Șiruri (tablouri unidimensionale)

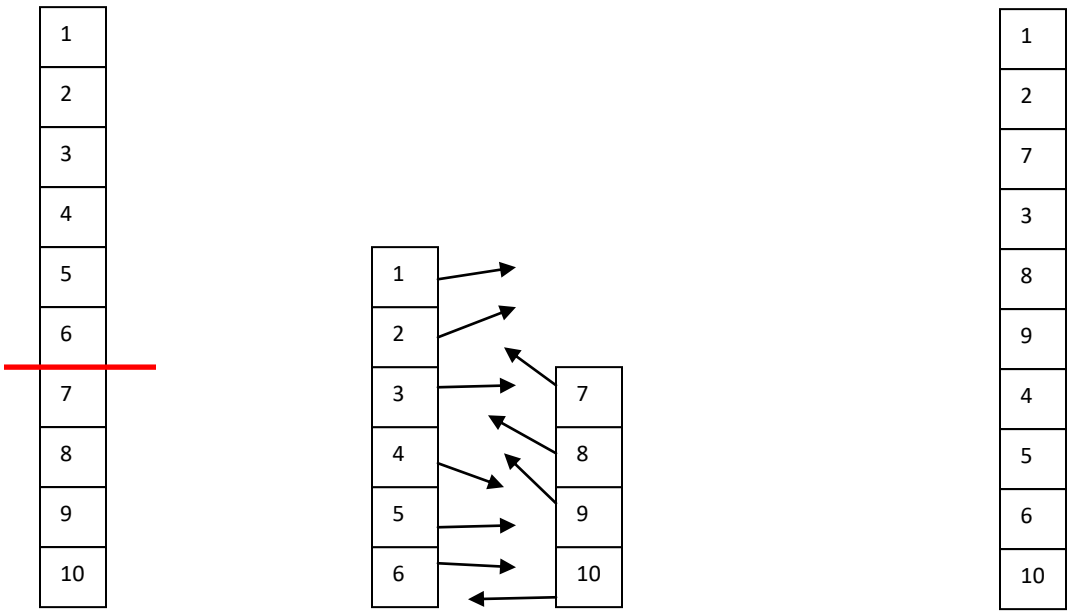
Problema 1

Avem un pachet de *n* cărți, și pe fiecare carte avem un număr unic de la 1 la *n*. Procesul de amestecare al cărților, numit ”riffle shuffle” împarte pachetul aleator în 2 jumătăți și intercalează jumătățile în mod aleator (vezi imaginea din dreapta). 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 sunt *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], putem ajunge la această ordonare cu un singur riffle shuffle (vedeți exemplul de mai jos), dar la ordonare_finală [1, 7, 3, 4, 2, 8, 9, 5, 6, 10] nu putem ajunge.

Pachetul inițial: **Procesul de ”riffle shuffle”** **Pachetul final**



Rezolvare problemei este compusă din 2 părți:

- Să determinăm punctul unde pachetul a fost tăiat (pe baza ordonării finale)
- Având cele 2 jumătăți de pachet (una de la început până la punctul de tăiere, și una de la punctul de tăiere până la capăt) să verificăm dacă se poate ajunge la ordonarea finală.

Determinarea punctului de tăiere

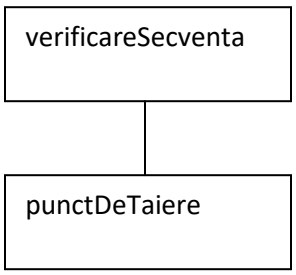
Știm că dacă tăiem pachetul de cărți la un punct *p,* vom avea 2 jumătăți: prima jumătate care începe la poziția 1 și se termină la poziția *p,* și a 2-a jumătate care începe la poziția *p+1* și se termină la poziția *n*. Dacă urmărim exemplul, observăm că ultimul element din ordonarea finală este ultimul element dintr-una dintre jumătăți. Dacă ultimul element din ordonare 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 ordonare inițială, ne uităm la penultimul element din ordonare finală, care trebuie să fie ori ultimul element din prima jumătate, ori penultimul element din a 2-a jumătate, etc.

Verificarea secvenței finale

Odată ce avem punctul de tăiere, parcurgem în paralel pachetul final și cele 2 jumătăți. Parcurgerile se fac de la finalul tabloului spre început (pentru că așa se construiește rezultatul amestecării) și la fiecare pas verificăm din care jumătate provine elementul curent din ordonare finală. În fiecare jumătate vom avea un element curent, și tot timpul unul dintre elementele curente trebuie să fie următorul element din ordonare finală.

Analiză

Identificarea subalgoritmilor



Specificarea subalgoritmilor

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

Proiectare

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

```
funcție punctDeTaiere(n, ordI, ordF) este:
	pozitieOrdI = n			{pornim de la capătul tablourilor}
	pozitieOrdF = n
	câttimp pozitieOrdF > 0 și ordI[pozitieOrdI] = ordF[pozitieOrdF] execută
		pozitieOrdI ← pozitieOrdI - 1
		pozitieOrdF ← pozitieOrdF - 1
	sf_câttimp
	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
		câttimp ordI[pozitieUltimElem] != ultimElemJumatate1 execută
			pozitieUltimElem ← pozitieUltimElem + 1
		sf_câttimp
		punctDeTaiere ← pozitieUltimElem
	sf_dacă
sf_functie

functie verificareSecventa(n, ordI, ordF) este:
	taietura ← punctDeTaiere(n, ordI, ordF)
	dacă taietura = -1 atunci
		verificareSecventa ← true
	altfel
		{verificam jumatatile de la capat si ordonare finala de la capat}
		pozJ1 ← taietura	{prima jumatate e de la poz 1 la poz taietura}
		pozJ2 ← n		{a 2-a jumatate e de la poz taietura+1 la poz n}
		pozOrdF ← n
		cattimp pozOrdF > 0 executa
			dacă pozJ1 > 0 si ordF[pozOrdF] = ordI[pozJ1] atunci
				pozOrdF ← pozOrdF - 1
				pozJ1 ← pozJ1 - 1
			altfel
				dacă pozJ2 > taietura si ordF[pozOrdF] = ordI[pozJ2] atunci
					pozOrdF ← pozOrdF - 1
					pozJ2 ← pozJ2 - 1
				altfel
					verificareSecventa ← false
{elementul din ordF nu se potriveste cu elementul curent din niciuna dintre jumatati}
				sf_daca
```

```
                sf_daca
            sf_cattimp
            verificareSecventa ← true
        sf_daca
sf_functie
```

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
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

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, nu s-a taiat pachetul, 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

Produs maxim numere abundente

Se dă un tablou unidimensional *T* cu *n* ($5 \leq n \leq 10\,000$) elemente numere naturale strict pozitive, mai mici sau egale cu 10000. Se cere un subalgoritm care determină patru numere abundente din tabloul *T* al căror produs este maxim. Subalgoritmul poate apela alți subalgoritmi.
Un număr *nr* se numeşte *abundent* dacă este mai mic decât suma *alicotă* a divizorilor săi. Suma *alicotă* este suma divizorilor pozitivi ai lui *nr*; se include 1, se exclude *nr*.

Parametrii de intrare ai subalgoritmului sunt *n* și *T*, iar cei de ieşire sunt *a*, *b*, *c* și *d*, reprezentând patru elemente din tabloul T, având proprietatea cerută. Dacă problema are mai multe soluții, se cere determinarea uneia singure.
Se garantează că *T* conține cel puțin 4 numere abundente.
Se cer 3 variante de soluții (cu complexități diferite) dar cu complexitate de spațiu constant (fără tablou suplimentar).

- Exemplu: dacă *n* = 10 și *T* = (12, 20, 5, 18, 48, 3, 4, 50, 60, 2), cele 4 numere sunt: a = 20, b = 18, c = 48, d = 60.

Idei:

1. "forța brută"
   - se parcurge tabloul cu 4 cicluri for imbricate, generându-se toate tuplurile (x_i, x_j, x_k, x_l) cu indici distincți, unde fiecare element este un număr abundent
   - se verifică toate produsele posibile
   - complexitate: $T(n) = O(n^4)$ (plus complexiatea verificării nrAbundent)
2. sortare
   - se sortează șirul
   - soluția conține cele mai mari 4 numere abundente
   - complexitate: T(n) = complexitatea algoritmului de sortare
   - se utilizează sortarea prin selecție $\Rightarrow T(n) = O(n^2)$ (plus complexitatea verificării nrAbundent)

3. 4 parcurgeri consecutive
   - se determină primele patru maxime care sunt și numere abundente utilizând un subalgoritm similar cu sortarea prin selecție, dar de complexitate liniară (nu ne trebuie sortare completă, doar cele mai mari 4 numere abundente)
   - cele mai mari patru numere abundente vor fi mutate la sfârșitul tabloului
   - complexitate: T(n) = O(n) (plus complexitatea verificării nrAbundent)

În legătură cu verificarea dacă un număr este abundent:

Numerele din tablou provin din intervalul [1, 10000]. Chiar dacă pe exemple pare că numerele abundente sunt doar numere pare, dacă verificăm fiecare număr din acest interval, vom găsi o serie de numere abundente care sunt impare. Un exemple de număr abundent impar este 945, suma divizorilor este 975 (1+3+315+5+189+7+135+9+105+15+63+21+45+27+35).

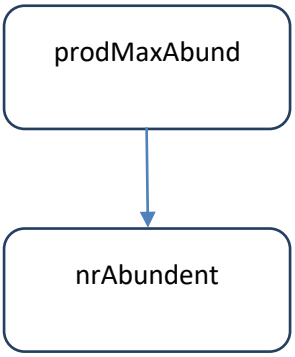
O altă idee menționată la pregătire era că dacă un număr are minim 5 divizori, el este abundent. Verificând numerele până la 10000, observăm că acest lucru NU este adevărat (un contraexemplu este numărul 32, care are 5 divizori 1, 2, 16, 4, 8, dar suma acestor divizori este 31). E adevărat că orice număr abundent are minim 5 divizori, dar această informație nu ne ajută la verificare mai eficientă.

