fişierelor de tip text**
- **Principii în proiectarea algoritmilor**
- **Prelucrarea numerelor**
- **Conversii între două baze de numeraţie**
- **Tablouri unidimensionale**
- **Subprograme**
- **Ordonări şi căutări**
- **Şiruri de caractere**
- **Tablouri bidimensionale**
- **Polinoame**
- **Mulţimi**
- **Tipul înregistrare**
- **Operaţii pe biţi**
- **Recursivitate**
- **Metoda backtracking**
- **Probleme propuse**

1.1. Prelucrarea fişierelor de tip text

Datele care urmează să fie prelucrate cu calculatorul sunt păstrate în fişiere. Programele care trebuie rezolvate la concursurile de programare, va trebui să citească datele de intrare din fişiere şi să scrie rezultatele în alte fişiere. Un *fişier de tip text* este o colecţie de înregistrări în care înregistrările sunt caractere (sau şiruri de caractere) structurate pe linii.

Variabilele de tip fişier le declarăm în secţiunea **`var`** a programului (subprogramului) şi le asociem fişierelor precizate în enunţ cu ajutorul procedurii predefinite `Assign(`*identificator, nume_fişier*`)`. Aceasta realizează legătura între variabila de tip fişier şi fişierul existent pe suport, unde *nume_fişier* este o expresie de tip *şir de caractere*.

Procedurile predefinite (în Pascal) cu care deschidem, respectiv închidem un fişier de tip text identificat prin variabila `f` sunt:

- `ReWrite(f)`: pentru crearea fişierului sau rescrierea acestuia, dacă el există;
- `Reset(f)`: pentru deschiderea fişierului pentru citire;

- `Append(f)`: pentru adăugare de înregistrări la sfârşitul fişierului;
- `Close(f)`: realizează închiderea fişierului.
- `Read(f,a1,a2,...,an)`: are ca efect citirea a *n* date din fişier;
- `ReadLn(f,a1,a2,...,an)`: are ca efect citirea a *n* date din fişier, după care următoarea citire se va face de pe linie nouă;
- `Write(f,a1,a2,...,an)`: se realizează scrierea a *n* date în fişier;
- `WriteLn(f,a1,a2,...,an)`: se realizează scrierea a *n* date în fişier şi a marcajului de sfârşit de linie;
- `Rename(f,nume_nou)`: redenumeşte un fişier la nivel fizic, (adică pe suport);
- `Eof(f)`: funcţie booleană care returnează valoarea `true` dacă următorul caracter care ar urma să fie citit din fişier este marcajul de sfârşit de fişier şi `false` altfel.
- `EoLn(f)`: funcţie booleană care returnează valoarea `true` dacă următorul caracter care ar urma să fie citit din fişier este marcajul de sfârşit de linie sau marcajul de sfârşit de fişier şi `false` în caz contrar.

1.2. Principii în proiectarea algoritmilor

1.2.1. Sume şi produse

Orice sumă se iniţializează cu elementul neutru faţă de adunare (0) şi orice produs cu elementul neutru faţă de înmulţire (1). Apoi se calculează suma, respectiv produsul. Singura problemă care poate să apară se referă la faptul că mediile de programare nu ne atenţionează când o astfel de sumă sau produs depăşeşte valoarea maximă posibilă de reprezentat în cadrul tipului variabilei sumă (sau produs).

În concluzie, este indicat să ne protejăm din program faţă de posibilele depăşiri. La concursurile de programare, de regulă, propunătorii precizează dimensiunea datelor precum şi mărimile componentelor. În concluzie, măsurile de protecţie vor conferi rezolvitorului o siguranţă mai mare în timpul lucrului, un mod de a cere avertisment după caz, chiar de către program.

1.2.2. Medii

Pentru a calcula o medie aritmetică, mai întâi se calculează o sumă. Dacă numărul termenilor din sumă se cunoaşte, media se calculează împărţind suma la acest număr. Dacă în sumă se adună doar anumite numere, acestea se vor număra, în paralel cu calcularea sumei. Înainte de împărţire vom verifica dacă numărul termenilor din sumă nu cumva este 0, pentru a evita o împărţire nepermisă. De asemenea, în acest caz fie se afişează un mesaj, fie se realizează sarcinile specificate în enunţul problemei.

1.2.3. Căutarea secvențială

Presupunem că trebuie să căutăm un element având o valoare cunoscută sau având o anumită proprietate. Reamintim că o valoare căutată şi găsită nu trebuie căutată în restul datelor. În concluzie nu vom căuta cu **for**, ci cu o structură repetitivă cu număr necunoscut de paşi. Dacă trebuie/dorim să lucrăm cu **for**, în Pascal avem posibilitatea să întrerupem căutarea în momentul găsirii elementului căutat, apelând subprogramul `Break`. Dacă trebuie să regăsim toate elementele având proprietatea dată, va trebui să prelucrăm toate datele.

1.2.4. Determinarea elementului minim/maxim

Rezolvarea acestor probleme necesită prelucrarea fiecărui element din mulțime/şir. Variabila desemnată să rețină valoarea minimului/maximului (*min/max*) va fi inițializată (dacă este posibil) cu prima valoare prelucrată. Inițializarea cu o valoare „străină" se realizează doar în cazul în care prima variantă nu este posibilă sau este mult prea anevoioasă, caz în care aceasta se alege cu mare atenție.

1.3. Prelucrarea numerelor

1.3.1. Construirea numărului întreg din caractere cifre. Schema lui Horner

Algoritmul cu care construim un număr întreg, pornind de la cifrele sale, (în ordinea în care acestea apar în număr) este următorul:

```
Algoritm Horner:
  n ← 0                                   { inițializarea numărului n }
  cât timp mai există cifre execută:
    citeşte cifra                  { cifra curentă de la stânga la dreapta }
    n ← n*10 + cifra                  { introducerea cifrei în numărul n }
  sfârşit cât timp
  scrie n
sfârşit algoritm
```

Acest algoritm poate fi completat astfel încât să se întrerupă în cazul în care a apărut un caracter care nu este cifră sau dacă numărul ar urma să devină „prea mare".

1.3.2. Descompunerea unui număr dat în cifre

Operația „inversă", adică furnizarea cifrelor unui număr este necesară ori de câte ori dorim să realizăm prelucrarea unui număr cifră cu cifră. În algoritmul următor „prelucrarea" va fi afişarea cifrelor.

```
Algoritm Invers:
  citeşte n
  cât timp n ≠ 0 execută:                    { cât timp n mai are cifre }
    cifra ← rest[n/10]                       { cifra curentă de la dreapta la stânga }
    scrie cifra
    n ← [n/10]                               { eliminăm ultima cifră din numărul n }
  sfârşit cât timp
sfârşit algoritm
```

1.3.3. Numere *Fibonacci*

Şirul numerelor care poartă numele celebrului matematician *Fibonacci* are primii 11 termeni: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

Generarea termenilor mai mici sau egali cu un număr natural $n > 1$ dat ai acestui şir este o problemă simplă şi se realizează cu următorul algoritm:

```
Algoritm Fibonacci:
  citeşte n                                  { valoarea cea mai mare a unui termen }
  a ← 1                                                    { primul termen }
  b ← 1                                                    { al doilea termen }
  c ← a + b                                                { al treilea termen }
  scrie a, b                                 { deoarece n >1, primii doi se pot scrie }
  cât timp c ≤ n execută:                    { cât timp suma celor doi termeni anteriori }
    scrie c                                                { satisface cerința, o scriem }
    a ← b                                                  { noul a va fi vechiul b }
    b ← c                                                  { noul b va fi vechiul c }
    c ← a + b                                              { calculăm noul termen }
  sfârşit cât timp
sfârşit algoritm
```

1.3.4. Numere prime

Pentru a verifica dacă un număr dat este prim sau nu, am putea porni de la definiția numărului prim (un număr prim are exact doi divizori: 1 şi el însuşi). În acest caz ar trebui să verificăm dacă numărul dat are vreun divizor printre numerele 2, 3, ..., $n - 1$.

Algoritmul poate fi îmbunătățit considerabil dacă tratăm separat cazul acelor numere care nu necesită nici o împărțire pentru stabilirea proprietății. Acestea sunt numărul 1 şi numerele pare. În rest, proprietatea de divizibilitate o vom verifica doar cu numere impare, ştiind că un număr impar nu poate avea nici un divizor par. În plus, nu vom genera divizori mai mari decât rădăcina pătrată a numărului.

```
Algoritm Prim:
  citeşte nr
```

```
  dacă nr = 1 atunci
    prim ← fals                                          { 1 nu este prim }
  altfel
    dacă nr este număr par atunci                        { numere pare }
      prim ← nr=2                                { singurul număr par prim este 2 }
    altfel                                               { numere impare }
      diviz ← 3                  { primul divizor posibil al unui număr impar }
      rad ← parte întreagă din rădăcina pătrată a lui nr
                { calculăm radicalul o singură dată în afara structurii repetitive }
      prim ← adevărat                    { presupunem că numărul este prim }
      cât timp (diviz ≤ rad) şi prim execută:
        dacă rest[nr/diviz]=0 atunci
          prim ← fals              { am găsit un divizor, deci nr nu este prim }
        altfel
          diviz ← diviz + 2
                     { un număr impar se poate divide doar cu numere impare }
        sfârşit dacă
      sfârşit cât timp
    sfârşit dacă
  sfârşit dacă
  afişare
sfârşit algoritm
```

1.4. Conversii între două baze de numerație

1.4.1. Conversia unui număr dintr-o bază dată în baza 10

Efectuând calculele cerute de descompunerea după puterile bazei, putem obține în cazul oricărui sistem de numerație valoare numărului dat în baza 10.

Pentru a converti un număr fracționar dintr-o bază *b* în baza 10 aplicăm același algoritm ca și pentru partea întreagă: dezvoltăm numărul cifră cu cifră după puterile bazei, apoi efectuăm calculele în baza 10.

1.4.2. Conversia din baza 10 într-o altă bază de numerație

Conversia unui număr natural reprezentat în baza 10 într-o altă bază de numerație *b* se face prin împărțiri succesive la *b*, valoarea numărului fiind înlocuită la fiecare pas prin câtul împărțirii. Resturile scrise în ordine inversă reprezintă cifrele numărului căutat.

Conversia unui număr fracționar reprezentat în baza 10 într-o bază *b* se face prin înmulțiri succesive cu *b*. La fiecare pas, partea întreagă a rezultatului înmulțirii ne dă câte o cifră din rezultat, înmulțirea continuând cu partea fracționară a rezultatului. Algoritmul va avea atâția pași câte cifre zecimale dorim să determinăm.

1.5. Tablouri unidimensionale

În diferitele medii de programare, *definirea unui tip tablou* înseamnă precizarea *mulțimii* din care acesta poate lua valori (*tipul de bază*), precum şi a *operațiilor* care se pot efectua asupra unei variabile de tip *tablou, la nivel global*.

Prin atribuire, o variabilă de tip tablou primeşte valoarea unei alte variabile de acelaşi tip tablou. Posibilitatea efectuării atribuirilor la nivelul unui întreg tablou reprezintă o facilitate importantă oferită de anumite limbaje de programare (de exemplu, Pascal). Această atribuire se poate realiza numai dacă tipul celor două tablouri este *identic*.

O componentă a tabloului se precizează prin specificarea variabilei de tip tablou, urmată între paranteze drepte de o expresie indiceală.

1.5.1. Declararea unui tablou în Pascal

Forma generală a unei declarații de tablou unidimensional este următoarea:

```
type nume_tip=array[tip_indice] of tip_de_bază;
var nume_tablou:nume_tip;
```

unde *nume_tip* este un identificator ales de utilizator, *tip_de_bază* (poate fi orice tip elementar sau structurat, cunoscut în momentul declarației) şi *tip_indice* (ordinal) este un tip predefinit sau tip utilizator.

