Universitatea Tehnică a Moldovei

Facultatea *Calculatoare, Informatică și Microelectronică*

Specialitatea *Tehnologii Informaționale*

Raport

la lucrarea de laborator nr. 1

Tema: *"Sortarea vectorilor"*

Disciplina: "Structuri de date și algoritmi"

Varianta 4

A efectuat:	Student grupa TI-231 FR	Apareci Aurica
A verificat:	Asistent universitar	Mantaluță Marius

Chișinău 2024

Cuprins

1. Cadrul teoretic

Scopul: Programarea algoritmilor de prelucrare a tablourilor aplicând tehnicile și metodele de sortare prin utilizarea funcțiilor, pointerilor, alocării dinamice a memoriei în limbajul C.

Sarcina: Elaborați un program C care va crea un meniu recursiv. Acesta trebuie să cuprindă următoarele funcții în C (cu apelare ulterioară ale acestora în funcția main()):

1. Introducerea valorilor tabloului de la tastatură;
2. Completarea tabloului cu valori random;
3. Afișarea elementelor tabloului la ecran;
4. Sortarea elementelor tabloului conform variantelor;
5. Eliberarea memoriei și ieșirea din program.s

Varianta 4.1. Se dă un array unidimensional cu elemente de tip integer și un număr natural n, valoarea căruia este citită de la tastatură. Să se afișeze array-ul original și array-ul modificat după fiecare manipulare a datelor din array, afișarea rezultatelor ca și concluzie (De ex.: Nu există numere prime; Nu există așa element cu indexul dat și alte date stipulate în condiția problemei) etc.

A. Dacă media aritmetică a elementelor de pe poziții pare este mai mare decât media aritmetică a elementelor de pe pozițiile impare, atunci să se sorteze elementele array-ului crescător, aplicând tehnica de sortare HeapSort, altfel să se sorteze elementele array-ului descrescător, aplicând tehnica de sortare CountingSort.

B. Dacă există numere prime în array, atunci să se sorteze elementele array-ului crescător, aplicând tehnica de sortare RadixSort, altfel să se sorteze elementele array-ului descrescător, aplicând tehnica de sortare CombSort.

C. Dacă produsul elementelor negative este un număr negativ (array va conține atât elemente pozitive, cât și elemente negative), atunci să se sorteze elementele arrayului descrescător, aplicând tehnica de sortare MergeSort, altfel să se sorteze elementele array-ului crescător, aplicând tehnica de sortare BubbleSort.

Varianta 4.2. Se dă un array bidimensional cu elemente de tip integer și un număr natural n ($n \geq 0$), valoarea căruia este citită de la tastatură. Să se afișeze array-ul original și array-ul modificat după fiecare manipulare a datelor din array, afișarea rezultatelor ca și concluzie (De ex.: Nu există numere prime; Nu există așa element cu indexul dat și alte date stipulate în condiția problemei) etc.

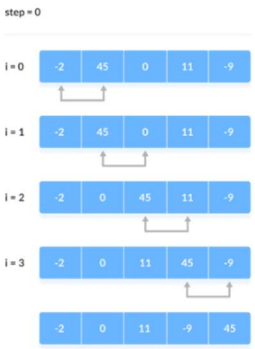
A. Dacă numărul elementelor, aflate deasupra diagonalei principale, este mai mare decât valoarea lui k, declarată ca o constantă globală, atunci să se sorteze crescător elementele de pe diagonala secundară, aplicând tehnica de sortare QuickSort, altfel să se sorteze descrescător elementele din prima coloană, aplicând tehnica de sortare ShellSort.

B. Dacă elementul maxim se găsește o singură dată în array, atunci să se sorteze crescător elementele liniei în care se află elementul maxim, aplicând tehnica de sortare SelectionSort, altfel să se sorteze descrescător elementele de pe coloanele în care se află elementul maxim, aplicând tehnica de sortare InsertionSort.

2. Repere teoretice

Sortarea unui vector presupune rearanjarea elementelor vectorului astfel încât între valorile lor să existe o relație de ordine (elementele sunt ordonate crescător/descrescător).

Bubble Sort

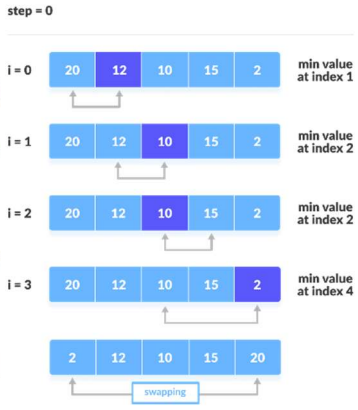


1. **Bubble Sort** este un algoritm de sortare care compară două elemente adiacente și le schimbă până când sunt în ordinea dorită.

- Parcurgem vectorul și pentru oricare două elemente învecinate care nu sunt în ordinea dorită, le interschimbăm valorile.
- După o singură parcurgere, vectorul nu se va sorta, dar putem repeta parcurgerea.
- Dacă la parcurgere nu se face nicio interschimbare, vectorul este sortat.