Observație: La fiecare variantă de soluție vom da specificații și implementare pseudocod doar pentru subalgoritmi și funcții care nu au fost definiți în variante anterioare. De exemplu, funcția *nrAbundent* este folosită de toate variantele de soluții, dar este implementată doar la prima. Presupunem că restul variantelor folosesc aceeași implementare.

1. T(n) = O(n^4) (plus complexitatea verificarii nrAbundent – sqrt(nr))

Analiză

Identificarea subalgoritmilor



Specificarea subalgoritmilor

Subalgoritmul **prodMaxAbund(T, n, a, b, c, d)**:
Descriere: determină patru numere abundente din vectorul *T*, astfel încât produsul lor să fie maxim
Date: *T* - tablou unidimensional, $T = (t_i | i = 1..n, t_i \in N^*, t_i \in [1, 10000])$
$n \in N$ - lungimea vectorului *T*, $5 \leq n \leq 10\,000$
Rezultate: $a, b, c, d \in N^*$ - patru numere naturale din vectorul *T*, astfel încât produsul lor să fie maxim și toate să fie abundente

Functia **nrAbundent(nr)**:
Descriere: determină dacă numărul natural pozitiv *nr* este abundent
Date: $n \in N^*$
Rezultate: returnează *true* dacă *nr* este abundent, *false* altfel

Proiectare

```
subalgoritmul prodMaxAbund(T, n, a, b, c, d) este:
	a ← 0
	b ← 0
	c ← 0
	d ← 0
	i ← 1
	cat-timp i <= n - 3 executa			{se testeaza toate produsele posibile}
		daca nrAbundent(T[i]) atunci {in cel mai rau caz, sunt C(n,4) produse}
			j ← i + 1
			cat-timp j <= n - 2 executa
				daca nrAbundent(T[j]) atunci
					k ← j + 1
					cat-timp k <= n - 1 executa
						daca nrAbundent(T[k]) atunci
							l ← k + 1
							cat-timp l <= n executa
								daca nrAbundent(T[l]) atunci
									daca  T[i]*T[j]*T[k]*T[l]>a*b*c*d
									atunci
										a ← T[i]
										b ← T[j]
										c ← T[k]
										d ← T[l]
									sf-daca
								sf-daca
								l ← l + 1
							sf-cat-timp
						sf-daca
						k ← k + 1
					sf-cat-timp
				sf-daca
				j ← j + 1
			sf-cat-timp
		sf-daca
		i ← i + 1
	sf-cat-timp
sf-subalgoritm

functia nrAbundent(nr) este:
	daca nr = 1 atunci
		nrAbundent ← false
	sf-daca
	s ← 1
	div ← 2
	câttimp div * div  <=  nr executa
		daca nr % div = 0 atunci
			s ← s + div
			daca nr/div != div atunci //avem 2 divizori diferiti
				s ← s + nr/div
			sf_daca
		sf-daca
		div ← div + 1
	sf-cattimp
	daca nr < s atunci
		nrAbundent ← true
	altfel
		nrAbundent ← false
sf-functie
```

Cod sursă C++

```
bool nrAbundent(int nr)
{
	if (nr == 1) {
		return false;
	}
	int s = 1; // 1 e tot timpul divizor
	int div = 2;
	while (div * div <= nr) {
		if (nr % div == 0) {
			s += div;
			if (nr / div != div) {//verificam sa avem 2 divizori diferiti
				s += nr / div;
			}
		}
		div++;
	}
	if (nr < s)
		return true;
	return false;
}


void prodMaxAbund(int T[], int n, int& a, int& b, int& c, int& d)
{ //forta bruta cu O(n^4)
	int i, j, k, l;
	a = 0
	b = 0
	c = 0
	d = 0
	i = 0;
	while (i < n - 3)
	{
		if (nrAbundent(T[i]))
		{
			j = i + 1;
			while (j < n - 2)
			{
				if (nrAbundent(T[j]))
				{
					k = j + 1;
					while (k < n - 1)
					{
						if (nrAbundent(T[k]))
						{
							l = k + 1;
							while (l < n)
							{
								if (nrAbundent(T[l]))
									if (((long(T[i])) * T[j] *
T[k] * T[l]) > (long(a)) * b * c * d)

									{
										a = T[i];
										b = T[j];
										c = T[k];
										d = T[l];
									}
								l += 1;
							}
						}
						k += 1;
```

```
                                }
                            }
                            j += 1;
                        }
                }
                i += 1;
        }
}
```

Cod sursă Pascal

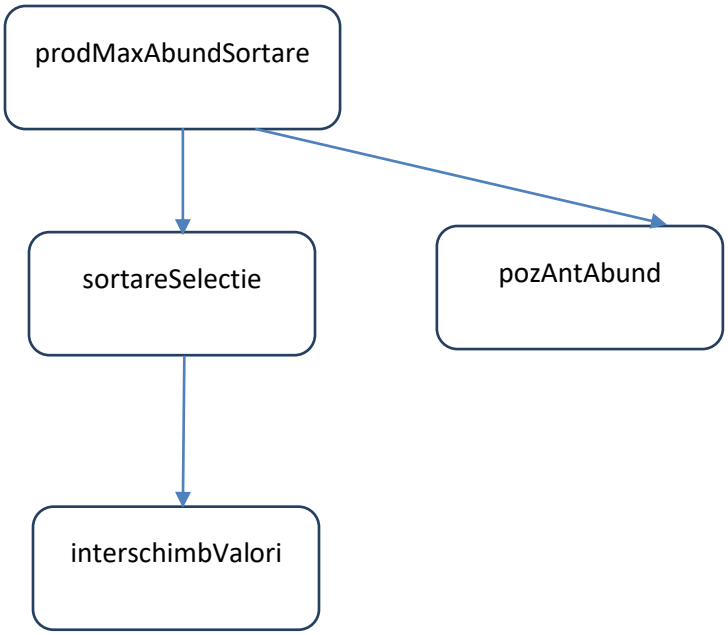
```
program Problema2;

type
    myArray = array[1..10000] of integer;

function nrAbundent(nr: integer): boolean;
var
    sumaDiv :integer;
    divCurent: integer;
    result: boolean;
begin
    if (nr = 1) then
        sumaDiv := 0
    else
    begin
        sumaDiv := 1;
        divCurent := 2;
        while (divCurent * divCurent <= nr) do
        begin
            if (nr mod divCurent = 0) then
            begin
                sumaDiv := sumaDiv + divCurent;
                if (nr  div divCurent <> divCurent) then
                begin
                    sumaDiv := sumaDiv + nr div divCurent;
                end;
            end;
            divCurent := divCurent + 1;
        end;
     end;
     if sumaDiv > nr then
         result:= true
     else
         result:= false;
    nrAbundent := result;
end;

procedure prodMaxAbund(elemente: myArray; n: integer; var a: integer; var b:
integer; var c:integer; var d:integer);
var
    i, j, k, l: integer;
    p1, p2: int64;
begin
    a := 0;
    b := 0;
    c := 0;
```

```
    d := 0;
    i := 1;
    while (i <= n-3) do
    begin
        if (nrAbundent(elemente[i])) then
        begin
            j := i + 1;
            while (j <= n - 2) do
            begin
                if (nrAbundent(elemente[j])) then
                begin
                    k := j + 1;
                    while (k <= n - 1) do
                    begin
                        if (nrAbundent(elemente[k])) then
                        begin
                            l := k + 1;
                            while (l <= n) do
                            begin
                                if (nrAbundent(elemente[l])) then
                                begin
                                    p1 := a * b * c * d;
                                    p2 := elemente[i] * elemente[j] * elemente[k]
* elemente[l] ;
                                    if (p2 > p1) then
                                    begin
                                        a := elemente[i];
                                        b := elemente[j];
                                        c := elemente[k];
                                        d := elemente[l];
                                    end;
                                end;
                                l := l + 1;
                            end;
                        end;
                        k := k + 1;
                    end;
                end;
                j := j + 1;
            end;
        end;
        i := i + 1;
    end;
end;
```

2. T(n) = O(n^2) (plus complexitatea verificarii nrAbundent – sqrt(nr))

Analiză

Identificarea subalgoritmilor



Specificarea subalgoritmilor

Subalgoritmul **prodMaxAbundSortare(T, n, a, b, c, d)**:
Descriere: determină patru numere abundente din vectorul *T*, astfel încât produsul lor să fie maxim
Date: *T* - tablou unidimensional, $T = (t_i | i = 1..n, t_i \in N^*, t_i \in [1, 10000])$
$n \in N$ - lungimea vectorului *T*, $5 \leq n \leq 10\,000$
Rezultate: $a, b, c, d \in N^*$ - patru numere naturale din vectorul *T*, astfel încât produsul lor să fie maxim și toate să fie abundente

Subalgoritmul **sortareSelectie(T, n)**:
Descriere: se sortează crescător tabloul *T* prin metoda selecției
Date: *T* - tablou unidimensional, $T = (t_i | i = 1..n, t_i \in N^*, t_i \in [1, 10000])$
$n \in N$ - lungimea vectorului *T*, $5 \leq n \leq 10\,000$
Rezultate: vectorul *T* sortat crescător

Subalgoritmul **interschimbValori(*a*, b*)**:
Descriere: interschimbă valorile *a* și *b*
Date: $a, b \in Z$
Rezultate: *a*, *b* actualizate - *a* conține vechea valoare a lui *b*, iar *b* conține vechea valoarea a lui *a*

Functia **pozAntAbund(T, poz)**:
Descriere: caută poziția primului număr abundent din vectorul *T*, începând de la poziția *poz* inclusiv și mergând către începutul vectorului
Date: *T* - tablou unidimensional, cu numere in intervalul [1, 10000]

poz ∈ N - poziția începând de la care se caută primul număr abundent

Rezultate: *k* ∈ N - poziția primului număr abundent găsit începând cu poziția *poz* și mergând către primul element; dacă există un astfel de număr, *1 ≤ k ≤ poz*; altfel, *k* = -1

