

## Algoritmi care lucreaza cu tablouri unidimensionale

13.11.2021

1. Implementati un algoritm care sa genereze o permutare circulara a unui sir  $x$  cu  $k$  pozitii stanga (24).

Fie un șir  $x$  cu  $n$  elemente numere naturale ( $3 \leq n \leq 10000$ ) și numărul natural  $k$  ( $1 \leq k < n$ ). Se cere generarea rezultatului fara a folosi un sir auxiliar, ci modificarea sirului  $x$  astfel incat in urma executiei algoritmului rezultatul permutarii sa poata fi accesat prin intermediul sirului dat ca si input.

De exemplu:

Pentru sirul:

1	2	3	4	5
---	---	---	---	---

Rezultatul permutarii cu 2 pozitii la stanga ar fi:

3	4	5	1	2
---	---	---	---	---

Pentru rezolvarea acestei probleme ni se ofera o varianta de pseudocod in ajutor – insa aceasta varianta are o mica problema, functioneaza doar in anumite cazuri. Trebuie sa gasim cazul general in care acest pseudocodul poate fi folosit cu success.

```
Subalgoritm permCirc(n, k, x)
  c ← k
  Pentru j = 1, c execută
    unde ← j
    nr ← x[unde]
    Pentru i = 1, n / c - 1 execută
      deUnde ← unde + k
      Dacă deUnde > n atunci
        deUnde ← deUnde - n
      SfDacă
      x[unde] ← x[deUnde]
      unde ← deUnde
    SfPentru
    x[unde] ← nr
  SfPentru
SfSubalgoritm
```

**CAZ I:**

Considerand  $k=2$ , pentru  $x$  ( $n=6$ ):

1	2	3	4	5	6
---	---	---	---	---	---

rezultatele intermediare ale fiecărei iteratii ar fi:

1.

3	2	5	4	1	6
---	---	---	---	---	---

2.

3	4	5	6	1	2
---	---	---	---	---	---

(rezultat final) - corect

**CAZ II:**

Considerand  $k=3$ , pentru  $x$  ( $n=8$ ):

1	2	3	4	5	6	7	8
---	---	---	---	---	---	---	---

rezultatele intermediare ale fiecărei iteratii ar fi:

1.

4	2	3	1	5	6	7	8
---	---	---	---	---	---	---	---

2.

4	5	3	1	2	6	7	8
---	---	---	---	---	---	---	---

3.

4	5	6	1	2	3	7	8
---	---	---	---	---	---	---	---

(rezultat final) - gresit

**CAZ III:**

Considerand  $k=3$ , pentru  $x$  ( $n=5$ ):

1	2	3	4	5
---	---	---	---	---

rezultatele intermediare ale fiecărei iteratii ar fi:

1.

1	2	3	4	5
---	---	---	---	---

2.

1	2	3	4	5
---	---	---	---	---

3.

1	2	3	4	5
---	---	---	---	---

(rezultat final) – gresit

#### CAZ IV:

Considerand  $k=4$ , pentru  $x (n=8)$ :

1	2	3	4	5	6	7	8
---	---	---	---	---	---	---	---

rezultatele intermediare ale fiecărei iteratii ar fi:

1.

5	2	3	4	1	6	7	8
---	---	---	---	---	---	---	---

2.

5	6	3	4	1	2	7	8
---	---	---	---	---	---	---	---

3.

5	6	7	4	1	2	3	8
---	---	---	---	---	---	---	---

4.

5	6	7	8	1	2	3	4
---	---	---	---	---	---	---	---

(rezultat final) - corect

Primul lucru care se poate observa din cazurile de mai sus este ca executia algoritmului cu un **P divizor a lui N** produce un rezultat **corect**.

Asta se intampla datorita conditiei  $n/k - 1$  prezenta in structura celui de-al doilea **pentru**, structura ce ne ajuta sa mutam la stanga din **k** in **k** pozitii elementele sirului, pornind de la o

poziție  $i$  ( $i \rightarrow k$ ) de  $n/k - 1$  ori. Faptul că  $n/k$  returnează partea întreagă a operației de împărțire, rezultă la ignorarea pozițiilor  $k$  ce se află în segmentul  $k * (n/k) \leq i < n$ .