Există posibilitatea de a declara un tablou descriind structura în secțiunea **var**, introducând astfel un tip *anonim*. Reamintim că două tipuri anonime se consideră diferite, chiar dacă, de fapt, descriu aceeaşi structură de date.

1.5.2. Citirea/afişarea tablourilor unidimensionale

O variabilă de tip tablou nu poate fi citită şi nu poate fi afişată în Pascal; aceste operații se realizează citind/afişând tabloul element cu element:

```
Algoritm CitireTablou:
  citeşte n                                  { citim dimensiunea tabloului }
  pentru i=1,n execută:
    citeşte x[i]                                    { citim elementele }
  sfârşit pentru
sfârşit algoritm

Algoritm AfişareTablou:
  pentru i=1,n execută:
    scrie x[i]
  sfârşit pentru
sfârşit algoritm
```

1.5.3. Prelucrarea şirurilor

A. Decizia

Vom întâlni probleme în care se cere verificarea unei proprietăţi a şirului. Aceasta poate să vizeze existenţa unui singur element având proprietatea respectivă sau o proprietate globală a şirului. Trăsătura comună a acestor probleme constă în felul în care algoritmul va furniza răspunsul. Soluţia va fi dată sub forma unui mesaj prin care se confirmă sau se infirmă proprietatea cerută. Acest mesaj poate fi simplu: `'DA'` sau `'NU'` şi se afişează în funcţie de valoarea unei variabile logice, stabilită în algoritm.

În următorul algoritm contorul *i* reprezintă indici în şirul *x*, iar variabila *găsit* nu va fi iniţializată; ea primeşte valoare în funcţie de motivul părăsirii ciclului **cât timp**. Dacă există $i \le n$ pentru care x_i are proprietatea cerută, *găsit* primeşte valoarea *adevărat*; dacă nici un element nu are proprietatea căutată, valoarea lui *i* este mai mare decât *n* şi *găsit* primeşte valoarea *fals*.

```
Algoritm Decizie_1:
  citire date
  i ← 1
  cât timp (i ≤ n) şi (x[i] nu are proprietatea căutată) execută:
    i ← i + 1
  sfârşit cât timp
  găsit ← i ≤ n                              { se evaluează expresia relaţională }
                          { şi valoarea de adevăr obţinută se atribuie variabilei găsit }
  afişare rezultat
sfârşit algoritm
```

Dacă se cere ca fiecare element al şirului să aibă o aceeaşi proprietate, adică trebuie să verificăm că în şir nu există nici un element care *nu* are proprietatea cerută, transformăm algoritmul `Decizie_1` prin aplicarea negaţiei în două locuri.

```
Algoritm Decizie_2:
  citire date
  i ← 1
  cât timp (i ≤ n) şi (x[i] are proprietatea căutată) execută:
                        { am negat subexpresia: (x[i] nu are proprietatea căutată) }
    i ← i + 1
  sfârşit cât timp
  toate ← i > n                                  { am negat subexpresia i ≤ n }
  afişare rezultat
sfârşit algoritm
```

B. Selecția

Să presupunem că enunțul garantează că în șirul dat există cel puțin un element având proprietatea dată. Deci, trebuie să găsim acel element şi să stabilim numărul său de ordine (*nr_ord*). Primul număr de ordine (indice) pentru care proprietatea este adevărată, constituie rezultatul algoritmului.

```
Algoritm Selecția:
  citire date
  nr_ord ← 1
  cât timp (x[nr_ord] nu are proprietatea căutată) execută:
    nr_ord ← nr_ord + 1
  sfârşit cât timp
  afişare rezultat
sfârşit algoritm
```

C. Numărarea

Trebuie să numărăm elementele din şir având o anumită proprietate. Enunțurile nu garantează că există cel puțin un astfel de element, deci soluția furnizată poate fi şi 0.

Având în vedere că trebuie verificat fiecare element, vom lucra cu o structură repetitivă de tipul **pentru**. Contorul cu care numărăm elementele având proprietatea dată este notat cu *buc*.

```
Algoritm Numărare:
  citire date
  buc ← 0
  pentru i=1,n execută:
    dacă x[i] are proprietatea căutată atunci
      buc ← buc + 1
    sfârşit dacă
  sfârşit pentru
  afişare rezultat
sfârşit algoritm
```

D. Selectarea elementelor

Dacă dorim să reținem toate pozițiile pe care se află elemente având o anumită proprietate, vom aplica un algoritm de tip `Selectare`. Dacă şirul pozițiilor ne este necesar în continuare în rezolvarea problemei, vom crea un tablou *y* în care vom reține aceste poziții. Contorizarea elementelor din acesta o realizăm cu variabila *buc*.

```
Algoritm Selectare_1:
  citire date
  buc ← 0
```

```
  pentru i=1,n execută:
    dacă x[i] are proprietatea căutată atunci
      buc ← buc + 1
      y[buc] ← i  { în y păstrăm indicii elementelor x[i], având proprietatea }
    sfârşit dacă                                                    { căutată }
  sfârşit pentru
  afişare rezultate
sfârşit algoritm
```

Pentru păstrarea rezultatului am avut nevoie de un nou tablou în cazul căruia nu cunoaştem exact câte elemente va cuprinde, deci va fi declarat având atâtea elemente câte are şirul dat. Modelul de mai sus „colecţionează" în vectorul *y* indicii elementelor având proprietatea precizată, şi totodată obţine şi numărul acestora în *buc*. Dacă nu avem nevoie de elementele respective, colecţionate în vectorul *y*, deoarece trebuie doar să le furnizăm ca rezultat, algoritmul este şi mai simplu:

```
Algoritm Selectare_2:
  citire date
  pentru i=1,n execută:
    dacă x[i] are proprietatea căutată atunci scrie x[i]
    sfârşit dacă
  sfârşit pentru
sfârşit algoritm
```

În a treia variantă a acestui algoritm considerăm că după selectarea elementelor căutate nu mai avem nevoie de şirul dat. Deci, în loc să folosim un al doilea şir pentru elementele selectate, vom suprascrie elementele de la începutul şirului dat cu cele selectate. Variabila *buc* reţine lungimea acestui şir „nou":

```
Algoritm Selectare_3:
  citire date
  buc ← 0
  pentru i=1,n execută:
    dacă x[i] are proprietatea căutată atunci
      buc ← buc +1
      x[buc] ← x[i]                          { sau i, în funcţie de cerinţe }
    sfârşit dacă
  sfârşit pentru
  afişare rezultate
sfârşit algoritm
```

Dacă se cere păstrarea elementelor care *nu* au această proprietate, vom nega condiţia din instrucţiunea **dacă**. Dacă vrem să păstrăm aceste elemente pe poziţiile origina-

le în şirul dat, putem anula aceste elemente, suprascriindu-le cu o valoare specială. Astfel, vom avea un algoritm care realizează o *selectare pe loc*. Această soluţie va fi avantajoasă doar dacă „evitarea" prelucrării acestor elemente, având valoarea specială, va fi mai simplu de realizat decât verificarea proprietăţii în sine.

```
Algoritm Selectare_4:
  citire date
  pentru i=1,n execută:
    dacă x[i] are proprietatea căutată atunci
      x[i] ← valoare_specială          { o valoare convenţional stabilită }
    sfârşit dacă
  sfârşit pentru
  afişare rezultate
sfârşit algoritm
```

E. Partiţionarea unui şir în două subşiruri

Vor fi probleme în care trebuie să descompunem un şir în două sau mai multe subşiruri. Mai întâi vom selecta acele elemente care au prima proprietate, apoi din şirul elementelor puse deoparte (rămase după selectare) selectăm elementele având cea de a doua proprietate şi aşa mai departe.

Acest tip de problemă se poate rezolva pe baza mai multor modele. În prima variantă elementele având proprietatea cerută şi cele care nu au această proprietate le vom aşeza în două şiruri noi. Aceste şiruri le vom declara având dimensiunea şirului dat, deoarece nu se poate anticipa numărul exact de elemente.

În algoritmul următor variabilele *bucy* şi *bucz* reprezintă numărul elementelor aşezate în şirul *y*, respectiv şirul *z*, şiruri noi, rezultate în urma partiţionării.

```
Algoritm Partiţionare_1:
  citire date
  bucy ← 0
  bucz ← 0
  pentru i=1,n execută:
    dacă x[i] are proprietatea căutată atunci
      bucy ← bucy + 1              { elementele care au proprietatea dată }
      y[bucy] ← x[i]                                  { le aşezăm în şirul y }
    altfel
      bucz ← bucz + 1           { elementele care nu au proprietatea dată }
      z[bucz] ← x[i]                                  { le aşezăm în şirul z }
    sfârşit dacă
  sfârşit pentru
  afişare rezultate
sfârşit algoritm
```

Dacă vrem să economisim spațiul risipit cu o astfel de soluție, putem utiliza un singur şir. Elementele selecționate le reținem în acesta, aşezându-le de la început spre sfârşit, iar cele rămase de la ultimul element spre primul. Evident, nu există pericolul să „ne ciocnim", deoarece avem exact *n* elemente care trebuie aşezate pe *n* locuri.

Să observăm că această rezolvare ne furnizează şirul elementelor neselectate în ordine inversă față de pozițiile elementelor sale în şirul dat.

```
Algoritm Partiționare_2:
  citire date
  bucy ← 0
  bucz ← 0
  pentru i=1,n execută:
    dacă x[i] are proprietatea căutată atunci
      bucy ← bucy + 1                    { elementele care au proprietatea dată }
      y[bucy] ← x[i]            { le aşezăm în şirul y, începând cu prima poziție }
    altfel                                { elementele care nu au proprietatea dată }
      bucz ← bucz + 1
      y[n-bucy+1] ← x[i]      { le aşezăm tot în y, începând cu ultima poziție }
    sfârşit dacă
  sfârşit pentru
  afişare rezultate
sfârşit algoritm
```

În cazul în care după partiționare nu avem nevoie de şirul dat în forma sa originală, partiționarea, exact ca selectarea, poate fi realizată folosind spațiul alocat şirului dat, adică *pe loc*.

Punem primul element deoparte, păstrându-i valoarea într-o variabilă auxiliară, şi căutăm un element având proprietatea cerută, pornind de la capătul din dreapta al şirului şi îl punem pe locul eliberat la începutul şirului. Acum vom căuta, pornind de la această poziție spre dreapta, un element care nu are proprietatea cerută şi îl aşezăm pe locul eliberat de la sfârşitul şirului. Continuăm acest procedeu până când, în urma avansării în cele două direcții, ne vom întâlni.

În algoritm variabila *start* reține indicele curent al elementului care are proprietatea cerută, parcurgând şirul de la stânga spre dreapta, iar *stop* reține indicele curent al elementului care *nu* are proprietatea cerută, parcurgând şirul de la dreapta spre stânga.

```
Algoritm Partiționare_3:
  citire date
  start ← 1
  stop ← n
  aux ← x[start]
```

```
  cât timp start < stop
    cât timp (start < stop) şi (x[stop] nu are proprietatea căutată):
      stop ← stop - 1
    sfârşit cât timp
    dacă start < stop atunci
      x[start] ← x[stop]
      start ← start + 1
      cât timp (start < stop) şi (x[start] are proprietatea căutată):
        start ← start + 1
      sfârşit cât timp
      dacă start < stop atunci
        x[stop] ← x[start]
        stop ← stop - 1
      sfârşit dacă
    sfârşit dacă
  sfârşit cât timp
  x[start] ← aux
  dacă x[start] are proprietatea căutată atunci buc ← start
                                        altfel buc ← start - 1
  sfârşit dacă
  afişare rezultate
sfârşit algoritm
```

1.5.5. Mulţimi reprezentate cu şiruri

Se ştie că o mulțime fie este vidă, fie conține un număr oarecare de obiecte. În acest sens, dacă o mulțime este reprezentată cu ajutorul unui şir, elementele sale trebuie să fie distincte (un acelaşi „obiect” este prezent o singură dată).