Proiectare

```
subalgoritmul prodMaxAbundSortare (T, n, a, b, c, d) este:
       sortareSelectie(T, n)      {sorteaza sirul prin metoda selectiei}
      {alege solutia - cele mai mari 4 numere care sunt si abundente }
       poz ← pozAntAbund(T,  n)
       a ← T[poz]
       poz ← pozAntAbund(T, poz-1)
       b ← T[poz]
       poz ← pozAntAbund(T, poz-1)
       c ← T[poz]
       poz ← pozAntAbund(T, poz-1)
       d ← T[poz]
sf-subalgoritm

functia pozAntAbund(T, poz) este:
       cat-timp poz >= 1 AND NOT nrAbundent(T[poz]) executa
              poz ← poz - 1
       sf-cat-timp
       daca poz = 0 atunci
              pozUrmAbund ← -1
       altfel
              pozUrmAbund ← poz
       sf-daca
sf-functie

subalgoritmul sortareSelectie(T, n) este:
       pentru i ← 1, n-1 executa  {parcurge sirul de la primul la penultimul element}
              pozitieMin ← i      {la fiecare pas alege minimul dintre valorile ramase}
              pentru j ← i+1, n executa        {si il plaseaza pe pozitia curenta, i}
                     daca T[j] < T[pozitieMin] atunci
                            pozitieMin ← j
                     sf-daca
              sf-pentru
              daca pozitieMin ≠ i atunci
                     interschimbValori(T[i], T[pozitieMin])
              sf-daca
       sf-pentru
sf-subalgoritm

subalgoritmul interschimbValori(a, b) este:
        aux ← a
        a ← b
        b ← aux
Sf-subalgoritm
```

Cod sursă C++

```
void  interschimbValori(int& a, int& b)
{
       int aux;
       aux = a;
       a = b;
       b = aux;
}
```

```
void sortareSelectie(int T[], int n)
{
	int pozMin, aux;
	for (int i = 0; i < n-1; i++)
	{
		pozMin = i;
		for (int j = i + 1; j < n; j++)
		{
			if (T[j] < T[pozMin])
				pozMin = j;
		}
		if (pozMin != i)
			interschimbValori(T[i], T[pozMin]);
	}
}

bool nrAbundent(int nr)
{
	if (nr == 1) {
		return false;
	}
	int s = 1; // 1 e tot timpul divizor
	int div = 2;
	while (div * div <= nr) {
		if (nr % div == 0) {
			s += div;
			if (nr / div != div) {//verificam sa avem 2 divizori diferiti
				s += nr / div;
			}
		}
		div++;
	}
	if (nr < s)
		return true;
	return false;
}

int pozAntAbund(int T[], int poz)
{
	while (poz >= 0 && !(nrAbundent(T[poz])))
		poz -= 1;
	if (poz == -1)
		return -1;
	return poz;
}

void prodMaxAbundSortare(int T[], int n, int& a, int& b, int& c, int& d)
{
	sortareSelectie(T, n);
	int poz = pozAntAbund(T, n-1);
	a = T[poz];
	poz = pozAntAbund(T, poz - 1);
	b = T[poz];
	poz = pozAntAbund(T, poz - 1);
	c = T[poz];
	poz = pozAntAbund(T, poz - 1);
	d = T[poz];
}
```

Cod sursă Pascal

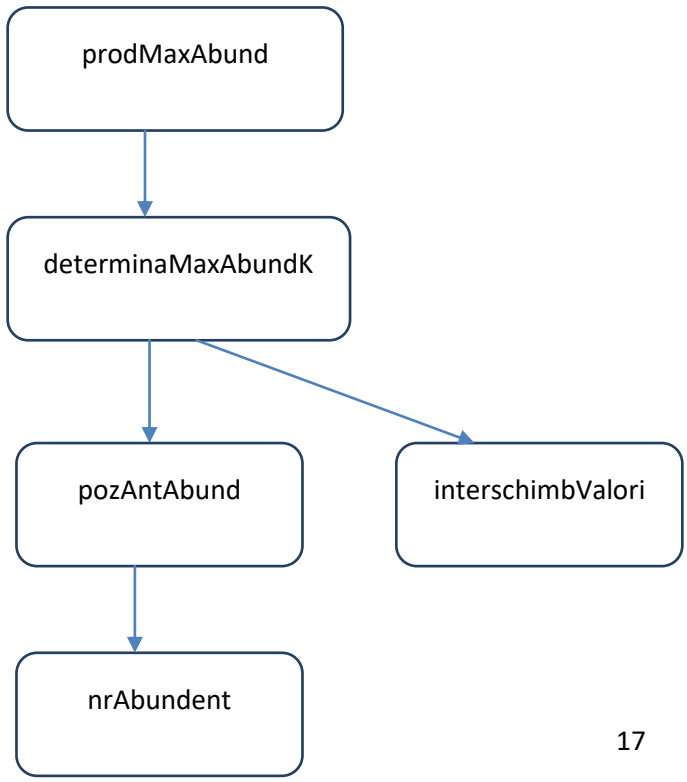
```
program Problema2;

type
    myArray = array[1..10000] of integer;

function nrAbundent(nr: integer): boolean;
var
    sumaDiv :integer;
    divCurent: integer;
    result: boolean;
begin
    if (nr = 1) then
        sumaDiv := 0
    else
    begin
        sumaDiv := 1;
        divCurent := 2;
        while (divCurent * divCurent <= nr) do
        begin
            if (nr mod divCurent = 0) then
            begin
                sumaDiv := sumaDiv + divCurent;
                if (nr  div divCurent <> divCurent) then
                begin
                    sumaDiv := sumaDiv + nr div divCurent;
                end;
            end;
            divCurent := divCurent + 1;
        end;
     end;
     if sumaDiv > nr then
         result:= true
     else
         result:= false;
    nrAbundent := result;
end;

function pozAntAbund(elemente: myArray; poz: integer):integer;
begin
    while ((poz >= 1) and (not nrAbundent(elemente[poz]))) do
    begin
        poz := poz - 1;
    end;
    if poz = 0 then
        pozAntAbund := -1
    else
        pozAntAbund := poz;
end;

procedure interSchimbValori(var a: integer; var b: integer);
var
    temp: integer;
begin
    temp := a;
    a := b;
    b := temp;
end;

procedure sortareSelectie(var elemente: myArray; n: integer);
var
    i,j, pozMin: integer;
begin
```

```
    for i := 1 to n-1 do
    begin
        pozMin := i;
        for j:= i+1 to n do
        begin
            if (elemente[j] < elemente[pozMin]) then
                pozMin := j;
        end;
        if (pozMin <> i) then
            interSchimbValori(elemente[i], elemente[pozMin]);
    end;
end;

procedure prodMaxAbundSortare(elemente: myArray; n:integer; var a:integer; var b: integer;
var c:integer; var d: integer);
var
    poz: integer;
begin
    sortareSelectie(elemente, n);
    poz := pozAntAbund(elemente, n);
    a := elemente[poz];
    poz := pozAntAbund(elemente, poz-1);
    b := elemente[poz];
    poz := pozAntAbund(elemente, poz-1);
    c := elemente[poz];
    poz := pozAntAbund(elemente, poz-1);
    d := elemente[poz];
end;
```

3. T(n) = O(n) (plus complexitatea verificarii sqrt(nr))

Analiză

Identificarea subalgoritmilor



Specificarea subalgoritmilor

Subalgoritmul **prodMaxAbund (T, n, a, b, c, d)**:
Descriere: determină patru numere abundente din vectorul *T*, astfel încât produsul lor să fie maxim
Date: *T* - tablou unidimensional, $T = (t_i | i = 1..n, t_i \in N^*, t_i \in [1, 10000])$
n ∈ N - lungimea vectorului *T*, 5 ≤ *n* ≤ 10 000
Rezultate: *a*, *b*, *c*, *d* ∈ N^* - patru numere naturale din vectorul *T*, astfel încât produsul lor să fie maxim și toate să fie abundente

Subalgoritmul **determinaMaxAbundK(T, n, k)**:
Descriere: determină primele k numere maxime care sunt abundente din tabloul T și le plasează pe ultimele k poziții în șir (k iterații dintr-o sortare prin selecție)
Date: *T* - tablou unidimensional, $T = (t_i | i = 1..n, t_i \in N^*, t_i \in [1, 10000])$
n ∈ N - lungimea vectorului *T*, 5 ≤ *n* ≤ 10 000
k ∈ N - numărul de numere maxime care se dorește a se determina
Rezultate: șirul *T* actualizat: ultimele *k* poziții sunt ocupate, în ordine crescătoare, de primele *k* numere maxime din *T* care sunt abundente
Obs. Pentru *k* = 4, T(n) = O(n) //

Proiectare

```
subalgoritmul prodMaxAbundSortare(T, n, a, b, c, d) este:
       {determina si plaseaza primele 4 maxime abundente pe ultimele 4 poziţii}
       determinaMaxAbundK(T, n, 4)
       a ← T[n]
       b ← T[n-1]
       c ← T[n-2]
       d ← T[n-3]
sf-subalgoritm

subalgoritmul determinaMaxAbundK(T, n, k) este:
       pentru i ← n, n-k+1,-1 executa
              poz ← i
              daca NOT nrAbundent(T[i]) atunci
                     l ← pozAntAbund(T, i-1)
                     interschimbValori(T[i], T[l])
              altfel
                     l ← i
              sf-daca
              pentru j ← l-1, 1, -1 executa
                     daca T[j] > T[poz] AND nrAbundent(T[j])  atunci
                            poz ← j
                     sf-daca
              sf-pentru
              daca i ≠ poz atunci
                     interschimbValori(T[i], T[poz])
              sf-daca
       sf-pentru
Sf-subalgoritm
```