2. Se consideră subalgoritmul `prelucreaza(v, k)`, unde  $v$  este un șir cu  $k$  numere naturale ( $1 \leq k \leq 1\,000$ ) (10).

```
Subalgoritm prelucreaza(v, k)
i ← 1, n ← 0
CâtTimp i ≤ k și vi ≠ 0 execută
    y ← vi, c ← 0
    CâtTimp y > 0 execută
        Dacă y MOD 10 > c atunci
            c ← y MOD 10
        SfDacă
        y ← y DIV 10
    SfcâtTimp
    n ← n * 10 + c
    i ← i + 1
SfcâtTimp
returnează n
SfSubalgoritm
```

Precizați pentru care valori ale lui  $v$  și  $k$  subalgoritmul returnează valoarea **928**.

Se poate observa din execuția algoritmului că prin structura primului **CâtTimp** se parcurge fiecare element al vectorului  $v$ , urmând ca prin execuția celui de-al doilea **CâtTimp** să iterăm fiecare cifră a elementului curent, salvându-se cifra cu valoarea maximă.

Cifra cu valoarea maximă rezultată din parcurgerea celui de-al doilea **CâtTimp** este adăugată la sfârșitul numărului  $n$ , număr ce reprezintă rezultatul execuției subalgoritmului **prelucreaza**.

De aceea am putea formaliza soluția algoritmului în una ce creează un număr  $n$  din cifrele maxime din  $k$  numere dintr-un vector  $v$ , ordinea cifrelor fiind dată de poziția numărului în vectorul  $v$ .

A.  $v = (194, 121, 782, 0)$  și  $k = 4$

$v[0] \rightarrow 194$ ,  $v[1] \rightarrow 121$ ,  $v[2] \rightarrow 782$ ,  $v[3] \rightarrow -$

$n = 928$

(rezultat final) - corect

B.  $v = (928)$  și  $k = 1$

$v[0] \rightarrow 928$

$n = 9$

(rezultat final) – gresit

C.  $v = (9, 2, 8, 0)$  și  $k = 4$

$v[0] \rightarrow 9$ ,  $v[1] \rightarrow 2$ ,  $v[2] \rightarrow 8$ ,  $v[3] \rightarrow -$

$n = 928$

(rezultat final) - corect

D.  $v = (8, 2, 9)$  și  $k = 3$

$v[0] \rightarrow 8$ ,  $v[1] \rightarrow 2$ ,  $v[2] \rightarrow 9$

(rezultat final) – gresit

3. Se consideră următorul program (12):

Varianta C	Varianta C++	Varianta Pascal
<pre>#include &lt;stdio.h&gt;  int prelVector(int v[], int *n) {     int s = 0; int i = 2;     while (i &lt;= *n) {         s = s + v[i] - v[i - 1];         if (v[i] == v[i - 1])             *n = *n - 1;         i++;     }     return s; }  int main(){     int v[8];     v[1] = 1; v[2] = 4; v[3] = 2;     v[4] = 3; v[5] = 3; v[6] = 10;     v[7] = 12;     int n = 7;     int rezultat = prelVector(v,     &amp;n);     printf("%d;%d", n, rezultat);     return 0; }</pre>	<pre>#include &lt;iostream&gt; using namespace std; int prelVector(int v[], int&amp;n) {     int s = 0; int i = 2;     while (i &lt;= n) {         s = s + v[i] - v[i - 1];         if (v[i] == v[i - 1])             n--;         i++;     }     return s; }  int main(){     int v[8];     v[1] = 1; v[2] = 4; v[3] = 2;     v[4] = 3; v[5] = 3; v[6] = 10;     v[7] = 12;     int n = 7;     int rezultat = prelVector(v, n);     cout &lt;&lt; n &lt;&lt;";" &lt;&lt; rezultat;     return 0; }</pre>	<pre>type vector=array [1..10] of integer; function prelVector(v: vector;     var n: integer): integer; var s, i: integer; begin     s := 0; i := 2;     while (i &lt;= n) do         begin             s := s + v[i] - v[i - 1];             if (v[i] = v[i - 1]) then                 n := n - 1;             i := i + 1;         end;     prelVector := s; end; var n, rezultat:integer; v:vector; begin     n := 7;     v[1] := 1; v[2] := 4; v[3] := 2;     v[4] := 3; v[5] := 3; v[6] := 10;     v[7] := 12;     rezultat := prelVector(v,n);     write(n, ';', rezultat); end.</pre>

Precizați care este rezultatul afișat în urma executării programului.

