Probleme consultații
Șiruri (tablouri unidimensionale)

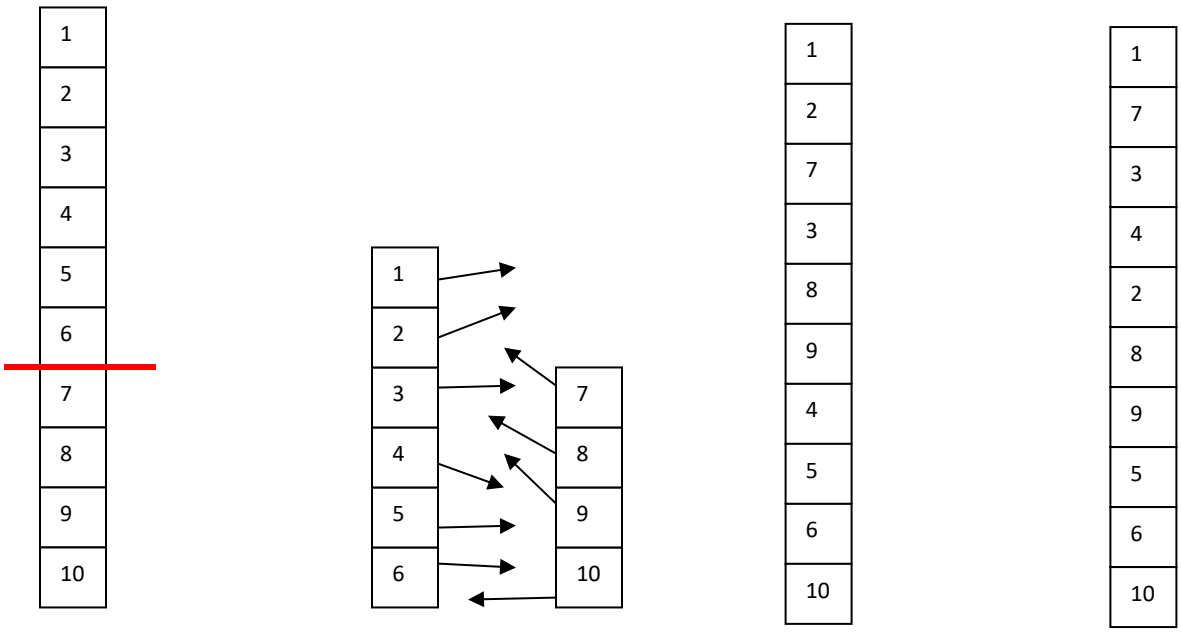
Problema 1 - Riffle Shuffle

Enunț

Considerăm un pachet de *n* cărți, fiecare carte având un număr unic de la 1 la *n*. Procesul de amestecare a cărților, numit ”riffle shuffle”, împarte pachetul aleatoriu în două și intercalează cele două “jumătăți” în mod aleatoriu (a se vedea imaginea din stânga). Având 2 ordonări ale cărților, *ordonare_inițială* și *ordonare_finală*, determinați dacă pornind de la *ordonare_inițială* putem ajunge, cu un singur ”riffle shuffle”, la *ordonare_finală*.

Date de intrare: *n* (numărul de cărți), *ordonare inițială* (un tablou cu n elemente) și *ordonare finală* (un tablou cu n elemente). Rezultatul este *adevărat* sau *fals*. De exemplu, dacă avem 10 cărți, *ordonare_inițială* este [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] și *ordonare_finală* este [1, 2, 7, 3, 8, 9, 4, 5, 6, 10], se poate ajunge de la ordonarea inițială la cea finală cu un singur ”riffle shuffle” (a se vedea exemplul de mai jos). Pornind de la aceeași *ordonare_inițială*, nu se poate ajunge la *ordonare_finală* = [1, 7, 3, 4, 2, 8, 9, 5, 6, 10].

