Determinarea tripletului cu produsul maxim într-un tablou

Se dă un tablou cu elemente întregi, atât pozitive cât și negative. Scrieți un algoritm care determină tripletul cu produsul maxim din acest tablou.

Cea mai simpla implementare ar fi să generăm toate tripletele posibile și să selectăm tripletul cu produsul maxim. Această implementare naivă are o complexitate $O(n^3)$, avem 3 bucle *for* imbricate.

Implementare

```
// se da un tablou cu numere intregi pozitive si negative
// scrieti un program care determina tripletul cu produsul maxim din tablou

// vom incepe cu o implementare brute-force in care vom genera toate tripletele
posibile si vom selecta tripletul cu produsul maxim
// aceasta metoda are complexitatea de O(n^3)
// arr - tabloul in care cautam tripletul cu produsul maxim
// sz - dimensiunea acestui tablou
void gaseste_triplet_v1(int a[], int sz) {
	int maxim = std::numeric_limits<int>::min();

	if (sz < 3) {
		cout << "Tabloul are mai putin de 3 elemente" << endl;
		return;
	}
	int max_el_1 = 0, max_el_2 = 0, max_el_3 = 0;

	// generam toate tripletele
	for (int i = 0; i < sz - 2; i++) {
		for (int j = i + 1; j < sz - 1; j++) {
			for (int k = j + 1; k < sz; k++) {
				int v = a[i] * a[j] * a[k];
				// si determinam tripletul cu produsul maxim
				if (v > maxim) {
					maxim = v;
					max_el_1 = a[i];
					max_el_2 = a[j];
					max_el_3 = a[k];
				}
			}
		}
	}
cout << "Tripletul cu produsul maxim este: (" << max_el_1 << ", " << max_el_2 << ", "
<< max_el_3 << ")" << endl;
```

```
}
```

Observăm că tripletul cu produs maxim este format:

- fie din primele 3 cele mai mari elemente
- fie din primele 2 cele mai mici elemente și elementul maxim din tablou.

O altă implementare ar fi să sortăm tabloul și să determinăm tripletul cu produs maxim din aceste două triplete candidat.

Implementare

```
// tripletul cu produsul maxim este format fie din primele cele mai trei ele mai mari
elemente, sau primele doua cele mai mici elemente
// si maximul din tablou. vom sorta tabloul si vom selecta aceste elemente
//
// aceasta metoda are complexitatea de O(n log n)
// arr - tabloul in care cautam tripletul cu produsul maxim
// sz - dimensiunea acestui tablou
void gaseste_triplet_v2(int a[], int sz) {
	if (sz < 3) {
		cout << "Tabloul are mai putin de 3 elemente" << endl;
		return;
	}

	// sortam tabloul
	std::sort(a, a + sz);

	// si returnam maximum dintre cele 2 candidate la tripletul cu produs maxim
	int candidate1 = a[0] * a[1] * a[sz - 1];
	int candidate2 = a[sz - 1] * a[sz - 2] * a[sz - 3];

	if (candidate1 > candidate2) {
		cout << "Tripletul cu produsul maxim este: (" << a[0] << ", " << a[1] <<
", " << a[sz - 1] << ")" << endl;
	}
	else {
		cout << "Tripletul cu produsul maxim este: (" << a[sz - 1] << ", " <<
a[sz - 2] << ", " << a[sz - 3] << ")" << endl;
	}
}
```

Cel mai costisitor pas din această implementare este sortare ($O(n \log n)$).

Dar, există o implementate și mai eficientă în care determinăm primele 3 cele mai mari elemente, cât și cele mai mici două elemente din tablou.