Cod sursă C++

```
void  interschimbValori(int& a, int& b)
{
	int aux;
	aux = a;
	a = b;
	b = aux;
}

bool nrAbundent(int nr)
{
	if (nr == 1) {
		return false;
	}
	int s = 1; // 1 e tot timpul divizor
	int div = 2;
	while (div * div <= nr) {
		if (nr % div == 0) {
			s += div;
			if (nr / div != div) {//verificam sa avem 2 divizori diferiti
				s += nr / div;
			}
		}
		div++;
	}
	if (nr < s)
		return true;
	return false;
}

int pozAntAbund(int T[], int poz)
{
	while (poz >= 0 && !(nrAbundent(T[poz])))
		poz -= 1;
	if (poz == -1)
		return -1;
	return poz;
}

void determinaMaxAbundK(int T[], int n, int k)
{
	int l;
	for (int i = n-1; i >=  n - k; i--)
	{                                   //pentru k=4 avem T(n)=O(n)
		int poz = i;
		if (!(nrAbundent(T[i])))
		{
			l = pozAntAbund(T, i - 1);
			interschimbValori(T[i], T[l]);
		}
		else
			l = i;

		for (int j = l - 1; j >= 0; j--)
			if (nrAbundent(T[j]) && T[j]>T[poz])
				poz = j;
		if (i != poz) {
			interschimbValori(T[i], T[poz]);
		}
	}
}
```

```
void prodMaxAbund(int T[], int n, int& a, int& b, int& c, int& d)
{
       determinaMaxAbundK(T, n, 4);
       a = T[n - 1];
       b = T[n - 2];
       c = T[n - 3];
       d = T[n - 4];
}
```

Cod sursă Pascal

```
program Problema2;

type
    myArray = array[1..10000] of integer;

function nrAbundent(nr: integer): boolean;
var
    sumaDiv :integer;
    divCurent: integer;
    result: boolean;
begin
    if (nr = 1) then
        sumaDiv := 0
    else
    begin
        sumaDiv := 1;
        divCurent := 2;
        while (divCurent * divCurent <= nr) do
        begin
            if (nr mod divCurent = 0) then
            begin
                sumaDiv := sumaDiv + divCurent;
                if (nr  div divCurent <> divCurent) then
                begin
                    sumaDiv := sumaDiv + nr div divCurent;
                end;
            end;
            divCurent := divCurent + 1;
        end;
     end;
     if sumaDiv > nr then
         result:= true
     else
         result:= false;
    nrAbundent := result;
end;

function pozAntAbund(elemente: myArray; poz: integer):integer;
begin
    while ((poz >= 1) and (not nrAbundent(elemente[poz]))) do
    begin
        poz := poz - 1;
    end;
    if poz = 0 then
        pozAntAbund := -1
    else
        pozAntAbund := poz;
end;
```

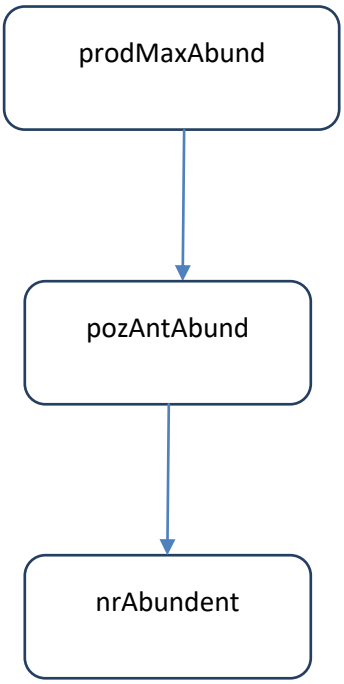
```
procedure interSchimbValori(var a: integer; var b: integer);
var
    temp: integer;
begin
    temp := a;
    a := b;
    b := temp;
end;

procedure determinaMaxAbundK(var elemente: myArray; n:integer; k:integer);
var
    i,j, l, poz: integer;
begin
    for i := n downto n-k+1 do
    begin
        poz := i;
        if (not nrAbundent(elemente[i])) then
        begin
            l := pozAntAbund(elemente, i-1);
            interschimbValori(elemente[l], elemente[i])
        end
        else
            l := i;
        for j := l-1 downto 1 do
        begin
            if ((elemente[j] > elemente[poz]) and (nrAbundent(elemente[j]))) then
                poz := j;
        end;
        if poz <> i then
            interschimbValori(elemente[i], elemente[poz]);
    end;
end;

procedure prodMaxAbund(elemente: myArray; n:integer; var a:integer; var b: integer; var
c:integer; var d: integer);
begin
   determinaMaxAbundK(elemente, n, 4);
   a := elemente[n];
   b := elemente[n-1];
   c := elemente[n-2];
   d := elemente[n-3];
end;
```

3 (varianta b) discutată la pregătire cu complexitate T(n) = O(n) (plus complexitatea verificarii sqrt(nr))

- Putem rezolva problema cu o singură parcurgere, reținând în cele 4 variabile de ieșire (a, b, c, d) cele mai mari numere abundente găsite până la momentul respectiv. Vom reține aceste 4 valori astfel încât a >= b >= c >= d (aceste relații ne ajută să știm exact care numere trebuie înlocuite când găsim un nou număr abundent).

Analiză

Identificarea subalgoritmilor



Specificarea subalgoritmilor

Subalgoritmul **prodMaxAbund (T, n, a, b, c, d)**:
Descriere: determină patru numere abundente din vectorul *T*, astfel încât produsul lor să fie maxim
Date: *T* - tablou unidimensional, $T = (t_i | i = 1..n, t_i \in N^*, t_i \in [1, 10000])$
$n \in N$ - lungimea vectorului *T*, $5 \leq n \leq 10\,000$
Rezultate: $a, b, c, d \in N^*$ - patru numere naturale din vectorul *T*, astfel încât produsul lor să fie maxim și toate să fie abundente

Proiectare

```
subalgoritmul prodMaxAbund (T, n, a, b, c, d):
	a ← 0
	b ← 0
	c ← 0
	d ← 0
	index ← n {incepem cautarea de la sfarsit}
	cattimp index > 0 executa
		index ← pozAntAbund(T, index)
		daca index ≠ -1 atunci
			daca T[index] >= a atunci {avem un nou maxim}
				d ← c
				c ← b
				b ← a
				a ← T[index]
			altfel
				daca T[index] >= b atunci {a ramane, restul se schimba}
					d ← c
					c ← b
					b ← T[index]
				altfel
					daca T[index] >= c atunci {a, b raman, c, d se schimba}
						d ← c
						c ← T[index]
					altfel
						daca T[index] > d atunci {doar d se schimba}
							d ← T[index]
						sf-daca
					sf-daca
				sf-daca
			sf-daca
		sf-daca
		index ← index - 1
	sf-cattimp
sf-subalgoritm
```

Cod sursă C++

```
bool nrAbundent(int nr) {
	if (nr == 1) {
		return false;
	}
	int s = 1;
	int div = 2;
	while (div * div <= nr) {
		if (nr % div == 0) {
			s += div;
			if (nr / div != div) {
				s += nr / div;
			}
		}
		div = div + 1;
	}
	if (s > nr) {
		return true;
	}
	else {
		return false;
```

```
	}
}

int pozAntAbund(int T[], int poz)
{
	while (poz >= 0 && !(nrAbundent(T[poz])))
		poz -= 1;
	if (poz == -1)
		return -1;
	return poz;
}

void prodMaxAbund(int T[], int n, int& a, int& b, int& c, int& d) {
	a = 0;
	b = 0;
	c = 0;
	d = 0;

	int index = n - 1;
	while (index > -1) {
		index = pozAntAbund(T, index);
		if (index != -1) {
			if (T[index] >= a) {
				d = c;
				c = b;
				b = a;
				a = T[index];
			}
			else if (T[index] >= b) {
				d = c;
				c = b;
				b = T[index];
			}
			else if (T[index] >= c) {
				d = c;
				c = T[index];
			}
			else if (T[index] > d) {
				d = T[index];
			}
		}
		index--;
	}
}
```

Cod sursă Pascal

```
program Problema2.3b;

type
    myArray = array[1..10000] of integer;

function nrAbundent(nr: integer): boolean;
var
    sumaDiv :integer;
    divCurent: integer;
    result: boolean;
begin
    if (nr = 1) then
        sumaDiv := 0
    else
    begin
        sumaDiv := 1;
        divCurent := 2;
        while (divCurent * divCurent <= nr) do
        begin
            if (nr mod divCurent = 0) then
            begin
                sumaDiv := sumaDiv + divCurent;
                if (nr  div divCurent <> divCurent) then
                begin
                    sumaDiv := sumaDiv + nr div divCurent;
                end;
            end;
            divCurent := divCurent + 1;
        end;
     end;
     if sumaDiv > nr then
         result:= true
     else
         result:= false;
    nrAbundent := result;
end;

function pozAntAbund(elemente: myArray; poz: integer):integer;
begin
    while ((poz >= 1) and (not nrAbundent(elemente[poz]))) do
    begin
        poz := poz - 1;
    end;
    if poz = 0 then
        pozAntAbund := -1
    else
        pozAntAbund := poz;
end;

procedure prodMaxAbund(elemente: myArray; n:integer; var a:integer; var b: integer; var
c:integer; var d: integer);
var
    index: integer;
begin
   a := 0;
   b := 0;
   c := 0;
   d := 0;
   index := n;
   while index > 0 do
```

```
    begin
        index := pozAntAbund(elemente, index);
        if index <> -1 then
        begin
            if elemente[index] >= a then
            begin
                d := c;
                c := b;
                b := a;
                a := elemente[index];
            end
            else if elemente[index] >= b then
            begin
                d := c;
                c := b;
                b := elemente[index];
            end
            else if elemente[index] >= c then
            begin
                d := c;
                c := elemente[index];
            end
            else if elemente[index] > d then
            begin
                d := elemente[index]
            end;
        end;
        index := index - 1;
    end;
end;
```

Exemple

Date de intrare		Rezultate
n	T	cele patru numere
6	99, 90, 20, 100, 2, 108	108, 100, 90, 20
7	24, 5, 2, 20, 18, 12, 1	24, 20, 18, 12
8	84, 30, 55, 70, 40, 10, 102, 60	84, 70, 102, 60

Tema

Ce alte soluții sunt posibile dacă renunțăm la cerința de a nu avea tablou auxiliar?
Idei:

- O parcurgere a tabloului inițial și copierea numerelor abundente într-un tablou auxiliar. Oricare dintre soluțiile de mai sus poate fi aplicată pe acest tablou auxiliar, unde nu vor mai fi necesare verificări (toate numerele sunt abundente).