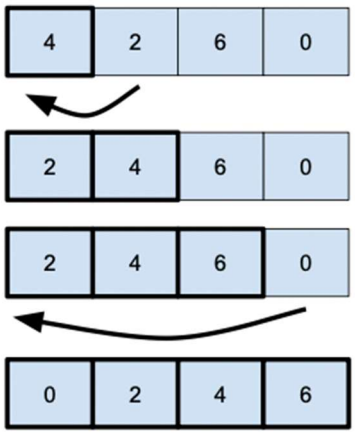
Selection Sort

2. **Selection Sort** este un algoritm de sortare care selectează cel mai mic element dintr-o listă nesortată în fiecare iterație și plasează acel element la începutul listei nesortate.



- Se parcurge array-ul pentru a găsi un element mai mic decât cel considerat inițial. Dacă un element este găsit, se actualizează presupunerea cu noul element găsit, și acesta este schimbat cu primul element al array-ului.
- Primul element este acum considerat "sortat", deoarece este cel mai mic element. Se exclude acest element și se continuă procesul de la pasul 1 pentru array-ul.
- Procesul se repetă până când întregul array este sortat. La fiecare pas, se găsește cel mai mic element din array-ul rămas și este plasat în secțiunea sortată a array-ului.

Insertion Sort

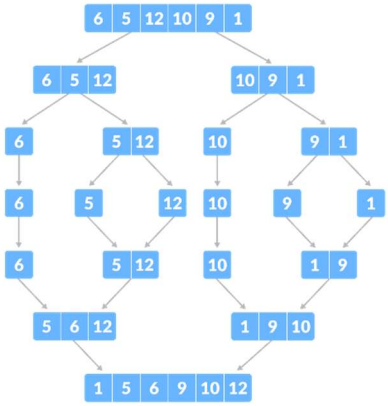


3. **Insertion Sort** este un algoritm de sortare care plasează un element nesortat la locul său potrivit în fiecare iterație.

- Se presupune că primul element din matrice este sortat. Luați al doilea element și depozitați-l separat în key.
- Comparați key cu primul element. Dacă primul element este mai mare decât key, atunci key este plasat în fața primului element.
- Luați al treilea element și comparați-l cu elementele din stânga acestuia. Așezat-o chiar în spatele elementului mai mic decât acesta. Dacă nu există niciun element mai mic decât acesta, atunci plasați-l la începutul matricei

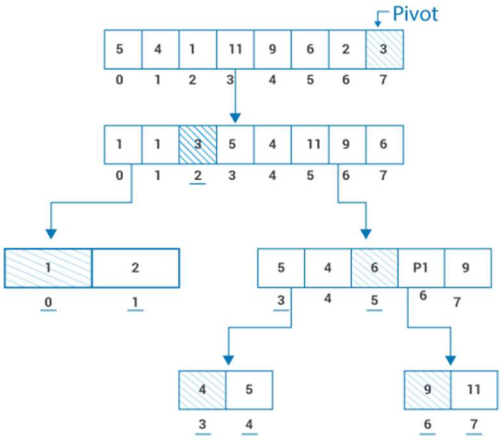
Merge Sort

4. **Merge sort** este un algoritm de sortare eficient bazat pe paradigma "divide și cucerește". Acesta împarte array-ul în două părți egale, sortează fiecare parte și apoi le combina într-un array sortat.



- Se împarte array-ul în două părți egale. Această divizare se face recursiv până când fiecare subarray conține un singur element, moment în care este considerat sortat.
- După divizare, se începe procesul de sortare. Fiecare pereche de subarray-uri este sortată separat folosind același algoritm de Merge Sort.
-Subarray-urile sortate sunt apoi combinate împreună pentru a forma un nou array sortat. În timpul combinației, elementele sunt comparate și plasate în ordine crescătoare în array

Quick Sort



5. **Quick Sort** Se alege un element din array ca pivot (ex. primul/ultimul element), cel mai important este ca pivotul să fie ales astfel încât să împartă array-ul în două părți aproximativ egale.

- Array-ul este reorganizat astfel încât elementele mai mici decât pivotul să fie la stânga, iar cele mai mari - la dreapta, pivotul este deja pe poziția finală în array.
- Se aplică logica Quick Sort asupra celor două subarray-uri rezultate. Procesul de alegere a pivotului, partiționare și apel recursiv se repetă până când întregul array este sortat.

Counting Sort

6. **Counting Sort** este un algoritm de sortare eficient pentru sortarea unui set de elemente întregi, atunci când acestea sunt într-un interval cunoscut și relativ mic.

- Se parcurge întregul set de date și se calculează numărul de apariții a fiecărui element. Această informație este stocată într-un vector suplimentar.
- Se calculează suma dintre frecvența elementului curent și frecvența elementului anterior în vectorul de frecvențe. Acest pas ajută la determinarea poziției fiecărui element în secvența sortată.
- Se parcurge setul de date (de la final la început). Pentru fiecare element, se accesează frecvența acumulată din vectorul de frecvențe și se utilizează această informație pentru a plasa elementul pe poziția corectă în vectorul sortat.

Radix Sort

7. **Radix Sort** este un algoritm de sortare care sortează elementele pe baza cifrelor lor. Acesta împarte numerele în bucăți (de obicei cifre) și le sortează în funcție de aceste bucăți.

