Tablouri unidimensionale

Capitolul

6

- **Atribuirea**
- **Selectarea unei componente**
- **Declararea tablourilor unidimensionale**
- **Citirea şi afişarea tablourilor unidimensionale**
- **Prelucrări simple pe tablouri**
- **Implementări sugerate**
- **Probleme propuse**
- **Soluțiile problemelor**

Pentru a defini structura de date ***tablou*** considerăm o mulțime M de valori oarecare. Aplicația f: $\{1, 2, ..., n\} \rightarrow M$ ataşează fiecărui indice $i \in \{1, 2, ..., n\}$ un element $m_i \in M$. Ea defineşte un tablou unidimensional T_n, numit ***vector*** sau ***şir***, având elementele $m_1 = f(1)$, $m_2 = f(2)$, ..., $m_n = f(n)$.

În memoria internă, elementele acestui tablou vor ocupa în ordine locații succesive de memorie. Presupunând că primul element al tabloului este memorat la adresa a_1 şi că un element ocupă l locații, adresa relativă a_i (față de începutul tabloului) a locației ocupate de elementul i va fi:

$a_i = a_1 + l \cdot (i - 1)$, $i \in \{1, 2, ..., n\}$.

Indicele este o expresie de tip ordinal care conține informații cu privire la poziția elementului în cadrul structurii.

Elementele tabloului trebuie să fie de acelaşi tip, adică tabloul are o *structură omogenă*. Un *tablou unidimensional* este o structură de date care cuprinde un număr de elemente mai mic sau egal cu un număr precizat prin *tipul indicelui*.

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

6.1. Atribuirea

Prin această operație, *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*.

6.2. Selectarea unei componente

O componentă a tabloului se precizează prin *specificarea variabilei de tip tablou, urmată între paranteze drepte de o expresie indiceală*. De exemplu: `a[i+1]`.

6.3. 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;
```

În forma generală de mai sus, *nume_tip* este un identificator ales de utilizator, *tip_de_bază* şi *tip_indice* sunt tipuri predefinite sau tipuri utilizator. *tip_indice* este, în mod obligatoriu, un tip ordinal şi, în cazul în care este de tip subdomeniu, poate fi precizat cu ajutorul unor expresii, caz în care în ea pot interveni doar constante şi constante simbolice. *tip_de_bază* poate fi orice tip elementar sau structurat, cunoscut în momentul declarației.

Exemplu

```
const n=10;
type Litere='a'..'z';
     indice=1..10*n;
     tablou=array[indice] of Integer;
     sir=array[1..n] of Litere;
var a:tablou; b:sir;
```

Există posibilitatea de a declara un tablou descriind structura în secțiunea **var**, introducând astfel un tip *anonim*:

```
var a:array[1..10] of Integer;
```

Menționăm că două tipuri anonime se consideră diferite, chiar dacă, de fapt, descriu aceeaşi structură de date:

```
var a:array[1..10] of Integer;
    b:array[1..10] of Integer;
```

În acest caz `a` şi `b` vor avea tipuri diferite, în concluzie, instrucțiunea de atribuire `a:=b;` nu este permisă. În schimb, datorită faptului că cele două tablouri au acelaşi tip de bază, instrucțiunea `a[1]:=b[3]` este corectă.

6.4. Citirea tablourilor unidimensionale

O variabilă de tip tablou nu poate fi citită în Pascal; această operație se realizează citind tabloul element cu element, precizând elementele prin indicii lor:

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

Dacă nu cunoaştem dimensiunea exactă a tabloului, dar cunoaştem un număr care exprimă valoarea maximă posibilă a acestuia, atunci vom proceda după cum urmează:

```
Algoritm CitireTablou:
  n ← 0
  cât timp nu urmează marca de sfârşit de fişier execută:
    n ← n + 1                          { dacă primul indice al tabloului este 1 }
    citeşte x[n]                                          { citim elementele }
  sfârşit cât timp                     { în n avem dimensiunea tabloului citit }
sfârşit algoritm
```

6.5. Afişarea tablourilor unidimensionale

O variabilă de tip tablou nu poate fi afişată în Pascal; această operaţie se realizează scriind pe suportul de ieşire tabloul element cu element, precizând elementele prin indicii lor:

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

Constante de tip tablou

În Pascal tablourile se pot iniţializa în secţiunea de declaraţii. Cu toate că ele se declară cu ajutorul cuvântului **const**, elementele unui tablou declarat astfel îşi pot schimba valorile. Acestea se declară în modul următor:

type *tip_tablou*=**array**[*tip_indice*] **of** *tip_bază*;
const *nume_tablou*: *tip_tablou*=(*constanta*$_1$, *constanta*$_2$, ...);
unde lista *constanta*$_1$, *constanta*$_2$, ... are exact atâtea constante de tipul *tip_bază* câte elemente are tipul *tip_indice*.

Exemplu

```
type sir=array[1..3] of Char;
const A:sir=('a','b','c');
```

6.6. Prelucrări simple pe tablouri

Problemele rezolvate până acum, în marea lor majoritate, pot fi abordate şi într-o manieră care presupune citirea tuturor datelor înainte de a fi prelucrate, ca atare se impune păstrarea lor în memorie. Acest lucru se realizează reținând aceste valori într-un *tablou unidimensional* (şir). Totuşi, în situația în care numărul datelor de prelucrat este mare, trebuie mai întâi să analizăm soluția aleasă pentru structurarea datelor, deoarece este posibil ca acestea să nu aibă suficient spațiu disponibil în memorie. De asemenea, atragem atenția că dimensiunea memoriei interne, disponibilă pentru datele care urmează să fie prelucrate de un anumit program, diferă de la un mediu de programare la altul.

Se recomandă memorarea datelor în tablouri, dacă astfel creşte lizibilitatea programului, dar mai ales atunci când este nevoie „atingerea" elementelor de mai multe ori pe parcursul prelucrării. Dacă nu lucrăm cu tablouri, aceste rezolvări ar putea necesita deschiderea fişierului de intrare de foarte multe ori şi citirea repetată a conținutului, ceea ce conduce la creşterea timpului de execuție.

În prima fază a analizei unei probleme de algoritmică (programare), de regulă, căutăm să stabilim *clasa* de probleme în care ar putea fi încadrată aceasta. În cele ce urmează vom trata subprobleme clasice care prelucrează şiruri.

6.6.1. Sume şi produse

Într-o clasă importantă de probleme cu care ne întâlnim frecvent, se cere determinarea *unui singur rezultat* pe baza unui *şir* de date de intrare. Este vorba de probleme în care trebuie să aplicăm un operator şi să determinăm o sumă sau un produs. Pentru aceste operații, mai întâi vom stabili elementul neutru, astfel încât operația efectuată asupra unui element y, ales arbitrar, să conducă la y.

În următorii algoritmi, prezentați în pseudocod, pentru a accentua prelucrarea propriu-zisă, nu vom explicita operațiile de citire a datelor de intrare şi de scriere a rezultatelor. Presupunem în continuare că, dacă nu va fi menționat altfel, se prelucrează un tablou unidimensional x având dimensiunea n şi elemente numere întregi.

Observație

După ce se vor dobândi cunoştințele privind subprogramele, aceşti algoritmi se vor putea implementa sub formă de funcții sau (în Pascal) proceduri şi datele primite spre prelucrare, respectiv cele obținute şi transmise celorlalte unități ale programului se vor menționa în lista de parametri.

În algoritmul cu care calculăm sume (sau produse) rezultatul obținut după prelucrarea elementelor şirului dat x (de dimensiune n) îl obținem în variabila *rezultat*, iar operația aplicată este notată cu *op*.