- Un tablou auxiliar de 10.000 de elemente (numărul maxim din tablou inițial este 10.000) booleane, aux[i] = true, dacă i este număr abundent, false altfel. După ce tabloul este construit, toate verificările de număr abundent pot fi făcute în Θ(1) în acest tablou. Dar asta înseamnă 10.000 de verificări număr abundent, se merită mai mult dacă ar trebui să citesc mai multe (foarte multe) șiruri sau numere.
- Un tablou auxiliar de 10.000 de elemente (numărul maxim din tablou inițial e 10.000) intregi (-1, 0, 1), inițial toate valorile fiind -1. aux[i] este -1 dacă nu știm dacă i este număr abundent sau nu, 0 dacă nu e abundent și 1 dacă este abundent. Când un număr, nr, trebuie verificat, verificăm aux[nr]. Dacă este -1 apelăm funcția de verificare, dacă este 0 sau 1 știm deja dacă este abundent sau nu. Astfel, verificăm doar acele numere care chiar sunt în șir (dar am un tablou auxiliar imens).

Problema 3

Se cere implementarea unei aplicații care gestionează situația absențelor într-un liceu. Între altele, aplicația va permite:

1. citirea a două numere naturale $p \in \{1, ..., 17\}$ și $q > 0$, a unei săptămâni de început s și a unei săptămâni de sfârșit $f, s, f \in \{1, ..., 36\}$, astfel încât $f - s >= 2*p$;

2. citirea numărului total de absențe și a numărului total de absențe motivate săptămânal, din săptămâna s până în săptămâna f inclusiv, pentru două clase de elevi, XII A și XII B; numărul săptămânal de absențe și numărul săptămânal de absențe motivate sunt numere naturale;

3. determinarea și afișarea celei mai lungi perioade de timp în care are loc următorul fenonem, atât în clasa XII A, cât și în clasa XII B: o perioadă cu p creșteri consecutive ale numărului de absențe nemotivate este urmată de o perioadă cu p scăderi consecutive ale numărului de absențe nemotivate și invers, o perioadă cu p scăderi consecutive ale numărului de absențe nemotivate este urmată de o perioadă cu p creșteri consecutive ale numărului de absențe nemotivate; pentru perioada identificata, secventa corespunzatoare clasei XII A poate fi simetrica cu cea a clasei XII B (de exemplu în clasa XII A p creșteri urmate de p scăderi și în XII B p scăderi urmate de p creșteri); se afișează și numărul de secvențe cu creșteri, respectiv scăderi; începând cu a doua săptămână s_i a perioadei identificate, se afișează și diferența dintre numărul de absențe nemotivate din săptămâna s_i și numărul de absențe nemotivate din săptămâna precedentă s_{i-1}; în cazul în care în două săptămâni consecutive numărul de absențe nemotivate este identic, se consideră că proprietatea nu este îndeplinită;

- cea mai lungă perioadă de timp căutată are cel puțin $2*p+1$ săptămâni consecutive: p creșteri urmate de p descreșteri au loc într-o perioadă care include $2*p+1$ săptămâni consecutinve;
- în cazul în care există mai multe soluții - mai multe perioade de timp care au aceeași lungime - se afișează una dintre ele;

4. determinarea celei mai lungi perioade în care numărul de absențe nemotivate scade, de la o săptămână la alta, cu cel puțin q% în XII B - **temă**.

Exemplu

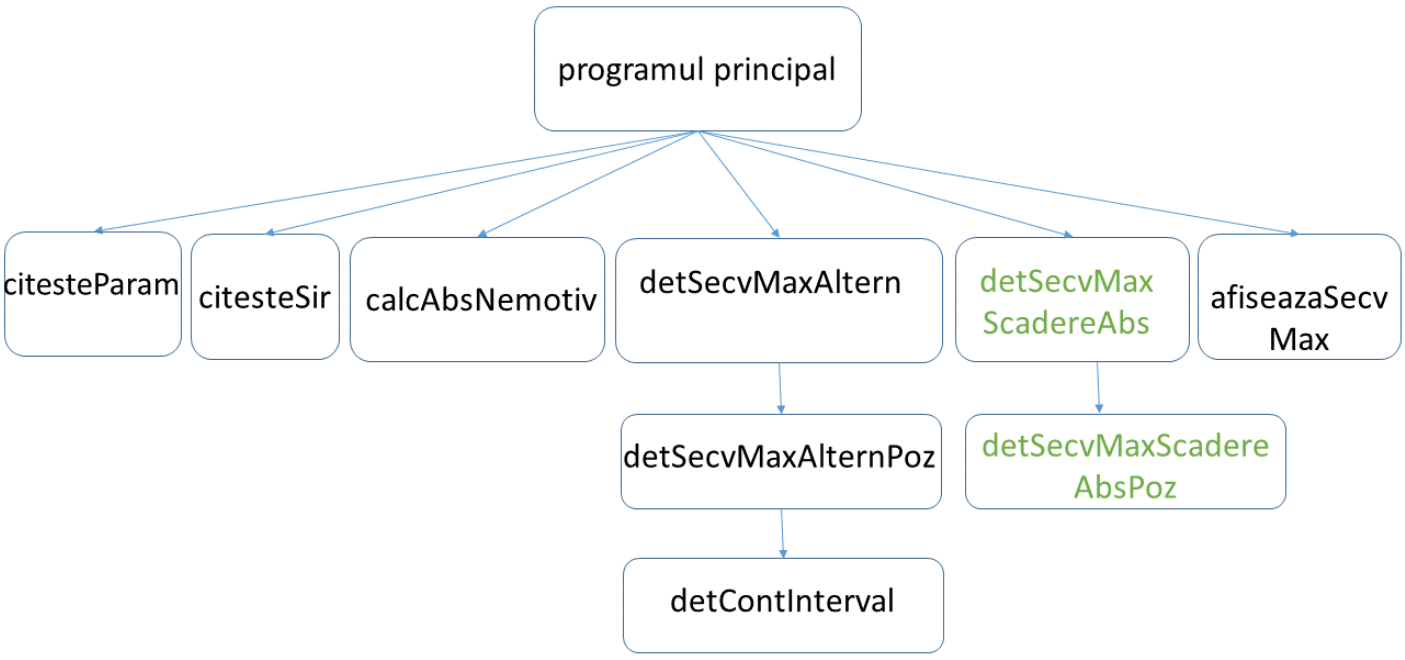
- săptămâna de început s = 4 și săptămâna de sfârșit f = 21
- număr de absențe din săptămâna 4 până în săptămâna 21 pentru clasa XII A: 450, 433, 410, 420, 455, 470, 481, 500, 490, 470, 482, 487, 496, 505, 501, 495, 515, 520
- număr de absențe motivate din săptămâna 4 până în săptămâna 21 pentru clasa XII A: 395, 378, 357, 367, 394, 400, 399, 420, 411, 397, 407, 396, 402, 415, 415, 406, 425, 425
- număr de absențe nemotivate din săptămâna 4 până în săptămâna 21 pentru clasa XII A: 55, 55, 53, 53, 61, 70, 82, 80, 79, 73, 75, 91, 94, 90, 86, 89, 90, 95

- număr de absențe din săptămâna 4 până în săptămâna 21 pentru clasa XII B: 431, 439, 435, 427, 437, 445, 459, 465, 460, 450, 470, 487, 497, 491, 482, 473, 473, 470
- număr de absențe motivate din săptămâna 4 până în săptămâna 21 pentru clasa XII B: 386, 388, 385, 378, 386, 390, 402, 410, 410, 410, 429, 432, 440, 436, 430, 422, 421, 417
- număr de absențe nemotivate din săptămâna 4 până în săptămâna 21 pentru clasa XII B: 45, 51, 50, 49, 51, 55, 57, 55, 50, 40, 41, 55, 57, 55, 52, 51, 52, 53
- pentru *p = 3*: cea mai lungă perioadă în care se înregistrează *p* scăderi consecutive urmate de *p* creșteri consecutive ale numărului de absențe nemotivate în ambele clase este s_7-s_{16}; sunt 3 secvențe de creșteri / scăderi în acest interval; XIIA înregistrează în această perioadă 53, 61, 70, 82, 80, 79, 73, 75, 91, respectiv 94 de absențe nemotivate iar XII B 49, 51, 55, 57, 55, 50, 40, 41, 55, respectiv 57 de absențe nemotivate, deci diferențele între numărul de absențe nemotivate dintr-o săptămână și cel din săptămâna precedentă, începând cu a doua săptămână din interval, sunt: pentru XII A (8, 9, 12, -2, -1, -6, 2, 16, 3), iar pentru XII B (2, 4, 2, -2, -5, -10, 1, 14, 2).

Este necesară utilizarea subprogramelor care comunică între ele și cu programul principal, precum și specificarea acestora.

