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 şir x cu n elemente numere naturale ($3 \leq n \leq 10000$) şi 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 algortimului 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 aceast 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 fiecarei 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 fiecarei 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 fiecarei 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 fiecarei 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 pozitie **i (i → k)** de ***n/k - 1*** ori**.** Faptul ca **n/k** returneaza partea intreaga a operatiei de impartire, rezulta la ignorarea pozitiilor **k** ce se afla in segmentul ***k * (n/k) <= i < n***.

2. Se consideră subalgoritmul prelucreaza(v, k), unde ***v*** este un șir cu ***k*** numere naturale (1 ≤ ***k*** ≤ 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 executia algoritmului ca prin structura primului **CatTimp** se parcurge fiecare element al vectorului ***v***, urmand ca prin executia celui de-al doilea **CatTimp** sa iteram fiecare cifra a elemetului current, salvandu-se cifra cu valoarea maxima.

Cifra cu valoarea maxima rezultata din parcurgerea celui de-al doile **CatTimp** este adaugata la sfarsitul numarului **n,** numar ce reprezinta rezultatul executiei subalgoritmului **prelucreaza**.

De aceea am putea formaliza solutia algoritmului in una ce creeaza un numar **n** din cifrele maxime din **k** numere dintr-un vector **v**, ordinea cifrelor fiind data de pozitia numarului in vectorul **v**.

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

   v[0] → 1**9**4 , v[1] → 1**2**1, v[2] → 7**8**2, v[3] → -

   n = 928

   (rezultat final) - corect

B. $\boldsymbol{v}$ = (928) și $\boldsymbol{k}$ = 1

v[0] → 928

n = 9

(rezultat final) – gresit

C. $\boldsymbol{v}$ = (9, 2, 8, 0) și $\boldsymbol{k}$ = 4

v[0] → **9** , v[1] → **2**, v[2] → **8**, v[3] → -

n = 928

(rezultat final) - corect

D. $\boldsymbol{v}$ = (8, 2, 9) și $\boldsymbol{k}$ = 3

v[0] → **8** , v[1] → **2**, v[2] → **9**

(rezultat final) – gresit

3. Se consideră următorul program (12):

Varianta C

```c
#include <stdio.h>

int prelVector(int v[], int *n) {
    int s = 0; int i = 2;
    while (i <= *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,
&n);
  printf("%d;%d", n, rezultat);
  return 0;
}
```

Varianta C++

```cpp
#include <iostream>
using namespace std;
int prelVector(int v[], int&n) {
    int s = 0; int i = 2;
    while (i <= 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 << n <<";" << rezultat;
    return 0;
}
```

Varianta Pascal

```pascal
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 <= 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.
```

Precizați care este rezultatul afișat în urma executării programului.

Algoritmul **prelVector** parcuge 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, valorea 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).

```
Subalgoritm reuniune(a, n, b, m, c, p):
  Dacă n = 0 atunci
    Pentru i ← 1, m execută
      p ← p + 1, c_p ← b_i
    SfPentru
  altfel
    Dacă nu aparține(a_n, b, m) atunci
      p ← p + 1, c_p ← a_n
    SfDacă
    reuniune(a, n - 1, b, m, c, p)
  SfDacă
SfSubalgoritm
```

```
Subalgoritm calcul(a, n, b, m, c, p):
    p ← 0
    reuniune(a, n, b, m, c, p)
SfSubalgoritm
```

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

Apelul functiei ***calcul*** declanseaza apelul recursiv al functiei ***reuniune*** ce implementeaza o logica ghidata de existenta unei structura **Daca**.

Astfel, daca dimensiunea lui **n** este diferita de 0, algoritmul verifica daca elementului de pe ultima pozitie a sirului **a** nu exista deja in sirul **b**, iar daca nu exista se adauga in sirul rezultat **c.**
Aceasta ramura a structurii **Daca** se incheie cu apelul recursiv **reuniune(a, n – 1, b, m, c, p)** ce micsoreaza valorea **n** cu 1 astfel incat urmatoarul apel va verifica penultima valoarea sirului **a** s.a.m.d pana cand nu mai exista elemente in **a** si **n=0.**

Cand **n=0** se activeaza conditia structurii **Daca**, unde se incheie recursivitatea prin adaugarea tuturor elementelor din sirul **b** in sirul rezultat **c.**

Putem formaliza solutia algoritmului astfel: se obtine un sir **c** ce contine in ordine pe primele pozitii toate elemente din **a** care nu se regasesc in **b,** urmate de toate elemente 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: In cazul in care **a** contine un singur element, ramura **Altfel** a structurii **Daca** se executa o singura data, urmand ca apoi sa se execute ramura principala a acesteia. Atat timp cat **n** se deprementeaza pe apelul recursiv, avem o conditie ce opreste 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: Adevarat

   Daca **a** contine 4 elemente, inseamna ca ramura **Altfel** va fi executata de acelasi numar de ori datorita faptului ca **n>0** cat timp este decrementat in fiecare apel recursiv de la linia 10 care se opreste cand **n=0** (conditia principala 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 adevarat ca instructiunea de la linia 2 se interpreteaza la fiecare apel de functie **reuniune (care in acest caz ar fi executata de 5 ori prin recursivitate)** dar nu se va executa ca si conditie de adevar decat odata cand **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: Adevarat

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 șirul (1, 2, 3, 2, 5, 2, 3, 7, 2, 4, 3, 2, 5, 11, ...) format astfel: plecând de la șirul 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 șir (n - număr natural, $1 \leq n \leq 1000$)? (**27**)

A.
```
Subalgoritm identificare(n):
    a ← 1, b ← 1, c ← 1
    CâtTimp c < n execută
        a ← a + 1, b ← a, c ← c + 1, d ← 2
        f ← false
        CâtTimp c ≤ n şi d ≤ a DIV 2 execută
            Dacă a MOD d = 0 atunci
                c ← c + 1, b ← d, f ← true
            SfDacă
            d ← d + 1
        SfCâtTimp
        Dacă f atunci
            c ← c - 1
        SfDacă
    SfCâtTimp
    returnează b
SfSubalgoritm
```

C.
```
Subalgoritm identificare(n):
    a ← 1, b ← 1, c ← 1
    CâtTimp c < n execută
        a ← a + 1, d ← 2
        CâtTimp c < n şi d ≤ a execută
            Dacă a MOD d = 0 atunci
                c ← c + 1, b ← d
            SfDacă
            d ← d + 1
       SfCâtTimp
    SfCâtTimp
    returnează b
SfSubalgoritm
```

B.
```
Subalgoritm identificare(n):
    a ← 1, b ← 1, c ← 1
    CâtTimp c < n execută
        c ← c + 1, d ← 2
        CâtTimp c ≤ n şi d ≤ a DIV 2 execută
            Dacă a MOD d = 0 atunci
                c ← c + 1, b ← d
            SfDacă
            d ← d + 1
        SfCâtTimp
        a ← a + 1, b ← a
    SfCâtTimp
    returnează b
SfSubalgoritm
```

D.
```
Subalgoritm identificare(n):
    a ← 1, b ← 1, c ← 1
    CâtTimp c < n execută
        b ← a, a ← a + 1, c ← c + 1, d ← 2
        CâtTimp c ≤ n şi d ≤ a DIV 2 execută
            Dacă a MOD d = 0 atunci
                c ← c + 1, b ← d
            SfDacă
            d ← 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.

<u>Sir divizori (X):</u>

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

<u>Sir numere</u> (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 exaustiva, 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 <u>A este varianta corecta.</u>

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 (valorea 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.**