A. Stabilirea proprietății de „mulțime”

```
Algoritm Mulţime_1:
  citire date
  i ← 1
  mulţime ← adevărat
  cât timp mulţime şi (i < n) execută:
    j ← i + 1
    cât timp (j ≤ n) şi (x[i] ≠ x[j]) execută:
      j ← j + 1
    sfârşit cât timp
    dacă j > n atunci
      mulţime ← adevărat
```

```
      altfel
        mulțime ← adevărat
    sfârşit dacă
    i ← i + 1
  sfârşit cât timp
  afişare rezultate
sfârşit algoritm
```

B. Eliminarea dublurilor

Transformarea unui şir, astfel încât să devină mulțime, înseamnă eliminarea dublurilor. Trebuie să fim atenți, ca după depistarea şi eliminarea unei dubluri x_j al lui x_i, să nu avansăm cu i, ci să efectuăm încă o căutare de dubluri pentru acelaşi x_i. După fiecare eliminare de un element, lungimea şirului va scădea cu 1.

```
Algoritm Mulțime_2:
  citire date
  i ← 1
  cât timp i < n execută:
    j ← i + 1
    cât timp (j ≤ n) şi (x[i] ≠ x[j]) execută:
      j ← j + 1
    sfârşit cât timp
    dacă j ≤ n atunci
      x[j] ← x[n]              { suprascriem dublura cu ultimul element }
      n ← n - 1                                          { scurtăm şirul }
    altfel
      i ← i + 1
    sfârşit dacă
  sfârşit cât timp
  afişare rezultate
sfârşit algoritm
```

C. Intersecția

Prin *intersecție* înțelegem şirul care conține elementele comune tuturor şirurilor date. Avem, deci o problemă în care se impune selectarea anumitor elemente pe baza proprietății de a fi prezente în fiecare şir dat.

Evident, în diverse limbaje de programare această problemă se rezolvă în funcție de instrumentele pe care acesta le pune la dispoziția programatorilor. De exemplu, în Pascal există tipul **set** care poate simplifica mult efectuarea acestei operații, dacă tipul de bază şi domeniul valorilor elementelor permit declararea datelor ca fiind de acest tip. Acum tratăm problema enunțată în condiții generale, independent de limba-

jul de programare ales, doar din punct de vedere algoritmic. De asemenea, presupunem că fiecare şir conţine elemente *distincte* şi că aceste şiruri *nu sunt ordonate*.

Prezentăm un algoritm general care stabileşte subşirul acelor elemente din şirul *x* care se găsesc şi în şirul *y*. Dimensiunile şirurilor sunt *n*, respectiv *m*, iar dimensiunea şirului intersecţie *z* va fi *buc*.

```
Algoritm Intersecţie:
  citire date
  buc ← 0
  pentru i=1,n execută:
  j ← 1
    cât timp (j ≤ m) şi (x[i] ≠ y[j]) execută:
      j ← j + 1
    sfârşit cât timp
    dacă j ≤ m atunci
      buc ← buc + 1
      z[buc] ← x[i]
    sfârşit dacă
  sfârşit pentru
  afişare rezultate
sfârşit algoritm
```

D. Reuniunea

Dacă vrem să rezolvăm probleme care prelucrează mulţimi implementate cu ajutorul şirurilor, trebuie să tratăm şi problema reuniunii.

Semnificaţia variabilelor în algoritmul următor este acelaşi ca mai înainte. Mai întâi vom copia toate elementele din şirul *x* în şirul *z*, iniţializând lungimea *buc* cu lungimea acestui şir. Apoi, vom selecta din şirul *y* acele elemente care nu se află în *x*.

```
Algoritm Reuniune:
  citire date
  z ← x
  buc ← n
  pentru j=1,m execută:
    i ← 1
    cât timp (i ≤ n) şi (x[i] ≠ y[j]) execută:
      i ← i + 1
    sfârşit cât timp
    dacă i > n atunci
      buc ← buc + 1
```

```
      z[buc] ← y[j]
    sfârşit dacă
  sfârşit pentru
  afişare rezultate
sfârşit algoritm
```

1.6. Subprograme

Noţiunea de subprogram a apărut din necesitatea de a diminua complexitatea unui program, şi anume prin descompunerea acestuia în subprograme.

În proiectarea subprogramelor se va urmări creşterea independenţei acestora pentru a mări şansele reutilizării lor în cât mai multe programe.

Datele care asigură comunicarea între subprogram şi modulul apelant sunt definite într-o interfaţă a subprogramului cu exteriorul care se materializează în *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 execuţia ultimei instrucţiuni a subprogramului.

Datele definite în programul principal se numesc *globale* şi vor fi „văzute" (cu excepţia celor redeclarate în subprogramul respectiv) în toate subprogramele.

Lista parametrilor formali este declarată în antetul subprogramului şi conţine *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 referinţă*.

1.6.1. Declararea şi apelul unui subprogram

Structura unui subprogram seamănă cu structura unui program:

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

unde prin *Tip_Subprogram* înţelegem precizarea tipului subprogramului (în Pascal *funcţie* sau *procedură*). *Corpul* subprogramului este format din declaraţiile locale ale sale şi o instrucţiune compusă.

Apelul subprogramului are loc în partea de acţiuni a modulului apelant. Dacă subprogramul returnează o singură valoare (avem subprogram de tip *funcţie*), apelul poate să apară într-o expresie, oriunde sintaxa limbajului permite prezenţa unei expresii. Dacă un subprogram calculează mai multe valori (în Pascal avem subprogram de tip *procedură*), apelul subprogramului are loc în *instrucţiunea procedurală* (*apel*).

În principiu, un *apel* se scrie precizând numele subprogramului, urmat, între paran-

teze, de lista *parametrilor actuali*. Efectul apelului constă în activarea subprogramului şi transferul execuţiei programului la prima instrucţiune 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 execuţie*), iar corespunzător parametrilor transmişi prin referinţă se comunică adresele parametrilor actuali. *Orice modificare de valoare asupra parametrului formal de tip referinţă (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 referinţă trebuie să fie o variabilă.

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

1.6.2. Funcţii şi proceduri (în Pascal)

În limbajul Pascal subprogramele pot fi:

- ***funcţii***: calculează mai multe valori; una se returnează prin numele funcţiei;
- ***proceduri***: calculează mai multe valori sau nici una.

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

Declararea unei funcţii în limbajul Pascal
`function` *Nume_funcţie* (*lista parametrilor formali*) **:** *tip_rezultat;*
declaraţiile şi definiţiile locale funcţiei
instrucţiune compusă

Printre instrucţiunile din cadrul instrucţiunii compuse a funcţiei (corpul funcţiei) trebuie să existe cel puţin una de atribuire prin care numelui funcţiei i se atribuie o valoare.

Declararea unei proceduri în limbajul Pascal
`procedure` *Nume_Procedură* (*listă parametrilor formali*) ;
declaraţii şi definiţii locale procedurii
instrucţiune compusă

În limbajul Pascal în faţa oricărui parametru formal transmis prin referinţă se scrie cuvântul cheie **`var`**. Parametrii formali care nu au cuvântul **`var`** în faţă se consideră a fi parametri transmişi prin valoare. Tipul parametrilor formali se poate preciza *doar cu identificator de tip*. Valoarea returnată de o funcţie (ţinând cont de faptul că aceasta returnează prin identificatorul său *o singură* valoare) poate fi de orice tip întreg, orice tip real, caracter, boolean, enumerare (având definit identificator de tip), subdomeniu (având definit identificator de tip), **`string`** sau pointer[*)].

[*)] Vezi capitolul 9.

1.6.3. 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 obţin 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 iniţiale, acestea fiind folosite pentru a construi module din ce în ce mai sofisticate care, asamblate să rezolve problema, se numeşte ***bottom up***.

A. 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 se testează şi se controlează mai uşor. Fiecare subprogram va putea apela numai subprograme declarate anterior propriei declaraţii. Din această restricţie de vizibilitate decurge şi dezavantajul acestui stil de lucru care constă în dependenţa subprogramelor de unele subprograme definite anterior.

B. 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 de vizibilitate ca între unitatea de program principal şi subprogram.

1.7. Ordonări şi căutări

Fie $A = (a_1, a_2,...,a_n)$ un tablou unidimensional cu n elemente. A *ordona* (sorta) tabloul înseamnă a ordona crescător sau descrescător elementele tabloului.

1.7.1. Metoda bulelor

Să presupunem că dorim o ordonare crescătoarea şirului. Algoritmul constă în parcurgerea tabloului A de mai multe ori, până când devine ordonat. La fiecare pas se compară două elemente alăturate. Dacă $a_i > a_{i+1}$, ($i = 1, 2, ..., n - 1$), atunci cele două valori se interschimbă între ele. Controlul acţiunii repetitive este dat de variabila booleană *ok*, care la fiecare reluare a algoritmului primeşte valoarea iniţială *adevărat*, care se schimbă în *fals*, dacă s-a efectuat o interschimbare de două elemente alăturate. În momentul în care tabloul A s-a parcurs fără să se mai efectueze nici o schimbare, *ok* rămâne cu valoarea iniţială *adevărat* şi algoritmul se termină, deoarece tabloul este ordonat.

Interschimbarea a două elemente se realizează prin intermediul variabilei auxiliare *aux* care are acelaşi tip ca şi elementele tabloului.

```
Subalgoritm Metoda_bulelor(a,n):
  repetă
    ok ← adevărat
    pentru i=1,n-1 execută:
      dacă a[i] > a[i+1] atunci
        ok ← fals
        aux ← a[i]
        a[i] ← a[i+1]
        a[i+1] ← aux
      sfârşit dacă
    sfârşit pentru
  până când ok
sfârşit subalgoritm
```

Deoarece la fiecare parcurgere ajunge cel puțin un element pe locul său definitiv în şirul ordonat, la următorul pas nu mai sunt necesare verificările în care intervine acest element şi cele care se află după el în şir. Rezultă că la fiecare parcurgere am putea micşora cu 1 numărul elementelor verificate. Dar este posibil ca la o parcurgere să ajungă mai multe elemente în locul lor definitiv. Dacă ținem minte indicele ultimului element, respectiv primului element care au intervenit în interschimbări şi efectuăm verificările doar între aceste elemente, obținem un subalgoritm îmbunătățit ca performanță.

1.7.2. Sortare stabilind poziția definitivă prin numărare

Această metodă constă în construirea unui tablou nou *B*, având aceeaşi dimensiune ca şi tabloul *A*, în care depunem elementele din *A*, ordonate crescător.

Vom analiza fiecare element comparând-le cu fiecare alt element din şir pentru a reține în variabila *k* numărul elementelor care sunt mai mici decât elementul considerat. Astfel, vom afla poziția pe care trebuie să-l punem pe acesta în şirul *B*. Dacă în problemă avem nevoie de şirul ordonat în tabloul *A*, vom copia în *A* întreg tabloul *B*.

```
Subalgoritm Numărare(n,a):
  pentru i=1,n execută:
    k ← 0
    pentru j=1,n execută:
      dacă (a[i] ≥ a[j]) şi (i ≠ j) atunci
        k ← k + 1  { numărăm câte elemente sunt mai mici sau egale cu a[i] }
      sfârşit dacă
    sfârşit pentru
    b[k+1] ← a[i]              { pe următoarea poziție îl punem pe a[i] în b}
  sfârşit pentru
  a ← b                                  { copiem peste şirul a întreg şirul b }
sfârşit subalgoritm
```

1.7.3. Sortare prin selecție directă

Metoda precedentă are dezavantajul că necesită de două ori mai multă memorie decât tabloul *A*. Dacă dorim să evităm această risipă, putem aplica metoda de ordonare prin selectarea unui element şi plasarea lui pe poziția sa finală.