```
Algoritm Prelucrare_şir:
  citire date
  rezultat ← element nul
  pentru i=1,n execută:                    { prelucrăm toate elementele }
    rezultat ← rezultat op x[i]
  sfârşit pentru
  afişare rezultat
sfârşit algoritm
```

6.6.2. 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.

Într-o primă variantă putem dezvolta algoritmul `Prelucrare_şir`. În acest algoritm rezultatul se păstrează în variabila *găsit*, a cărei valoare de adevăr constituie răspunsul la întrebarea pusă în problemă.

```
Algoritm Decizie_1:
  citire date
  găsit ← fals
  pentru i=1,n execută:
    găsit ← găsit sau x[i] are proprietatea căutată
  sfârşit pentru
  afişare rezultat
sfârşit algoritm
```

Să observăm că dacă, la un moment dat, valoarea variabilei `găsit` devine *adevărat*, valoarea ei nu se va mai schimba până la sfârşitul executării algoritmului. În concluzie, vom modifica, pe baza acestei observaţii, algoritmul de mai sus.

În primul rând vom schimba structura repetitivă de tipul **`pentru`** într-o structură cu număr necunoscut de paşi, pentru a putea întrerupe execuţia în momentul în care am obţinut răspunsul la întrebare. Evident, dacă nici un element al şirului analizat nu are proprietatea căutată, se va parcurge întreg şirul.

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_2:
  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
```

Să analizăm acum problemele din cealaltă categorie, mai exact, cele în care se verifică fiecare element al şirului dat. Acestea trebuie să aibă o aceeaşi proprietate. Această caracteristică a şirului se poate formula şi în felul următor: *în şir nu există nici un element care nu are proprietatea cerută*. Transformând algoritmul `Decizie_2` prin aplicarea negaţiei în două locuri, obţinem algoritmul de rezolvare pentru această clasă de probleme. Deoarece semnificaţia deciziei se referă la toate elementele şirului, înlocuim variabila `găsit` cu `toate`.

```
Algoritm Decizie_3:
  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
```

În momentul codificării algoritmilor de tip `Decizie` pot apărea diverse probleme. În unele limbaje de programare o expresie logică formată din subexpresii între care apar operatorii logici **şi**, respectiv **sau**, se evaluează în întregime, dar există şi limbaje în care, în funcţie de operator şi valoarea *primei* subexpresii, evaluarea se întrerupe după stabilirea valorii primei subexpresii. De exemplu, dacă valoarea primei subexpresii este *adevărat* şi urmează operatorul **sau**, cea de-a doua subexpresie nu se mai evaluează, deoarece indiferent de valoarea acestuia, rezultatul final va fi *adevărat* (*adevărat* **sau** *adevărat* este *adevărat* şi *adevărat* **sau** *fals* este tot *adevărat*). De asemenea, dacă valoarea de adevăr a primei subexpresii este *fals* şi urmează operatorul **şi**, cea de-a doua subexpresie nu se mai evaluează, deoarece indiferent de valoarea acesteia, rezultatul final va fi *fals* (*fals* **şi** *adevărat* este *fals*, respectiv *fals* **şi** *fals* este tot *fals*).

În cazul acelor limbaje de programare care, în schimb, nu întrerup evaluarea acestor expresii logice, sau dacă programatorul nu alege atent ordinea subexpresiilor în cazul primei categorii de limbaje de programare, codificarea deciziei, conform algoritmilor prezentaţi, pot genera erori neplăcute. Fie expresia logică
`(i ≤ n)` **`şi`** `(x[i]` *are proprietatea căutată*`)`
din structura repetitivă **`cât timp`** din algoritmul `Decizie_3`. Dacă $i > n$, în urma evaluării subexpresiei `(i ≤ n)` obţinem *fals*, dar se evaluează şi subexpresia `(x[i]` *are proprietatea căutată*`)` în condiţiile în care ştim deja că $i > n$, deci se va verifica un element `x[i]` care nu există în şirul dat!

În concluzie, în implementarea algoritmilor de tipul celor prezentaţi trebuie să căutăm soluţii prin care să evităm producerea unor erori în timpul executării programului sau obţinerea unor rezultate incorecte. În cele ce urmează propunem câteva soluţii pentru evitarea acestei „capcane”:

- **a.** putem crea un al $(n + 1)$-lea element fictiv (numit frecvent *santinelă*);
- **b.** putem opri reluarea executării nucleului structurii repetitive cu *un pas mai repede*, urmând să analizăm motivele ieşirii din ciclu;
- **c.** putem despărţi subexpresiile, urmând ca pe cea de-a doua s-o evaluăm doar atunci când acest lucru este necesar;
- **d.** putem să evităm folosirea, în cea de-a doua subexpresie, a valorii `x[i]`; acest lucru este posibil dacă introducem o variabilă logică pentru a reţine valoarea de adevăr a subexpresiei `(x[i]` *are proprietatea căutată*`)` de la pasul precedent, astfel permiţând ieşirea din structura repetitivă în momentul în care avem confirmarea proprietăţii, sau după parcurgerea întregului şir, în cazul în care acest lucru nu a avut loc.

6.6.3. Selecţia

Soluţiile problemelor rezolvate cu algoritmi de tipul `Decizie` verifică o anumită proprietate într-un şir de date şi comunică prezenţa sau absenţa ei. Analizând mai atent rezolvările, constatăm că am putea fructifica algoritmul, afişând în plus informaţii cum ar fi *poziţia* pe care se află elementul cu proprietatea cerută.