Pachetul inițial: **Procesul de ”riffle shuffle”** **Pachetul final posibil:** **Pachet final imposibil:**



Exemple

Date de intrare			Rezultat
n	ordI	ordF	
10	[1,2,3,4,5,6,7,8,9,10]	[1,2,7,3,8,9,4,5,6,10]	True
10	[1,2,3,4,5,6,7,8,9,10]	[1,4,5,6, 2,3,7,8,9,10]	True
10	[1,2,3,4,5,6,7,8,9,10]	[1,8,2,3,4,5,6,7,9,10]	True
10	[1,2,3,4,5,6,7,8,9,10]	[10,9,8,7,6,5,4,3,2,1]	False
10	[1,2,3,4,5,6,7,8,9,10]	[1,6,2,8,3,4,5,7,9,10]	False
10	[1,2,3,4,5,6,7,8,9,10]	[1,2,3,4,5,6,7,8,9,10]	True*
15	[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]	[9,10,1,2, 11,3,5,12,13,4,6,7,14,15,8]	False
15	[3,7,1,8,10,9,15,2,13,14,4,6,11,5,12]	[3, 13,14,7,4,1,8,6,10,11,5,9,12,15,2]	True

Analiză

Rezolvarea problemei se descompune în:

1. Determinarea punctului în care a fost tăiat pachetul (pe baza ordonării finale);
2. Având cele 2 "jumătăți" ale pachetului (una de la început până la punctul de tăiere, iar a doua de la punctul de tăiere până la capăt), verificarea posibilității de a se ajunge la ordonarea finală printr-un singur "riffle shuffle".

1. Determinarea punctului de tăiere

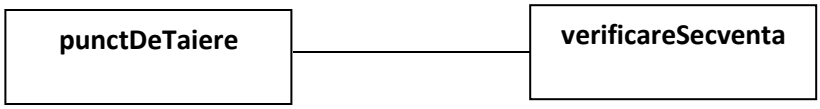
Știm că dacă tăiem pachetul de cărți la un punct p, vom avea 2 "jumătăți": prima "jumătate" începe la poziția 1 și se termină la poziția p, iar cea de-a 2-a începe la poziția p+1 și se termină la poziția n. Dacă urmărim exemplul, observăm că ultimul element din ordonarea finală este ultimul element dintr-una dintre jumătăți.
Dacă ultimul element din ordonarea finală nu este egal cu ultimul element din ordonarea inițială, el trebuie să fie ultimul element din prima "jumătate". Cum numerele de pe cărți sunt unice, putem găsi imediat punctul de tăiere. Dacă ultimul element din ordonarea finală este egal cu ultimul element din ordonarea inițială, ne uităm la penultimul element din ordonarea finală; acesta din urmă trebuie să fie ori ultimul element din prima "jumătate", ori penultimul element din a 2-a "jumătate" ș.a.m.d.

2. Verificarea secvenței finale

Odată identificat punctul de tăiere, parcurgem în paralel pachetul final și cele 2 "jumătăți". Parcurgerile se efectuează de la finalul tabloului spre început (doar deoarece astfel se construiește rezultatul amestecării, putându-se efectua, totuși, și în sens invers). La fiecare pas verificăm din care "jumătate" provine elementul curent din ordonarea finală. În fiecare "jumătate" vom avea un element curent, a fiecare pas unul dintre elementele curente din cele 2 "jumătăți" trebuind să fie următorul element din ordonarea finală (în sensul parcurgerii).

Observatie: Inclusiv punctul de taiere se poate determina pacurgând ordonarea finală în sens invers, începând cu prima poziție, nu cu ultima.

Identificarea subalgoritmilor



Specificarea subalgoritmilor

Funcție **punctDeTaiere(n, ordI, ordF):**
Descriere: caută punctul de tăiere în *ordF*, pornind de la ordinea inițială *ordI*
Date: n ∈ N*, n este numărul de cărți (lungimea tablourilor ordI și ordF)
ordI – tabloul inițial cu cărți, ordI = ($ordI_i$ |i =1…n, $ordI_i$ ∈ {1,…,n})
ordF - tabloul final cu cărți, ordF = ($ordF_i$ |i =1…n, $ordF_i$ ∈ {1,…,n})
Rezultate: returnează poziția de sfârsit pentru prima jumătate (poziția după care ordI a fost tăiat; poziția ∈ 1 ... n) sau -1 dacă cele 2 ordonări sunt identice.