De exemplu, în caz de ordonare crescătoare, pornind de la primul element vom căuta valoarea minimă din tablou. Aceasta o aşezăm pe prima poziție printr-o interschimbare între elementul de pe prima poziție şi elementul minim. Reluăm algoritmul, pornind de la a doua poziție şi căutând minimul între elementele a_2, ..., a_n. Acesta îl interschimbăm cu al doilea, dacă este cazul. Continuăm procedeul până la ultimul element.

```
Subalgoritm Selecție(n,a):
  pentru i=1,n-1 execută:
    min ← a[i]
    pentru j=i+1,n execută:
      dacă min > a[j] atunci
        min ← a[j]
        k ← j
      sfârşit dacă
      dacă min ≠ a[i] atunci
        aux ← a[i]
        a[i] ← a[k]
        a[k] ← aux
      sfârşit dacă
    sfârşit pentru
  sfârşit pentru
sfârşit subalgoritm
```

1.7.4. Sortarea prin inserție directă

Ideea algoritmului este de a insera fiecare element pe locul său în subtabloul ordonat până la momentul respectiv. Acest loc s-ar putea să nu fie definitiv, deoarece este posibil ca următorul element să se insereze undeva în fața acestuia.

Dacă am ajuns la elementul a_j, (j = 1, 2, ..., n) şi elementele a_1, a_2, ..., a_{j-1} au fost ordonate, înseamnă că trebuie să-l inserăm pe a_j între acestea. Îl comparăm pe rând cu a_{j-1}, a_{j-2}, ... până când ajungem la un prim element a_i având proprietatea $a_i < a_j$. Prin urmare a_j se inserează după a_i. Această inserare provoacă deplasarea elementelor care îl succed pe a_i cu o poziție spre dreapta.

```
Subalgoritm Inserție(n,a):
  pentru i=2,n execută:
    copie ← a[i]  { salvăm al i-lea element (s-ar putea pierde cu translatările) }
    p ← 0                          { în p vom avea indicele unde îl vom pune }
```

```
    j ← i - 1          { vom căuta locul printre elementele din fața lui a[i] }
    găsit ← fals
    cât timp (j ≥ 1) şi nu găsit execută:
      dacă a[j] ≤ copie atunci
        p ← j              { dacă a[j] ≤ a[i], aici trebuie sa-l punem }
        găsit ← adevărat                                  { am găsit locul }
      altfel
        a[j+1] ← a[j] { până nu am găsit locul, mutăm elementele la dreapta }
      sfârşit dacă
      j ← j - 1
    sfârşit cât timp
    a[p+1] ← copie                  { îl punem pe a[i] pe locul „eliberat" }
  sfârşit pentru
sfârşit subalgoritm
```

1.7.5. Sortare prin numărarea aparițiilor

Algoritmii prezentați anterior sunt relativ mari consumatori de timp.

În comparație cu aceştia, dacă avem un şir de elemente de tip ordinal, care sunt dintr-un interval de cardinalitate nu foarte mare, vom putea realiza o *ordonare liniară*. Corespunzător fiecărei valori întâlnite în şir, în timpul prelucrării mărim cu 1 valoarea elementului având indicele (în şirul de contoare) egal cu valoarea elementului în şirul dat. În final, vom suprascrie în şirul dat atâtea elemente cu valori ai indicilor elementelor diferite de 0 cât este valoarea elementului în acest şir a numărului de apariții.

Este important să reținem particularitățile pe care trebuie să le aibă şirul dat pentru ca această metodă să se poată aplica:

- valorile elementelor trebuie să fie de tip ordinal;
- numărul elementelor mulțimii din care şirul primeşte valori trebuie să fie relativ mic (dacă în program nu există alte structuri de date, din cei 64KB ai memoriei, scădem spațiul necesar şirului de ordonat şi putem calcula spațiul pe care îl avem la dispoziție);
- valorile posibile în şirul dat trebuie să fie din intervalul [*x*..*y*], unde $y - x + 1$ va fi dimensiunea şirului de contoare.

Exemplu

```
var sir:array[1..20000] of Integer;          { cel mult 20000 de numere }
    frecv:array[-500..500] of Integer; { valori posibile între -500 şi 500 }

Subalgoritm Ordonare_cu_Şir_de_Frecvențe(n,a,x,y):
  pentru i=x,y execută:
    frecv[i] ← 0
  sfârşit pentru
```

```
  pentru i=1,n execută:
    frecv[a[i]] ← frecv[a[i]] + 1
  sfârşit pentru
  k ← 0
  pentru i=x,y execută:
    pentru j=1,frecv[i] execută:
      k ← k + 1
      a[k] ← i
    sfârşit pentru
  sfârşit pentru
sfârşit subalgoritm
```

1.7.6. Căutare binară

Dacă tabloul A este sortat în ordine crescătoare, atunci se poate realiza o căutare mai rapidă decât cea secvențială.

Presupunem că avem $a_1 \le a_2 \le ... \le a_{m-1} \le a_m \le a_{m+1}, ... \le a_n$, unde a_m reprezintă elementul din mijloc, ($m = [(n + 1)/2]$). Fie valoarea căutată p. Notăm cu s extremitatea stângă şi cu d extremitatea dreaptă a intervalului în care efectuăm căutarea. Inițial $s = 1$ şi $d = n$.

Comparăm valoarea p cu a_m. Dacă $p = a_m$, atunci căutarea se încheie cu succes. Dacă $p < a_m$, atunci căutarea se va continua în prima jumătate a şirului, iar dacă $p > a_m$, atunci căutarea se transferă în a doua jumătate a tabloului. Algoritmul se continuă până când fie la un moment dat $a_m = p$, ceea ce înseamnă că am încheiat căutarea cu succes, fie ajungem la un subşir curent vid, adică având marginea din stânga mai mare decât marginea din dreapta ($s > d$), ceea ce înseamnă că avem o căutare fără succes.

```
Subalgoritm Căutare_binară(n,a,p):
  s ← 1
  d ← n
  este ← fals
  cât timp s ≤ d şi nu este execută
    m ← [(s+d)/2]
    dacă p = a[m] atunci
      este ← adevărat
    altfel
      dacă p < a[m] atunci
        d ← m - 1
      altfel
        s ← m + 1
      sfârşit dacă
    sfârşit dacă
  sfârşit cât timp
```

```
  dacă este atunci
    scrie 'Valoarea ',p,' se afla pe pozitia ',m
  altfel
    scrie 'Valoarea ',p,' nu se afla in sir.'
  sfârşit dacă
sfârşit subalgoritm
```

1.7.7. Interclasarea a două tablouri unidimensionale

Am văzut cum procedăm în cazul în care dorim să creăm un şir nou din două şiruri date, astfel încât fiecare valoare să apară o singură dată, simulând mulţimi. În cazul în care şirurile sunt ordonate, problema „intersecţiei" şi „reuniunii" se prezintă altfel.

Fie $A = (a_1, a_2, ..., a_n)$ cu elementele $a_1 \le a_2 ... \le a_n$ şi $B = (b_1, b_2, ..., b_m)$ cu elementele $b_1 \le b_2 \le ... \le b_m$. Se cere să se construiască tabloul $C = (c_1, c_2, ..., c_{n+m})$ cu elementele $c_1 \le c_2 \le ... \le c_{n+m}$ care să conţină toate elementele lui A şi B.

În rezolvare vom avansa în paralel în cele două şiruri date. Primul element în şirul nou va fi fie a_1, fie b_1. Dacă a_1 este mai mic decât b_1 atunci pe acesta îl „consumăm" din şirul a şi pregătim indicele elementului următor din acesta. În caz alternativ, procedăm la fel cu indicele j, după ce aşezăm elementul b_1 în şirul c. Vom efectua aceşti paşi până când careva din şiruri se sfârşeşte, moment în care copiem elementele rămase din celălalt şir în şirul nou. Din moment ce nu putem şti care şir s-a „consumat" integral, vom scrie secvenţa respectivă pentru ambele şiruri.

```
Subalgoritm Interclasare1(n,a,m,b,buc,c):
  buc ← 0
  i ← 1
  j ← 1
  cât timp (i ≤ n) şi (j ≤ m) execută:
    buc ← buc + 1
    dacă a[i] < b[j] atunci
      c[buc] ← a[i]
      i ← i + 1
    altfel
      c[buc] ← b[j]
      j ← j + 1
    sfârşit dacă
  sfârşit cât timp
  cât timp i ≤ n execută:
    buc ← buc + 1
    c[buc] ← a[i]
    i ← i + 1
  sfârşit cât timp
```

```
  cât timp j ≤ m execută:
    buc ← buc + 1
    c[buc] ← b[j]
    j ← j + 1
  sfârşit cât timp
sfârşit subalgoritm
```

Putem aşeza două elemente fictive (numite frecvent *santinele*) după ultimele elemente din cele două şiruri. Vom fi atenţi să aibă valorile mai mari decât ultimul (cel mai mare) din celălalt şir (în subalgoritmul următor l-am notat cu `infinit`).

```
Subalgoritm Interclasare2(n,a,m,b,buc,c):
  buc ← 0
  i ← 1
  j ← 1
  a[n+1] ← infinit
  b[m+1] ← infinit
  cât timp (i < n+1) sau (j < m+1) execută:
    buc ← buc + 1
    dacă a[i] < b[j] atunci
      c[buc] ← a[i]
      i ← i + 1
    altfel
      c[buc] ← b[j]
      j ← j + 1
    sfârşit dacă
  sfârşit cât timp
sfârşit subalgoritm
```

Deoarece se ştie că numărul elementelor în şirul *c* este $n + m$, (se admit şi elemente identice în cele două şiruri date) structura repetitivă utilizată va fi de tip **pentru**.

```
Subalgoritm Interclasare3(n,a,m,b,buc,c):
  i ← 1
  j ← 1
  a[n+1] ← infinit
  b[m+1] ← infinit
  pentru buc=1,n+m execută:
    dacă a[i] < b[j] atunci
      c[buc] ← a[i]
      i ← i + 1
```

```
    altfel
      c[buc] ← b[j]
      j ← j + 1
    sfârşit dacă
  sfârşit pentru
sfârşit subalgoritm
```

1.8. Şiruri de caractere

Tipul *şir de caractere* se identifică în Pascal prin termenul *string*, deoarece în acest limbaj de programare există tipul predefinit **`string`** pentru tablouri unidimensionale care au elemente câte un caracter. O variabilă de tip **`string`** are implicit cel mult 255 de elemente. Dacă la declarare se specifică pentru dimensiunea maximă a unei variabile de tip **`string`** un număr mai mic decât 255, va fi valabilă declaraţia făcută explicit.

1.8.1. Operaţii cu variabile de tip `string`

A. Citirea

O variabilă de tip **`string`** se poate citi în Pascal, specificându-i numele. În momentul în care am apăsat **Enter**, sau, în caz de citire dintr-un fişier de tip `Text`, atunci când urmează caracterul sfârşit de linie, *string*-ul „s-a terminat" şi prima componentă a variabilei (cea cu indice 0) primeşte o valoare egală cu numărul caracterelor citite.

B. Afişarea

Afişarea se realizează specificând o expresie de tip **`string`**.

C. Atribuirea

Unei variabile de tip **`string`** i se poate atribui valoarea unei expresii de tip **`string`**. Dacă variabila are un tip care precizează o lungime mai mică decât *string*-ul din expresie, caracterele care depăşesc lungimea variabilei se pierd.

D. Concatenarea

Concatenarea se poate realiza utilizând operatorul `'+'`, dar în Pascal există şi o procedură predefinită în acest scop (`Concat`).

E. Operaţii relaţionale

În cazul expresiilor de tip **`string`** se pot folosi operatorii relaţionali cunoscuţi: <, <=, <>, >, >=, =. Rezultatul a două comparaţii este o constantă logică (*adevărat* sau *fals*).

F. Accesul la o componentă
Componenta `x[i]` reprezintă al `i`-lea caracter din variabila `x`, de tip **`string`**.

1.8.2. Subprograme predefinite pentru *string*-uri

Unit-ul *System* din Borland Pascal 7.0 conține mai multe subprograme predefinite care pot fi utilizate în prelucrarea *string*-urilor. Fie *s* şi *ss* variabile de tip **`string`**.