Să presupunem că enunţul garantează apartenenţa a cel puţin unui astfel de element. Rezultă că trebuie doar 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.

Nu vom verifica dacă valoarea contorului a ajuns la dimensiunea şirului dat, tocmai datorită faptului că enunţul garantează existenţa în şir a valorii căutate. Elementul cu proprietatea cerută va fi găsit cel târziu atunci când prelucrăm ultimul element.

```
Algoritm Selecţia:
  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
```

6.6.4. Căutarea (secvenţială)

Acum vom presupune că enunţul ***nu*** garantează prezenţa a cel puţin unui element având proprietatea cerută în şirul dat. În acest tip de problemă cele două cerinţe din ultimele două clase de probleme se combină, respectiv, după aflarea răspunsului la întrebarea „există?”, afişăm, de exemplu, poziţia pe care se află elementul căutat.

Evident, modelul de rezolvare se bazează pe modelele prezentate până acum. Variabilele folosite au semnificaţiile din algoritmii precedenţi.

```
Algoritm Căutare:
  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        { dacă am părăsit ciclul înainte ca i să devină mai mare }
                                { decât n, înseamnă că am găsit elementul căutat }
  afişare rezultat
sfârşit algoritm
```

Se observă că soluţia porneşte cu modelul algoritmului `Decizie` care se completează cu determinarea numărului de ordine conform algoritmului `Selecţie`.

Dacă într-o problemă de căutare trebuie să regăsim toate elementele având o proprietate dată, evident vom parcurge şirul până la capăt, ceea ce înseamnă că vom lucra cu o structură repetitivă de tip **`pentru`**.

Dacă, după căutare, se cere eliminarea elementului respectiv din şirul dat, în funcţie de specificaţiile din enunţ, vom alege una din următoarele posibilităţi:

- dacă se cere eliminarea elementului şi nu contează dacă se schimbă ordinea, copiem ultimul element peste cel care trebuie eliminat şi scurtăm şirul cu 1;
- dacă eliminarea este „temporară”, vom păstra un şir cu valori logice în care valoarea de adevăr a elementului corespunzător celui eliminat va fi modificat după caz;

- dacă se cere eliminarea elementului şi se impune păstrarea ordinii şirului, vom translata elementele în aşa fel încât cel care trebuie eliminat să se suprascrie cu următorul element din şir; după terminarea translatării scurtăm şirul cu 1;
- există şi posibilitatea creării unui şir nou în care se vor copia doar elementele care nu trebuie eliminate, urmând ca în final variabila de tip tablou care păstrează şirul iniţial să se suprascrie cu variabila de tip tablou nou creat.

O căutare specială se referă la depistarea *dublurilor* într-un şir. De regulă, o astfel de prelucrare se finalizează fie cu un şir care conţine elemente distincte (după eliminarea dublurilor), fie cu unul care conţine dublurile. Prima este echivalentă cu transformarea şirului în mulţime şi va fi tratată la sfârşitul acestui capitol.

6.6.5. Numărarea

În această clasă de probleme se cere *numărul* elementelor din şir având o anumită proprietate. Enunţurile nu garantează că există cu siguranţă un astfel de element, deci soluţia furnizată poate fi şi 0.

Având în vedere că fiecare element trebuie verificat (este posibil ca oricare dintre elemente să aibă proprietatea cerută), 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
```

6.6.6. Selectarea elementului maxim

Problema determinării minimului, respectiv maximului am mai tratat-o în capitolul 2. Rezolvările acestor probleme vor fi asemănătoare, deoarece în acestea se cere stabilirea unei valori care este fie cea mai mare, fie cea mai mică în *şirul* dat. Bineînţeles, pot exista mai multe astfel de valori egale cu valoarea minimă sau maximă şi este posibil ca în anumite probleme să trebuiască să le numărăm.

Algoritmul care rezolvă această problemă (în termeni generali) va lucra cu o buclă de tip **pentru**, deoarece fiecare element trebuie prelucrat. Valoarea elementului maxim se reţine în variabila *max*.

```
Algoritm Maxim:
  citire date
  max ← x[1]
  pentru i=2,n execută:
    dacă max < x[i] atunci
      max ← x[i]
    sfârşit dacă
  sfârşit pentru
  afişare rezultat
sfârşit algoritm
```

Reamintim că se recomandă inițializarea maximului (minimului) cu o valoare existentă în şir, şi, deoarece, de regulă, prelucrăm şirurile începând cu primul element, cel mai firesc este ca în inițializarea respectivă să folosim această valoare.

Utilizând acest algoritm vom obține indicele *primului* element având valoarea maximă (minimă). Dacă ni s-ar cere ultimul astfel de element, atunci ar fi suficient să schimbăm operatorul relațional „<” în „≤”.

În continuare prezentăm o alternativă interesantă care poate fi utilă în cazul unui şir în care un element ocupă un spațiu relativ mare de memorie, respectiv, dacă se cere şi poziția elementului maxim. În loc să se lucreze cu variabila *max*, având acelaşi tip ca elementele din şir, se va păstra doar indicele său (*ind_max*). Pentru afişarea valorii maxime, se va specifica `x[ind_max]`.

```
Algoritm Maxim_optimizat:
  citire date
  ind_max ← 1          { ind_max este inițializat cu indicele primului element }
  pentru i=2,n execută:
    dacă x[ind_max] < x[i] atunci
      ind_max ← i                          { indicele elementului maxim }
    sfârşit dacă
  sfârşit pentru
  afişare rezultate
sfârşit algoritm
```

6.6.7. Selectarea elementelor

Am văzut că algoritmul `Selecție` determină poziția unui singur element având proprietatea dată. Dacă dorim să reținem toate pozițiile pe care se află elemente având acea proprietate, vom aplica un algoritm de tip `Selectare` care, pe de o parte seamănă cu *căutarea liniară*, pe de altă parte cu *numărarea*. 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
```

Se observă că pentru păstrarea rezultatului am avut nevoie de un tablou *nou* în cazul căruia nu cunoaştem exact câte elemente va cuprinde. Dar, din moment ce nu este exclus ca fiecare element al şirului dat să aibă proprietatea dată, acesta va fi declarat având atâtea elemente câte are şirul dat. Modelul de mai sus „adună" indicii elementelor având proprietatea precizată în vectorul *y*, 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 folosi chiar şirul dat. Ideea este de a suprascrie elementele de la începutul şirului dat cu cele selectate. Variabila *buc* reţine lungimea acestui şir „nou". Pseudocodul acestui algoritm este următorul:

```
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
```

Se poate observa că în caz extrem, fiecare element are proprietatea cerută, ceea ce va conduce la un şir „nou” *x*, identic cu cel dat.