Algoritmul **prelVector** parcurge vector **v** incepand cu a doua pozitie, insumand in variabila **s** rezultatul operatiei **v[i] – v[i-1]**, iar in cazul in care cele 2 valori **v[i]** si **v[i-1]** sunt egale se decrementeaza valoarea lui **n**.

Astfel incat, valoarea lui **s** in urma parcurgerii **i <= n** din exemplul dat ar fi:

- I.  $i=2, n=7$   
 $s = 0 + 4 - 1$
- II.  $i=3, n=7$   
 $s = 3 + 2 - 4$
- III.  $i=4, n=7$   
 $s = 1 + 3 - 2$
- IV.  $i=5, n=7$   
 $s = 2 + 3 - 3$  (pentru ca  $v[i] == v[i-1]$ ,  $n$  devine 6);
- V.  $i=6, n=6$   
 $s = 2 + 10 - 3 = 9$

Rezultat corect: 6;9

4. Fie subalgoritmii `reuniune(a, n, b, m, c, p)` și `calcul(a, n, b, m, c, p)`, descriși mai jos, unde **a**, **b** și **c** sunt șiruri care reprezintă mulțimi de numere naturale cu **n**, **m** și respectiv **p** elemente ( $1 \leq n \leq 200, 1 \leq m \leq 200, 1 \leq p \leq 400$ ). Parametrii de intrare sunt **a**, **n**, **b**, **m** și **p**, iar parametrii de ieșire sunt **c** și **p** (16).

<ol style="list-style-type: none"> <li>1. Subalgoritm reuniune(a, n, b, m, c, p):</li> <li>2. Dacă <math>n = 0</math> atunci</li> <li>3. Pentru <math>i \leftarrow 1, m</math> execută</li> <li>4. <math>p \leftarrow p + 1, c_p \leftarrow b_i</math></li> <li>5. SfPentru</li> <li>6. altfel</li> <li>7. Dacă nu aparține(<math>a_n, b, m</math>) atunci</li> <li>8. <math>p \leftarrow p + 1, c_p \leftarrow a_n</math></li> <li>9. SfDacă</li> <li>10. reuniune(a, <math>n - 1, b, m, c, p</math>)</li> <li>11. SfDacă</li> <li>12. SfSubalgoritm</li> </ol>	<ol style="list-style-type: none"> <li>1. Subalgoritm calcul(a, n, b, m, c, p):</li> <li>2. <math>p \leftarrow 0</math></li> <li>3. reuniune(a, n, b, m, c, p)</li> <li>4. SfSubalgoritm</li> </ol>
--	---

Se considera ca subalgoritmul `aparține(x, a, n)` verifică dacă un număr natural **x** aparține mulțimii **a** cu **n** elemente; **a** este un șir cu **n** elemente și reprezintă o mulțime de numere naturale ( $1 \leq n \leq 200, 1 \leq x \leq 1000$ ):

```
int aparține(int x, int a[], int n) {
    for (int i=0; i<n; i++) {
        if (x == a[i]) {
            return 1;
        }
    }
    return 0;
}
```

Apelul funcției **calcul** declanșează apelul recursiv al funcției **reuniune** ce implementează o logică ghidată de existența unei structuri **Daca**.

Astfel, dacă dimensiunea lui **n** este diferită de 0, algoritmul verifică dacă elementul de pe ultima poziție a sirului **a** nu există deja în sirul **b**, iar dacă nu există se adaugă în sirul rezultat **c**.

Această ramură a structurii **Daca** se încheie cu apelul recursiv **reuniune(a, n - 1, b, m, c, p)** ce micșorează valoarea **n** cu 1 astfel încât următorul apel va verifica penultima valoare a sirului **a** s.a.m.d până când nu mai există elemente în **a** și **n=0**.

Când **n=0** se activează condiția structurii **Daca**, unde se încheie recursivitatea prin adăugarea tuturor elementelor din sirul **b** în sirul rezultat **c**.

Putem formaliza soluția algoritmului astfel: se obține un sir **c** ce conține în ordine pe primele poziții toate elementele din **a** care nu se regăsesc în **b**, urmate de toate elementele din **b**.

1. când mulțimea **a** conține un singur element, apelul subalgoritmului **calcul(a, n, b, m, c, p)** provoacă apariția unui ciclu infinit

**R: Fals**

**Explicatie:** În cazul în care **a** conține un singur element, ramura **Altfel** a structurii **Daca** se execută o singură dată, urmând ca apoi să se execute ramura principală a acesteia. Atât timp cât **n** se deprementează pe apelul recursiv, avem o condiție ce oprește recursivitatea.