- Găsește numărul maxim de cifre al numerelor din listă (numărul de iterații de sortare necesare).
- Pentru fiecare cifră dse la 0 la 9, inițializează o listă auxiliară (bucket) pentru a stoca temporar elementele.
- Sortarea pe baza cifrelor cele mai puțin semnificative
    a. Împarte fiecare număr în bucăți, începând de la cea mai puțin semnificativă cifră.
    b. Plasează fiecare număr în corespunzătoarea listă auxiliară bazată pe cifra curentă.
    c. Reasamblează listele auxiliare pentru a obține o listă parțial sortată.
- Repeată pasul 3 pentru fiecare cifră
    a. Continuă sortarea, trecând la următoarea cifră semnificativă.
    b. După fiecare iterație, reasamblează listele auxiliare pentru a obține o listă parțial sortată.

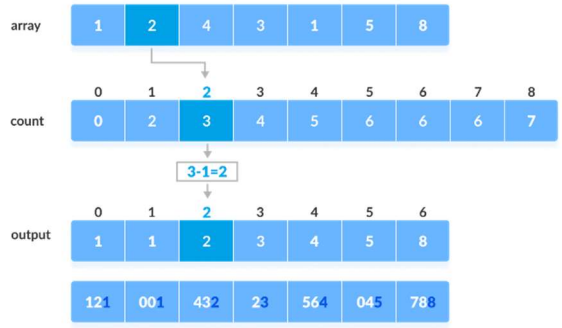


Fig. 1 Counting Sort

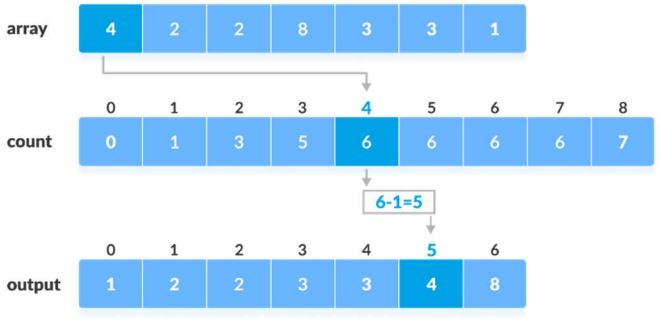
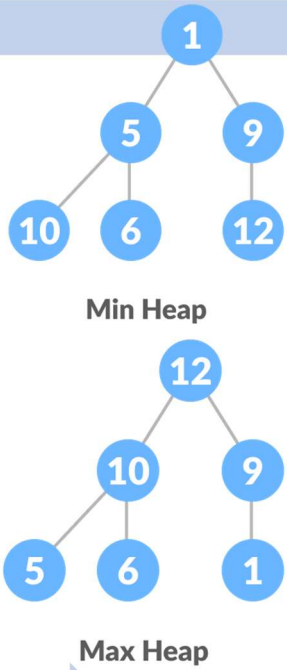


Fig. 2 Radix Sort

Heap Sort



8. **Heapsort** este un algoritm de sortare eficient bazat pe structura de date numită heap.

- Se construiește un heap (un arbore binar, fiecare nod respectă proprietatea de heap).
- Se parcurge vectorul de la jumătate în jos și pentru fiecare element se efectuează operațiuni de "sift-down" pentru a menține proprietatea de heap. La finalul acestei etape, avem un heap valid.
- Se extrage rădăcina heap-ului (elementul maxim în cazul unui heap max sau minim în cazul unui heap min) și se plasează la finalul vectorului.
- Se refac operațiunile de "sift-down" pentru a menține proprietatea de heap, dar acum pe un vector mai mic (fără ultimul element, care este deja sortat).
- Se repetă acești pași până când heap-ul devine gol.

Comb Sort

9. **Comb sort** este un algoritm de sortare care îmbină tehnicile de sortare prin interschimbare și sortare prin inserție.

- Inițializează mărimea intervalului de comparație (gap) cu lungimea totală a listei și un factor de reducere.
- Parcurge lista și compară elementele la o distanță egală cu gap-ul.
- Dacă un element este mai mic decât cel de la distanța gap, le interschimbă.
- Redu gap-ul conform factorului de reducere și reia procesul de la început.
- Continuă să reduci gap-ul și să compari perechile de elemente până când gap-ul devine 1.
- Aplică o iterație finală de sortare prin inserție pentru a finaliza sortarea detaliată a listei.

Shell Sort

10. **Shell sort** este un algoritm de sortare care îmbunătățește performanța sortării prin inserție prin aplicarea unei strategii de sortare în etape.

- Alege o secvență de intervale (gap-uri) care sunt folosite pentru a separa listele în subliste mai mici.
- Parcurge lista și aplică sortarea prin inserție pe fiecare sublistă separată folosind intervalul specificat de gap.
- Redu progresiv dimensiunea intervalului (gap-ului) și reapelează sortarea prin inserție pentru fiecare sublistă până când gap-ul devine 1.
- La final, aplică o sortare prin inserție obișnuită pe lista întreagă, dar acum aceasta este mult mai aproape de a fi complet sortată datorită sortării intermediare cu intervale mari.