Analiză

Identificarea subalgoritmilor



Specificarea subalgoritmilor

Subalgoritmul **citesteParam(p, q, s, f)**:

Descriere: Citește săptămâna de la care începe analiza, cea în care se încheie analiza, numărul natural *p* - parametru utilizat în problemă - și procentul *q*. Perioada analizată trebuie să includă cel puțin 2*p+1 săptămâni consecutive.
Date: -
Rezultate: *p, q, s, f*
p - număr natural utilizat în problemă, $p \in \{1, ..., 17\}$
q - procent utilizat în problemă, $q \in N^*$
s - săptămâna de la care începe analiza, $s \in \{1, ..., 36\}$
f - săptămâna în care se încheie analiza, $f \in \{1, ..., 36\}$
*f - s >= 2*p*

Subalgoritmul **citesteSir(T, saptInc, saptSf, n, tipDateT)**:
Descriere: Citește un șir de numere naturale, câte un număr pentru fiecare săptămână din perioada determinată de *saptInc* și *saptSf*; calculează lungimea șirului
Date: *saptInc* - prima săptămână pentru care se citește un număr natural
saptSf - ultima săptămână pentru care se citește un număr natural
tipDateT - semnificația unui număr natural din șir: număr total de absențe sau număr de absențe motivate
Rezultate: *T* - șir de numere naturale, câte unul pentru fiecare săptămână aflată în perioada determinată de *saptInc* și *saptSf*
$T = (t_i \in N | i=1..n, n \in N^*)$
n - lungimea șirului de numere citite

Subalgoritmul **calcAbsNemotiv(abs, absMotiv, lungAbs, absNemotiv)**:
Descriere: Date fiind un șir care reține numărul săptămânal de absențe pentru mai multe săptămâni consecutive și un șir cu numărul săptămânal de absențe motivate în aceeași perioadă, se determină un șir cu numărul de absențe nemotivate în fiecare săptămână din perioada considerată.
Date: *abs* - un șir de numere naturale cu numărul total de absențe pe săptămână
absMotiv - un șir de numere naturale cu numărul de absențe motivate pe săptămână
lungAbs - lungimea șirurilor *abs* și *absMotiv*
$abs = (a_i \in N | i=1..lungAbs, lungAbs \in N^*)$
$absMotiv = (am_i \in N | i=1..lungAbs, lungAbs \in N^*)$
Rezultate: *absNemotiv* - șir de numere naturale în care an_i reprezintă diferența între a_i și am_i - numărul de absențe nemotivate în săptămâna considerată, pentru *i* între 1 și *lungAbs*; are aceeași lungime cu cea a șirurilor de intrare, *lungAbs*
$absNemotiv = (an_i \in N | i=1..\ lungAbs, an_i = a_i - am_i)$

Subalgoritmul **detSecvMaxAltern (absNemotivA, absNemotivB, lungAbsNemotiv, p, lungSecvMax, poz)**:
Descriere: Calculează cea mai lungă secvență în care *p* creșteri consecutive ale numărului de absențe nemotivate sunt urmate de *p* scăderi consecutive ale numărului de absențe nemotivate și invers, în ambele clase analizate
Date: *absNemotivA* - un șir de numere naturale cu numărul de absențe nemotivate pe săptămână în prima clasă
absNemotivB - un șir de numere naturale cu numărul de absențe nemotivate pe săptămână în a doua clasă
$absNemotivA = (ana_i \in N | i=1..lungAbsNemotiv, lungAbsNemotiv \in N^*)$
$absNemotivB = (anb_i \in N | i=1..lungAbsNemotiv, lungAbsNemotiv \in N^*)$

lungAbsNemotiv - lungimea șirurilor cu numărul săptămânal de absențe nemotivate
lungAbsNemotiv ∈ N*
p - număr natural utilizat în problemă, *p* ∈ {1, ..., 17}
Rezultate: *poz* ∈ N* - pentru cea mai lungă secvență de *p* scăderi / creșteri alternative, *poz* reprezintă poziția din șir a primei săptămâni din secvență
lungSecvMax ∈ N - lungimea celei mai lungi secvențe în care *p* creșteri consecutive ale numărului de absențe nemotivate sunt urmate de *p* scăderi consecutive ale numărului de absențe nemotivate și invers, în ambele clase analizate; reprezintă numărul de săptămâni din secvență
lungSecvMax >= 2*p+1

Subalgoritmul **detSecvMaxAlternPoz (absNemotivA, absNemotivB, lungAbsNemotiv, p, poz, lungSecv)**:
Descriere: Calculează lungimea secvenței de câte *p* creșteri consecutive ale numărului de absențe nemotivate urmate de câte p scăderi consecutive ale numărului de absențe nemotivate și invers, relativ la o poziție dată.
Date: *absNemotivA* - un șir de numere naturale cu numărul de absențe nemotivate pe săptămână în prima clasă
absNemotivB - un șir de numere naturale cu numărul de absențe nemotivate pe săptămână în a doua clasă
lungAbsNemotiv - lungimea șirurilor cu numărul săptămânal de absențe nemotivate
lungAbsNemotiv ∈ N*
p - număr natural utilizat în problemă, *p* ∈ {1, ..., 17}
poz ∈ N - reprezintă poziția din șir începând de la care se verifică proprietatea cerută
Rezultate: *lungSecv* ∈ N - lungimea secvenței de absențe nemotivate care îndeplinesc proprietatea cerută relativ la poziția dată

Functia **detContInterval(T, pozSt, pozSf)**
Descriere: determină dacă șirul T are doar creșteri succesive sau doar descreșteri succesive între pozSt și pozSf+1
Date: *T* - șir de numere naturale
pozSt, pozSf ∈ N*
Rezultate: returnează *true* dacă T are doar creșteri succesive sau doar descreșteri succesive între pozSt și pozSf+1, *false* altfel

Subalgoritmul **afiseazaSecvMax(absNemotivA, absNemotivB, lungAbsNemotiv, inceputInterval, lungimeSecventa, saptInceput, p)**:
Descriere: Afișează perioada de timp solicitată în problemă
Date: *absNemotivA* - un șir de numere naturale cu numărul de absențe nemotivate pe săptămână în prima clasă
absNemotivB - un șir de numere naturale cu numărul de absențe nemotivate pe săptămână în a doua clasă
absNemotivA = (ana_i ∈ N|*i*=1..*lungAbsNemotiv*, *lungAbsNemotiv* ∈ N*)
absNemotivB = (anb_i ∈ N|*i*=1..*lungAbsNemotiv*, *lungAbsNemotiv* ∈ N*)
lungAbsNemotiv - lungimea șirurilor cu numărul săptămânal de absențe nemotivate
lungAbsNemotiv ∈ N*
inceputInterval ∈ N* - poziția începând de la care începe secvența identificată; *inceputInterval* ∈ {1, ...,*lungAbsNemotiv*}
lungimeSecventa ∈ N - lungimea secvenței care trebuie afișată
saptInceput ∈ N - săptămâna începând cu care se afișează diferențele între numerele de absențe săptămânale din săptămâni succesive, pentru ambele clase analizate

p - număr natural utilizat în problemă, *p* ∈ {1, ..., 17}
Rezultate: se afișează intervalul solicitat în problemă

Proiectare

```
subalgoritmul citesteParam(p, q, s, f) este:
	repeta
		@tipareste „ Introduceti valoarea parametrului p: ”
		@citeste p
	pana-cand p >= 1 AND p <= 17
	repeta
		@tipareste „ Introduceti valoarea procentului q: ”
		@citeste q
	pana-cand q >= 0
	repeta
		@tipareste „Introduceti sapt de start s si de final f a perioadei analizate
(f - s>= 2 *p): ”
		repeta
			@tipareste „ Introduceti s: ”
			@citeste s
		pana-cand s >= 1 AND s <= 36
		repeta
			@tipareste „ Introduceti f: ”
			@citeste f
		pana-cand f >= 1 AND f <= 36
	pana-cand f - s >= 2 * p
sf-subalgoritm

subalgoritmul citesteSir(T, saptInc, saptSf, n, tipDateT) este:
	n ← saptSf – saptInc + 1
	pentru i ← 1, n executa
		repeta
			@tipareste "Numarul de ", tipDateT, " in saptamana ”,saptInc+i-1, ": "
			@citeste T[i]
		pana-cand T[i] >= 0
	sf-pentru
sf-subalgoritm

subalgoritmul calcAbsNemotiv(abs, absMotiv, lungAbs, absNemotiv) este:
	pentru i ← 1, lungAbs executa
		absNemotiv[i] ← abs[i] - absMotiv[i]
	sf-pentru
sf-subalgoritm

functia detContInterval(T, pozSt, pozSf) este:
	i ← pozSt
	cont ← true

	cat-timp i < pozSf AND cont executa
		daca (T[i+1]-T[i])*(T[i+2]-T[i+1]) <= 0 atunci
				cont ← false
		altfel
			i ← i+1
		sf-daca
	sf-cat-timp
	detContInterval ← cont
sf-functie
```

```
subalgoritmul detSecvMaxAltern(absNemotivA, absNemotivB, lungAbsNemotiv, p, lungSecvMax,
poz) este:
       i ← 1
       cat-timp i <= lungAbsNemotiv - p executa
              lungSecv ← 0
              detSecvMaxAlternPoz(absNemotivA, absNemotivB, lungAbsNemotiv, p, i, lungSecv)
              daca lungSecv > lungSecvMax atunci
                     lungSecvMax ← lungSecv
                     poz ← i
              sf-daca
              daca lungSecv = 0 atunci
                     i ← i + 1
              altfel
                     i ← i + lungSecv - (p+1)
              sf-daca
       sf-cat-timp
sf-subalgoritm

subalgoritmul detSecvMaxAlternPoz(absNemotivA, absNemotivB, lungAbsNemotiv, p, poz,
lungSecv) este:
        lungSecv ← 0
        i ← poz
        contor ← 0
        cont ← true
       cat-timp i <= lungAbsNemotiv - p AND cont executa
              daca     detContInterval(absNemotivA,     i,     i     +     (p-1))     AND
       detContInterval(absNemotivB, i, i + (p- 1)) atunci
                     i ← i + p
                     contor ← contor + 1
              altfel
                     cont ← false
              sf-daca
              daca i+1 <= lungAbsNemotiv AND NOT (
                     (absNemotivA[i]-absNemotivA[i-1])*(absNemotivA[i+1]-absNemotivA[i])  <=
0 AND
                     (absNemotivB[i]-absNemotivB[i-1])*(absNemotivB[i+1]-absNemotivB[i])  <=
0
                                                          ) atunci
                     cont ← false
              sf-daca
       sf-cat-timp
       daca contor >= 2 atunci
              lungSecv ← i - poz + 1
       sf-daca
sf-subalgoritm

subalgoritmul afiseazaSecvMax(absNemotivA, absNemotivB, lungAbsNemotiv, inceputInterval,
lungimeSecventa, saptInceput, p) este:
       daca inceputInterval = 0 atunci
              @tipareste "Nu exista niciun interval de timp cu proprietatea dorita. "
       altfel
              saptIncInterval ← saptInceput + inceputInterval - 1
              saptSfInterval ← saptIncInterval + lungimeSecventa - 1
              @tipareste "Cea mai lunga perioada care indeplineste proprietatea ceruta
       este:
              ", saptIncInterval , " - ", saptSfInterval
              @tipareste "Nr de cresteri / scaderi in perioada respectiva este:
              ", (saptSfInterval - saptIncInterval)/p

              @tipareste "Diferente intre nr de absente nemotivate din 2 sapt succesive in
       perioada considerata, pentru ambele clase: "
              i ← inceputInterval + 1
              cat-timp i <= inceputInterval + lungSecv - 1 executa
```

```
                    @tipareste absNemotivA[i] - absNemotivA[i-1], " "
                    @tipareste absNemotivB[i] - absNemotivB[i-1], " "
                    i ← i + 1
             sf-cat-timp
      sf-daca
sf-subalgoritm

Algoritmul MonitorizareAbsente este:
      s ← 0
      f ← 0
           p ← 0
      q ← -1
      lungAbs ← 0
      lungAbsMotiv ← 0

      citesteParam(p, q, s, f)
      citesteSir(absA, s, f, lungAbs, "absente XII A")
      citesteSir(absMotivA, s, f, lungAbs, "absente motivate XII A ")
      citesteSir(absB, s, f, lungAbs, "absente XII B")
      citesteSir(absMotivB, s, f, lungAbs, "absente motivate XII B ")

      calcAbsNemotiv(absA, absMotivA, lungAbs, absNemotivA)
      calcAbsNemotiv(absB, absMotivB, lungAbs, absNemotivB)

      lungSecv ← 0
      poz ← 0
      detSecvMaxAltern(absNemotivA, absNemotivB, lungAbs, p, lungSecv, poz)
      afiseazaSecvMax(absNemotivA, absNemotivB, lungAbs, poz, lungSecv, s, p)
Sf-Algoritm
```