Implementare

```
// tripletul cu produsul maxim este format fie din primele cele mai trei cele mai mari
// elemente, sau primele doua cele mai mici elemente
// si maximul din tablou. parcurgem o singura data tabloul si determinam aceste 5
// elemente
//
// aceasta metoda are complexitatea de O(n)
// arr - tabloul in care cautam tripletul cu produsul maxim
// sz - dimensiunea acestui tablou

void gaseste_triplet_v3(int arr[], int sz) {
	if (sz < 3) {
		cout << "Tabloul are mai putin de 3 elemente" << endl;
		return;
	}

    // calculam primele 3 cele mai mari elemente din tablou
    int maxA = std::numeric_limits<int>::min(), maxB = std::numeric_limits<int>::min(),
maxC = std::numeric_limits<int>::min();

    // si primele 2 cele mai mici elemente din tablou
    int minA = std::numeric_limits<int>::max(), minB = std::numeric_limits<int>::max();

    for (int i = 0; i < sz; i++)
    {
        if (arr[i] > maxA){
            maxC = maxB;
            maxB = maxA;
            maxA = arr[i];
        }
        else if (arr[i] > maxB){
            maxC = maxB;
            maxB = arr[i];
        }
        else if (arr[i] > maxC)
            maxC = arr[i];


        if (arr[i] < minA){
            minB = minA;
            minA = arr[i];
        }else if (arr[i] < minB)
            minB = arr[i];
    }

       int candidate1 = maxA * maxB * maxC;
       int candidate2 = minA * minB * maxA;

       if (candidate1 > candidate2) {
		cout << "Tripletul cu produsul maxim este: (" << maxA << ", " << maxB <<
", " << maxC << ")" << endl;
	}
	else {
		cout << "Tripletul cu produsul maxim este: (" << maxA << ", " << minA <<
", " << minB << ")" << endl;
```

```
        }
}
```

n $f(n)$	$\lg n$	n	$n \lg n$	n^2	2^n	$n!$
10	0.003 μs	0.01 μs	0.033 μs	0.1 μs	1 μs	3.63 ms
20	0.004 μs	0.02 μs	0.086 μs	0.4 μs	1 ms	77.1 years
30	0.005 μs	0.03 μs	0.147 μs	0.9 μs	1 sec	8.4×10^{15} yrs
40	0.005 μs	0.04 μs	0.213 μs	1.6 μs	18.3 min	
50	0.006 μs	0.05 μs	0.282 μs	2.5 μs	13 days	
100	0.007 μs	0.1 μs	0.644 μs	10 μs	4×10^{13} yrs	
1,000	0.010 μs	1.00 μs	9.966 μs	1 ms		
10,000	0.013 μs	10 μs	130 μs	100 ms		
100,000	0.017 μs	0.10 ms	1.67 ms	10 sec		
1,000,000	0.020 μs	1 ms	19.93 ms	16.7 min		
10,000,000	0.023 μs	0.01 sec	0.23 sec	1.16 days		
100,000,000	0.027 μs	0.10 sec	2.66 sec	115.7 days		
1,000,000,000	0.030 μs	1 sec	29.90 sec	31.7 years		

Figure 2.4: Growth rates of common functions measured in nanoseconds

Găsirea elementului care nu este duplicat într-un tablou

Se dă un tablou unidimensional de numere întregi pozitive în care toate elementele, mai puțin unul, apar de două ori. Scrieți un algoritm care determină numărul care apare o singură dată în tablou.
Altfel spus, tabloul de intrare are $2n + 1$ elemente, din care n elemente apar de două ori, iar noi trebuie să găsim elementul care apare o singură dată.

Implementarea 1

Cea mai simplă implementare ar fi să parcurgem tabloul cu două bucle for imbricate pentru a genera toate perechile posibile din tablou, și să determinăm elementul care "nu are pereche".
Această metodă este foarte simplu de implementat, dar are complexitatea de $O(n^2)$.

Implementare

```
// se da un tablou unidimensional de dimensiune impara (2k + 1), in care toate
// elementele, mai putin unul, sunt duplicate
// aceasta functie determina elementul care nu se repeta in acest tablou
// metoda brute-force in care generam toate perechile posibile pentru fiecare numar, si
// apoi selectam elementul care nu apare de 2 ori
int gaseste_element_unic_v1(int arr[], int n) {

    for (int i = 0; i < n; i++) {
        bool found = false;
        for (int j = 0; j < n && !found; j++)
            if (i != j && arr[i] == arr[j]) {
                found = true;
            }
        if (!found)
            return arr[i];
    }

    return -1;
}
```

Implementarea 2

O metodă puțin mai eficientă ar fi să sortăm tabloul (după cum am menționat anterior, dacă folosim o metodă de sortare eficientă, acest pas are o complexitate de $O(n \log n)$. După ce am sortat tabloul, îl putem parcurge din 2 în 2 și să căutăm primul element care nu este urmat de un element egal.