- Funcția `Pos(ss,s)` returnează valoarea poziției începând de la care se regăseşte *ss* în *s*; dacă *ss* nu se găseşte în *s*, atunci *i* ia valoarea 0.
- Funcția `Copy(s,i,j)` returnează *substring*-ul din *s* care începe la poziția *i* şi are *j* caractere. Dacă în *s* nu există *j* caractere începând cu poziția *i*, se vor returna atâtea câte sunt.
- Funcția `Length(s)` returnează lungimea *stringului* (numărul caracterelor din *s*).
- Procedura `Delete(s,i,j)` are ca efect ştergerea din *s*, începând de la poziția *i*, a *j* caractere. Dacă în *s* nu există *j* caractere începând cu poziția *i*, se vor şterge atâtea câte sunt.
- Procedura `Insert(ss,s,i)` inserează caracterele din *string*-ul *ss* în *s*, începând de la poziția *i*.

În problemele de concurs pot să apară cerințe care impun prelucrarea unor numere întregi „mari” care nu pot fi reprezentate cu toate cifrele lor exacte, folosind tipurile întregi cunoscute de limbajele de programare. Reprezentarea lor ca reale nu este o soluție, deoarece nici în tipurile reale nu putem lucra cu un număr mare de cifre exacte. Dacă trebuie să efectuăm calcule cu aceste numere şi cerințele sunt de aşa manieră încât trebuie determinate toate cifrele exacte ale numărului, trebuie implementată „aritmetica numerelor mari”.

1.9. Tablouri bidimensionale

1.9.1. Operații cu tablouri bidimensionale

A. Declararea tablourilor bidimensionale
Declarația are următoarea formă generală:

```
type TipTab=array[tip_indice1, tip_indice2] of tip_de_bază;
var tablou:TipTab;
```

unde prin tip_indice_1, tip_indice_2 se înțeleg tipurile valorilor din care se alimentează indicii (obligatoriu tipuri ordinale), iar *tip_de_bază* este tipul elementelor tabloului. Acest tip de bază poate fi orice: putem avea elemente de orice tip numeric, de caracte-

re sau şiruri de caractere, valori booleene, înregistrări (vezi tipul **record**), tablouri de cele mai diverse tipuri etc. *tip_indice*$_1$, *tip_indice*$_2$ pot fi identificatori de tip predefinit sau definit de utilizator şi, de asemenea, pot fi expresii, cu mențiunea că în acestea pot interveni doar constante şi constante simbolice.

Dacă structura unui tablou este descrisă în secțiunea **var**, atunci el va avea un tip *anonim*. Asociind tipului respectiv un identificator de tip într-o declarație **type**, acesta va putea fi folosit în program, oriunde vrem să referim acest tip.

Reamintim că două tablouri declarate în două declarații anonime diferite vor fi de tipuri diferite, chiar dacă în ele s-a descris aceeaşi structură.

B. Citirea tablourilor bidimensionale

În programe, de regulă, prima operație va fi citirea unui astfel de tablou. În limbajele Pascal şi C nu se poate citi o variabilă de tip tablou. Această operație se va realiza element cu element.

```
Subalgoritm Citire(n,x):
  citeşte m,n                          { tabloul x are m linii şi n coloane }
  pentru i=1,m execută:
    pentru j=1,n execută:
      citeşte x[i,j]
    sfârşit pentru
  sfârşit pentru
sfârşit subalgoritm
```

C. Afişarea tablourilor bidimensionale

Afişarea tablourilor se realizează asemănător. Dacă dorim să realizăm o afişare linie după linie a tabloului `x` de tipul `tablou`, având `m` linii şi `n` coloane, în Pascal vom scrie procedura în felul următor:

```
procedure Afişare(m,n:Byte; x:tablou);
begin
  for i:=1 to m do begin
    for j:=1 to n do
      Write(x[i,j],' ');
    WriteLn                                { trecem la linie nouă }
  end
end;
```

D. Atribuirea

Atribuirea la nivel de tablou este permisă doar dacă identificatorii menționați în partea stângă şi cea dreaptă a operației de atribuire au acelaşi tip.

1.10. Polinoame

1.10.1. Definirea noțiunii de polinom

Fie mulțimea şirurilor (infinite) de numere complexe $f = (a_0, a_1, a_2, ..., a_n, ...)$, care au numai un număr finit de termeni nenuli, adică există un număr natural m, astfel încât $a_i = 0$ pentru orice $i > m$.

Pe această mulțime se definesc două operații algebrice:

A. Adunarea

$f + g = (a_0 + b_0, a_1 + b_1, a_2 + b_2, ...)$.

B. Înmulțirea

$f \cdot g = (c_0, c_1, c_2, ...)$, unde

$c_0 = a_0 \cdot b_0$

$c_1 = a_0 \cdot b_1 + a_1 \cdot b_0$

$c_2 = a_0 \cdot b_2 + a_1 \cdot b_1 + a_2 \cdot b_0$

...

$a_r = a_0 \cdot b_r + a_1 \cdot b_{r-1} + a_2 \cdot b_{r-2} + ... + a_r \cdot b_0$

Se observă că suma $f + g$ şi produsul $f \cdot g$ aparțin aceleiaşi mulțimi.

Fiecare element al mulțimii definite anterior pe care sunt definite cele două operații, se numeşte *polinom*.

Dacă $f = (a_0, a_1, a_2, ..., a_n,...)$ este un polinom, numerele $a_0, a_1, a_2, ...$ se numesc *coeficienții lui f*.

C. Forma algebrică a polinoamelor

Prin convenție, vom nota polinomul (0, 1, 0, 0,...) cu X şi îl vom citi *nedeterminata X*.

Folosim înmulțirea şi adunarea pentru a scrie: $f = a_0 + a_1X + a_2X^2 + ... + a_mX^m$, unde $a_0, a_1, a_2, ..., a_m$ sunt coeficienții polinomului f.

Polinoamele de forma aX^n, unde $a \in \mathbf{C}$ (mulțimea numerelor complexe) şi n este un număr natural se numesc *monoame*.

D. Reprezentarea polinoamelor prin şirul coeficienților

Coeficienții unui polinom se pot păstra într-un tablou unidimensional în ordine crescătoare (sau descrescătoare) după puterea lui X. Numim *gradul* polinomului puterea cea mai mare a lui X, pentru care coeficientul este diferit de 0. În concluzie, şirul corespunzător coeficienților va avea cel puțin atâtea elemente + 1 (pentru termenul liber), cât este gradul polinomului. Acest mod de reprezentare are avantajul că în cazul a două polinoame coeficienții aceloraşi puteri ale lui X sunt aşezați în cei doi vectori pe poziții corespunzătoare.

E. Reprezentarea polinoamelor prin monoamele sale

Dacă un polinom are mulți coeficienți egali cu 0, reprezentarea prin şirul coeficienților nu mai este avantajoasă. În acest caz, se preferă reprezentarea prin şirul monoamelor. De asemenea, dacă aceste monoame nu sunt accesibile în ordinea crescătoare (sau descrescătoare) a puterii lui X, nu vor apărea probleme, deoarece în cazul fiecărui monom se precizează două informații: valoarea coeficientului şi gradul termenului. Reprezentarea prin monoame se poate realiza cu mai multe tipuri de structuri de date.

- cu două şiruri: unul pentru coeficienții diferiți de 0 ai polinomului şi unul pentru gradele corespunzătoare ale termenilor;
- cu un şir de articole;
- cu ajutorul unei liste liniare, alocată dinamic[*)].

F. Valoarea unui polinom

Prin definiție numărul $f(\alpha) = a_0 + a_1\alpha + a_2\alpha^2 + ... + a_n\alpha^n$ se numeşte *valoarea polinomului f în α*.

S-a demonstrat că cea mai rapidă metodă de a calcula valoarea unui polinom este cea bazată pe schema lui *Horner*.

Polinomul P(X) se scrie sub forma:

$P(X) = (...(((a_n \cdot x + a_{n-1}) \cdot x + ... + a_2) \cdot x + a_1) \cdot x + a_0$

şi se aplică un algoritm simplu în care x este argumentul pentru care se calculează valoarea polinomului P de coeficienți a_i şi grad n:

```
Subalgoritm Horner(n,x,P):
  P ← a[n]
  pentru i=n-1,0 execută:
    P ← P*x + a[i]
sfârşit subalgoritm
```

D. Împărțirea polinoamelor

Fiind date două polinoame oarecare cu coeficienții complecşi f şi g, unde $g \neq 0$, există două polinoame cu coeficienți de tip complex q şi r, astfel încât $f = g \cdot q + r$, unde gradul polinomului r este mai mic decât gradul polinomului g. Polinoamele q şi r sunt unice dacă satisfac această proprietate.

Fie f şi g două polinoame. Spunem că *polinomul g divide polinomul f* (sau f este divizibil prin g, sau g este un divizor al lui f, sau f este un multiplu al lui g) dacă există un polinom h, astfel încât $f = g \cdot h$.

Fie f un polinom nenul cu coeficienți de tip complex. Un număr complex a este *rădăcină* a polinomului f, dacă $f(a) = 0$.

[*)] Vezi capitolul 9.

Fie $f \neq 0$ un polinom nenul. Numărul $a \in \mathbf{C}$ *este rădăcină a polinomului f* dacă şi numai dacă $X - a$ îl divide pe f.

Fie $f = a_0 + a_1X + a_2X^2 + ... + a_nX^n$ un polinom cu $a_n \neq 0$, $n \geq 1$. Dacă $x_1, x_2, ..., x_n$ sunt rădăcinile lui f, atunci $f = a_n \cdot (X - x_1) \cdot (X - x_2) \cdot ... \cdot (X - x_n)$ şi, în plus, această descompunere a lui f în factori liniari este unică.

Restul împărţirii unui polinom $f \neq 0$ prin binomul $X - a$ este egal cu valoarea $f(a)$ a polinomului f în a.

Schema lui Horner oferă un procedeu de aflare a câtului şi a restului împărţirii polinomului f prin binomul $X - a$. În cazul în care restul împărţirii este 0, a este rădăcină a polinomului. Orice rădăcină a polinomului este divizor al numărului a_0/a_n (din ultima relaţie a lui *Viéte*).

1.11. Mulţimi

1.11.1. Tipul mulţime

Mulţimea este o structură statică de date cu ajutorul căreia se poate reprezenta o colecţie finită de elemente. Caracterul finit ne permite indexarea elementelor sale: $M = \{m_1, m_2, ..., m_n\}$ ceea ce înseamnă că la nivelul reprezentării interne mulţimea poate fi asimilată cu un şir $(m_1, m_2, ..., m_n)$.

Tipul mulţime este un tip structurat, diferit de celelalte tipuri structurate; el corespunde noţiunii de mulţime din matematică.

O valoare a unui tip mulţime este o submulţime a valorilor unui tip ordinal, standard sau definit de utilizator, numit tipul de bază al mulţimii. (De exemplu, în limbajul Pascal tipul de bază al mulţimii are cel mult 256 de valori; valorile ordinale ale acestora trebuie să aparţină intervalului [0, 255].)

Un tip mulţime are atâtea valori câte submulţimi admite mulţimea valorilor tipului de bază şi mulţimea vidă, respectiv 2^n submulţimi, dacă tipul de bază are n valori.

1.11.2. Atribuirea

Unei variabile de tip mulţime i se atribuie valoarea unei expresii de acelaşi tip mulţime.

1.11.3. Declararea tipului mulţime (Pascal)

`type` *nume_tip*=**`set of`** *tip_de_bază*;
`var` *identificator_variabilă*:*nume_tip*;
unde *tip_de_bază* este tipul de bază al mulţimii şi are cel mult 256 de valori.

1.11.4. Operaţii cu mulţimi

Operaţiile cunoscute din teoria mulţimilor – se implementează în funcţie de instrumentele limbajului de programare.

A. Operatori binari

- Reuniunea a două mulțimi: `+`
- Intersecția a două mulțimi: `*`
- Diferența a două mulțimi: `-`

B. Operatori relaționali

≤ (incluziune), ≥ (include pe), = (egalitate), <> (diferite).