Funcție **verificareSecventa(n, ordI, ordF)**:
Descriere: verifică dacă se poate ajunge de la ordI la ordF cu un singur "*riffle shuffle*"
Date: n ∈ N*, n este numărul de cărți (lungimea tablourilor ordI și ordF)
ordI – tabloul inițial cu cărți, ordI = ($ordI_i$ |i =1…n, $ordI_i$ ∈ {1,…,n})
ordF - tabloul final cu cărți, ordF = ($ordF_i$ |i =1…n, $ordF_i$ ∈ {1,…,n})
Rezultate: returnează *true* dacă se poate ajunge din ordI la ordF cu un singur riffle shuffle, *false* altfel

Proiectare

Observație: La toate problemele din acest document indexarea tablourilor începe de la:
- 1 la Pseudocod și Pascal
- 0 la C++

```
funcție punctDeTaiere(n, ordI, ordF):
      pozitieOrdI = n            {pornim de la ultima pozitie a tablourilor}
      pozitieOrdF = n            {OBS: parcuregerea poate fi efectuata cu un singur
indice curent, intrucat pozitieOrdI si pozitieOrdF coincid, la fiecare pas}
cat-timp pozitieOrdF > 0 și ordI[pozitieOrdI] = ordF[pozitieOrdF] execută  {cât timp
elementul curent din ordonarea finală provine din cea de-a doua "jumătate", avansăm (în
sens invers) în cele două ordonări}
            pozitieOrdI ← pozitieOrdI - 1
            pozitieOrdF ← pozitieOrdF - 1
      sf-cat-timp
      dacă pozitieOrdF = 0 atunci {cele 2 tablouri sunt identice, returnăm -1}
```

```
                punctDeTaiere ← -1
        altfel {cautam pozitia elementului cu care se incheie prima jumatate in ordI}
                ultimElemJumatate1 ← ordF[pozitieOrdF]
                pozitieUltimElem ← 1
                cat-timp ordI[pozitieUltimElem] != ultimElemJumatate1 execută
                        pozitieUltimElem ← pozitieUltimElem + 1
                sf-cat-timp
                punctDeTaiere ← pozitieUltimElem
        sf_dacă
sf_functie

functie verificareSecventa(n, ordI, ordF):
        taietura ← punctDeTaiere(n, ordI, ordF)
        dacă taietura = -1 atunci
                verificareSecventa ← true {pachetul final este identic cu cel initial*}
        altfel
                {parcurgem cele 2 "jumatati" si ordonarea finala in sens invers, incepand cu
ultima pozitie}
                pozJ1 ← taietura     {prima "jumatate" incepe cu pozitia 1 si se incheie cu
pozitia taietura}
                pozJ2 ← n            {a 2-a "jumatate" incepe cu pozitia taietura+1 si se
incheie cu pozitia n}
                pozOrdF ← n
                cat-timp pozOrdF > 0 executa
                        dacă pozJ1 > 0 si ordF[pozOrdF] = ordI[pozJ1] atunci  {verificam daca
elementul curent din ordonarea finala este egal cu elementul curent din prima "jumatate"}
                                pozOrdF ← pozOrdF - 1
                                pozJ1 ← pozJ1 - 1
                        altfel
                                dacă pozJ2 > taietura si ordF[pozOrdF] = ordI[pozJ2] atunci
                        {verificam daca elementul curent din ordonarea finala este egal cu
                        elementul curent din a doua "jumatate"}
                                        pozOrdF ← pozOrdF - 1
                                        pozJ2 ← pozJ2 - 1
                                altfel
                                        verificareSecventa ← false {elementul curent din ordF nu
se potriveste cu elementul curent din niciuna dintre "jumatati"; decucem ca pachetul final
nu poate fi obtinut din cel initial printr-un riffle-shufle}
                                sf_daca
                        sf_daca
                sf-cat-timp
                verificareSecventa ← true
        sf_daca
sf_functie
```