Implementare

```
// se da un tablou unidimensional de dimensiune impara (2k + 1), in care toate
elementele, mai putin unul, sunt duplicate
// aceasta functie determina elementul care nu se repeta in acest tablou
// metoda bazata pe sortarea tabloului
int gaseste_element_unic_v2(int arr[], int n) {

    // sortam tabloul
    std::sort(arr, arr + n);

    // apoi cautam elementul care nu se repeta
    for (int i = 0; i < n - 1; i+=2) {
        if (arr[i] != arr[i + 1]) {
            return arr[i];
        }
    }
    return arr[n-1];
}
```

Implementarea 3

A treia implementare este cea mai eficientă și implică folosirea operației SAU EXCLUSIV. Complexitatea metodei este $O(n)$.

Operația SAU-EXCLUSIV se aplică la nivel de bit. Din tabelul de adevăr al funcției XOR observăm cu ușurință ca operația A xor A= 0 și A xor 0 = A. Astfel dacă aplicăm operația xor între toate elementele tabloului rezultatul va fi elementul care nu se repetă.

A	**B**	**A xor B**
0	0	0
0	1	1
1	0	1
1	1	0

Implementare

```
// se da un tablou unidimensional de dimensiune impara (2k + 1), in care toate
// elementele, mai putin unul, sunt duplicate
```

```
// aceasta functie determina elementul care nu se repeta in acest tablou
// metoda crae foloseste operatia sau exclusiv
int gaseste_element_unic_v3(int arr[], int n) {

    int non_repeating = 0;
    // ^ - operatia de sau exclusiv
    for (int i = 0; i < n; i++) {
        non_repeating = non_repeating ^ arr[i];
    }

    return non_repeating;
}
```

Găsirea elementului majoritar într-un tablou sortat

Un element se numește majoritar dacă apare de mai mult de [n/2] ori într-un tablou, unde n este dimensiunea tabloului.
Fiind dat un număr v și tablou unidimensional arr sortat în ordine crescătoare, scrieți un algoritm care determină dacă elementul v este majoritar în acest tablou.

Implementarea 1

Cea mai simplă implementare ar fi să parcurgem tabloul pentru a găsi prima apariție a elementului v. Să numim această poziție f. Apoi, știind că tabloul este sortat, verificăm și dacă elementul de pe f + n/2 este egal cu v.

Implementare

```
// Un element se numeste majoritar daca apare de mai mult de [n/2] intr-un tablou, unde n
// este dimensiunea tabloului.
// Fiind dat un numar v si tablou unidimensional arr sortat in ordine crescatoare, acesta
// functie determina daca v este element majoritar in arr
// aceasta metoda foloseste cautarea secventiala
bool este_majoritar_v1(int arr[], int n, int v)
{
	cout << n << endl;
	// cautam prima aparitie a lui v in tablou
	for (int i = 0; i < n; i++)
	{
		// daca elementul este curent este egal cu v si elementul dupa n/2 pozitii
este tot v
		// atunci v este element majoritar
		if (arr[i] == v && (i + n / 2) < n && arr[i + n / 2] == v)
			return 1;
	}
	return 0;
}
```

O versiune mai eficientă ar fi să folosim căutarea binară pentru a determina prima apariție a elementului în tablou.

Implementare

```
// cauta in tabloul sortat arr de dimensiune v, prima apartitie a valorii v si returneaza
// aceasta pozitie
// daca valoarea v in apare in tabloul sortat arr, atunci functia returneaza valoarea -1
int cautare_binara(int arr[], int sz, int v) {
	int start = 0;
	int fin = sz - 1;
	while (start < fin) {
		int mid = start + (fin - start) / 2;
		if (arr[mid] >= v) {
```

```
                        fin = mid;
                }
                else {
                        start = mid + 1;
                }
        }

        return (arr[start] == v) ? start : -1;
}



// Un element se numeste majoritar daca apare de mai mult de [n/2] intr-un tablou, unde n
// este dimensiunea tabloului.
// Fiind dat un numar v si tablou unidimensional arr sortat in ordine crescătoare, acesta
// functie determina daca v este element majoritar in arr
// aceasta metoda foloseste cautarea binara
bool este_majoritar_v2(int arr[], int n, int v)
{
        // gasim prima apartie a lui v in tablou
        int i = cautare_binara(arr, n, v);

        // daca elementul nu apare in tablou, returnam false
        if (i == -1)
                return false;

        // altfel verificam ca elementul sa apara de cel putin n/2 ori
        if (((i + n / 2) < n) && arr[i + n / 2] == v)
                return true;

        return false;

}
```