Exemple

Date de intrare	Rezultate
vezi enunt	

Cod sursă C++

```
#include "stdafx.h"
#include <iostream>
using namespace std;


#include "stdafx.h"
#include <iostream>
using namespace std;


void citesteParam(int& p, int& q, int& s, int& f)
{
      do
      {
            cout << "Introduceti valoarea parametrului p: ";
            cin >> p;
      } while (p < 1 || p > 17);
```

```
	do
	{
		cout << "Introduceti valoarea procentului q: ";
		cin >> q;
	} while (q < 0);

	do
	{
		cout << "Introduceti saptamana de start s si de final f a perioadei analizate
(f - s >= 2 * p): ";
		do
		{
			cout << "Introduceti s: ";
			cin >> s;
		} while (s < 1 || s > 36);
		do
		{
			cout << "Introduceti f: ";
			cin >> f;
		} while (f < 1 || f > 36);
	} while (f - s < 2 * p);
}


void citesteSir(int T[], int saptInc, int saptSf, int& n, const char tipDateT[50])
{
	n = saptSf - saptInc + 1; // se citeste un sir de numere naturale
	for (int i = 0; i < n; i++)
	{
		do
		{
			cout << "Numarul de "<<tipDateT<<" in saptamana "<<saptInc+i  << ": ";
			cin >> T[i];
		} while (T[i] < 0);
	}
}


void calcAbsNemotiv(int abs[], int absMotiv[], int lungAbs, int absNemotiv[])
{
	for (int i = 0; i < lungAbs; i++)
		absNemotiv[i] = abs[i] - absMotiv[i];
}


bool detContInterval(int T[], int pozSt, int pozSf)
{
	int i = pozSt;
	bool cont = true;

	while (i < pozSf && cont)
		if ((T[i + 1] - T[i])*(T[i + 2] - T[i + 1]) <= 0)
			cont = false;
		else i++;
	return cont;
}

void detSecvMaxAlternPoz(int absNemotivA[], int absNemotivB[], int lungAbsNemotiv, int p,
int poz, int& lungSecv)
{
	lungSecv = 0;
	int i = poz;
	int contor = 0;
```

```
	bool cont = true;
	while (i < lungAbsNemotiv - p && cont)
	{
		if (detContInterval(absNemotivA, i, i + (p - 1)) &&
detContInterval(absNemotivB, i, i + (p - 1)))
		{
			i += p;
			contor++;
		}
		else
			cont = false;

		if ((i+1 <= lungAbsNemotiv)&& !(((absNemotivA[i] - absNemotivA[i - 1]) *
(absNemotivA[i + 1] - absNemotivA[i]) < 0) &&
			((absNemotivB[i] - absNemotivB[i - 1]) * (absNemotivB[i + 1] -
absNemotivB[i]) < 0))
			)
			cont = false;
	}
	if (contor >= 2)
		lungSecv = i - poz + 1;
}


void detSecvMaxAltern(int absNemotivA[], int absNemotivB[], int lungAbsNemotiv, int p,
int& lungSecvMax, int& poz)
{
	int i = 0;
	while (i < (lungAbsNemotiv - p))
	{
		int lungSecv = 0;
		detSecvMaxAlternPoz(absNemotivA, absNemotivB, lungAbsNemotiv, p, i,
lungSecv);
		if (lungSecv > lungSecvMax)
		{
			lungSecvMax = lungSecv;
			poz = i;
		}
		if (lungSecv == 0)
			i++;
		else
			i += lungSecv - (p+1);
	}
}

void afiseazaSecvMax(int absNemotivA[], int absNemotivB[], int lungAbsNemotiv, int
inceputInterval, int lungimeSecventa, int saptInceput, int p)
{
	if (inceputInterval == -1)
	{
		cout << "Nu exista niciun interval de timp cu proprietatea dorita" << endl;
		return;
	}
	int saptIncInterval = saptInceput + inceputInterval;
	int saptSfInterval = saptIncInterval + lungimeSecventa - 1;

	cout << "Cea mai lunga perioada care indeplineste proprietatea ceruta este: " <<
saptIncInterval << " - " << saptSfInterval << endl;
	cout << "Nr de cresteri / scaderi in perioada respectiva este: " << ( saptSfInterval
- saptIncInterval ) / p << endl;

	int i;
```

```
	cout << "Diferente intre nr de absente nemotivate din 2 sapt succesive in perioada
considerata: " << endl;
	i = inceputInterval + 1;
	while (i <= inceputInterval + lungimeSecventa - 1)
	{
		cout << "abs nemotiv[" << i << "] - abs nemotiv[" << i - 1 << "]: " ;
		cout << "(XII A: )   ";
		cout << absNemotivA[i] - absNemotivA[i-1] << "  " <<endl;

		i++;
	}

	i = inceputInterval + 1;
	while (i <= inceputInterval + lungimeSecventa - 1)
	{
		cout << "abs nemotiv[" << i << "] - abs nemotiv[" << i - 1 << "]: ";
		cout << "(XII B: )   ";
		cout << absNemotivB[i] - absNemotivB[i - 1] << "  " << endl;

		i++;
	}
}


int main()
{
	int absA[36], absMotivA[36], absNemotivA[36], absB[36], absMotivB[36],
absNemotivB[36];
	int p, q, s, f, lungAbs, lungAbsMotiv, lungSecv, poz;
	char tipDate[20];

	s = 0, f = 0, p = 0, q = -1, lungAbs = 0, lungAbsMotiv = 0; //citire param, absente
	citesteParam(p, q, s, f);
	citesteSir(absA, s, f, lungAbs, "absente XII A");
	citesteSir(absMotivA, s, f, lungAbsMotiv, "absente motivate XII A ");
	citesteSir(absB, s, f, lungAbs, "absente XII B ");
	citesteSir(absMotivB, s, f, lungAbsMotiv, "absente motivate XII B ");


	//calcul sir absente nemotivate
	calcAbsNemotiv(absA, absMotivA, lungAbs, absNemotivA);
	calcAbsNemotiv(absB, absMotivB, lungAbs, absNemotivB);

	// identificam cea mai lunga secventa cu p scaderi urmate de p cresteri ale
numerelor de absente nemotivate, sau invers
	lungSecv = 0;
	poz = -1;
	detSecvMaxAltern(absNemotivA, absNemotivB, lungAbs, p, lungSecv, poz);

	afiseazaSecvMax(absNemotivA, absNemotivB, lungAbs, poz, lungSecv, s, p);

}
```

Cod sursă Pascal

```
program Problema3;

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

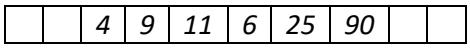
```
procedure citesteParam(var p: integer; var q: integer; var s:integer; var f:integer);
begin
    repeat
        writeln('Introduceti valoarea parametrului p: ');
        readln(p);
    until ((p >= 1) and (p <= 17));
    repeat
        writeln('Introduceti valoarea procentului q: ');
        readln(q);
    until (q >= 0);
    repeat
        writeln('Introduceti sapt de start s si de final f a perioadei analizate (f-s >=
2*p)');
        repeat
            writeln('Introduceti s: ');
            readln(s);
        until((s >= 1) and (s <= 36));
        repeat
            writeln('Introduceti f: ');
            readln(f);
        until((f >= 1) and (f <= 36));
    until ( f-s >= 2 * p);
end;

procedure citesteSir(var elemente: myArray; saptInc: integer; saptSf: integer; var
n:integer; tipDate: string);
var
    i:integer;
begin
    n := saptSf - saptInc + 1;
    for i:= 1 to n do
    begin
        repeat
            writeln('Numarul de ', tipDate, ' in saptamana ', (saptInc+i-1), ': ');
            readln(elemente[i]);
        until (elemente[i] >= 0);
    end;
end;

procedure calcAbsNemotiv(abs: myArray; absMotiv: myArray; lungAbs:integer; var absNemotiv:
myArray);
var
    i: integer;
begin
    for i:= 1 to lungAbs do
    begin
        absNemotiv[i] := abs[i] - absMotiv[i];
    end;
end;

function detContInterval(elemente: myArray; pozSt: integer; pozSf: integer): boolean;
var
    i:integer;
    cont: boolean;
begin
    i := pozSt;
    cont := true;
    while ((i < pozSf) and cont) do
    begin
        if ((elemente[i+1] - elemente[i]) * (elemente[i+2] - elemente[i+1]) <= 0) then
            cont := false
        else
```