Cod sursă C++

```
int punctDeTaiere(int n, int ordI[], int ordF[])

{

        int pozitieOrdF = n - 1;
```

```
	int pozitieOrdI = n - 1;

	while (pozitieOrdF >= 0 && ordI[pozitieOrdI] == ordF[pozitieOrdF])
	{
		pozitieOrdF--;
		pozitieOrdI--;
	}

	if (pozitieOrdF == -1)
	{
		//cele 2 tablouri sunt identice, returnam -1
		return -1;
	}
	else
	{
		int ultimElemJumatate1 = ordF[pozitieOrdF];
		int pozitieUltimElem = 0;
		while (ordI[pozitieUltimElem] != ultimElemJumatate1)
		{
			pozitieUltimElem++;
		}
		return pozitieUltimElem;
	}
}


bool verificareSecventa(int n, int ordI[], int ordF[])
{

	int taietura = punctDeTaiere(n, ordI, ordF);

	if (taietura == -1)
	{
		//cele 2 tablouri sunt identice
		return true;
	}
	else
	{
		int pozJumatate1 = taietura;
		int pozJumatate2 = n-1;
		int pozOrdF = n-1;

		while (pozOrdF >= 0)
		{
			if (pozJumatate1 >= 0 && ordF[pozOrdF] == ordI[pozJumatate1])
			{
				pozOrdF--;
				pozJumatate1--;
			}
			else if (pozJumatate2 > taietura && ordF[pozOrdF] ==
					ordI[pozJumatate2]) {
				pozOrdF--;
				pozJumatate2--;
			}
			else
			{
				return false;
			}
		}
```

```
                return true;
        }
}
```

Cod sursă Pascal

```
program Problema1;

type
    myArray = array[1..100] of integer;

function punctDeTaiere(n:integer; ordI: myArray; ordF: myArray): integer;
var
   pozOrdI: integer;
   pozOrdF: integer;
   result: integer;
   pozUltimElem: integer;
   ultimElemJumatate1: integer;
begin
    pozOrdI := n;
    pozOrdF := n;
    while ((pozOrdF > 0) and (ordI[pozOrdI] = ordF[pozOrdF])) do
    begin
        pozOrdF := pozOrdF - 1;
        pozOrdI := pozOrdI - 1;
    end;
    if pozOrdF = 0 then
        result := -1
    else
        begin
            ultimElemJumatate1 := ordF[pozOrdF];
            pozUltimElem := 1;
            while ordI[pozUltimElem] <> ultimElemJumatate1 do
            begin
                pozUltimElem := pozUltimElem + 1;
            end;
            result := pozUltimElem;
        end;
    punctDeTaiere:= result;
end;

function verificareSecventa(n: integer; ordI: myArray; ordF: myArray): boolean;
var
    result: boolean;
    taietura: integer;
    pozJumatate1: integer;
    pozJumatate2: integer;
    pozOrdF: integer;
    continua: boolean;
begin
    taietura := punctDeTaiere(n, ordI, ordF);
    if taietura = -1 then
        result := true
    else
    begin
        pozJumatate1 := taietura;
        pozJumatate2 := n;
```

```
        pozOrdF := n;
        continua := true;
        while ((continua = true) and (pozOrdF > 0)) do
        begin
            if ((pozJumatate1 > 0) and (ordI[pozJumatate1] = ordF[pozOrdF])) then
            begin
                pozJumatate1 := pozJumatate1 - 1;
                pozOrdF := pozOrdF - 1;
            end
            else if ((pozJumatate2>taietura) and (ordI[pozJumatate2]=ordF[pozOrdF])) then
            begin
                pozJumatate2 := pozJumatate2 - 1;
                pozOrdF := pozOrdF - 1;
            end
            else
                continua := false;
        end;
        result:= continua;
    end;
    verificareSecventa := result;
end;
```

Problema 2 - Găsește duplicatul

Enunț

Se dă un tablou cu n+1 elemente, în care fiecare element este un număr întreg din intervalul [1,n]. Prin urmare, există cel puțin un element care se repetă. Este posibil ca toate elementele să fie egale. Să se găsească unul dintre elementele care se repetă.

Exemple

Date de intrare		Rezultat
Tablou	**n**	
[1,2,3,4,5,6,2]	7	2
[1,1,1,1,1,1,1]	7	1
[1,2,3,1,2,3,1,2,3,1,2,3]	12	1 sau 2 sau 3
[6,1, 5, 2, 3, 2, 4, 7]	8	2
[1, 1, 1, 1, 7, 7, 7, 7]	8	1 sau 7

Analiză

Varianta 1

- Se utilizează 2 cicluri repetitive (*for*) imbricate, comparându-se elementele două câte două

Complexitate de spațiu extra (ignorând tabloul de intrare): Θ(1)
Complexitate de timp: $O(n^2)$
($\Theta(n^2)$ în caz defavorabil, când ultimele 2 elemente sunt egale, iar restul unice, caz în care se execută ambele cicluri *for* complet, dar execuția se poate opri și mai repede, având, în caz favorabil complexitate Θ(1))