Sortarea unui vector ce conține doar 3 valori (0, 1 și 2)

Se dă un tablou unidimensional care conține doar valorile 0, 1 și 2. Scrieți un algoritm care separă elementele din tablou în trei regiuni distincte. Această problemă se mai numește și *Dutch national flag* problem.
De exemplu, daca avem tabloul { 0, 1, 2, 2, 2, 1, 1, 0, 0, 1 } algoritmul nostru ar trebui să rearanjeze elementele din tablou astfel încât să obținem rezultatul {0, 0, 0, 1, 1, 1, 1, 2, 2, 2}.

Cea mai simplă implementare ar fi să sortăm tabloul. Această implementare are o complexitate de $O(n \log n)$, dacă folosim o metodă de sortare cum ar fi merge-sort sau heap-sort.

Implementare

```
// se da un tablou unidimensional care contine doar 3 valori (0, 1, si 2). Scrieți un
// algoritm care sorteaza acest tablou
// metoda care foloseste sortare
void sortare_culori_v1(int a[], int sz) {
	// sortam tabloul
	std::sort(a, a + sz);
}
```

Să vedem acum o implementare mai eficientă în timp liniar O(N).
Împărțim tabloul în 4 secțiuni determinate de pozițiile *low, mid* și *high:*

- Elementele de la [0, *low-1*] vor conține valoarea 0 (roșu)
- Elementele de la [*low, mid - 1*] vor conține valoarea 1 (alb)
- Elementele de la [*mid, high*] vor fi neprocesate (gri)
- Elementele de la [*high+1, sz*), unde *sz* este dimensiunea tabloului vor fi egale cu 2 (albastru).

0	0	0	1	1	?	?	?	2	2
			Low		Mid		High		

La fiecare iterație a algoritmului vom micșora porțiunea cu elementele neprocesate, menținând întotdeauna condițiile de mai sus. Ne uităm la primul element neprocesat *arr[mid]* și, în funcție de culoarea acestuia, efectuăm următoarele operații:

- Daca este 0 (roșu), interschimbăm arr[*low*] și arr[*mid*], apoi incrementăm atât *low*, cât și *mid*.
- Daca este 1 (alb), incrementăm *mid*.
- Dacă este 2 (albastru), interschimbăm arr[*mid*] și arr[*high*] și decrementăm *high*

0	1	2	2	2	1	1	0	0	1

La început *low* = *mid* = *0* și *high* = *sz -1*. Cu alte cuvinte toate elementele vor fi neprocesate.

0	1	2	2	2	1	1	0	0	1
Low, Mid									High

0	1	2	2	2	1	1	0	0	1
	Low Mid								High

0	1	2	2	2	1	1	0	0	1
	Low	Mid							High

0	1	1	2	2	1	1	0	0	2
	Low	Mid						High	

0	1	1	2	2	1	1	0	0	2
	Low		Mid					High	

0	1	1	0	2	1	1	0	2	2
	Low		Mid				High		

0	0	1	1	2	1	1	0	2	2
		Low		Mid			High		

0	0	1	1	0	1	1	2	2	2
		Low		Mid		High			

0	0	0	1	1	1	1	2	2	2
			Low		Mid	High			

0	0	0	1	1	1	1	2	2	2

			Low			High Mid			

0	0	0	1	1	1	1	2	2	2
			Low			High Mid			

Implementare

```
// se da un tablou unidimensional care contine doar 3 valori (0, 1, si 2). scrieti un
algoritm care sorteaza acest tablou
void sortare_culori_v2(int a[], int sz) {
    int lo = 0;
    int hi = sz - 1;
    int mid = 0;

    while (mid <= hi) {
        if (a[mid] == 0) {
            swap(a[lo++], a[mid++]);
        }
        else {
            if (a[mid] == 1) {
                mid++;
            }
            else {
                swap(a[mid], a[hi--]);
            }
        }
    }
}
```