C. Test de apartenență a unui element la o mulțime

Pentru a verifica dacă o valoare dată se află sa nu într-o submulțime, în Pascal se foloseşte operatorul relațional **`in`**.

D. Citirea unei variabile de tip mulțime

În limbajul Pascal nu este permisă citirea unei variabile de tip mulțime. În concluzie, se va proceda similar cu citirea unei variabile de tip tablou, citind componentele mulțimii una câte una:

```
Subalgoritm Citire_Mulțime(m):
  m ← ∅      { în Pascal mulțimea vidă se notează cu ajutorul constructorului: [] }
  cât timp mai există elemente execută:
    citeşte element
              { element trebuie să fie de acelaşi tip cu tipul de bază al mulțimii }
    m ← m + [element]
           { operația de reuniune se efectuează între doi operanzi de tip mulțime }
  sfârşit dacă
sfârşit subalgoritm
```

E. Scrierea unei variabile de tip mulțime

În limbajul Pascal nu este permisă afişarea unei variabile de tip mulțime. Şi de data aceasta se va proceda similar cu scrierea unei variabile de tip tablou, scriind componentele mulțimii una câte una.

În subalgoritmul următor presupunem că mulțimea are tipul de bază `Byte`.

```
Subalgoritm Scriere_Mulțime(m):
  pentru element=0,255 execută:
            { generăm fiecare element care aparține tipului de bază al mulțimii }
    dacă element ∈ m atunci          { dacă element aparține mulțimii }
      scrie element                                  { atunci îl scriem }
    sfârşit dacă
  sfârşit pentru
sfârşit subalgoritm
```

Nu orice structură de date de tip mulţime poate fi implementată cu un tip predefinit în limbajul de programare respectiv. De exemplu, chiar dacă în Pascal există posibilitatea folosirii tipului **`set`**, acesta nu este utilizabil în cazul unei mulţimi având cardinalitate mai mare decât 256, sau în cazul în care elementele mulţimii nu sunt de tip ordinal. În aceste cazuri vom implementa un tip mulţime propriu, apelând la tipul tablou (**`array`**). În cazul implementărilor proprii va trebui să creăm subalgoritmii care realizează operaţiile specifice cum sunt reuniunea şi intersecţia.

1.12. Tipul înregistrare

Articolul (înregistrarea) reprezintă reuniunea unui număr fix (sau variabil) de componente, numite *câmpuri*, (de regulă având tipuri diferite) care constituie logic o unitate de prelucrare. Tipurile diferite ale câmpurilor din componenţa sa dau articolului caracterul de structură *neomogenă*. Dacă L_i este lungimea câmpului i (i = 1, 2, ..., n), atunci valoarea adresei câmpului i este: $a_i = a_1 + L_1 + L_2 + ... + L_{i-1}$.

Caracterul neomogen al articolului, precum şi lipsa unei relaţii liniare între numărul de ordine al câmpului şi adresa acestuia fac ca regăsirea unui câmp să se realizeze cu ajutorul unui identificator asociat câmpului respectiv la definirea articolului.

Tipul articol (înregistrare) este o structură statică de date flexibilă, datorită faptului că, spre deosebire de tipul tablou, cuprinde un număr fix sau variabil de componente care pot fi de tipuri diferite. *Componentele* nu se indexează printr-o expresie (ca în cazul tablourilor), ci *se selectează prin intermediul identificatorilor de câmp* care se definesc la declararea tipului.

1.12.1. Declararea unui tip articol în limbajul Pascal

Forma generală a definiţiei unui *tip înregistrare fixă* este:

`type` *identificator_tip*=**`record`**
nume_câmp$_1$`:`*tip*$_1$`;`
nume_câmp$_2$`:`*tip*$_2$`;`
`...`
nume_câmp$_n$`:`*tip*$_n$
`end;`

Tipul unui identificator de câmp poate fi la rândul lui tot articol.

Un identificator de câmp trebuie să fie unic în cadrul propriului tip *înregistrare*.

Dacă se doreşte includerea în structura articolului a unor informaţii care depind de o altă informaţie deja prezentă în structură se formează cea de-a doua categorie, şi anume date de tip *înregistrare cu variante*.

1.12.2. Operații

A. Atribuirea
Unei variabile de tip înregistrare i se atribuie valoarea unei alte variabile de același tip *înregistrare*. Fiind date două variabile de tip înregistrare *a* şi *b*, care au același tip, se poate realiza operația de atribuire $a \leftarrow b$, ceea ce înseamnă că valorile câmpurilor lui *a* se substituie cu valorile din *b*.

B. Selectarea unui câmp
La un câmp al unei variabile de tip înregistrare ne referim prin numele variabilei, urmat de punct, apoi numele câmpului.

O astfel de scriere este greoaie şi pentru evitarea specificării identificatorului de tip înregistrare în fața fiecărui câmp care intervine în prelucrare, în Pascal se poate folosi instrucțiunea **`with`**, care simplifică modul de scriere a selectării componentelor.

1.13. Operații pe biți

În memoria calculatorului toate numerele sunt reprezentate în sistemul de numerație binar. Noi le vedem în sistemul de numerație zecimal doar pentru că sistemul efectuează permanent conversii pentru datele introduse de la tastatură spre memorii şi de la memorie spre monitor pentru ca utilizatorii să nu fie obligați să descifreze numere reprezentate în sistemul binar. Ca atare, cu ajutorul operatorilor pe biți putem acționa direct asupra reprezentării binare fără a fi nevoie de conversii.

Deoarece operatorii pe biți se execută mai rapid decât operatorii aritmetici, în implementarea algoritmilor se preferă frecvent utilizarea lor în schimbul celor aritmetici.

1.13.1. Negarea

Negarea este un operator unar care acționează asupra unui singur operand de tip întreg având ca efect schimbarea fiecărui bit al reprezentării numărului din 1 în 0 şi din 0 în 1 (**`not`** `a` în limbajul Pascal şi `~a` în limbajul C^{++}).

1.13.2. Disjuncția

Disjuncția, (numită *sau* logic) este un operator binar care acționează asupra a doi operanzi întregi, bit cu bit, având ca efect setarea bitului rezultat pe 1, dacă cel puțin unul dintre biți este 1 şi pe 0 în caz contrar (`a` **`or`** `b` în limbajul Pascal şi `a | b` în C^{++}).

17.3.3. Conjuncția

Conjuncția, (numită *şi* logic) este un operator binar care acționează asupra a doi operanzi întregi având ca efect setarea bitului rezultat pe 1, dacă ambii biți sunt 1 şi pe 0 în caz contrar (`a` **`and`** `b` în limbajul Pascal şi `a & b` în limbajul C^{++}).

1.13.3. „Sau exclusiv”

Sau exclusiv este un operator binar care acționează asupra a doi operanzi întregi bit cu bit având ca efect setarea bitului rezultat pe 1, dacă cei doi biți nu au aceeași valoare și pe 0 în caz contrar (`a` **`xor`** `b` în limbajul Pascal și `a ^ b` în limbajul C++).

1.13.4. Rotiri pe biți

Operatorul de rotire a biților spre dreapta și cel pentru rotirea biților spre stânga sunt operatori binari, primul operand fiind numărul ai cărui biți vor fi rotiți, iar cel de-al doilea reprezintă numărul de rotații. În locul biților deplasați din margine, numărul se completează cu 0-uri (**`shl`** (rotire la stânga) și **`shr`** (rotire la dreapta).

Orice rotire (*translatare*) la stânga cu o poziție este echivalentă cu o înmulțire a numărului cu 2 și orice rotire la dreapta cu o poziție este echivalentă cu o împărțire a numărului la 2.

1.14. Recursivitate

1.14.1. Noțiunea de recursivitate

Noțiunea de *recursivitate* din programare derivă în mod natural din noțiunea matematică cunoscută sub numele de *recurență*. Dar și viața de zi cu zi oferă nenumărate exemple. O funcție este recurentă dacă este definită prin sine însăși.

1.14.2. Proiectarea unui algoritm recursiv

În programare, recursivitatea se realizează prin autoapelul unui subprogram. Altfel spus, în corpul subprogramului apare un apel al subprogramului însuși în timp ce acesta este activ. Pentru ca apelul să nu se realizeze la infinit, este necesară existența în subprogram a unei condiții corecte de oprire a acestor apeluri.

Vom întâlni probleme în care putem aplica o formulă de recurență dată, dar și probleme în care această relație trebuie determinată pe baza enunțului.

1.14.3. Execuția apelurilor recursive

Reamintim că orice apel de subprogram (chiar și atunci când este vorba de autoapel) are ca efect salvarea pe stiva calculatorului a adresei de revenire, a valorilor parametrilor transmiși prin valoare și a adresei parametrilor transmiși prin referință, precum și alocarea de spațiu pe stivă pentru variabilele locale ale subprogramului.

1.14.4. Recursivitate indirectă (încrucişată)

În situaţia în care două sau mai multe subprograme se apelează reciproc, recursivitatea este *indirectă* sau încrucişată. Pentru a asigura ieşirea din recursivitate, va exista o condiţie de ieşire cel puţin într-unul dintre subprograme.

1.15. Metoda backtracking

Metoda *backtracking* este o metodă de programare cu ajutorul căreia se rezolvă problemele în care:

- soluţia se poate reprezenta sub forma unui tablou $X = (x_1, x_2, \ldots, x_n)$ cu $x_1 \in M_1$, $x_2 \in M_2$ etc. $M_1, M_2, ..., M_n$ fiind mulţimi finite având $s_1, s_2, ..., s_n$ elemente.
- între elementele tabloului X există anumite legături impuse în enunţ.

În fiecare problemă se precizează anumite relaţii care trebuie să existe între componentele $x_1, x_2, ..., x_n$ ale vectorului X, numite *condiţii interne*.

Mulţimea $M_1 \times M_2 \times ... \times M_n$ se numeşte *spaţiul soluţiilor posibile*.

Dacă la pasul k condiţiile interne sunt satisfăcute, algoritmul se continuă în funcţie de cerinţe. Dacă mai trebuie căutate componente, deoarece numărul lor se cunoaşte şi încă nu le-am obţinut pe toate, sau componentele încă nu constituie o soluţie pe baza unor proprietăţi, *condiţiile de continuare* vor permite continuarea algoritmului.

Acele soluţii dintre soluţiile posibile care satisfac condiţiile impuse de problemă (condiţiile interne şi condiţiile de continuare nu cer componente în plus) se numesc *soluţii rezultat*.

O metodă de rezolvare a acestei categorii de probleme ar fi determinarea tuturor soluţiilor posibile şi apoi căutarea acelora care satisfac condiţiile interne. Dezavantajul este uşor de observat, deoarece timpul cerut de această căutare este foarte mare.

1.15.1. Modul de lucru al metodei backtracking

- Elementele din tabloul X primesc *pe rând* valori, adică lui x_k i se atribuie o valoare numai dacă elementelor $x_1, x_2, ..., x_{k-1}$ au fost deja atribuite valori.
- În plus, lui x_k i se atribuie o valoare numai dacă pentru valorile $x_1, x_2, ..., x_{k-1}$ şi valoarea propusă pentru x_k sunt îndeplinite condiţiile interne şi cele de continuare.
- Dacă la pasul k condiţiile de continuare nu sunt îndeplinite trebuie să se facă o altă alegere pentru x_k din mulţimea M_k.
- Dacă mulţimea M_k a fost epuizată (s-au testat toate valorile din mulţime) se decrementează k şi se încearcă o altă alegere pentru x_{k-1} din mulţimea M_{k-1}.