Dacă se cere păstrarea elementelor care *nu* au această proprietate, vom nega condiția din **dacă**. Dacă vrem să păstrăm aceste elemente pe pozițiile originale în şirul dat, putem anula prezența acestor elemente, suprascriindu-le cu o valoare specială, aleasă în acest scop. Astfel, vom avea un algoritm care realizează o *selectare pe loc*. Această soluție va fi avantajoasă doar dacă „evitarea” prelucrării acestor elemente, având valoarea specială, va fi mai simplu de realizat decât verificarea proprietății.

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

6.6.8. Partiționarea unui şir în două subşiruri

În secțiunea precedentă am prezentat mai mulți algoritmi cu care rezolvăm probleme în care se dă un şir şi se cere selectarea mai multor elemente, având o aceeaşi proprietate dată. Se pune întrebarea ce se întâmplă cu elementele neselectate?

Evident, vor fi probleme în care se va cere descompunerea unui şir în două sau mai multe subşiruri.

Prima particularitate a acestor probleme constă în faptul că *se dă un şir* şi *se cer mai multe*. Practic, va trebui să efectuăm, una după alta, mai multe selectări ținând cont de proprietățile precizate. 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. Rezultă că algoritmul constă din aplicarea repetată a *partiționării* în două şiruri a şirului dat. Dacă partiționarea se face în două subşiruri, este clar că elementele neselecționate pentru primul şir vor forma pe cel de al doilea.

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. Este posibil ca toate elementele să fie selectate în primul şir sau acesta să nu aibă nici un element.

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

```
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 spaţiul risipit cu o astfel de soluţie, putem utiliza un singur şir. Elementele selecţionate le reţinem 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 faţă de poziţiile elementelor sale în şirul dat.

```
Algoritm Partiţionare_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 poziţie }
    altfel                       { elementele care nu au proprietatea dată }
      bucz ← bucz + 1
      y[n-bucy+1] ← x[i]     { le aşezăm tot în y, începând cu ultima poziţie }
    sfârşit dacă
  sfârşit pentru
  afişare rezultate
sfârşit algoritm
```

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

Ne putem imagina o soluție care ar folosi un algoritm conform căruia am avansa în şir până la descoperirea primului element care *nu* are proprietatea cerută şi l-am interschimba cu primul care *are* proprietatea cerută.

Această soluție, nefiind suficient de eficientă, propunem următorul algoritm: 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
```

6.6.9. Mulţimi reprezentate cu şiruri

Deocamdată ne vom baza pe cunoştinţele privind mulţimile, dobândite la disciplina matematică. Se ştie că o mulţime fie este vidă, fie conţine un număr oarecare de obiecte. În acest sens, dacă o mulţime este reprezentată cu ajutorul unui şir, elementele sale trebuie să fie distincte (un acelaşi „obiect" este prezent o singură dată).

A. Stabilirea proprietăţii de „mulţime"

Ne punem întrebarea: cum stabilim că, de exemplu, un şir de numere reale constituie sau nu o mulţime? Altfel spus, cum verificăm dacă elementele unui şir dat sunt distincte?

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

B. Eliminarea dublurilor

Transformarea unui şir, astfel încât să devină mulţime, înseamnă eliminarea dublurilor. Trebuie să fim atenţi, 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 Mulţime_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

Am văzut cum se procedează atunci când se dă un şir şi trebuie create mai multe şiruri pe baza unei proprietăți care permite partiționarea şirului. Evident, se poate pune întrebarea cum procedăm când se dau mai multe şiruri şi trebuie să creăm un singur şir care să conțină intersecția lor. Aici 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** (vom învăța despre acest tip mai târziu), care poate simplifica mult efectuarea acestei operații, dacă tipul de bază şi domeniul valorilor elementelor permit declararea datelor ca fiind de acest tip. Acum, ne-am propus să tratăm problema enunțată în condiții generale, independent de limbajul de programare ales, doar din punct de vedere algoritmic. De asemenea, presupunem că fiecare şir conține 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 intersecție *z* va fi *buc*.

```
Algoritm Intersecție:
  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 mulțimi implementate cu ajutorul şirurilor, trebuie să tratăm şi problema reuniunii.

Semnificația variabilelor în algoritmul următor este acelaşi ca mai înainte. În rezolvare, mai întâi vom copia toate elementele din şirul *x* în şirul *z*, inițializând 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
```

6.7. Implementări sugerate

Pentru a vă familiariza cu modul în care se abordează problemele în care se prelucrează tablouri unidimensionale, vă sugerăm să implementați algoritmi pentru:

1. determinarea sumei elementelor unui şir;
2. determinarea minimului şi maximului unui şir;
3. verificarea existenței a cel puțin unui element având o proprietate dată dintr-un şir;
4. căutarea liniară a unui element într-un şir;
5. căutarea liniară a unui element într-un şir şi determinarea tuturor aparițiilor;
6. căutarea liniară a unui element având o proprietate dată într-un şir şi eliminarea lui

7. căutarea liniară a tuturor elementelor având o proprietate dată într-un şir şi eliminarea lor;
8. citirea a două şiruri (cu număr diferit de elemente) şi determinarea şirului care are suma elementelor maximă/minimă;
9. citirea a *n* şiruri (cu număr diferit de elemente) şi determinarea şirului care are suma elementelor maximă/minimă;
10. citirea a *n* şiruri (cu număr diferit de elemente) şi identificarea şirului care are cele mai multe elemente;
11. verificarea proprietăţii de mulţime a unui şir;
12. determinarea dublurilor într-un şir şi eliminarea lor;
13. determinarea reuniunii a două mulţimi, reprezentate cu şiruri;
14. determinarea intersecţiei a două mulţimi, reprezentate cu şiruri;
15. eliminarea unui element dintr-un şir cu păstrarea ordinii;
16. determinarea dintr-un şir a două subşiruri pe baza unei proprietăţi date (creând două subşiruri noi, apoi creând subşiruri noi, „pe loc").