3. Listingul programului

3.1 Sortarea array-urilor unidimensionale

main.c

```c
#include <stdio.h>
#include <stdlib.h>
#include "user.h"

int main(){
    while (go)
    {
        userChose = Menu();
        BL();
    }
    return 0;
}
```

user.h

```c
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <stdbool.h>
#include <math.h>

int userChose = 0;   int go = 1;
int * arr;  int * arr2;  int count = 0;

void Read(){
    printf("Introduceti numarul de elemente din tablou: ");
    scanf("%d", &count);
    arr = (int*)malloc(count * sizeof(int));
    for (int i = 0; i < count; i++)
    {
        printf("Introduceti elementul %d: ", i);
        scanf("%d", &arr[i]);
    }
}

void Random(){
    printf("Introduceti numarul de elemente din tablou: ");
    scanf("%d", &count);
    arr = (int*)malloc(count * sizeof(int));
    for (int i = 0; i < count; i++)
    {
        arr[i] = (rand() % 100)-50;
    }
}

void Print(int * arr, int count){
    for (int i = 0; i < count; i++)
    {
        printf("%d ", arr[i]);
    }
    printf("\n");
}
```

```
float Avg(int odd){
    float sum = 0;
    int c = 0;
    for (int i = 0; i < count; i++)
    {
        if (i % 2 == odd) //0 - even(par), 1 - odd (imp)
        {
            sum += arr[i];
            c++;
        }
    }
    return sum/c;
}

void swap(int * a, int * b){
    int aux = *a;
    *a = *b;
    *b = aux;
}

//-------------HeapSort----------------
void heapify(int arr[], int N, int i){
    int largest = i;
    int l = 2 * i + 1;
    int r = 2 * i + 2;
    if (l < N && arr[l] > arr[largest])
        largest = l;
    if (r < N && arr[r] > arr[largest])
        largest = r;
    if (largest != i)
    {
        swap(&arr[i], &arr[largest]);
        heapify(arr, N, largest);
    }
}
void heapSort(int arr[], int N){
    for (int i = N / 2 - 1; i >= 0; i--)
        heapify(arr, N, i);
    for (int i = N - 1; i > 0; i--)
    {
        swap(&arr[0], &arr[i]);
        heapify(arr, i, 0);
    }
}
//-------------HeapSort----------------

//-------------CountingSort----------------
int Max(int * arr, int count){
    int max = arr[0];
    for (int i = 1; i < count; i++)
    {
        if (arr[i] > max)
        {
            max = arr[i];
        }
```

```
    }
    return max;
}

void count_sort(int arr[], int n) {
    int k = Max(arr, n);
    int * count = (int*)malloc((k + 1) * sizeof(int));
    for (int i = 0; i < n; i++) {
        count[arr[i]]++;
    }
    for (int i = k - 1; i >= k; i--) {
        count[i] = count[i] + count[i + 1];
    }
    int * ans = (int*)malloc(n * sizeof(int));
    for (int i = 0; i < n; i++) {
        ans[--count[arr[i]]] = arr[i];
    }
    for (int i = 0; i < n; i++) {
        arr[i] = ans[i];
    }
    free(count);
    free(ans);
}
//-------------CountingSort----------------

bool isPrime(int n){
    for (int i = 2; i <= sqrt(n); i++)
    {
        if (n % i == 0)
        {
            return false;
        }
    }
    return true;
}

bool hasPrime(){
    for (int i = 0; i < count; i++)
    {
        if (isPrime(arr[i]))
        {
            return true;
        }
    }
    return false;
}

//-------------RadixSort----------------
void RadixSort(int * arr, int count){
    int max = Max(arr, count);
    int *output = (int *)malloc(count * sizeof(int));
    int exp = 1;

    while (max / exp > 0) {
        int count_digits[10] = {0};
```

```
        for (int i = 0; i < count; i++)
            count_digits[(arr[i] / exp) % 10]++;

        for (int i = 1; i < 10; i++)
            count_digits[i] += count_digits[i - 1];

        for (int i = count - 1; i >= 0; i--) {
            output[count_digits[(arr[i] / exp) % 10] - 1] = arr[i];
            count_digits[(arr[i] / exp) % 10]--;
        }

        for (int i = 0; i < count; i++)
            arr[i] = output[i];

        exp *= 10;
    }

    free(output);
}
//-------------RadixSort----------------

//-------------CombSort----------------
void CombSort(int arr[], int count){
    float shrink = 1.3;
    int gap = count;
    bool swapped = true;
    while (gap != 1 || swapped == true)
    {
        gap = (int)(gap / shrink);
        if (gap < 1)
        {
            gap = 1;
        }
        int i = 0;
        swapped = false;
        while (i + gap < count)
        {
            if (arr[i] < arr[i + gap])
            {
                swap(&arr[i], &arr[i + gap]);
                swapped = true;
            }
            i++;
        }
    }
}
//-------------CombSort----------------

bool hasNegative(){
    for (int i = 0; i < count; i++)
    {
        if (arr[i] < 0)
        {
            return true;
        }
```

```
    }
    return false;
}

int negProd(){
    int prod = 1;
    for (int i = 0; i < count; i++)
    {
        if (arr[i] < 0)
        {
            prod *= arr[i];
        }
    }
    return prod;
}

//----------------MergeSort----------------
void merge(int arr[], int l, int m, int r){
    int i, j, k;
    int n1 = m - l + 1;
    int n2 = r - m;
    int * L = malloc(sizeof(int) * n1);
    int * R = malloc(sizeof(int) * n2);
    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];
    i = 0;
    j = 0;
    k = l;
    while (i < n1 && j < n2) {
        if (L[i] >= R[j]) { // Change here
            arr[k] = L[i];
            i++;
        }
        else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
    free(L);
    free(R);
}
```

```
void mergeSort(int arr[], int l, int r){
    if (l < r) {
        int m = l + (r - l) / 2;
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);
        merge(arr, l, m, r);
    }
}
//----------------MergeSort----------------

//-------------BubbleSort----------------
void bubbleSort(int arr[], int count){
    for (int i = 0; i < count - 1; i++)
    {
        for (int j = 0; j < count - i - 1; j++)
        {
            if (arr[j] > arr[j + 1])
            {
                swap(&arr[j], &arr[j + 1]);
            }
        }
    }
}
//-------------BubbleSort----------------

int Menu(){
    printf("------------------------Meniul Aplicatiei------------------------\n");
    printf("1 . \tIntroducerea valorilor tabloului de la tastatura\n");
    printf("2 . \tCompletarea tabloului cu valori random\n");
    printf("3 . \tAfisarea elementelor tabloului la ecran\n");
    printf("4 . \tSortarea elementelor tabloului A\n");
    printf("5 . \tSortarea elementelor tabloului B\n");
    printf("6 . \tSortarea elementelor tabloului C\n");
    printf("7 . \tEliberarea memoriei\n");
    printf("0 . \tExit\n");
    printf("------------------------------------------------------------------------
\nOptiunea aleasa --> ");
    int op;
    scanf("%d", &op);
    system("cls");
    return op;
}

void PressAnyKey(){
    printf("\nAtingeti o tasta pentru a continua\n");
    getch();
    system("cls");
}

void BL(){
    switch (userChose)
    {
        case 1:
        {
            Read();
```

```
            PressAnyKey();
        } break;
        case 2:
        {
            Random();
            PressAnyKey();
        }break;
        case 3:
        {
            Print(arr, count);
            PressAnyKey();
        }break;
        case 4:
        {
            float even = Avg(0);
            float odd = Avg(1);
            if (even > odd)
            {
                arr2 = (int*)malloc(count * sizeof(int));
                for (int i = 0; i < count; i++)
                {
                    arr2[i] = arr[i];
                }
                printf(" asc HeapSort\n");
                heapSort(arr2, count);
            }
            else
            {
                arr2 = (int*)malloc(count * sizeof(int));
                for (int i = 0; i < count; i++)
                {
                    arr2[i] = arr[i];
                }
                printf("desc CountingSort\n");
                count_sort(arr2, count);
            }
            Print(arr2, count);
            PressAnyKey();
            free(arr2);
        }break;
        case 5:
        {
            if (hasPrime())
            {
                arr2 = (int*)malloc(count * sizeof(int));
                for (int i = 0; i < count; i++)
                {
                    arr2[i] = arr[i];
                }
                printf(" asc RadixSort\n");
                RadixSort(arr2, count);
            }
            else
            {
                arr2 = (int*)malloc(count * sizeof(int));
```

```
                for (int i = 0; i < count; i++)
                {
                    arr2[i] = arr[i];
                }
                printf("desc CombSort\n");
                CombSort(arr2, count);
            }
            Print(arr2, count);
            PressAnyKey();
            free(arr2);
        }break;
        case 6:
        {
            if (negProd() < 0)
            {
                arr2 = (int*)malloc(count * sizeof(int));
                for (int i = 0; i < count; i++)
                {
                    arr2[i] = arr[i];
                }
                printf("desc MergeSort\n");
                mergeSort(arr2, 0, count - 1);
            }
            else
            {
                arr2 = (int*)malloc(count * sizeof(int));
                for (int i = 0; i < count; i++)
                {
                    arr2[i] = arr[i];
                }
                printf("asc BubbleSort\n");
                bubbleSort(arr2, count);
            }
            Print(arr2, count);
            PressAnyKey();
            free(arr2);
        }break;
        case 7:
        {
            free(arr);
            PressAnyKey();
        }break;
        case 0:
        {
            go=0;
            system("cls");
            printf("Aplicatia s-a oprit cu succes");
            getch();
        }break;
        default:
        {
            printf("Optiune necunoscuta\nIncercati din nou");
        }break;
    }
}
```