Pseudocod

```
functie duplicat1(elemente, lungime):
      i ← 1
      cont  ← true
      cat-timp i <= lungime-1 si cont executa
            j ← i+1
            cat-timp j <= lungime si cont executa
                  daca elemente[i] = elemente[j] atunci
                        duplicat1 ← t[i]
```

```
                        cont ← false
                    altfel
                        j ← j+1
                    sf-daca
                sf-cat-timp
                i ← i+1
            sf-cat-timp
sf_functie
```

Cod sursă C++

```
int duplicat1(int elemente[], int lungime)
{
	for (int i = 0; i < lungime; i++)
	{
		for (int j = i + 1; j < lungime; j++)
		{
			if (elemente[i] == elemente[j])
			{
				return elemente[i];
			}
		}
	}
}
```

Cod sursă Pascal

type

```
T = array[1..100] of integer;

function duplicat1(tablou: T; n: integer):integer;
var
    i, j, result: integer;
    cont: Boolean;
begin
    i := 1;
    cont := True;
    result := 0;
    while (i < n) and cont do
    begin
        j := i + 1;
        while (j <= n ) and cont do
        begin
            if (tablou[i] = tablou[j]) then
            begin
                result := tablou[i];
                cont := False;
            end;
```

```
        j := j +1;
        end;
    i := i + 1;
    end;
    duplicat1 := result;
end;
```

Varianta 2

- Se folosește un tablou auxiliar de n elemente booleene (având valoarea *true* sau *false)*
- Inițial, toate elementele din tabloul auxiliar au valoarea *false.*
- Se parcurge tabloul cu numere întregi și, pentru fiecare număr *e*, se setează valoarea de pe poziția *e* din tabloul auxiliar la *true,* dacă valoarea curentă este *false*. Dacă valoarea este deja *true*, se returnează *e*, *e* fiind un element duplicat.

Complexitate de spațiu extra: Θ(n)
Complexitate de timp: O(n)

Pseudocod

```
functie duplicat2(elemente, lungime):
      pentru i = 1,lungime executa
            aux[i] = false
      sf-pentru
      cont  ← true
      i ← 1
      cat-timp (i<=lungime) si cont executa
            elem  ← elemente[i]
            daca aux[elem] = true atunci
                  duplicat2 ← elem
                  cont ← false
            altfel
                  aux[elem] ← true
                  i ← i+1
            sf-daca
      sf-cat-timp
sf_functie
```

Cod sursă C++

```
int duplicat2(int elemente[], int lungime)
{

      //sunt n+1 elemente din intervalul 1..n; folosim un vector auxiliar de n+1 elemente,
nu utilizam pozitia 0
      bool* auxiliar = new bool[lungime];
```

```
	for (int i = 0; i < lungime; i++)
	{
		auxiliar[i] = false;
	}
	for (int i = 0; i < lungime; i++)
	{
		int elem = elemente[i];
		if (auxiliar[elem] == true)
		{
			delete[] auxiliar;
			return elem;
		}
		else
		{
			auxiliar[elem] = true;
		}
	}
}
```

Cod sursă Pascal

```
type
T = array[1..100] of integer;
TBool = array[1..100] of Boolean;

function duplicat2(tablou: T; n: integer): integer;
var
    i, elem, result: integer;
    aux: TBool;
    cont : Boolean;
begin
    for i:= 1 to n do
        aux[i] := False;
    cont := True;
    i := 1;
    while (i <= n) and cont do
    begin
        elem := tablou[i];
        if (aux[elem] = true) then
        begin
            cont := False;
            result := elem;
        end
        else
            aux[elem] := True;
        i := i + 1;
    end;
    duplicat2 := result;
end;
```

Varianta 3

Se folosește o metodă similară cu căutarea binară:

- vectorul de elemente nu se împarte (ca în cazul căutării binare), ci se încearcă reducerea intervalului în care se găsește un element duplicat;
- elementele sunt din intervalul [1...n]; dacă se împarte intervalul în 2 subintervale, (cel puțin) unul dintre acestea ar trebui să conțină mai multe elemente decât lungimea lui, întrucât cele 2 subintervale au împreună lungimea n, dar vectorul conține n+1 elemente; un element care se repetă este, sigur, în subintervalul respectiv, putându-se continua căutarea în acel subinterval.

