****

**MINISTERUL EDUCAȚIEI, CULTURII ȘI CERCETĂRII**

**AL REPUBLICII MOLDOVA**

**Universitatea Tehnică a Moldovei**

**Facultatea Calculatoare, Informatică şi Microelectronică**

**Raport**

**pentru lucrarea de laborator Nr.2**

***la cursul de “Analiza și proectarea algoritmilor ”***

***Tema:Algoritmi de Sortare.***

Efectuat: Studentul gr. TI-207 **Bunescu Gabriel**

Verificat:  *asist. Univ* **Bîtca Ernest**

**Chișinău – 20****21**

Cuprins:

[1. Quick sort 3](#_Toc85900854)

[2. Merge sort 7](#_Toc85900855)

[3. Bubble sort 12](#_Toc85900856)

[4. Reprezentarea rezultatelor obținute și compararea lor cu ajutorul graficelor: 15](#_Toc85900857)

[5. Concluzie: 16](#_Toc85900858)

***Scopul lucrării:***

1. De aplicat in cod algoritmele de sortare : Mergesort , Quicksort si Bubblesort.

2. De aflat numarul de iteratii la fiecare sortare.

3. De introdus timpul de executare in cod.

4. De construit tabel dupa timpul de executare si numarul de iteratii.

### Quick sort

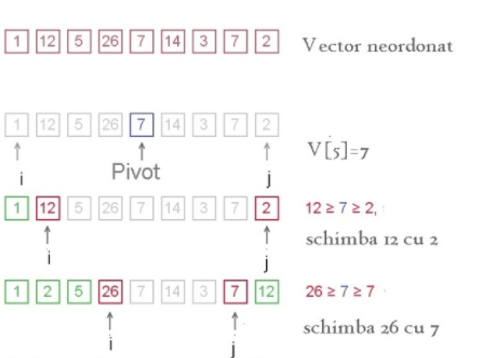
* timp mediu: O(N log N)
* timp la limită: O(N^2)
* memorie: O(log N)
* Stabil: NU

**Descriere :**

Quick Sort este unul dintre cei mai rapizi şi mai utilizaţi algoritmi de sortare până în acest moment,bazându-se pe tehnica „**Divide et impera**“.Deşi cazul cel mai nefavorabil este O(N^2), în practică, QuickSort oferă rezultate mai bune decât restul algoritmilor de sortare din clasa „O(N log N)“.

**Algoritmul se bazează pe următorii paşi:**

* alegerea unui element pe post de **pivot**
* parcurgerea vectorului din două părţi(de la stânga la pivot, de la dreapta la pivot, ambele în acelaşi timp)
* interschimbarea elementelor care se află pe „**partea greşită**“ a pivotului(mutăm la dreapta pivotului elementele mai mari, la stânga pivotului elementel mai mici)
* divizarea algoritmului: după ce mutăm elementele pe „**partea corectă**“ a pivotului, avem **2 subşiruri de sortat**, iar pivotul se află pe poziţia bună.



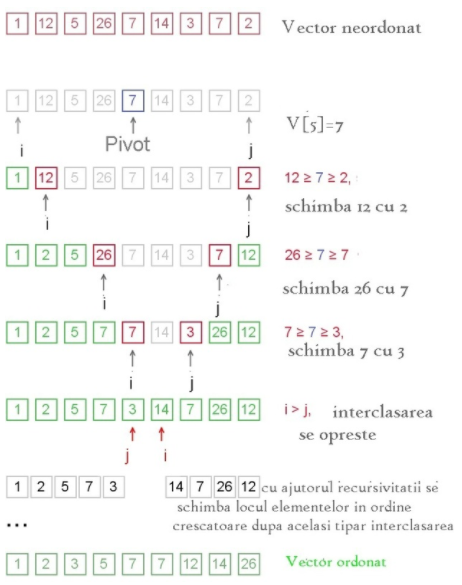


Fig.1.1.Quick sort

* 1. Codul sursă Quicksort:

#include<conio.h>

#include<iostream>

#include<time.h>

#include <vector>

using namespace std;

int it;

const int n = 1000;

int tab[n];

// Quicksort .

int partition(int InitialTablou[], int top, int bottom)

{

int x = InitialTablou[top];

int i = top - 1;

int j = bottom + 1;

int temp;

do

{

do

{

j--;

} while (x < InitialTablou[j]);

do

{

i++;

} while (x > InitialTablou[i]);

if (i < j)

{

temp = InitialTablou[i];

InitialTablou[i] =InitialTablou[j];

InitialTablou[j] = temp;

}

} while (i < j); it = i + j; return j;

}

void quicksort(int InitialTablou[], int top, int bottom)

{

int middle;

if (top < bottom)

{

middle = partition(InitialTablou, top,bottom);

quicksort(InitialTablou, top, middle); quicksort(InitialTablou, middle + 1,bottom);

}

}

void AfisareTablou(int InitialTablou[], int size)

{

int i;

for (i = 0; i < size; i++)

{

cout << InitialTablou[i] << ' ';

}

cout << "\n\n";

}

int main()

{

int initialTablou[1000];

int n;

cout << "Dati n:",

cin>>n;

cout << "\n";

rand();

int marime = n;

for (int i = 0;i < marime; i++)

tab[i] = rand() % 1000 - 100;

cout << "Tabloul nesortat:"<< endl;

AfisareTablou(tab, marime);

clock\_t start, end;

start = clock();

quicksort(tab, 0,marime-1);

cout << "Tabloul sortat prinmetoda quiksort este:" << endl;

AfisareTablou(tab, marime );

printf("\nTimpul de executieal algoritmului Quicksort este:: %.8f ", ((double) (clock() - start)) / CLOCKS\_PER\_SEC);

end = clock();

cout<<"\nNumarul de iteratii :"<<it<< endl;

getch();

return 0;