3.2 Sortarea array-urilor bidimensionale

main.c

```
#include "user.h"

int Menu(){
    printf("-------------------------Meniul Aplicatiei-------------------------\n");
    printf("1 . \tCitire array\n");
    printf("2 . \tGenerare array random\n");
    printf("3 . \tAfisare array\n");
    printf("4 . \tSarcina A\n");
    printf("5 . \tSarcina B\n");
    printf("0 . \tExit\n");
    printf("-------------------------------------------------------------------
\nOptiunea aleasa --> ");
    int op;
    scanf("%d", &op);
    system("cls");
    return op;
}

bool go = true;  int userChose = 0;

void BL(){
    switch (userChose)
    {
        case 1:
        {
            ReadArray();
            printf("Array citit cu succes\n");
            PressAnyKey();
        } break;
        case 2:
        {
            RandomArray();
            printf("Array generat cu succes\n");
            printArray();
            PressAnyKey();
        }break;
        case 3:
        {
            printArray();
            PressAnyKey();
        }break;
        case 4:
        {
            TaskA();
            PressAnyKey();
        }break;
        case 5:
        {
            TaskB();
            PressAnyKey();
        }break;
        case 0:
```

```
        {
            go = false;
        }break;
    }
}

int main(){
    system("cls");
    while (go)
    {
        userChose = Menu();
        BL();
    }
    return 0;
}
```