Complexitate de spațiu extra: $\Theta(1)$
Complexitate de timp: $\Theta(n*\log_2 n)$

Pseudocod

```
functie duplicat3(elemente, lungime):
      inceput ← 1
      sfarsit ← lungime
      cat-timp inceput < sfarsit executa
             mijloc ← [(inceput + sfarsit) / 2]
             count ← 0
             pentru i ← 1,n executa
                    daca elemente[i] >= inceput si elemente[i] <= mijloc atunci
                          count ← count + 1
                    sf-daca
             sf-pentru
             lungimeInterval ← mijloc - inceput + 1
            daca lungimeInterval < count
                  sfarsit ← mijloc
            altfel
                  inceput ← mijloc + 1
           sf-daca
      sf-cat-timp
      duplicat3 ← inceput
sf_functie
```

Cod sursă C++

```
int duplicat3(int elemente[], int lungime)
{
       int inceput = 0;
       int sfarsit = lungime - 1;

       while (inceput < sfarsit)
       {
              int mijloc = (inceput + sfarsit) / 2;
              int count = 0;
```

```
            for (int i = 0; i < lungime; i++)
            {
                  if (elemente[i] >= inceput && elemente[i] <= mijloc)
                  {
                        count++;
                  }
            }
            int lungimeInterval = mijloc - inceput + 1;
            if (lungimeInterval < count)
            {
                  sfarsit = mijloc;
            }
            else
            {
                  inceput = mijloc + 1;
            }
      }
      return inceput;
}
```

Cod sursă Pascal

```
type
T = array[1..100] of integer;
function duplicat3(tablou: T; n: integer): integer;
var
    inceput, sfarsit, mijloc, result, count, lungimeInterval, i: integer;
begin
    inceput := 1;
    sfarsit := n;
    while (inceput < sfarsit) do
    begin
        mijloc := (inceput + sfarsit) div 2;
        count := 0;
        //numaram cate elemente sunt in intervalul [inceput, mijloc]
        for i := 1 to n do
        begin
            if (tablou[i] >= inceput) and (tablou[i] <= mijloc) then
                count := count + 1;
        end;
        lungimeInterval := mijloc - inceput + 1;
        if (lungimeInterval < count) then
            sfarsit := mijloc
        else
            inceput := mijloc + 1;
    end;
    duplicat3 := inceput;
end;
```

Întrebări grilă

Întrebarea 1a.

Considerați subalgoritmul următor:

```
subalgoritm ceFace(tablou, n):
      pentru i=1,10 executa
            pentru j=1,n-1 executa
                  daca (tablou[j]>tablou[j+1]) atunci
                        tmp ← tablou[j]
                        tablou[j] ← tablou[j+1]
                        tablou[j+1] ← tmp
                  sf-daca
            sf-pentru
      sf-pentru
sf-subalgoritm
```

Care dintre următoarele afirmații despre subalgoritmul `ceFace` sunt adevărate, presupunând că, la apel, primește un tablou și lungimea lui?

a. Subalgoritmul `ceFace` sortează un tablou indiferent de lungimea tabloului.
b. Subalgoritmul `ceFace` sortează un tablou dacă are lungimea 10.
c. Subalgoritmul `ceFace` sortează un tablou dacă are lungimea mai mică decât 10.
d. Subalgoritmul `ceFace` sortează un tablou dacă are lungimea mai mare decât 10.

Întrebarea 1b.

Considerați același subalgoritm `ceFace` (de la întrebarea 1a). Care dintre următoarele afirmații sunt adevărate, indiferent de lungimea tabloului?

a) Pe ultima poziție va fi cel mai mare element din tablou.
b) Pe ultima poziție va fi cel mai mic element din tablou.
c) Pe prima poziție va fi cel mai mic element din tablou.
d) Pe prima poziție va fi cel mai mare element din tablou.

Răspunsuri:

1a. - b, c

1b. - a

Întrebarea 2.