2. când mulțimea **a** conține 4 elemente, apelul subalgoritmului **calcul(a, n, b, m, c, p)** provoacă executarea instrucțiunii de pe linia 10 a subalgoritmului **reuniune** de 4 ori

**R: Adevărat**

Dacă **a** conține 4 elemente, înseamnă că ramura **Altfel** va fi executată de același număr de ori datorită faptului că **n > 0** cât timp este decrementat în fiecare apel recursiv de la linia 10 care se oprește când **n=0** (condiția principală din **Daca**).

3. când mulțimea **a** conține 5 elemente, apelul subalgoritmului **calcul(a, n, b, m, c, p)** provoacă executarea instrucțiunii de pe linia 2 a subalgoritmului **reuniune** de 5 ori

**R: Fals**

Este adevărat că instrucțiunea de la linia 2 se interpretează la fiecare apel de funcție **reuniune** (**care în acest caz ar fi executată de 5 ori prin recursivitate**) dar nu se va executa ca și condiție de adevăr decât odată când **n=0**.

4. când mulțimea **a** are aceleași elemente ca și mulțimea **b**, în urma execuției subalgoritmului **calcul(a, n, b, m, c, p)** mulțimea **c** va avea același număr de elemente ca și mulțimea **a**

**R: Adevărat**

Daca sirurile **a** si **b** au acelasi numar de elemente, asta inseamna ca in **c** nu vor fi adaugate deloc elemente din **a** ci doar elementele din **b**. Astfel ca **c** va fi identic cu **b**, care este identic cu **a**, atunci si **a** este identic cu **c**.

5. Se consideră şirul (1, 2, 3, 2, 5, 2, 3, 7, 2, 4, 3, 2, 5, 11, ...) format astfel: plecând de la şirul numerelor naturale, se înlocuiesc numerele care nu sunt prime cu divizorii lor proprii, fiecare divizor **d** fiind considerat o singură dată pentru fiecare număr. Care dintre subalgoritmi determină al **n**-lea element al acestui şir (**n** - număr natural,  $1 \leq n \leq 1000$ )? (27)

A. Subalgoritm identificare(n):  
 $a \leftarrow 1, b \leftarrow 1, c \leftarrow 1$   
**CâtTimp**  $c < n$  execută  
 $a \leftarrow a + 1, b \leftarrow a, c \leftarrow c + 1, d \leftarrow 2$   
 $f \leftarrow \text{false}$   
**CâtTimp**  $c \leq n$  și  $d \leq a \text{ DIV } 2$  execută  
**Dacă**  $a \text{ MOD } d = 0$  atunci  
 $c \leftarrow c + 1, b \leftarrow d, f \leftarrow \text{true}$   
**SfDacă**  
 $d \leftarrow d + 1$   
**SfCâtTimp**  
**Dacă**  $f$  atunci  
 $c \leftarrow c - 1$   
**SfDacă**  
**SfCâtTimp**  
 returnează  $b$   
**SfSubalgoritm**

C. Subalgoritm identificare(n):  
 $a \leftarrow 1, b \leftarrow 1, c \leftarrow 1$   
**CâtTimp**  $c < n$  execută  
 $a \leftarrow a + 1, d \leftarrow 2$   
**CâtTimp**  $c < n$  și  $d \leq a$  execută  
**Dacă**  $a \text{ MOD } d = 0$  atunci  
 $c \leftarrow c + 1, b \leftarrow d$   
**SfDacă**  
 $d \leftarrow d + 1$   
**SfCâtTimp**  
**SfCâtTimp**  
 returnează  $b$   
**SfSubalgoritm**

B. Subalgoritm identificare(n):  
 $a \leftarrow 1, b \leftarrow 1, c \leftarrow 1$   
**CâtTimp**  $c < n$  execută  
 $c \leftarrow c + 1, d \leftarrow 2$   
**CâtTimp**  $c \leq n$  și  $d \leq a \text{ DIV } 2$  execută  
**Dacă**  $a \text{ MOD } d = 0$  atunci  
 $c \leftarrow c + 1, b \leftarrow d$   
**SfDacă**  
 $d \leftarrow d + 1$   
**SfCâtTimp**  
 $a \leftarrow a + 1, b \leftarrow a$   
**SfCâtTimp**  
 returnează  $b$   
**SfSubalgoritm**