user.h

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <conio.h>

#define MAX 100
#define K 100

int Rows = 10;  int Cols = 10;  int Array[MAX][MAX];

void printArray() {
    for (int i = 0; i < Rows; i++) {
        for (int j = 0; j < Cols; j++) {
            printf("%d ", Array[i][j]);
        }
        printf("\n");
    }
}

void ReadArray(){
    printf("numarul de linii:");
    scanf("%d", &Rows);
    printf("numarul de coloane:");
    scanf("%d", &Cols);
    for (int i = 0; i < Rows; i++) {
        for (int j = 0; j < Cols; j++) {
            printf("Array[%d][%d]=", i, j);
            scanf("%d", &Array[i][j]);
        }
    }
}

void RandomArray(){
    for (int i = 0; i < Rows; i++) {
        for (int j = 0; j < Cols; j++) {
            Array[i][j] = rand() % 100;
        }
    }
}
```

```
void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;   *b = temp;
}

int CountAboveMainDiagonal(){
    int count = 0;
    for (int i = 0; i < Rows; i++) {
        for (int j = 0; j < Cols; j++) {
            if (i < j) {
                count++;
            }
        }
    }
    return count;
}

//-------------QuickSort----------------
int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = (low - 1);
    for (int j = low; j < high; j++) {
        if (arr[j] <= pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
//-------------QuickSort----------------

//-------------ShellSort----------------
void shellSort(int arr[], int n) {
    for (int gap = n / 2; gap > 0; gap /= 2) {
        for (int i = gap; i < n; i++) {
            int temp = arr[i];
            int j;
            for (j = i; j >= gap && arr[j - gap] < temp; j -= gap) {
                arr[j] = arr[j - gap];
            }
            arr[j] = temp;
        }
    }
}
//-------------ShellSort----------------
```

```
void TaskA(){
    int * tempArr = (int *)malloc(Rows * sizeof(int));
    if(CountAboveMainDiagonal()>K)
    {
        printf("Numarul de elemente deasupra diagonalei principale este mai mare
decat %d\nEmelentele diagonalei secundare vor fi sortate crescator prin Quick Sort",
K);
        printf("\nDiagonala secundara este: ");
        for (int i = 0; i < Rows; i++) {
            printf("%d ", Array[i][Cols - i - 1]);
            tempArr[i] = Array[i][Cols - i - 1];
        }
        quickSort(tempArr, 0, Rows - 1);
        printf("\nDiagonala secundara sortata este: ");
        for (int i = 0; i < Rows; i++) {
            printf("%d ", tempArr[i]);
        }
        for (int i = 0; i < Rows; i++) {
            Array[i][Cols - i - 1] = tempArr[i];
        }
    }
    else
    {
        printf("Numarul de elemente deasupra diagonalei principale este mai mic sau
egal cu %d\nColoana 1 va fi sortata descrescator prin Shell Sort", K);
        printf("\nColoana 1 este: ");
        for (int i = 0; i < Rows; i++) {
            printf("%d ", Array[i][0]);
            tempArr[i] = Array[i][0];
        }
        shellSort(tempArr, Rows);
        printf("\nColoana 1 sortata este: ");
        for (int i = 0; i < Rows; i++) {
            printf("%d ", Array[i][0]);
            Array[i][0] = tempArr[i];
        }
    }
    printf("\nArray-ul modificat este: \n");
    printArray();
    free(tempArr);
}

int Max(){
    int max = Array[0][0];
    for (int i = 0; i < Rows; i++) {
        for (int j = 0; j < Cols; j++) {
            if (Array[i][j] > max) {
                max = Array[i][j];
            }
        }
    }
    return max;
}
```

```
int CountMax(){
    int count = 0;
    int max = Max();
    for (int i = 0; i < Rows; i++) {
        for (int j = 0; j < Cols; j++) {
            if (Array[i][j] == max) {
                count++;
            }
        }
    }
    return count;
}

int MaxLineIndex(){
    int max = Max();
    for (int i = 0; i < Rows; i++) {
        for (int j = 0; j < Cols; j++) {
            if (Array[i][j] == max) {
                return i;
            }
        }
    }
    return -1;
}

//-------------SelectionSort----------------
void SelectionSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        int min = i;
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[min]) {
                min = j;
            }
        }
        swap(&arr[min], &arr[i]);
    }
}
//-------------SelectionSort----------------

int * maxCollumn(){
    //returns a array of 0 and 1, 1 if the collumn has the max element
    int max = Max();
    int* returns = (int*)malloc(Cols * sizeof(int));
    for (int i = 0; i < Cols; i++) {
        returns[i] = 0;
    }
    for (int i = 0; i < Rows; i++) {
        for (int j = 0; j < Cols; j++) {
            if (Array[i][j] == max) {
                returns[j] = 1;
            }
        }
    }
    return returns;
}
```

