Algoritmi care lucrează cu tablouri unidimensionale

➢ Determinarea elementului care se repetă

Enunț

Se dă un tablou *t* cu *n+1* elemente, în care fiecare element este un număr întreg din intervalul *[1,n]*, *t = (t[1], t[2], ..., t[n+1]), t[i] ∈ {1,2,...,n},* pentru ***i*** *= 1, 2, ..., n+1*. Prin urmare, există cel puțin un element care se repetă. Este posibil inclusiv 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

Variante de rezolvare

Varianta 1

- Se utilizează 2 cicluri repetitive imbricate, comparându-se elementele două câte două

Pseudocod

```
functie duplicat1(t, n):
      i ← 1
      cont  ← true
      cat-timp i <= n si cont executa
              j ← i+1
              cat-timp j <= n+1 si cont executa
                    daca t[i] = t[j] atunci
                          duplicat ← t[i]
                          cont ← false
                    altfel
                          j ← j+1
                    sf-daca
              sf-cat-timp
              i ← i+1
      sf-cat-timp
      duplicat1 ← duplicat
sf_functie
```

OBS: Fiind n+1 elemente în tablou, indicii sunt între 1 și n+1.

Analiza complexității

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

Varianta 2

- Se utilizează 2 cicluri repetitive imbricate, pentru fiecare element posibil (din intervalul [1,n]) numărându-se aparițiile. Se va returna primul element pentru care numărul de apariții este mai mare decât 1.

Pseudocod

```
functie duplicat2(t, n):
     i ← 1
     ap ← 0
     cat-timp i <= n si ap < 2 executa
            j ← 1
            ap ← 0
            cat-timp j <= n+1 si ap < 2  executa
                  daca i = t[j] atunci
                        ap ← ap + 1
                  sf-daca
                  j ← j + 1
            sf-cat-timp
            daca  ap >= 2 atunci
                 duplicat ← i
            altfel
                 i ← i+1
            sf-daca
     sf-cat-timp
     duplicat2 ← duplicat
sf_functie
```

Analiza complexității

- Complexitate de spațiu extra (ignorând tabloul de intrare): Θ(1)
- Complexitate de timp: $O(n^2)$
  - Justificare: $\Theta(n^2)$ în caz defavorabil, când cel mai mic element care se repetă este n, anterioarele fiind unice, caz în care se execută ambele cicluri complet, dar execuția se poate opri și mai repede (de pildă, când 1 este elementul care se repetă), având, în caz favorabil complexitate Θ(1)

Varianta 3

- Se folosește un tablou unidimensional 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.

OBS: Alternativ, se poate folosi un vector de frecvențe, inițializat cu valori de 0.

Pseudocod

```
functie duplicat3(t, n):
      pentru i ← 1,n executa
            aux[i] = false
      sf-pentru
      cont  ← true
      i ← 1
      cat-timp (i<=n+1) si cont executa
             elem  ← t[i]
             daca aux[elem] = true atunci
                    duplicat ← elem
                    cont ← false
             altfel
                   aux[elem] ← true
                   i ← i+1
            sf-daca
      sf-cat-timp
      duplicat3 ← duplicat
sf_functie
```

Analiza complexității

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

Varianta 4

- 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 numere întregi din intervalul [1...n]. Dacă se împarte intervalul valorilor în 2 jumătăți, șirul ar trebui să conțină mai multe elemente dintr-una dintre jumătăți decât lungimea ei (adică numărul de valori întregi din subintervalul respectiv), întrucât cele 2 subintervale au împreună lungimea n, dar șirul conține n+1 elemente. Elementul care se repetă este în jumătatea respectivă și se poate continua căutarea în acel subinterval.

Exemplu

Pentru n=7 și t =[1,2,3,4,5,6,2], se împarte intervalul valorilor posibile, și anume [1,7] în [1,4] și [5,7]. Se numără câte valori din intervalul [1,4] se găsesc în tabloul t. Se obține 5, care este mai mare decât lungimea subintervalului, adică 4. Prin urmare, se poate concluziona că o valoarea care se repetă se poate găsi în prima jumătate. Se continuă căutarea valorii în prima jumătate. Se imparte intervalul valorilor [1,4] în două subintervale: [1,2] și [3,4]. Se numără câte valori din intervalul [1,2] sunt conținute de șir, obținându-se 3. Prin urmare, se poate concluziona că valoarea care se repetă se poate găsi în primul subinterval, și anume [1,2]. Se împarte subintervalul [1,2] în două subintervale constând în câte un singur element, [1] și [2] și se numără elementele din tablou egale cu 1. Se obține 1 și prin urmare, se deduce că elementul care se repetă aparține intervalului [2]. Cum începutul și finalul intervalului sunt identice, se oprește ciclul repetitiv principal și se returnează 2.

Pseudocod

```
functie duplicat4(t, n):
      inceput ← 1
      sfarsit ← n
      cat-timp inceput < sfarsit executa
              mijloc ← [(inceput + sfarsit) / 2]
              count ← 0
              pentru i ← 1,n+1 executa
                    daca t[i] >= inceput si t[i] <= mijloc atunci
                        count ← count + 1
                 sf-daca
              sf-pentru
              lungimeInterval ← mijloc - inceput + 1
               daca lungimeInterval < count
                    sfarsit ← mijloc //Se continuă căutarea în prima jumătate a intervalului
               altfel
                    inceput ← mijloc + 1 //Se continuă căutarea în a doua jumătate a intervalului
              sf-daca
     sf-cat-timp
     duplicat4 ← inceput
sf_functie
```

Analiza complexității

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

Varianta 5

- Se sortează mai întâi tabloul (crescător sau descrescător), iar apoi se parcurge, putându-se returna valoarea primului element egal cu elementul care îi urmează în tablou.