D. Subalgoritm identificare(n):  
 $a \leftarrow 1, b \leftarrow 1, c \leftarrow 1$   
**CâtTimp**  $c < n$  execută  
 $b \leftarrow a, a \leftarrow a + 1, c \leftarrow c + 1, d \leftarrow 2$   
**CâtTimp**  $c \leq n$  și  $d \leq a \text{ DIV } 2$  execută  
**Dacă**  $a \text{ MOD } d = 0$  atunci  
 $c \leftarrow c + 1, b \leftarrow d$   
**SfDacă**  
 $d \leftarrow d + 1$   
**SfCâtTimp**  
**SfCâtTimp**  
 returnează  $b$   
**SfSubalgoritm**

Pentru o intelegere pe deplin daca variantele propuse sunt corecte sau nu, este essential sa intelegem problema abordata ca si caz general.

### Sir divizori (X):

1	2	3	2	5	2	3	7	2	4	3	2	5	11
---	---	---	---	---	---	---	---	---	---	---	---	---	----

### Sir numere (Y):

(divizorii corespondenti numerelor din sirul de mai jos sunt colorati la fel):

1	2	3	4	5	6	7	8	9	10	11
---	---	---	---	---	---	---	---	---	----	----



Ca si variabile, fiecare din cele 4 solutii foloseste o serie de variabile care au aceeasi destinatie:

- **a** – stocheaza valoarea numarului curent din sirul numerelor;
- **c** – reprezinta pozitia curenta din sirul de divizori calculati pentru numarul **a**;
- **b** – reprezinta valoarea ultimului divizor calculat, pentru care **c == n**;

Desi la o prima vedere toate solutiile par sa calculeze ceea ce ne dorim, si mai mult decat atat solutiile unele solutii se aseamana, trebuie sa descoperim daca conditiile de iteratie peste celor 2 siruri **X** (sir divizori) si **Y** (sir numere) reusesc sa prinda mai multe cazuri de testare.

In acest context, putem apela la o metoda **brut force** de eliminare a variantelor propuse folosind mai multe cazuri de testare.

Din definitia problemei observam ca sirul de divizori este creat folosind sirul numerelor naturale (1, 2, 3, 4, 5..), de aceea putem stabili o strategie de interpretare a fiecare variante propuse folosind numerele naturale incepand de la 1, eliminand pe rand variante pana cand ramanem doar cu o singura varianta corecta. Desi poate parea o metoda exhaustiva, va fi inevitabila generalizarea fiecarui algoritm astfel incat s-ar putea sa obtinem varianta corecta inainte de a elimina candidatii cu implementari eronate pe baza cazurilor de testare.

#### Cazuri de testare:

I.  $n = 1$

A	1
B	1
C	1
D	1

II.  $n = 2$

A	2
B	2
C	2
D	1

III.  $n = 3$

A	3
B	3
C	3
D	2

IV.  $n = 4$

A	2
B	4
C	2

D	3
---	---

V. n = 5

A	5
B	5
C	4
D	4

Din I, II, III, IV, V concluzionam ca varianta A este varianta corecta.

Corectitudinea variantei A este data de conditiile structurilor iterative (**CatTimp**) prezente in algoritm.

Daca ne folosim de ordinea celor 2 structuri, primul **CatTimp** ne asigura repetarea calcului divizorilor atat timp cat **c** (pozitia curenta in sirul de divizori) este strict mai mic decat **n** (pozitia divizorului in sirul de divizori pe care dorim sa il returnam).

Cel de-al doile **CatTimp** este destinat calculului de divizori.

Particularitati:

- A. Calculeaza divizorii incepand de la 2 pana la jumatatea numarului current sau pana cand s-au gasit deja **c divizori**, iar de fiecare data cand se gaseste un divizor se incrementeaza variabila **c** si se seteaza pe **true** un flag **f**.

Astfel incat de fiecare data cand este gasit un divizor pentru numarul curent (valoarea lui **f este setata pe true**), **c** este decrementat deoarece **c** este incrementat "din oficiu" inca inainte de inceperea calcului de divizori (calculul incepe cu **d=2**).

- B. B este o varianta gresita din start, deoarece nu returneaza ultimul divizor, ci ultimul numar luat in considerare pentru calculul divizorilor (**b = a**)
- C. Este o variatie a variantei A pentru care nu este decrementat **c** desi el este incrementat inca dinainte gasirii unui divizor in calcului din cel de-al doilea **CatTimp**.
- D. Este o variatie a variante A, dar cu aceeasi problema ca si varianta **C**.