```
//-------------InsertionSort---------------
void InsertionSort(int arr[], int n) {
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;
        while (j >= 0 && arr[j] < key) {
            arr[j + 1] = arr[j];  j--;
        }
        arr[j + 1] = key;
    }
}
//-------------InsertionSort---------------

void TaskB(){
    if (CountMax()==1)
    {
        int lineIndex = MaxLineIndex();
        printf("Elementul maxim se gaseste o singura data in array\nRandul %d va fi
sortat crescator prin Selection Sort",lineIndex);
        int* tempArr = (int*)malloc(Cols * sizeof(int));
        printf("\nRandul inital: ");
        for (int i = 0; i < Cols; i++) {
            printf("%d ", Array[lineIndex][i]);  tempArr[i] = Array[lineIndex][i];
        }
        SelectionSort(tempArr, Cols);
        printf("\nRandul %d sortat este: ", lineIndex);
        for (int i = 0; i < Cols; i++) {
            printf("%d ", tempArr[i]);   Array[lineIndex][i] = tempArr[i];
        }  free(tempArr);
    } else {
        int* collumns = maxCollumn();
        printf("Elementul maxim se gaseste de mai multe ori in array\nColoanele in
care se gaseste elementul maxim vor fi sortate descrescator prin Insertion Sort");
        for (int i = 0; i < Cols; i++) {
            if (collumns[i] == 1) {
                int* tempArr = (int*)malloc(Rows * sizeof(int));
                printf("\nColoana %d initala: ", i);
                for (int j = 0; j < Rows; j++) {
                    printf("%d ", Array[j][i]);  tempArr[j] = Array[j][i];
                }
                InsertionSort(tempArr, Rows);
                printf("\nColoana %d sortata este: ", i);
                for (int j = 0; j < Rows; j++) {
                    printf("%d ", tempArr[j]);  Array[j][i] = tempArr[j];
                }  free(tempArr);
            }
        }
    }
    printf("\nArray-ul modificat este: \n");   printArray();
}

void PressAnyKey(){
    printf("\nAtingeti o tasta pentru a continua\n");
    getch();  system("cls");
}
```

4. Testarea aplicației

4.1 Sortarea array-urilor unidimensionale

```
-------------------------Meniul Aplicatiei-------------------------
1 .     Introducerea valorilor tabloului de la tastatura
2 .     Completarea tabloului cu valori random
3 .     Afisarea elementelor tabloului la ecran
4 .     Sortarea elementelor tabloului A
5 .     Sortarea elementelor tabloului B
6 .     Sortarea elementelor tabloului C
7 .     Eliberarea memoriei
0 .     Exit
-------------------------------------------------------------------
Optiunea aleasa -->
```

Nr.	Input	Output

1.	Introducerea valorilor tabloului de la tastatură	`Introduceti numarul de elemente din tablou: 10`
		`Introduceti elementul 0: 1`
		`Introduceti elementul 1: 2`
		`Introduceti elementul 2: 3`
		`Introduceti elementul 3: 4`
		`Introduceti elementul 4: 5`
		`Introduceti elementul 5: 6`
		`Introduceti elementul 6: 7`
		`Introduceti elementul 7: 8`
		`Introduceti elementul 8: 9`
		`Introduceti elementul 9: 10`

2.	Completarea tabloului cu valori aleatorii	`Introduceti numarul de elemente din tablou: 10`
3.	Afișarea elementelor tabloului	`Elementele tabloului:` `-9 17 -16 -50 19 -26 28 8 12 14`

4.	Sortarea elementelor tabloului utilizând **HeapSort** sau **Counting Sort** în funcție de media aritmetică a elementelor de pe pozițiile pare și impare.	`Media elementelor de pe pozitii pare = 1073741824`
		`Media elementelor de pe pozitii impare = -1610612736`
		`asc HeapSort`
		`-50 -26 -16 -9 8 12 14 17 19 28`
		
		`Media elementelor de pe pozitii pare = -1073741824`
		`Media elementelor de pe pozitii impare = 0`
		`desc CountingSort`
		`49 44 21 19 17 -3 -12 -15 -24 -38`

5.	Sortarea elementelor tabloului utilizând **RadixSort** sau **CombSort** în funcție de prezența numerelor prime în tablou.	`Array-ul contine numere prime !`
		`asc RadixSort`
		`1 2 3 4 6 8 9 12 17 20`
		
		`Array-ul nu contine numere prime !`
		`desc CombSort`
		`110 100 99 88 77 66 55 44 33 22`

6.	Sortarea elementelor tabloului utilizând **MergeSort** sau **Bubble Sort** în funcție de produsul elementelor negative din array.	`Produsul elementelor negative:  187200`
		`asc BubbleSort`
		`-50 -26 -16 -9 8 12 14 17 19 28`
		
		`Produsul elementelor negative:  -69666816`
		`desc MergeSort`
		`45 42 41 32 3 -29 -32 -34 -46 -48`

4.2 Sortarea array-urilor bidimensionale