6.8. Probleme propuse

6.8.1. Sumă

Se consideră *p* numere întregi. Să se calculeze suma:

$$S = 1 + \frac{1}{n_1} + \frac{1}{n_2} + ... + \frac{1}{n_p}.$$

Dacă există $i \in \{1, 2, \ldots, p\}$ astfel încât $n_i = 0$, termenul corespunzător nu se calculează. Totodată să se numere valorile egale cu 0 printre numerele date.

Date de intrare

Pe prima linie a fişierului **`SUMA.IN`** se află scris numărul natural *p*. Pe următoarele *p* linii se află câte un număr întreg.

Date de ieşire

Pe prima linie a fişierului de ieşire **`SUMA.OUT`** se va scrie suma calculată, iar pe a doua numărul numerelor egale cu 0.

Restricţii şi precizări

- $1 \le p \le 100$;
- $-10000 \le n_i \le 10000$, $i = 1, 2, ..., p$.

Exemplu

SUMA.IN	**SUMA.OUT**
5 1 0 1 1 1	5.00 1

6.8.2. Perechi

Se consideră *n* numere întregi. Să se afişeze numerele de ordine ale perechilor de elemente formate din două numere întregi egale.

Date de intrare

Pe prima linie a fişierului **PERECHI.IN** se află numărul natural *n*. Pe fiecare linie din următoarele *n* linii se află un număr întreg.

Date de ieşire

În fişierul de ieşire **PERECHI.OUT** se vor scrie pe câte o linie perechile de numere de ordine, separate printr-un spațiu, ale elementelor consecutive egale ale şirului.

Restricţii şi precizări

- $1 \le n \le 100$;
- $-10000 \le nr_i \le 10000$, $i = 1, 2, ..., n$.

Exemplu

PERECHI.IN	**PERECHI.OUT**
1 2 -3 -3	3 4

6.8.3. Raport

Se consideră *n* numere întregi. Să se determine raportul dintre numărul numerelor pare şi numărul numerelor impare date.

Date de intrare

Pe prima linie a fişierului de intrare **RAPORT.IN** se află un număr natural *n*. Următoarele *n* linii conțin fiecare câte un număr întreg.

Date de ieşire
Fişierul de ieşire **RAPORT.OUT** va conţine o singură linie pe care se va scrie raportul cerut (cu două zecimale exacte). Dacă în fişierul de intrare nu există nici un număr impar, în fişierul de ieşire se va scrie mesajul: `'Nu exista numere impare!'`

Restricţii şi precizări
- $1 \le n \le 100$;
- $-10000 \le nr_i \le 10000$, $i = 1, 2, ..., n$.

Exemplu

RAPORT.IN
```
5
1
2
3
4
5
```

RAPORT.OUT
```
0.67
```

6.8.4. Temperaturi

Pe durata lunii iunie s-a măsurat temperatura apei Mării Negre şi valorile s-au scris într-un fişier. Stabiliţi dacă apa mării s-a încălzit încontinuu sau nu.

Date de intrare
În fişierul de intrare **TEMP.IN** se află 30 de numere reale, reprezentând temperatura apei.

Date de ieşire
Dacă temperaturile formează un şir monoton crescător, în fişierul de ieşire **TEMP.OUT** se va scrie `DA`, altfel se va scrie `NU`.

Exemplu

TEMP.IN
```
19.8
22.5
21.3
...
```

PRIM.OUT
```
NU
```

Explicaţie
Aici nu sunt menţionate toate cele 30 de temperaturi, din motive de spaţiu, dar este clar că temperaturile nu au crescut monoton.

6.8.5. Întrebarea şefului

Recent s-a realizat recensământul populaţiei din România. Un angajat curios a creat un fişier în care a păstrat doar anul naşterii persoanelor care locuiesc în comuna sa natală. Şeful lui, observând că pierde timpul cu lucruri neimportante, îl acuză că fişierul res-

pectiv nu este bun la nimic şi pentru a fi convingător îl întreabă dacă poate să spună câte persoane născute anterior anului 1989 trăiesc în comuna lui?

Scrieți un program eficient care răspunde la întrebarea şefului şi în plus, creează un fişier nou în care se vor afla anii de naştere mai mici decât 1989.

Date de intrare
Pe prima linie a fişierului de intrare **SEF.IN** se află numărul *n*, reprezentând numărul locuitorilor din comuna natală a angajatului. Pe următoarele *n* linii se află numere naturale, reprezentând anul naşterii al locuitorilor.

Date de ieşire
Pe prima linie a fişierului de ieşire **SEF.OUT** se va scrie numărul persoanelor născute anterior anului 1989. Pe următoarele linii se vor afla anii de naştere respectivi.

Restricții şi precizări
- $5 \leq n \leq 30000$;
- dacă nu se găseşte nici o persoană născută anterior anului 1989, în fişier se va scrie mesajul `'NU'`;
- fişierul de ieşire nu trebuie să conțină numere distincte.

Exemplu

SEF.IN	**SEF.OUT**
5	3
1920	1920
2000	1975
1989	1945
1975	
1945	

6.8.6. Diriginta

La sfârşitul anului, diriginta clasei trebuie să completeze un formular în care este întrebată câți elevi din clasă au media finală între 9 şi 10 inclusiv, câți au media între 8 şi 9 inclusiv etc. Dar cum nu îi place să lucreze cu statisticile, vă roagă să scrieți un program care îi va lista datele necesare. În plus, ea vă solicită ca lista să conțină şi mediile din fiecare categorie.

Date de intrare
Pe prima linie din fişierul de intrare **DIRIG.IN** se află un număr natural *n*, reprezentând numărul elevilor din clasă. Pe următoarele *n* linii se află, în ordinea din catalog, mediile generale ale lor. Mediile sunt numere reale scrise cu doua zecimale exacte.

Date de ieşire
În fişierul de ieşire **DIRIG.OUT** veți scrie statistica sub următoarea formă:

- Pe prima linie se va scrie textul: `'Medii in intervalul (9,10]: '`, urmat de nr_1, unde nr_1 reprezintă numărul elevilor având media generală între 9 şi 10 inclusiv. Pe următoarea linie veți scrie mediile elevilor, despărțite prin câte un spațiu, în ordinea din fişierul de intrare, aparținând categoriei respective.
- Veți proceda la fel în cazul fiecărei categorii: medii generale între 8 şi 9 inclusiv, între 7 şi 8 inclusiv, între 6 şi 7 inclusiv, respectiv între 5 şi 6 inclusiv.
- Dacă numărul mediilor dintr-o categorie este 0, linia corespunzătoare mediei rămâne vidă în fişier.