}

**Rezultatele obținute:**

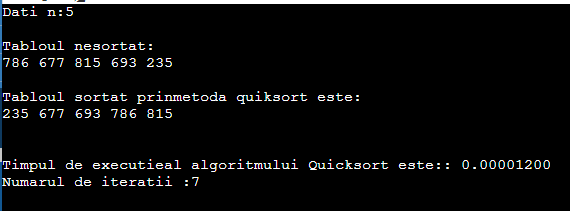


Fig.1.2.tablou din 5 numere

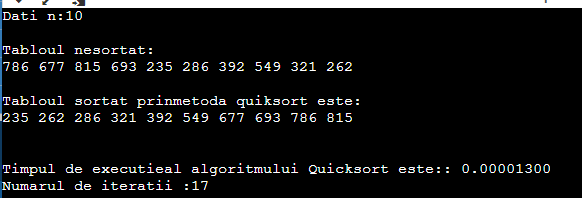


Fig.1.3.tablou din 10 numere

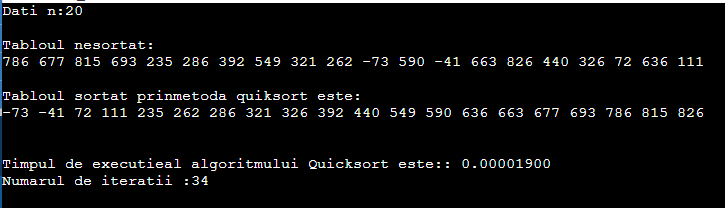


Fig.1.4.tablou din 20 numere

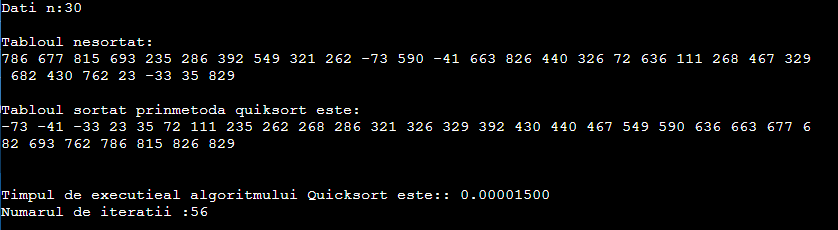


Fig.1.5.tablou din 30 numere

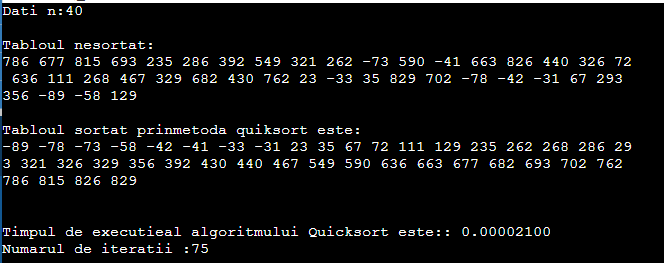


Fig.1.6.tablou din 40 numere

### Merge sort

* timp mediu: O(N log N)
* timp la limită: O(N log N)
* memorie: O(N)
* Stabil: DA

#### Descriere :

În cazul sortării prin interclasare, vectorii care se interclasează sunt două secvenţe ordonate din acelaşi vector. Sortarea prin interclasare utilizează metoda **Divide et Impera**:

* Se împarte vectorul în secvenţe din ce în ce mai mici, astfel încât fiecare secvenţă să fie

ordonată la un moment dat şi interclasată cu o altă secvenţă din vector corespunzătoare.

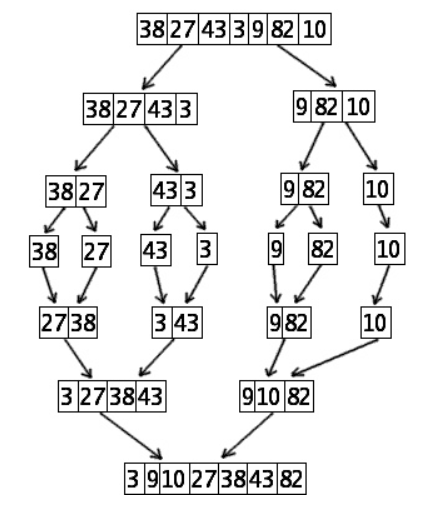
* ****Practic, interclasarea va începe când se ajunge la o secvenţă formată din două elemente. Aceasta, odată ordonată, se va interclasa cu o alta corespunzătoare(cu 2 elemente). Cele două secvenţe vor alcătui un subşir ordonat din vector mai mare(cu 4 elemente) care, la rândul lui, se va interclasa cu un subşir corespunzător(cu 4 elemente) ş.a.m.d.

Fig.2.1. Merge sort

#### Codul sursă Merge sort:

#include<conio.h>

#include<iostream>

#include<time.h>

using namespace std;

const int n = 100;

int tab[n];

int it;

//MergeSort

void MS( int aux1,int aux2,int aux3, int aux4)

{

int i, j;

for (j = aux1;j <= aux2;j++)

for (i = aux3;i <= aux4;i++)

if (tab[i]<tab[j])

{

tab[i] += tab[j];

tab[j] = tab[i] -tab[j];

tab[i] -= tab[j];

it = i + j;

}

}

void AF(int tab[], int size)

{

int i;

for (i = 0; i < size; i++)

{

cout << tab[i] << ' ';

}

cout << "\n\n";

}

void insert(int i, int j)

{

if (tab[i]>tab[j])

{

tab[i] += tab[j];

tab[j] = tab[i] - tab[j];

tab[i] -= tab[j];

}

}

int MergeSort(int i, int j)

{

if (j - i <= 1)insert(i, j);

else

{

MergeSort(i, (i + j) / 2);