1.15.2. Backtracking iterativ

Condiţiile interne se verifică în subalgoritmul `Posibil(k)`. Acesta returnează *adevărat* dacă adăugarea componentei x_k a şirului soluţie este posibilă şi *fals* în caz contrar.

```
Subalgoritm Posibil(k):
  dacă condițiile interne nu sunt îndeplinite atunci
    Posibil ← fals
    ieşire forțată
  sfârşit dacă
  Posibil ← adevărat
sfârşit subalgoritm
```

Generarea soluțiilor se realizează cu subalgoritmul `Back`.

```
Subalgoritm Back:
  k ← 1
  x[k] ← 0
  cât timp k > 0 execută:
    ok ← fals
    cât timp (x[k]< valoare maximă permisă) şi nu ok execută:
    { mai sunt valori netestate în mulțime, deci lui x[k] i se atribuie o altă valoare }
      x[k] ← x[k] + 1
      ok  ← Posibil(k)        { poate am găsit valoare posibilă pentru x[k] }
    sfârşit cât timp
    dacă nu ok atunci
      { dacă nu, se decrementează k pentru a alege o valoare nouă pentru aceasta }
      k ← k - 1
    altfel                                  { dacă am ajuns la sfârşitul generării }
      dacă k = numărul de elemente cerut atunci          { sau altă condiție }
        scrie soluția
      altfel
        k ← k + 1                  { trecem la următorul element din soluție }
        x[k] ← 0                   { inițializăm noul element din tabloul soluție }
      sfârşit dacă
    sfârşit dacă
  sfârşit cât timp
sfârşit subalgoritm
```

1.15.3. Backtracking recursiv

Algoritmul poate fi implementat şi recursiv (funcția `Posibil(k)` este cea cunoscută):

```
Subalgoritm Back(k):
  pentru i = 1,valoarea maximă impusă de problemă execută:
    x[k] ← i
    dacă Posibil(k) atunci
            { dacă am ajuns la sfârşitul generării sau altă condiție de continuare }
      dacă (*) k = numărul de elemente cerut atunci scrie soluția(k)
```

```
      altfel
        Back(k+1)
      sfârşit dacă
    sfârşit dacă
  sfârşit pentru
sfârşit subalgoritm
```

Dacă în condiția de continuare `(*)` expresia este formată din subexpresii logice (relaționale) şi condiția nu este îndeplinită, pe ramura **`altfel`** va trebui să decidem motivul pentru care condiția nu este îndeplinită. Dacă de exemplu, numărul de elemente cerut s-a atins, dar din alte motive soluția obținută nu este corectă şi deci nu se scrie, atunci vom ieşi din subprogram, asigurând totodată revenirea la pasul precedent şi căutarea unei valori noi pentru $k - 1$.

1.16. Probleme propuse

Problemele propuse în această secțiune pot fi rezolvate în cadrul unei evaluări/testări efectuate de profesor, după recapitulare, cu scopul stabilirii nivelului de cunoştințe a elevilor. Problemele sunt grupate pe 3 niveluri:

1. începători
2. avansați
3. excelenți

1.16.1. Începători

A. Medie aritmetică

Se citesc numere întregi pozitive de pe mai multe linii ale unui fişier text. De pe fiecare linie se aleg doar atâtea numere cât reprezintă numărul de ordine a liniei. Creați un fişier de ieşire care să conțină pe fiecare linie media aritmetică a numerelor pare de pe linia corespunzătoare din fişierul de intrare.

Date de intrare

Pe fiecare linie a fişierului de intrare **`MEDIA.IN`** sunt scrise mai multe numere întregi separate prin unul sau mai multe spații.

Date de ieşire

Fişierul de ieşire **`MEDIA.OUT`** are tot atâtea linii ca şi fişierul de intrare, iar pe fiecare a i-a linie este scrisă media aritmetică cu două zecimale exacte a i numere citite de pe a i-a linie a fişierului de intrare.

Restricţii şi precizări
- dacă pe linia *i* nu există *i* numere, se calculează media numerelor găsite;
- dacă din numerele citite nici unul nu este par media aritmetică este 0.00;
- scrierea în fişierul de ieşire se face pe şase caractere din care două sunt pentru partea zecimală;
- numerele din fişier nu depăşesc valoarea 999.

Exemplu

MEDIA.IN	**MEDIA.OUT**
1 2 3 4 5 6 11 2	0.00
12 1 343 2 45 12	12.00
123 23 14 2	14.00
2 2 2 2 4 44	2.00
1 2 4 6 66 7 888 999	19.50

Timp maxim de executare/test: 1 secundă

B. Multiplicare

De pe fiecare linie a unui fişier text se citeşte o cifră şi un caracter. Să se formeze un alt fişier în care pe fiecare linie să apară cifra multiplicată de atâtea ori cât reprezintă codul ASCII al caracterului de pe linia respectivă.

Date de intrare

Pe fiecare linie a fişierului **MULTI.IN** este scris o cifră, un spaţiu şi un caracter.

Date de ieşire

Fişierul de ieşire **MULTI.OUT** are tot atâtea linii ca şi fişierul de intrare, iar pe fiecare linie se va scrie cifra de atâtea ori cât reprezintă codul ASCII al caracterului citit. Între două apariţii ale numărului în fişierul de ieşire nu există spaţii sau alţi separatori.

Restricţii şi precizări
- cifrele din fişier sunt cifre în sistemul de numeraţie zecimal.

Exemplu

MULTI.IN	**MULTI.OUT**
1 (	1111111111111111111111111111111111111
2 2	22222222222222222222222222222222222222222222222
3 0	333333333333333333333333333333333333333333333333

Explicaţie

Codul ASCII al lui `'('` este 40, al lui `'2'` este 50, al caracterului `'0'` este 48.

Timp maxim de executare/test: 1 secundă

C. Florărie

Angajații unei florării au stabilit dimensiunile parcelelor şi tipurile de flori care se vor planta. După plantare au dorit să verifice respectarea dimensiunilor parcelelor prin compararea planului inițial cu partajarea actuală a întregii grădini. Au verificat dacă a fost plantată fiecare tip de floare şi dacă dimensiunea fiecărei parcele existente corespunde cu dimensiunea din planul inițial.

Presupunând că inițial nu s-a stabilit şi ordinea parcelelor în grădină, realizați un program care să-i ajute pe cei care efectuează controlul să verifice corectitudinea angajaților.

Tipurile de flori sunt codificate prin numere naturale (1 reprezintă bujori, 2 reprezintă narcise...etc.).

Date de intrare

Grădina se memorează sub forma unui tablou bidimensional în care elementele reprezintă tipurile de flori. Datele se citesc din fişierul de intrare **`GRADINA.IN`**, unde pe prima linie se află dimensiunile grădinii *n* şi *m*. De pe următoarele *n* linii se citesc câte *m* numere separate prin spațiu care reprezintă tipul de floare. De pe următoarele linii se citesc câte trei numere care reprezintă dimensiunile parcelelor şi tipul florii care trebuia să fie plantată.

Date de ieşire

Rezultatul analizei asupra corectitudinii angajaților legat de plantarea fiecărei parcele se va scrie în fişierul de ieşire **`GRADINA.OUT`**, pe linii distincte sub forma:
numar_floare mesaj, unde *mesaj* înseamnă pentru fiecare parcelă:

- `'Corect'`, dacă plantarea s-a făcut corect;
- `'Parcela inexistenta'`, dacă un anumit tip de floare nu apare;
- `'Dimensiuni eronate'`, dacă plantarea este incorectă, deoarece nu s-au respectat dimensiunile.

Restricții şi precizări

- caracteristicile unei parcele sunt: *nn*, *mm*, *tip_floare*;
- dimensiunile parcelelor se presupun a fi corecte, deci nu necesită validare;
- $1 \le n, m \le 20$;
- $1 \le nn_i \le n$;
- $1 \le mm \le m$;
- $1 \le tip\_floare \le m \cdot n$;
- dacă o anumită parcelă are dimensiuni mai mari decât cele prevăzute inițial se consideră a fi corectă, deoarece dimensiunile prevăzute inițial se includ în cele actuale.

Exemplu

GRADINA.IN
```
5 10
1 1 1 2 2 2 2 3 3 4
1 1 1 2 2 2 2 3 3 4
5 5 5 2 2 2 2 7 7 4
5 5 5 6 6 6 6 7 7 4
5 5 5 6 6 6 6 7 7 4
2 3 1
3 4 2
2 2 3
5 1 4
3 4 5
2 4 6
3 2 7
4 4 8
```

GRADINA.OUT
```
1 Corect
2 Corect
3 Corect
4 Corect
5 Dimensiuni eronate
6 Corect
7 Corect
8 Parcela inexistenta
```

Timp maxim de executare/test: 1 secundă

D. Concatenare

Să se construiască numele unei persoane din caracterele numelui.

Date de intrare

În fişierul de intrare **CONCAT.IN** sunt scrise caractere (litere şi asteriscuri), câte un caracter pe o linie. Caracterul asterisc separă numele de prenume etc.

Date de ieşire

Numele persoanei se va scrie în fişierul de ieşire **CONCAT.OUT**. Corespunzător caracterului asterisc se va insera în nume un spațiu.

Restricţii şi precizări

- caracterele sunt litere mici şi mari din alfabetul englez;
- numele are cel mult 255 caractere litere şi spații.

Exemplu

CONCAT.IN
```
A
l
i
*
B
a
b
a
```

CONCAT.OUT
```
Ali Baba
```

Timp maxim de executare/test: 1 secundă

E. Linia de orizont

Desenați toate linie de orizont care se pot imagina cu ajutorul a *n* caractere `'/'` şi *n* caractere `'\'`. O linie de orizont începe cu caracterul `'/'` şi se termină cu caracterul `'\'` pe linia orizontală corespunzătoare „nivelului mării”.

Date de intrare

Din fişierul de intrare **`ORIZONT.IN`** se citeşte numărul natural *n*.

Date de ieşire

Liniile de orizont se vor scrie, despărțite prin câte o linie vidă în fişierul de ieşire **`ORIZONT.OUT`**.

Restricții şi precizări

- $1 \leq n \leq 10$.

Exemplu

```
4
```

```
   /\
  /  \

 /\/\
```

Timp maxim de executare/test: 1 secundă

1.16.2. Avansați

A. Numere *Niven*

Numerele care sunt divizibile cu suma cifrelor lor, se numesc numere *Niven*. De exemplu, 21 este număr *Niven*, deoarece 2 + 1 = 3, îl divide pe 21. Această proprietate se poate extinde şi pentru alte baze de numerație. De exemplu, 110 este număr *Niven* în baza 2: 1 + 1 + 0 = 2 és $(110)_2 = 6$ este divizibil cu 2.

Scrieți un program care, stabileşte dacă numărul *n*, dat într-o bază de numerație *b*, stabileşte dacă este număr *Niven* sau nu.

Date de intrare

Din fişierul de intrare **`NIVEN.IN`** se citesc mi multe perechi de numere *n* şi *b*, reprezentând numărul de verificat şi baza de numerație în care s-a dat.

Date de ieşire

Fişierul de ieşire **`NIVEN.OUT`** va conține atâtea linii câte perechi de numere s-au dat în fişierul de intrare. Pe fiecare linie se va scrie mesajul `'DA'`, respectiv `'NU'` în funcție de constatarea făcută privind proprietatea numărului *n*.

Restricții și precizări

- $2 \le b \le 10$;
- $2 \le b \le 2000000000$.

Exemplu

NIVEN.IN	**NIVEN.OUT**
110 2	DA
123 10	NU

Timp maxim de executare/test: 1 secundă

B. Stânga-dreapta

În 15 septembrie domnul director îi așează pe toți copiii de clasa I într-un șir cu fața spre el. Apoi le cere să se întoarcă: „La stânga!”. Copiii, nesiguri pe ei, se întorc fie la stânga, fie la dreapta. Cei care văd spatele vecinului sau nu văd pe nimeni, consideră că s-au întors corect. Cei care constată că s-au întors față în față cu un vecin, crezând că au greșit, se întorc cu spatele la acesta. Toți copiii care cred că au greșit, vor face o întoarcere cu 180 grade simultan, într-o unitate de timp.