Restricții şi precizări
- $5 < medii \leq 10$;
- în această clasă nu sunt corigenți.

Exemplu

DIRIG.IN
```
5
6.52
9.67
9.34
5.25
9.00
```

DIRIG.OUT
```
Medii in intervalul (9,10]: 2
 9.34  9.67
Medii in intervalul (8,9]: 1
 9.00
Medii in intervalul (7,8]: 0

Medii in intervalul (6,7]: 1
 6.52
Medii in intervalul (5,6]: 1
 5.25
```

6.8.7. Generare şir

Se consideră următorul şir, construit astfel încât fiecare element al lui, cu excepția primului, se obține din cel precedent: 1, 11, 21, 1211, 111221, ... Regula de generare a termenilor este următoarea:

- se numără de la stânga la dreapta câte cifre există în termenul precedent;
- în termenul nou se trece, pentru fiecare cifră, numărul de apariții a cifrei şi cifra
  Să se determine al n-lea element din şir.

Date de intrare
Numărul natural n se citeşte din fişierul de intrare **GENERARE.IN**.

Date de ieşire
Al n-lea termen din şir se va scrie în fişierul **GENERARE.OUT**.

Restricţii şi precizări
- $n \leq 20$.

Exemplu

GENERARE.IN	**GENERARE.OUT**
4	1211

6.8.8. Suprapunere

Se consideră un tablou unidimensional *A* care conţine *n* numere naturale şi un alt tablou *B* (având elemente egale cu 1) de lungime *k*, unde $k \leq n$. Prin suprapunerea tabloului *B* peste tabloul *A* elementele acoperite din *A* îşi micşorează valoarea cu o unitate. Să se stabilească dacă este posibil ca prin aplicări succesive ale tabloului *B* peste tabloul *A* să se obţină în *A* numai valori egale cu 0.

Date de intrare

Datele de intrare se citesc din fişierul de intrare **SUPRA.IN**, în care pe prima linie este scris numărul natural *n*, pe a doua linie este scris numărul natural *k*, iar pe a treia linie sunt cele *n* elemente din tabloul *A*.

Date de ieşire

Datele de ieşire se vor scrie în fişierul **SUPRA.OUT**. Dacă problema nu admite soluţie în fişierul de ieşire se va scrie mesajul `'Imposibil'`, altfel se va scrie o succesiune de indici ai şirului *A* peste care se aplică poziţia de început a tabloului *B*.

Restricţii şi precizări
- $2 \leq n \leq 1000$;
- $1 \leq k \leq n$;
- $0 \leq a_i \leq 100$;
- o suprapunere este posibilă dacă întreg tabloul *B* „încape” în *A*.

Exemple

SUPRA.IN	**SUPRA.OUT**
8 4 1 2 3 4 0 0 0 0	Imposibil

SUPRA.IN	**SUPRA.OUT**
20 4 2 2 2 3 1 1 1 0 0 0 2 2 2 3 1 1 1 0 0 0	1 1 4 11 11 14

6.9. Soluțiile problemelor

6.9.1. Suma

Se citeşte dimensiunea *p* a şirului. Suma *s* se inițializează în această problemă cu valoarea 1, conform expresiei date, iar contorul zerourilor cu 0.

Fiecare număr din şir se verifică pentru a putea decide dacă participă la sumă sau, fiind 0 se măreşte contorul corespunzător valorilor nule. Dacă numărul citit este nul, atunci nu participă la sumă, deoarece nu putem efectua împărțire cu 0. În final se scrie valoarea sumei în fişierul de ieşire, precum şi numărul elementelor egale cu 0.

```
Algoritm Suma:
  citeşte p
  pentru i=1,p execută:
    citeşte n[i]
  sfârşit pentru
  zerouri ← 0                                   { numărul zerourilor }
  s ← 1                          { iniţializăm suma cu 1, conform expresiei date }
  pentru i=1,p execută:
    citeşte n[i]
    dacă n[i] ≠ 0 atunci
      s ← s + 1/n[i]
    altfel
      zerouri ← zerouri + 1
    sfârşit dacă
  sfârşit pentru
  scrie s, zerouri
sfârşit algoritm
```

6.9.2. Perechi

După citirea dimensiunii şirului şi a elementelor sale, prelucrarea se realizează cu o structură repetitivă de tip **pentru**. În această instrucțiune trebuie să fim atenți la valoarea finală a contorului pentru a nu compara ultimul element cu un al $n + 1$-lea, care nu există. Dacă vom avea un contor care porneşte cu valoarea 1, atunci valoarea finală va fi $n - 1$, dacă pornim cu 2, valoarea finală va fi n. În primul caz vom compara elementele de indice i şi $i + 1$, în al doilea caz comparăm elementele de indice $i - 1$ şi i.

```
Algoritm Perechi:
  citeşte n                                     { dimensiunea şirului }
  pentru i=1,n execută:
    citeşte x[i]
  sfârşit pentru
```

```
  pentru i=1,n-1 execută:      { echivalent cu pentru i=2,n execută: }
    dacă x[i]=x[i+1] atunci          { dacă x[i-1]=x[i] atunci... }
      scrie i,' ',i+1
    sfârşit dacă
  sfârşit pentru
sfârşit algoritm
```

6.9.3. Raport

Notăm cu *nrpare* numărul numerelor pare şi cu *nrimpare* numărul numerelor impare găsite. Ambele variabile se iniţializează cu 0.

Urmează o prelucrare simplă a elementelor şirului, în care contorizăm numerele impare şi pare.

În final, anterior calculării şi afişării raportului trebuie să verificăm dacă am întâlnit cel puţin un număr impar pentru a nu risca să împărţim cu 0. În cazul în care nu a existat nici un număr impar, în fişier scriem mesajul corespunzător.

```
Algoritm Raport:
  citeşte n
  pentru i=1,n execută:
    citeşte a[i]
  sfârşit pentru
  nrpare ← 0
  nrimpare ← 0
  pentru i=1,n execută:
    dacă a[i] este impar atunci
      nrimpare ← nrimpare + 1
    altfel
      nrpare ← nrpare + 1
    sfârşit dacă
  sfârşit pentru
  dacă nrimpare > 0 atunci
    scrie nrpare/nrimpare
  altfel
    scrie 'Nu exista numere impare!'
  sfârşit dacă
sfârşit algoritm
```

6.9.4. Temperaturi

Vom rezolva problema implementând algoritmul `Decizie_3`. După citirea şirului de temperaturi vom compara elementele două câte două.