MergeSort(1 + (i + j) / 2, j);

MS(i, (i + j) / 2, 1 + (i + j) / 2, j);

}

return 0;

}

int main()

{

int initialTablou[10000];

int n;

cout << "Dati n:",

cin>>n;

cout << "\n";

rand();

int marime = n;

for (int i = 0;i < marime; i++)

tab[i] = rand() % 1000 - 100;

cout << "Tabloul nesortat:" << endl;

AF(tab, marime);

{

clock\_t start, end;\

start = clock();

for (int i = 0; i < n; i++)

MergeSort(0, n - 1);

start = clock();

cout << "Tabloul sortat prinmetoda MergeSort este:" << endl;

MergeSort(0, n - 1);

AF(tab, marime);

printf("Timpul de executie alalgoritmului MergeSort este: %.8f\n", ((double)(clock() - start)) / CLOCKS\_PER\_SEC);

cout << "Numarul de iteratii: "<<it << endl;

}

getch();

return 0;

}

**Rezultatele obținute:**

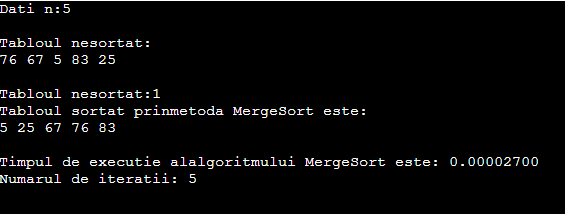


Fig.2.2.tablou din 5 numere

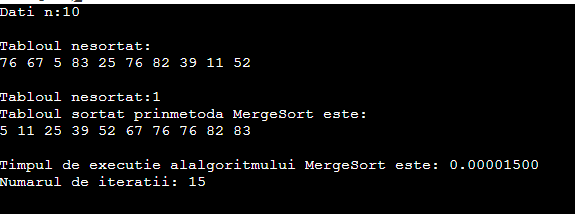


Fig.2.3.tablou din 10 numere

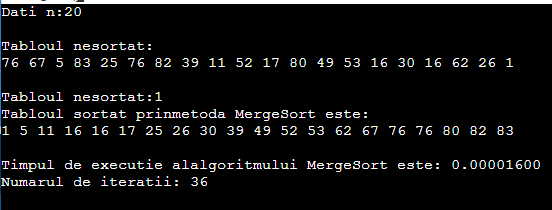


Fig.2.4.tablou din 20 numere

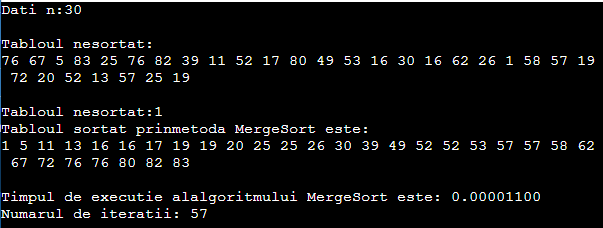


Fig.2.5.tablou din 30 numere

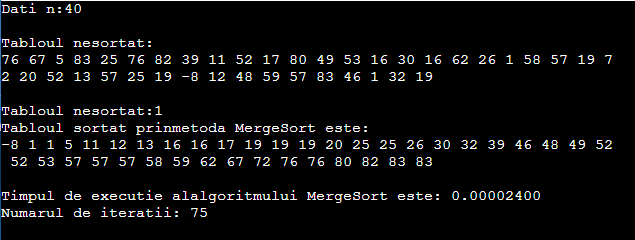


Fig.2.6.tablou din 40 numere

### Bubble sort

* timp mediu: O(N^2)
* timp la limită: O(N^2)
* memorie: O(1)
* Stabil: DA

#### Descriere :

Sortarea prin metoda bulelor se consideră drept una din cele mai puţin efective metode de sortare, dar cu un algoritm mai simplu.