Pentru o configurație inițială de întoarceri după comanda „La stânga!”, să se afișeze numărul unităților de timp care se scurg până când nu se mai întoarce nimeni (indiferent dacă poziția lor finală este corectă sau nu, conform comenzii date).

Date de intrare

Fișierul de intrare **COPII.IN** conține un șir de caractere, reprezentând pentru fiecare poziție *i* din șir, direcția în care s-a întors al *i*-lea copil.

Date de ieșire

Pe prima linie a fișierului **COPII.OUT** se va afișa șirul de caractere care reprezintă pozițiile (se uită în stânga sau în dreapta) copiilor după ce nu se mai mișcă nimeni. Pe cea de a doua linie se va afișa un număr întreg care reprezintă numărul unităților de timp care s-au scurs până la liniștire (când nu se mai mișcă nimeni). În cazul în care nu sunt întoarceri de corecție, această valoare va fi 0.

Restricții și precizări

- Șirul de caractere este format din cel mult 1600 de litere `'s'` și `'d'` nedespărțite prin spațiu.

Exemplu

COPII.IN	**COPII.OUT**
sdsds	sssdd
	2

Timp maxim de executare/test: 1 secundă

C. Împărțeală

Ancuța a primit de ziua ei *n* bomboane şi ar vrea să le împartă printre prietenii cei mai apropiați. În scopul unei împărțeli speciale, ea a notat pe o hârtie numele celor *k* prieteni în ordinea importanței lor în viața ei şi ar vrea ca:

1. fiecare prieten să primească cel puțin o bomboană, şi
2. un prieten nu poate primi mai multe bomboane decât un altul care are numărul de ordine mai mic.

Observând cât de multe posibilități are, Ancuța s-a zăpăcit, dar totuşi ar vrea să ştie din câte posibilități va trebui să aleagă. Ajutați-o, scriind un program care calculează numărul posibilităților de a împărți cele *n* bomboane printre cei *k* prieteni respectând cele două restricții stabilite de Ancuța.

Date de intrare

Fişierul de intrare **ANCA.IN** conține două numere naturale, despărțite printr-un spațiu, reprezentând numărul bomboanelor şi numărul prietenilor.

Date de ieşire

În fişierul de ieşire **ANCA.OUT** se va scrie un număr natural care reprezintă numărul posibilităților pe care le are Ancuța la dispoziție pentru a împărți bomboanele printre prietenii săi.

Restricții şi precizări

- $1 \le n \le 100$;
- $1 \le k \le 12$;
- Se împart toate bomboanele.

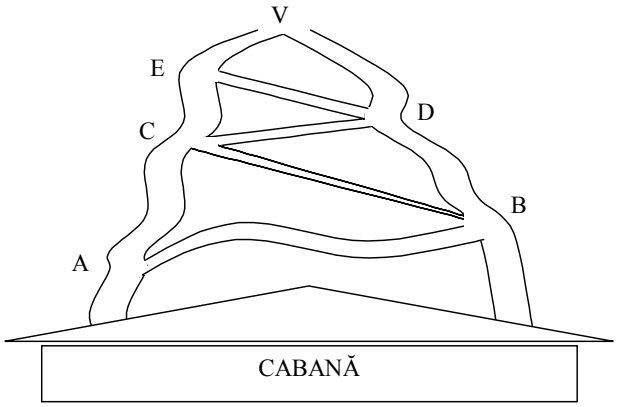
Exemplu

ANCA.IN	**ANCA.OUT**
10 3	8

Timp maxim de executare/test: 1 secundă

D. Excursie

Pe un vârf înalt din Munții Retezat se poate urca pe două poteci abrupte care pornesc de la cabană. Cele două poteci sunt legate din când în când printr-o serpentină, adică o potecă mai comodă care leagă două puncte oarecare de pe cele două poteci abrupte, astfel încât pe aceste porțiuni se poate urca mai domol. Un grup de turişti ar vrea să ştie în câte feluri ar putea să urce spre vârf, folosind toate modalitățile posibile de a urca pe porțiuni de potecă abruptă, apoi pe porțiuni de serpentine, iar pe potecă abruptă etc.



Determinați numărul tuturor traseelor pe care turiştii ar putea să urce spre vârf, ştiind că ei întotdeauna vor merge ***în sus*** spre vârf. De exemplu, corespunzător figurii sunt posibile porțiunile *cabană* $\rightarrow A$, $A \rightarrow C$, sau $A \rightarrow B$, apoi *cabană* $\rightarrow B$ etc., dar nu sunt posibile $C \rightarrow A$, $B \rightarrow A$ etc.

Date de intrare
În fişierul de intrare **`EXCURSIE.IN`** se află un număr natural n, reprezentând numărul total de intersecții ale potecilor abrupte şi ale serpentinelor.

Date de ieşire
În fişierul de ieşire **`EXCURSIE.OUT`** se va scrie un singur număr, reprezentând numărul traseelor diferite dintre punctul de pornire şi vârf.

Restricții
- $1 < n \leq 40$.

Exemplu

`EXCURSIE.IN`	**`EXCURSIE.OUT`**
`5`	`13`

Timp maxim de executare/test: 1 secundă

1.16.3.Excelenți

A. Loto1
Un jucător alege n numere şi completează un bilet de *Super Loto Special*. Să se stabilească ce câştigă acesta, în funcție de numărul numerelor câştigătoare ale lui, ştiind că o extragere conține 6 numere.

Pentru ca jucătorul să câştige, el trebuie să ghicească cel puțin o variantă de patru numere în ordinea în care acestea au fost extrase. Dacă el ghiceşte numerele, dar în altă ordine, nu câştigă nimic. De asemenea, dacă jucătorului i-au ieşit 1, 2 sau 3 numere, el nu câştigă.

Date de intrare
Numărul de numere jucate se găseşte pe prima linie a fişierului de intrare **LOTO.IN**. Numerele de pe bilet se găsesc pe a doua linie a fişierului, iar numerele extrase sunt scrise pe a treia linie. Numerele de pe o aceeaşi linie sunt separate prin câte un spațiu.

Date de ieşire
În fişierul **LOTO.OUT** se va afişa, după caz, unul din mesajele:
```
castig cu 4 numere
castig cu 5 numere
castig cu 6 numere
nimic
```

Restricții şi precizări
- $n \geq 6$

Exemplu

LOTO.IN
```
8
1 3 4 5 10 2 9 6
2 3 4 5 6 7
```

LOTO.OUT
```
castig cu 4 numere
```

B. Ali Baba

Ali Baba i-a surprins pe cei 40 de hoți, în timp ce cotrobăiau printre comorile lui. Hoții erau mulți, el era singur, aşa că a încercat să negocieze cu ei. Avea o lădiță specială, pe care era notat numărul diamantelor aflate în aceasta, un număr având cel mult 40 de cifre, într-o bază b. Ali Baba a propus conducătorului hoților să elimine din număr b cifre, după care poate să plece cu atâtea diamante câte reprezintă numărul rămas.

Hoțul, mai întâi a stabilit valoarea cea mai mică posibilă b, deoarece a vrut să stabilească un număr cât mai mic posibil de cifre pe care urmează să le elimine. Apoi a început tăierea cifrelor. În timpul eliminării a urmărit ca numărul rămas să fie cât se poate de mare.

Scrieți un program care îl ajută pe hoț (Ali Baba are destule comori...).

Date de intrare
Pe prima linie a fişierului de intrare **ALIBABA.IN** se află un şir de cifre, nedespărțite prin spații.

Date de ieşire
Pe prima linie a fişierului de ieşire **ALIBABA.OUT** se va scrie un număr natural *b*, reprezentând numărul cifrelor eliminate din numărul dat. Pe următoarele *b* linii se va scrie pe fiecare câte un număr, reprezentând numărul rămas după eliminarea unei cifre.

Restricţii şi precizări
- $2 \le b \le 10$.

Exemplu

ALIBABA.IN	**ALIBABA.OUT**
323332112	4 33332112 3333212 333322 33332

Timp maxim de executare/test: 1 secundă

C. Loto2

Vasile ar vrea cu orice preţ să câştige la loto. El îşi notează toate numerele posibile de jucat, apoi scrie un program care să-i genereze configuraţii de numere care ar putea să fie trecute pe biletele de loto. Programul lui generează toate variantele posibile (combinaţii de 6 numere din 49) şi de asemenea, numerotează variantele începând cu numărul de ordine 1. Vasile şi-a dat seama că numărul variantelor fiind foarte mare, nu are nici o şansă să cumpere atâtea bilete câte variante sunt. În concluzie, el hotărăşte să aleagă prima variantă, o trece pe un bilet, apoi selectează fiecare a *k*-a variantă din lista generată de program. Atunci când ajunge la capătul listei, continuă numărătoarea de la începutul listei, ca şi cum aceasta ar fi circulară. El termină selectarea variantelor atunci când, datorită numărătorii efectuate, nimereşte o variantă pe care a ales-o deja.

Vasile totuşi este supărat şi îşi imaginează un joc de loto în care se extrag cel mult *y* numere din totalul de *x*.

Ajutaţi-l pe Vasile şi scrieţi un program care calculează numărul total *n* al variantelor pe baza numerelor *x* şi *y*. Pe baza numerelor *n* şi *k* date, determinaţi numărul variantelor pe care Vasile ***nu le va juca*** dintre cele posibile, dacă el va aplica strategia descrisă.

Date de intrare
Pe prima linie a fişierului de intrare **LOTO.IN** sunt scrise numerele naturale nenule *x* şi *y*, despărţite printr-un spaţiu. Pe a doua linie a fişierului de intrare se află două numere naturale nenule *n* şi *k*, despărţite printr-un spaţiu, reprezentând numărul de variante şi numărul *k* ales de Vasile.

Date de ieşire
Pe prima linie a fişierului de ieşire **LOTO.OUT** veți scrie numărul variantelor calculat pe baza numerelor *x* şi *y*, citite din fişierul de intrare. Pe a doua linie a fişierului de ieşire veți scrie numărul variantelor pe care nu le va juca Vasile.

Restricții şi precizări
- Două variante care conțin aceleaşi numere, dar în ordine diferită, se consideră identice;
- $1 < n \leq 13983816$;
- $1 < k < n$;
- $1 < x \leq 49$;
- $1 < y \leq 6$;
- Punctajul acordat pentru fiecare dintre cele două programe este 50 de puncte.

Exemple

LOTO.IN
```
4 2
6 2
```

LOTO.OUT
```
6
3
```

LOTO.IN
```
30 4
27405 5
```

LOTO.OUT
```
27405
21924
```

Timp maxim de executare/test: 1 secundă

D. Suma maximă

Se consideră un tablou bidimensional *a* având elemente numere naturale, distincte. Determinați acel „drum" în tabloul dat care atinge o singură dată fiecare poziție în caroiajului corespunzător tabloului, şi pentru care următoarea sumă este maximă:

$$\sum_{k=1}^{n*m} k * a_{ij}$$

În expresia de mai sus *k* este pasul la care elementul a_{ij} se atinge pe drumul care parcurge tabloul dat.

Date de intrare
Pe prima linie a fişierului de intrare **SUMAMAX.IN** se află două numere naturale *n* şi *m*, separate printr-un spațiu, reprezentând numărul de linii, respectiv de coloane ale tabloului. Pe următoarele *n* linii se află, despărțite prin câte un spațiu câte *m* numere naturale, reprezentând elementele tabloului dat.

Date de ieşire
Pe prima linie a fişierului de ieşire se va scrie un număr natural, reprezentând suma maximă posibilă. Pe următoarea linie se vor scrie elementele tabloului dat, în ordinea în care au fost atinse pe drumul care are proprietatea cerută.

Restricţii şi precizări
- $1 \leq n, m \leq 10$;
- se garantează că suma obținută va fi mai mică decât 2 miliarde;
- dacă există mai multe soluții se va afişa una singură.

Exemplu

SUMAMAX.IN	**SUMAMAX.OUT**
2 3 1 2 3 6 5 4	21 1 2 3 4 5 6

Timp maxim de execuție/test: 1 secundă