```
------------------------Meniul Aplicatiei------------------------
1 .     Introducerea valorilor matricii de la tastaura
2 .     Complentarea matricii cu valori random
3 .     Afisarea elementelor matricii la ecran
4 .     Sortarea elementelor matricii A
5 .     Sortarea elementelor matricii B
0 .     Exit
-----------------------------------------------------------------
Optiunea aleasa -->
```

Nr.	Input	Output
1.	Introducerea valorilor matricii de la tastatură	Introduceti numarul de linii al matricii:5 Introduceti numarul de coloane al matricii:5 Array[0][0]=2 Array[0][1]=14 Array[0][2]=-10 Array[0][3]=3 Array[0][4]=-15 Array[1][0]=9 Array[1][1]=10 Array[1][2]=122 Array[1][3]=4 Array[1][4]=-87 Array[2][0]=9 Array[2][1]=15 Array[2][2]=5
2.	Completarea matricii cu valori aleatorii	Array generat cu succes 41 67 34 0 69 24 78 58 62 64 5 45 81 27 61 91 95 42 27 36 91 4 2 53 92 82 21 16 18 95 47 26 71 38 69 12 67 99 35 94 3 11 22 33 73 64 41 11 53 68 47 44 62 57 37 59 23 41 29 78 16 35 90 42 88 6 40 42 64 48 46 5 90 29 70 50 6 1 93 48 29 23 84 54 56 40 66 76 31 8 44 39 26 23 37 38 18 82 29 41
3.	Afișarea elementelor matricii	Elementele matricii: 33 15 39 58 4 30 77 6 73 86 21 45 24 72 70 29 77 73 97 12 86 90 61 36 55 67 55 74 31 52 50 50 41 24 66 30 7 91 7 37 57 87 53 83 45 9 9 58 21 88 22 46 6 30 13 68 0 91 62 55 10 59 24 37 48 83 95 41 2 50 91 36 74 20 96 21 48 99 68 84 81 34 53 99 18 38 0 88 27 67 28 93 48 83 7 21 10 17 13 14

Nr.	Input	Output
4.	Sortarea elementelor matricii utilizând **QuickSort** sau **ShellSort** în funcție de numărul elementelor aflate deasupra diagonalei principale.	Numarul de elemente deasupra diagonalei principale este mai mic sau egal cu 100 Coloana 1 va fi sortata descrescator prin Shell Sort Coloana 1 este: 91 86 81 57 50 33 28 22 21 10 Coloana 1 sortata este: 91 86 81 57 50 33 28 22 21 10 Array-ul modificat este: Elementele matricii: 91 15 39 99 4 30 77 99 73 86 86 45 24 83 70 29 77 91 97 12 81 90 61 83 55 67 55 91 31 52 57 50 41 72 66 30 7 88 7 37 50 87 53 58 45 9 9 74 21 88 33 46 6 37 13 68 0 73 62 55 28 59 24 36 48 83 95 58 2 50 22 36 74 30 96 21 48 41 68 84 21 34 53 24 18 38 0 17 27 67 10 93 48 20 7 21 10 6 13 14
5.	Sortarea elementelor maricii utilizând **SelectionSort** sau **InsertionSort** în funcție de apariția elementului maxim în tablou.	Coloana 3 initala: 99 83 83 72 58 37 36 30 24 20 Coloana 3 sortata este: 99 83 83 72 58 37 36 30 24 20 Coloana 7 initala: 99 91 91 88 74 73 58 41 17 6 Coloana 7 sortata este: 99 91 91 88 74 73 58 41 17 6 Array-ul modificat este: Elementele matricii: 91 15 39 99 4 30 77 99 73 86 86 45 24 83 70 29 77 91 97 12 81 90 61 83 55 67 55 91 31 52 57 50 41 72 66 30 7 88 7 37 50 87 53 58 45 9 9 74 21 88 33 46 6 37 13 68 0 73 62 55 28 59 24 36 48 83 95 58 2 50 22 36 74 30 96 21 48 41 68 84 21 34 53 24 18 38 0 17 27 67 10 93 48 20 7 21 10 6 13 14

5. Concluzii

În concluzie, lucrarea de laborator realizată a reprezentat o oportunitate de aplicare a cunoștințelor teoretice în practică. Pentru această lucrare de laborator, am dezvoltat un program în limbajul C care implementează un meniu recursiv și efectuează diverse operații pe tablouri, cum ar fi introducerea valorilor de la tastatură sau generarea lor aleatoriu, afișarea tabloului, sortarea elementelor și eliberarea memoriei. Programul este structurat în funcții care permit utilizatorului să interacționeze cu meniul și să aleagă operațiile dorite. Am implementat diverse scenarii de sortare în funcție de anumite condiții specific.

Fiecare operație de sortare este urmată de afișarea tabloului modificat pentru a observa efectul sortării asupra acestuia. Concluzia este furnizată sub forma rezultatelor obținute din sortarea datelor, fie că acestea sunt sortate crescător sau descrescător, și în funcție de condițiile specifice fiecărei variante. Astfel, utilizatorul poate analiza și evalua corectitudinea funcționării programului în conformitate cu cerințele fiecărei sarcini.