* Ideea de bază a sortării prin metoda bulelor este în a parcurge tabloul, de la stânga spre dreapta,

fiind comparate elementele alăturate **a[i] si a[i+1]**. Dacă vor fi găsite 2 elemente neordonate, valorile lor vor fi interschimbate.

* Parcurgerea tabloului de la stânga spre dreapta se va repeta atât timp cât vor fi întâlnite

elemente neordonate.

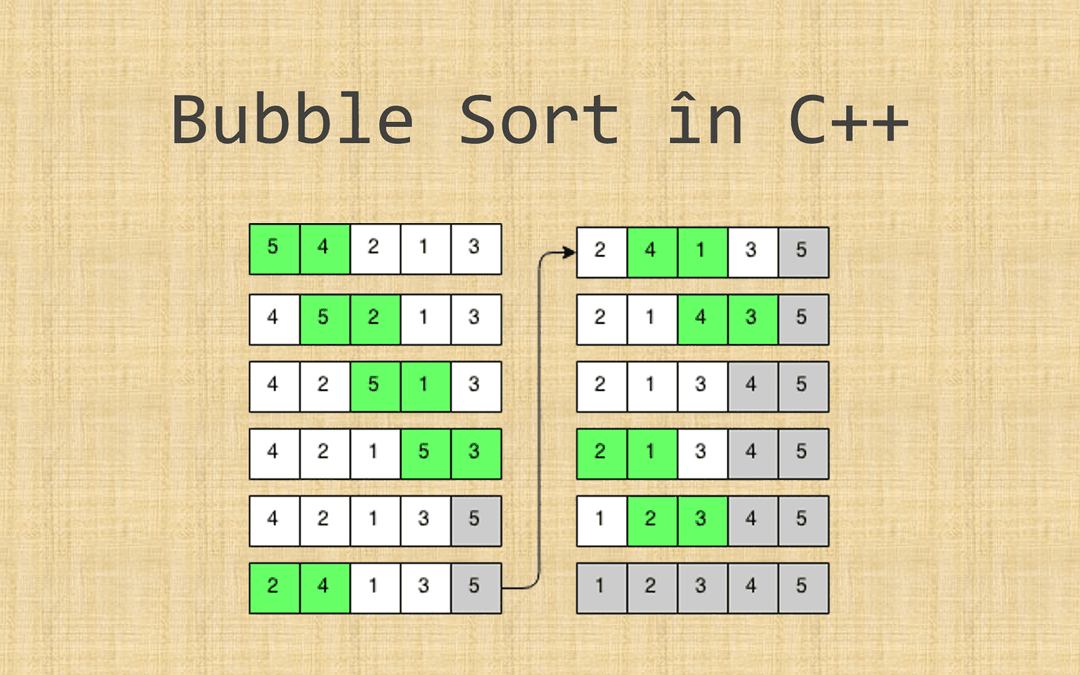


Fig.3.1. Bubble sort

#### Codul sursă Bubble sort:

#include<conio.h>

#include<iostream>

#include<time.h>

#include <vector>

using namespace std;

int it;

const int n = 100;

int tab[n];

void bubble\_sort(int iarr[], int num) { int i, j=0, k, temp;

bool swapped = true;

while (swapped) {

swapped = false;

j++;

for (i = 0; i < num-j; i++) {

if (iarr[i] > iarr[i + 1]) {

temp = iarr[i];

iarr[i] = iarr[i + 1];

iarr[i + 1] = temp;

swapped = true;

}

}

}

it = ((i)\*(j));

}

void AfisareTablou(int iarr[], int size)

{

int i;

for (i = 0; i < size; i++)

{

cout << iarr[i] << ' ';

}

cout << "\n\n";

}

int main()

{

int initialTablou[1000];

int n;

cout << "Dati n:",

cin>>n;

cout << "\n";

rand();

int marime = n;

for (int i = 0;i < marime; i++)

tab[i] = rand() % 100 - 10;

cout << "Tabloul nesortat:" << endl;