```
            i := i + 1;
    end;
    detContInterval := cont;
end;

procedure detSecvMaxAlternPoz(absNemotivA: myArray; absNemotivB: myArray; lungAbs:
integer; p: integer; var poz: integer; var lungSecv: integer);
var
    i:integer;
    cont: boolean;
    contor :integer;
begin
    lungSecv := 0;
    i := poz;
    contor := 0;
    cont := true;

    while ((i <= lungAbs - p) and cont) do
    begin
        if (detContInterval(absNemotivA, i, i+p-1) and detcontInterval(absNemotivB, i,
i+p-1)) then
        begin
            i := i + p;
            contor := contor + 1;
        end
        else
            cont := false;
        if ((i + 1 <= lungAbs) and not (((absNemotivA[i] - absNemotivA[i-1]) *
(absNemotivA[i+1]- absNemotivA[i]) < 0)
                            and ((absNemotivB[i] - absNemotivB[i-1]) * (absNemotivB[i+1] -
absNemotivB[i]) < 0))) then
            cont := false;
    end;
    if contor >= 2 then
        lungSecv := i - poz + 1;
end;

procedure detSecvMaxAltern(absNemotivA: myArray; absNemotivB: myArray; lungAbs: integer;
p: integer; var lungSecvMax: integer; var poz: integer);
var
    i: integer;
    lungSecv: integer;
begin
    i := 1; {inceputul unei secvente}
    while (i <= lungAbs - p) do
    begin
        lungSecv := 0;
        detSecvMaxAlternPoz(absNemotivA, absNemotivB, lungAbs, p, i, lungSecv);
        if lungSecv > lungSecvMax then
        begin
            lungSecvMax := lungSecv;
            poz := i;
        end;
        if lungSecv = 0 then
            i := i + 1
        else
            i := i + lungSecv - p + 1;
    end;
end;

procedure afiseazaSecvMax(absNemotivA: myArray; absNemotivB: myArray; lungAbs: integer;
inceputInterval: integer; lungimeSecv: integer; saptInceput: integer; p:integer);
var
```

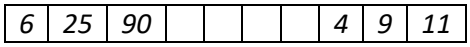
```
    saptIncInterval: integer;
    saptSfInterval: integer;
    i:integer;
begin
    if lungimeSecv = 0 then
        writeln('Nu exista secventa')
    else
    begin
        saptIncInterval := inceputInterval + saptInceput-1;
        saptSfInterval := saptIncInterval + lungimeSecv - 1;
        writeln('Cea mai lunga secventa care indeplineste proprietate: ', saptIncInterval,
'- ', saptSfInterval);
        writeln('Nr de cresteri/scaderi: ', (saptSfInterval-saptIncInterval) div p);
        writeln('Diferente: ');
        i := inceputInterval + 1;
        while (i <= inceputInterval + lungimeSecv -1) do
        begin
            writeln((absNemotivA[i] - absNemotivA[i-1]), ' ', (absNemotivB[i] -
absNemotivB[i-1]));
            i := i + 1;
        end;
    end;
end;

var
    p, q, s, f, n:integer;
    absA, absB, absMotivA, absMotivB, absNemotivA, absNemotivB: myarray;
    lungSecv, poz: integer;
begin
    citesteParam(p, q, s, f);
    citesteSir(absA, s, f, n, 'absente totale XII A');
    citesteSir(absMotivA, s, f, n, 'absente motivate XII A');
    citesteSir(absB, s, f, n, 'absente totale XII B');
    citesteSir(absMotivB, s, f, n, 'absente motivate XII B');
    calcAbsNemotiv(absA, absMotivA, n, absNemotivA);
    calcAbsNemotiv(absB, absMotivB, n, absNemotivB);

    lungSecv := 0;
    poz := 0;
    detSecvMaxAltern(absNemotivA, absNemotivB, n, p, lungSecv, poz);
    afiseazaSecvMax(absNemotivA, absNemotivB, n, poz, lungSecv, s, p);
end.
```

Întrebări grilă

1. Avem un tablou circular care are capacitatea *n* (adică maxim *n* elemente pot fi stocate în tablou), dar în care avem doar *k* elemente (1 <= k <= n). Aceste *k* elemente se află pe poziții consecutive, începând de la poziția *inceput*, și până la poziția *sfarsit* (ambele poziții sunt inclusive). Elementele pot fi aranjate în tablou în 2 moduri (în ambele exemple ordinea elementelor este 4,9,11,6,25,90):



n = 10
inceput = 3
sfarsit = 8



n = 10
inceput = 8
sfarsit = 3

Întrebare 1:

Cu care dintre următoarele expresii putem verifica dacă tabloul este plin?

a. inceput = sfarsit
b. inceput = sfarsit + 1
c. inceput = sfarsit + 1 SI (inceput = 1 SAU sfarsit = n)
d. inceput = sfarsit + 1 SAU (inceput = 1 SI sfarsit = n)
e. inceput = sfarsit – 1 SAU inceput = 1 SAU sfarsit = n
f. inceput = sfarsit SAU (inceput = 1 SI sfarsit = n)
g. inceput = sfarsit SAU (sfarsit = 1 SI inceput = n)

Întrebare 2:

Care dintre următorii subalgoritmi afișează toate elementele din tabloul circular în ordinea în care urmează (4, 9, 11, 6, 25, 90 pentru ambele exemple de mai sus). Pentru fiecare subalgoritm *tablou* este tabloul cu elemente, *n* este capacitatea (numărul maxim de elemente) tabloului*, inceput* este poziția din tablou unde încep elementele, *sfarsit* este poziția ultimului element. Presupunem că în tablou este cel puțin un element și cel mult *n* elemente.

a. **subalgoritm** print1(tablou, n, inceput, sfarsit):
    pentru i ← 1, n, 1 *executa*
        scrie tablou[i]
    sf_pentru
    sf_subalgoritm

b. **subalgoritm** print2(tablou, n, inceput, sfarsit)
```
        index ← -2
        repeata
                index ← index + 1
                scrie tablou[(index + inceput) % n + 1]
        pana_cand (index+inceput)% n + 1 = sfarsit
```
sf_subalgoritm

c. **subalgoritm** print3(tablou, n, inceput, sfarsit)
```
        index ← 1
        cattimp index < = n executa
                daca inceput <= index SI index <= sfarsit atunci
                        scrie tabou[index]
                        index ← index + 1
                altfel
                        index ← index + 1
                sf_daca
        sf_cattimp
```
sf_subalgoritm

d. **subalgoritm** print4(tablou, n, inceput, sfarsit)
```
        index ← inceput
        cattimp index != sfarsit executa
                scrie tablou[index]
                index ← index + 1
                daca index > n atunci
                        index ← 1
                sf_daca
        sf_cattimp
```
sf_subalgoritm

e. **subalgoritm** print5(tablou, n, inceput, sfarsit)
```
        index ← inceput – 1
        repeta
                index ← index + 1
                dacă index > n atunci
                        index ← 1
                sf_daca
                scrie tablou[index]
        pana_cand (index = sfarsit)
```
sf_subalgoritm

Să considerăm funcția următoare, care primește ca parametru un tablou numit *tablou*, și 2 valori intregi *a*, *b*

```
functie magie(t, a, b):
        daca a = b atunci:
                magie ← t[a]
        altfel
                daca t[a] <= t[a+1] atunci
                        magie ← magie(t, a+1, b)
                altfel
                        temp ← t[a]
                        t[a] ← t[a+1]
                        t[a+1] ← temp
                        magie ← magie(t, a+1, b)
                sf_daca
        sf_daca
sf_functie
```

Întrebare 1:
De câte ori se va executa linia *temp← t[a]* dacă facem apelul *magie([5, 1, 7, 9, 10, 17, 1, 6, 55], 1, 9)* ?

a. 1
b. 2
c. 3
d. 4
e. 5
f. 9

Întrebare 2:
Cât va fi rezultatul apelului *magie([6, 18, 3, 22, 1, 4, 19, 31, 27, 45], 1, 9)?*

a. 6
b. 45
c. 27
d. 1
e. 31
f. 18

Întrebare 3:
Avem tabloul t = [8, 1, 11, 32, 5, 7, 17, 3, 9]. Apelăm funcția *magie* cu t, a = 2, b = 9. Cum va arăta tabloul după apel?

a. [8, 1, 11, 32, 5 ,7, 17, 3, 9]
b. [1, 3, 5, 7, 8, 9, 11, 17, 32]
c. [8, 1, 3, 5, 7, 9, 11, 17, 32]
d. [8, 1, 11, 5, 7, 17, 3, 9, 32]
e. [1, 8, 11, 5, 7, 17, 3, 9, 32]
f. [8, 1, 11, 5, 32, 7, 17, 3, 9]

3. Alegeți varianta care corespunde afișării tuturor numerelor impare între 1 și 100 o singură dată:

```
a.
pentru i ← 1, 100 executa
       daca i%2 != 0 atunci
              @tipareste i
              i ← i + 2
       sf-daca
sf-pentru


b.
pentru i ← 1, 100 executa
       daca i%2 != 0 atunci
              @tipareste i
       altfel
              @tipareste i+1
       sf-daca
sf-pentru

c.
pentru i ← 1, 100 executa
       daca i%2 != 0 atunci
              @tipareste i
       altfel
              i ← i + 1
       sf-daca
sf-pentru

d.
pentru i ← 1, 100 executa
       daca i%2 != 0  atunci
              @tipareste i
       sf-daca
       i ← i + 1
sf-pentru
```

e. niciuna dintre variantele de mai sus nu afișează toate numerele impare între 1 și 100 o singură dată.

4. Ce se afișează la apelul afisare(20) pentru subalgoritmul *afisare* (scris în C++) de mai jos?

```
void afisare(int A)
{
	if (A>10)
	{
		afisare(A / 10);
		cout << A++;
		cout << --A;
	}
}
```

a. 2020
b. 2021
c. 2120
d.1920
e. nicio variantă nu este corectă