Considerați problema verificării existenței unui element întreg *e* într-un *tablou* de întregi cu lungimea *n*. Care dintre următorii subalgoritmi reprezintă rezolvări corecte pentru această problemă?

a. Subalgoritmul a1

```
subalgoritm a1(tablou, n, e):
        g ← false
        i ← 1
        cat-timp i <= n executa
              daca tablou[i] = e atunci:
                    g ← true
              altfel
                    g ← false
              sf-daca
              i ← i + 1
        sf-cat-timp
        @returneaza g
sf-subalgoritm
```

b. Subalgoritmul a2

```
subalgoritm a2(tablou, n, e):
        g ← false
        i ← 1
        cat-timp i <= n si g = false executa
              daca tablou[i] = e atunci:
                    g ← true
              altfel
                    g ← false
              sf-daca
              i ← i + 1
        sf-cat-timp
        @returneaza g
sf-subalgoritm
```

c. Subalgoritmul a3

```
subalgoritm a3(tablou, n, e):
        c ← 0
        pentru i = 1,n executa
                daca tablou[i] = e atunci:
                        c ← c + 1
                altfel
                        c ← c - 1
                sf-daca
        sf-cat-timp
        @returneaza -1 * n ≠ c
sf-subalgoritm
```

d. Subalgoritmul a4

```
subalgoritm a4(tablou, n, e):
        g ← false
        i ← 1
        cat-timp i <= n executa
                daca tablou[i] < e + 1 si tablou[i] % e = 0 atunci:
                        g ← true
                sf-daca
                i ← i + 1
        sf-cat-timp
        @returneaza g
sf-subalgoritm
```

Răspunsuri: b, c

Întrebarea 3a.

Considerați subalgoritmul următor:

```
subalgoritm g(tablou, n, p):
      daca tablou[p] == tablou[n] atunci
            @returneaza true
      altfel
            daca p = n + 1 atunci
                  @returneaza false
            altfel
                  @returneaza g(tablou, n, p+2)
            sf-daca
       sf-daca
sf-subalgoritm
```

Pentru care dintre următoarele date de intrare, algoritmul va returna `true`?

a. tablou = [1, 2, 3, 4, 5, 6, 7, 5], n = 8, p = 1
b. tablou = [1, 2, 3, 4, 4, 6, 7, 7, 5, 5], n = 10, p = 1
c. tablou = [1, 2, 3, 4, 5, 6, 7, 5], n = 6, p = 1
d. tablou = [1, 2, 3, 4, 5, 6, 7], n = 7, p = 1

Întrebarea 3b.

Considerați același subalgoritm `g` (de la întrebarea 3a). Care dintre următoarele afirmații sunt adevărate?

a) Pentru orice tablou cu elemente distincte, n = lungimea tabloului și p = 1, subalgoritmul returnează `false`
b) Pentru orice tablou cu elemente distincte și lungime pară, n = lungimea tabloului și p = 1, subalgoritmul returnează `false`
c) Pentru orice tablou în care ultimul element nu este unic, n = lungimea tabloului și p = 1, subalgoritmul returnează `true`
d) Dându-se un tablou, n = lungimea tabloului și p = 1, subalgoritmul returnează `true` dacă și numai dacă tabloul are un număr impar de elemente, iar ultimul element nu este duplicat

Răspunsuri:

3a. - a, d
3b. - b

Întrebarea 4a.

Considerați subalgoritmul următor:

```
subalgoritm s(tablou, n, p, c):
      daca p = n + 1 atunci
            @returneaza c
      altfel
            daca c > tablou[p] atunci
               @retuneaza s(tablou, n, p+1, c)
            sf-daca
            daca c < tablou[p] atunci
               @returneaza s(tablou, n, p+1, tablou[p])
            sf-daca
            @returneaza 0
      sf-daca
sf-subalgoritm
```

Ce va returna subalgoritmul `s`, dacă îl apelăm cu *tablou = [1, 2, 3, 4, 5, 5], n = 6, p = 1, c = -1*?

a. -1
b. 0
c. 4
d. 5

Întrebarea 4b.

Considerați același subalgoritm `s` (de la întrebarea 4a). De câte ori se execută comparația `c < tablou[p]` dacă apelăm `s` cu *tablou = [-1, 7, 3, 9, 2, 11], n = 6, p = 2, c = -1*?

a. 0
b. 1
c. 3
d. 5

Răspunsuri:

4a. - b
4b. - c