Analiza complexității

- Complexitatea soluției va fi dată de complexitatea algoritmului de sortare aplicat, complexitatea-timp a parcurgerii pentru identificarea unui element repetitiv fiind O(n), iar cea spațiu Θ(1).
    - Dacă se aplică algoritmul de sortare prin interclasare (*Merge Sort*), complexitatea timp va fi $\Theta(n \log_2 n)$, iar cea spațiu va fi Θ(n).
    - Dacă se aplică algoritmul de sortare rapidă (*Quick Sort*), complexitatea timp va fi $O(n^2)$ (deși complexitatea medie rămâne $\Theta(n \log_2 n)$, iar cea spațiu va fi constantă, Θ(1)

Probleme de tip grilă

➢ 1. Considerați problema verificării existenței unui element *e* într-un tablou unidimensional, format din numere naturale, cu lungimea *n, (tablou[1], tablou[2], tablou[3], …, tablou[n])*, unde *tablou[i]* este număr natural. 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
```

- Subalgoritmul nu reprezintă o rezolvare corectă deoarece se va returna true dacă și numai dacă ultimul element al tabloului este egal cu *e*, structura repetitivă efectuând întotdeauna *n* pași, iar g actualizându-se pentru fiecare element al tabloului, inclusiv ultimul.
- Contraexemplu: Pentru e=1, n = 5 și tabloul [1,2,3,4,5] se va returna *false*.

b. Subalgoritmul a2

```
subalgoritm a2(tablou, n, e):
        g ← false
        i ← 1
        cat-timp i <= n și 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
```

- Subalgoritmul este corect.

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
```

- Subalgoritmul este corect întrucât returnează *false* dacă și numai dacă c = -n după efectuarea ciclului repetitiv, condiție îndeplinită doar dacă la fiecare pas al ciclului c este decrementat, adică dacă elementul curent al tabloului este diferit de elementul căutat. Așadar, dacă există cel puțin un element al tabloului egal cu elementul căutat, c va fi mai mare decât -n și se va returna *true*.

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
```

- Subalgoritmul este incorect întrucât returnează *true* dacă și numai dacă există cel puțin un element al tabloului `tablou[i]` pentru care are loc condiția compusă `tablou[i] < e + 1 si tablou[i] % e = 0`, dar această condiție este îndeplinită nu doar dacă `tablou[i] = e`, ci și dacă `tablou[i] = 0`.
- Contraexemplu: Pentru e = 2, n = 5 și tabloul = [1,0,3,4,5] se returnează *true*.

Răspunsuri corecte: b, c

➢ 2. 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
```

2.1. Care dintre următoarele afirmații despre subalgoritmul `ceFace`este adevărată?

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.

2.2. Care dintre următoarele afirmații este adevărată, 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:
2.1. b, c
2.2. a

- 3. Se consideră algoritmul *ceFace(sir, a, b)*, unde *sir* este un vector format din numere naturale nenule distincte ordonate crescător (sir[1], sir[2], ..., sir[n]), sir[i] ∈ N*, pentru i = 1, 2, ..., n și sir[i] < sir[i+1], pentru i = 1, 2, ..., n-1.

```
subalgoritm ceFace(sir, a, b):
   dacă a > b atunci
         returează a
   sf-dacă
   c ← a + [(b - a) / 2] // jumatatea intervalui dintre a si b
   dacă sir[c] = c atunci
         returnează ceFace(sir, c + 1, b)
   altfel
         returnează ceFace(sir, a, c - 1)
   sf-dacă
sf-subalgoritm
```

Care dintre următoarele afirmații sunt adevărate, considerând că apelul inițial este `ceFace`(sir, 1, ***n***)?

a. Dacă vectorul ***șir*** este conține doar elemente din mulțimea primelor ***n*** numere naturale, atunci algoritmul ce face va returna ***n+1***

Fiind ordonat strict crescător și având lungime n, faptul că șirul conține doar elemente din mulțimea primelor n numere naturale implică faptul că șirul este sir=(1,2,3,..n). Prin urmare fiecare element al șirului este egal cu poziția lui. Aceasta înseamnă că verificarea sir[c] = c va fi întotdeauna evaluată la true, deci la fiecare pas în recursivitate se va apela ceFace(sir, mijlocul intervalului + 1, n), al treilea argument nefiind modificat. Așadar, reursivitate se va opri când al doilea argument va ajunge mai mare decât n, adică n+1, returnându-se n+1.

b. Algoritmul `ceFace` returnează poziția *p* cea mai apropiată de [***n***/2] pentru care ***sir****[p]* = *p*

c. Algoritmul `ceFace` returnează poziția *p* cea mai apropiată de [***n***/2] pentru care ***sir****[p]* ≠ *p*

d. Algoritmul `ceFace` returnează cel mai mic număr natural nenul care nu apare în vectorul ***sir***

Răspunsuri corecte: A, D

Contraexemple pentru răsp greșite: Contra-exemplu pentru B si C:
`ceFace`([2, 3, 4, 6, 7, 8], 1, 6) => 1

`ceFace`([2, 3, 4, 6, 7, 8], 1, 6) = ceFace([2, 3, 4, 5, 6, 7, 8],1,3]) = ceFace([2, 3, 4, 5, 6, 7, 8],1,1]) = ceFace([2, 3, 4, 5, 6, 7, 8],1,0]) = 1