În algoritm nu vom citi dimensiunea, deoarece ştim că luna iunie are 30 de zile.

```
Algoritm Temperaturi:
  pentru i=1,30 execută:
    citeşte t[i]
  sfârşit pentru
  i ← 1
    { cât timp în a i-a zi temperatura este mai mică sau egală cu cea din a i + 1-a zi }
  cât timp (i < 30) şi (t[i] ≤ t[i+1]) execută:
    i ← i + 1                                   { trecem la următoarea zi }
  sfârşit cât timp
          { dacă valoarea lui i a atins valoarea 30, înseamnă că până acum toate }
                  { perechile succesive de temperaturi au respectat proprietatea }
  dacă i = 30 atunci
    scrie 'DA'
  altfel
    scrie 'NU'
  sfârşit dacă
sfârşit algoritm
```

6.9.5. Întrebarea şefului

Problema se rezolvă cu un algoritm de selectare. Deoarece în fişierul de ieşire mai întâi trebuie să scriem numărul persoanelor selectate, nu putem să-i scriem direct în fişier pe baza proprietăţii, ci trebuie să creăm şirul lor.

Vom citi şirul dat şi îl vom reţine integral în memorie. Pentru şirul numerelor selectate nu putem declara un alt şir, deoarece în memorie nu am avea loc şi pentru acesta. În concluzie, vom lucra „pe loc", adică vom păstra în acest şir original elementele care ne interesează. (Putem „strica" şirul citit, deoarece şirul tuturor anilor de naştere rămâne nealterat în fişierul de intrare.)

Păstrarea elementelor care ne interesează din şirul dat se poate realiza în mai multe feluri, dar deoarece ni s-a cerut un program rapid, nu vom efectua translatări.

```
Algoritm Selectare_3:
  citeşte n                                  { numărul locuitorilor din comună }
  pentru i=1,n execută:
    citeşte an[i]                                        { anii de naştere }
  sfârşit pentru
  nr ← 0              { deocamdată nu am găsit nici un an mai mic decât 1989 }
  pentru i=1,n execută:
    dacă an[i] < 1989 atunci
      nr ← nr +1                  { creşte numărul anilor mai mici decât 1989 }
      an[nr] ← an[i]   { peste elementul an[nr] se copiază valoarea an[i] }
    sfârşit dacă
  sfârşit pentru
```

```
  dacă nr = 0 atunci scrie 'NU'
  altfel scrie nr          { numărul locuitorilor născuți anterior lui 1989 }
  pentru i=1,nr execută:
    scrie an[i]                                          { anii de naştere }
  sfârşit pentru
sfârşit algoritm
```

6.9.6. Diriginta

Problema face parte din categoria acelora care se rezolvă cu algoritmul de partiționare. Vom alege modelul `Partiționare_3`, chiar dacă această problemă nu ridică probleme de spațiu de memorare, pentru a oferi un model de rezolvare pentru problemele în care metoda nu poate fi ocolită.

În algoritm vom avansa cu *primul* de la stânga la dreapta în căutarea mediilor care nu intră în categoria curentă şi cu ultim de la dreapta la stânga în căutarea mediilor care trebuie să intre în categoria curentă. Cu variabila *limita* definim categoria de medii care se caută, iar *buc* reprezintă indicele ultimului element care face parte din subşirul curent. În *buc* se va determina lungimea subşirului curent, necesar în afişare.

```
Algoritm Clasa:
  citeşte n                                    { numărul elevilor din clasă }
  pentru i=1,n execută:
    citeşte m[i]                                                { mediile }
  sfârşit pentru
  primul ← 1                                  { primul indice al şirului curent }
  ultim ← n                                   { ultimul indice al şirului curent }
  pentru limita:=9,5 execută:                                  { medii posibile }
    p ← primul      { variabilă auxiliară, reține primul indice al şirului curent }
    aux ← m[primul]                          { prima medie o punem deoparte }
    cât timp primul < ultim execută:
          { căutăm primul element având proprietatea de la dreapta spre stânga }
      cât timp (primul<ultim) şi nu (m[ultim]>limita) execută:
        ultim ← ultim - 1
              { m[ultim] are proprietatea cerută, îl scriem peste m[primul] }
      sfârşit cât timp
      dacă primul < ultim atunci
        m[primul] ← m[ultim]
        primul ← primul + 1
        { căutăm primul element care nu are proprietatea de la stânga la dreapta }
        cât timp (primul<ultim) şi  (m[primul]>limita) execută:
          primul ← primul + 1
        sfârşit cât timp
```

```
            { m[primul] nu are proprietatea cerută, îl scriem peste m[ultim] }
        dacă primul < n atunci
          m[ultim] ← m[primul]
          ultim ← ultim - 1          { avansăm de la sfârşit spre început }
        sfârşit dacă
      sfârşit dacă
    sfârşit cât timp
    m[primul] ← aux          { elementul pus în aux se pune pe locul „liber" }
                    { dacă elementul care a fost în aux are proprietatea cerută }
                                                { face parte din subşirul curent }
    dacă m[primul] > limita atunci
                  { indicele ultimului element care face parte din subşirul curent }
      buc ← primul
    altfel
        { m[primul] face parte din subşirul care se va partiţiona în continuare }
      buc ← primul - 1
    sfârşit dacă
    scrie buc - p + 1              { numărul elementelor din subşirul curent }
    pentru j=p,buc execută:
      scrie m[j]
    primul ← buc + 1 { indicele de început al subşirului care se va partiţiona }
    ultim ← n             { refacem variabila ultim pentru o nouă prelucrare }
  sfârşit cât timp
sfârşit algoritm
```

Această rezolvare respectă întocmai cerinţele problemei, respectiv mediile elevilor sunt afişate în ordinea din şirul dat. Dacă nu ar fi existat această cerinţă, şirul de medii putea fi ordonat.

6.9.7. Generare şir

Deoarece, regula de generare a elementelor s-a dat în enunţ, vom considera un exemplu şi vom descrie algoritmul de rezolvare.

Să presupunem că dorim afişarea celui de-al 6-lea termen al şirului. Generăm pe rând toţi termenii până la al 6-lea:

- $T_1 = 1$ conţine o singură cifră de 1, deci $T_2 = 11$.
- T_2 conţine două cifre de 1, avem $T_3 = 21$.
- Deoarece în T_3 avem o cifră de 2 şi una de 1, rezultă $T_4 = 1211$.
- În T_4 avem un 1, o cifră 2, apoi două cifre 1, deci $T_5 = 111221$.
- În T_6 vom avea $T_6 = 312211$, deoarece în $T5$ am avut 3 de 1, 2 de 2 şi un 1.