AfisareTablou(tab, marime);

clock\_t start, end;

start = clock();

bubble\_sort(tab , marime );

cout << "Tabloul sortat prin metoda bubble sort este:" << endl;

AfisareTablou(tab, marime);

printf("\nTimpul de executie al algoritmului bubble sort este:: %.8f ", ((double)(clock() - start)) / CLOCKS\_PER\_SEC);

end = clock();

cout << "\nNumarul de iteratii : " << it <<endl;

return 0;

getch();

}

**Rezultatele obținute:**

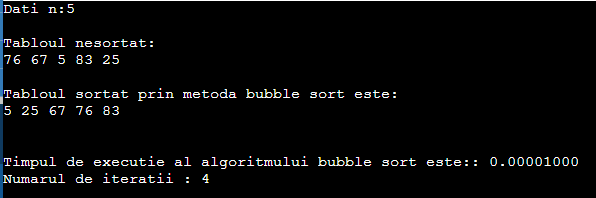


Fig.3.2.tablou din 5 numere

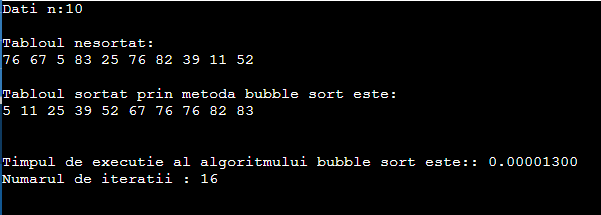


Fig.3.3.tablou din 10 numere

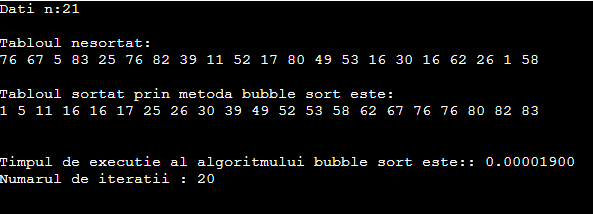


Fig.3.4.tablou din 21 numere

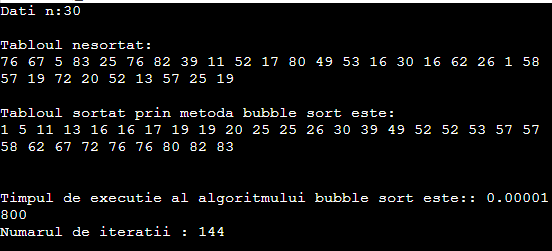


Fig.3.5.tablou din 30 numere

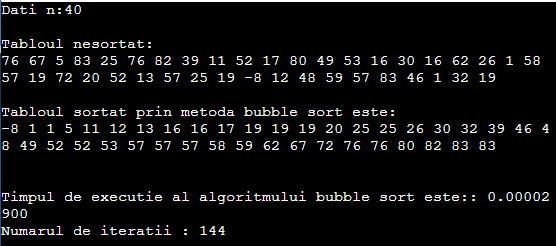


Fig.3.6.tablou din 40 numere

### Reprezentarea rezultatelor obținute și compararea lor cu ajutorul graficelor:

Fig.4.1.Iterațiile necesare pentru diferite valori

Fig.4.2. Timpul necesar pentru diferite valori

### Concluzie:

La crearea acestui laborator eu am facut cunoștință cu algoritmii de sortare ca Mergesort , Quicksort și Bubblesort. Pentru fiecare din ei am aflat cîte iterații și cit timp este necesar pentru diferite valorii(5-40). Comparîndule între ele aceste 3 metode folosind graficele (Fig.4.1. si Fig.4.2.) observînd cum timpul și numarul de iterații variază între ele,.pot spune că din ce în ce valoarea va fi mai mare bubble sort va avea nevoie de mai multe iterații și respectiv de mai mult timp decît Mergesort si Quicksort.