Vom lucra cu ajutorul a două *tablouri de cifre*. Unul dintre tablouri se foloseşte pentru a reține cifrele din care se formează prin alipire ultimul termen generat (*vechi*) şi celălalt pentru termenul care se generează pe baza precedentului (*nou*). Am convenit ca numărul de elemente pentru tabloul *vechi* să fie reținut în variabila *kvechi*, iar pentru şirul *nou* în *knou*. Vom lucra doar cu aceste două şiruri, deoarece nu dorim să păstrăm toate elementele şirului, având în vedere că se cere doar al *n*-lea termen. După construirea termenului *nou*, termenul *vechi* ia valoarea termenului *nou* precum şi *kvechi* se substituie cu *knou*, urmând să se genereze un alt termen *nou*.

```
Algoritm Generare:
  vechi[1] ← 1       { primul termen se inițializează şi devine imediat „vechi” }
  kvechi ← 1                                  { lungimea şirului acum este 1 }
  k ← 1                                       { contor pentru termenii şirului }
  cât timp k < n execută:
                    { algoritmul se termină, când am determinat al n-lea termen }
    knou ← 0               { deocamdată nu avem nici o cifră în noul termen }
    i ← 1
    cât timp i ≤ kvechi execută: { vom parcurge cifrele vechiului termen }
      cont ← 1      { în cont se numără toate elementele egale cu vechi[i] }
      cât timp (i < kvechi) şi (vechi[i] = vechi[i+1]) execută:
        cont ← cont + 1
        i ← i + 1
      sfârşit cât timp
      knou ← knou + 1                  { şirul va avea cu un termen mai mult }
      nou[knou] ← cont       { se trece în şirul nou numărul cifrelor curente }
      knou ← knou + 1                       { mai avem nevoie de o poziție }
      nou[knou] ← vechi[i]                        { pentru cifra curentă }
      i ← i + 1             { trecem la următoarea cifră din termenul vechi }
    sfârşit cât timp
    vechi ← nou
    kvechi ← knou
    k  ← k + 1
  sfârşit cât timp
  pentru i=1,knou execută:
    scrie nou[i]
  sfârşit pentru
sfârşit algoritm
```

6.9.8. Suprapunere

Pe parcursul algoritmului vom fi atenți ca printr-o suprapunere, (deci decrementare a valorilor elementelor succesive din *A*) să nu apară elemente negative.

Exemplu
$n = 11$, $k = 3$, iar tabloul A este:

A	1	1	1	0	1	2	2	1	0	0	0

Se realizează prima suprapunere, începând de la poziția 1:

A	**0**	**0**	**0**	0	1	2	2	1	0	0	0

În tabloul *rez* reținem prima poziție de suprapunere:

rez	1

Acum analizăm elementele tabloului A:
$a_1 = 0$, deci se încearcă suprapunere începând de la poziția 2
$a_2 = 0$, deci, nici aici nu este necesară suprapunere, se verifică poziția 3
$a_3 = 0$, nu necesită suprapunere
$a_4 = 0$, deci se încearcă suprapunere începând cu poziția 5
$a_5 = 1$, se realizează suprapunere începând de la poziția 5

A	0	0	0	0	**0**	**1**	**1**	1	0	0	0

În tabloul *rez* reținem a doua poziție de suprapunere:

rez	1	5

$a_6 > 0$, deci se realizează o suprapunere de la poziția 6

A	0	0	0	0	0	**0**	**0**	**0**	0	0	0

În tabloul *rez* reținem şi această poziție de suprapunere:

rez	1	5	6

În continuare, se încearcă suprapunere la poziția 7, 8 şi 9. Începând cu poziția 10 nu are sens să încercăm suprapunere, deoarece tabloul B nu se mai poate suprapune nefiind suficiente elemente în tabloul A.

Pe parcursul prelucrării se verifică dacă elementele tabloului A au rămas nenegative şi în cazul în care apare un număr negativ se întrerupe prelucrarea.

În final se verifică dacă şirul A mai are sau nu elemente nenule. Acestea pot fi doar la sfârşitul şirului, deoarece dacă şirul B nu poate fi suprapus, algoritmul se opreşte. Dacă toate elementele sunt nule, atunci se scriu elementele tabloului *rez* în fişierul de ieşire, altfel se scrie mesajul `'Imposibil'`.

Algoritmul care realizează suprapunerea este:

```
Algoritm Suprapunere:
   citirea datelor
   i ← 1
```

```
  p ← 0
  pozitiv ← adevărat              { iniţial toate numerele din A sunt nenegative }
                  { cât timp numerele din A sunt nenegaitive şi B „începe" în A }
  cât timp pozitiv şi (i ≤ n-k+1) execută:
              { dacă elementul curent este diferit de 0, vom realiza o suprapunere }
    dacă a[i] ≠ 0 atunci
      j ← 1
      p ← p + 1                          { vom avea un element nou în rez }
      cât timp pozitiv şi (j ≤ k) execută:
                        { scădem 1 din k elemente din A, începând cu a[i] }
        a[i+j-1] ← a[i+j-1] - 1
                       { verificăm dacă noile elemente din A sunt nenegative }
        dacă a[i+j-1] < 0 atunci
          pozitiv ← fals
        j ← j + 1
      sfârşit cât timp
      dacă pozitiv atunci
        rez[p] ← i                       { reţinem poziţia de suprapunere }
      sfârşit dacă
    sfârşit dacă
    dacă a[i] = 0 atunci    { dacă elementul curent are valoarea 0 avansăm }
      i ← i + 1
    sfârşit dacă    { altfel, încercăm o nouă suprapunere începând cu poziţia i }
  sfârşit cât timp
  dacă pozitiv atunci  { dacă prin suprapunere nu am obţinut număr negativ }
    i ← n
  { verificăm elementele de la sfârşitului şirului A, pentru a vedea dacă nu sunt nule }
    cât timp (a[i] = 0) şi (i ≥ 1) execută:
      i ← i - 1
    sfârşit cât timp
                     { dacă toate elementele din A au valoarea 0, avem rezultat }
    dacă i=0 atunci ok ← adevărat
             altfel ok ← fals
  sfârşit dacă
  dacă pozitiv şi ok atunci
    pentru i=1,p execută:
      scrie rez[i]
    sfârşit pentru
  altfel
    scrie 'Imposibil'
  sfârşit dacă
sfârşit algoritm
```