

**MINISTERUL EDUCAȚIEI, CULTURII ȘI CERCETĂRII**

**AL REPUBLICII MOLDOVA**

**Universitatea Tehnică a Moldovei**

**Facultatea Calculatoare, Informatică şi Microelectronică**

**Departamentul Informatică şi Ingineria Sistemelor**

**gr. IA-233, Chistol Maxim**

**Raport**

**pentru lucrarea de laborator Nr.2**

***la cursul de “Structuri de Date si Algoritmi”***

Verificat:

**Maria Guțu,** *Lector Universitar.*

Departamentul Informatică şi IS,

Facultatea FCIM, UTM

**Chișinău 2024**

Cuprins:

[Scopul lucrării: 3](#_Toc159316590)

[Sarcina 4 4](#_Toc159316591)

[Codul: 5](#_Toc159316592)

[Codul: 19](#_Toc159316593)

[Rezultatele: 29](#_Toc159316594)

[Concluzie: 30](#_Toc159316595)

Scopul lucrării:

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

# Sarcina 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.

2. Se dă un array bidimensional cu elemente de tip integer și un număr natural n (n ≥ 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.

# Codul:

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <time.h>**

**void meniu(int \*vector, int n);**

**void citire(int \*vector, int n);**

**//void alocare(int \*\*vector, int n);**

**void afisare(int \*vector, int n);**

**void random\_arr(int \*vector, int n);**

**void freemem(int \*vector);**

**double media\_aritmetica\_pozitii(int \*vector, int n, int pozitii[]);**

**void sortare\_bubblesort(int \*vector, int n);**

**void manipulare\_array(int \*vector, int n);**

**int main() {**

**int n;**

**printf("Introduceți dimensiunea vectorului (număr par): ");**

**scanf("%d", &n);**

**if (n % 2 != 0) {**

**printf("Dimensiunea vectorului trebuie să fie un număr par. Ieșire din program.\n");**

**return 1;**

**}**

**printf("\nSe aloca memorie pentru vector......\n");**

**int \*vector = (int \*)malloc(n \* sizeof(int));**

**if (vector == NULL) {**

**printf("Eroare. Nu s-a alocat memorie.\n");**

**return 1;**

**}**

**printf("\nMemoria sa alocat cu succes!!!\n");**

**meniu(vector, n);**

**return 0;**

**}**

**void meniu(int \*vector, int n) {**

**int optiune;**

**// meniul ...**

**printf("\n\_\_\_\_\_\_\_\_\_\_Meniu\_\_\_\_\_\_\_\_\_\n"**

**"\n 1.Citire de la Tastatura \n"**

**"\n 2.Random Valori\n"**

**"\n 3.Afisare Vector\n"**

**"\n 4.Efectuarea conditiiei din Varianta\n"**

**"\n 5.Esire din Program\n"**

**"\n Alegeti optiunea:");**

**scanf("%d", &optiune);**

**printf("\n\n");**

**switch (optiune) {**

**case 1:**

**// citire de la tastatura a valorilor**

**citire(vector, n);**

**meniu(vector, n);**

**break;**

**case 2:**

**// random completare**

**random\_arr(vector, n);**

**meniu(vector, n);**

**break;**

**case 3:**

**// afisare a vector**

**afisare(vector, n);**

**break;**

**case 4:**

**// Efectuarea conditiei din varianta**

**manipulare\_array(vector, n);**

**meniu(vector, n);**

**break;**

**case 5:**

**// Iesire din program**

**freemem(vector);**

**exit(0);**

**default:**

**printf("Optiune invalida! Introduceti din nou.\n");**

**break;**

**}**

**meniu(vector, n); // Apelul recursiv al meniului**

**}**

**void citire(int \*vector, int n) {**

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

**printf("Introduceti valoarea pentru vector[%d]: ", i);**

**scanf("%d", &vector[i]);**

**}**

**}**

**void afisare(int \*vector, int n) {**

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

**printf("%d ", vector[i]);**

**}**

**printf("\n");**

**}**

**void random\_arr(int \*vector, int n) {**

**srand(time(NULL));**

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

**vector[i] = rand() % 100;**

**}**

**printf("Vectorul a fost completat cu valori random.\n");**

**}**

**void freemem(int \*vector) {**

**free(vector);**

**}**

**double media\_aritmetica\_pozitii(int \*vector, int n, int pozitii[]) {**

**double suma = 0;**

**int count = 0;**

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

**if (pozitii[i] < n) {**

**suma += vector[pozitii[i]];**

**count++;**

**}**

**}**

**return count > 0 ? suma / count : 0;**

**}**

**void schimba(int \*a, int \*b) {**

**int temp = \*a;**

**\*a = \*b;**

**\*b = temp;**

**}**

**void heapify(int \*vector, int n, int i) {**

**int largest = i;**

**int left = 2 \* i + 1;**

**int right = 2 \* i + 2;**

**if (left < n && vector[left] > vector[largest])**

**largest = left;**

**if (right < n && vector[right] > vector[largest])**

**largest = right;**

**if (largest != i) {**

**schimba(&vector[i], &vector[largest]);**

**heapify(vector, n, largest);**

**}**

**}**

**void build\_max\_heap(int \*vector, int n) {**

**for (int i = n / 2 - 1; i >= 0; i--) {**

**heapify(vector, n, i);**

**}**

**}**

**void sortare\_heapsort(int \*vector, int n) {**

**build\_max\_heap(vector, n);**

**for (int i = n - 1; i > 0; i--) {**

**schimba(&vector[0], &vector[i]);**

**heapify(vector, i, 0);**

**}**

**}**

**void sortare\_countingsort(int \*vector, int n) {**

**if (n <= 1) {**

**return; // Vectorul este deja sortat sau gol**

**}**

**// Găsește valoarea maximă din vector**

**int max = vector[0];**

**for (int i = 1; i < n; i++) {**

**if (vector[i] > max) {**

**max = vector[i];**

**}**

**}**

**// Alocă memorie pentru vectorul de frecvențe**

**int \*frecvente = (int \*)malloc((max + 1) \* sizeof(int));**

**if (frecvente == NULL) {**

**printf("Eroare la alocarea de memorie\n");**

**return;**

**}**

**// Inițializează vectorul de frecvențe cu 0**

**for (int i = 0; i <= max; i++) {**

**frecvente[i] = 0;**

**}**

**// Calculează frecvențele**

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

**frecvente[vector[i]]++;**

**}**

**// Reconstruiește vectorul sortat**

**int index = 0;**

**for (int i = 0; i <= max; i++) {**

**while (frecvente[i] > 0) {**

**vector[index] = i;**

**index++;**

**frecvente[i]--;**

**}**

**}**

**// Eliberează memoria alocată**

**free(frecvente);**

**}**

**int getMax(int vector[], int n) {**

**int max = vector[0];**

**for (int i = 1; i < n; i++)**

**if (vector[i] > max)**

**max = vector[i];**

**return max;**

**}**

**// Funcție pentru a efectua sortarea prin counting sort bazat pe un anumit digit (exp)**

**void countingSort(int vector[], int n, int exp) {**

**const int radix = 10; // Baza pentru numărul de cifre**

**int output[n]; // Vectorul de ieșire**

**int count[radix];**

**for (int i = 0; i < radix; i++)**

**count[i] = 0;**

**// Contorizează frecvența fiecărei cifre la poziția curentă**

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

**count[(vector[i] / exp) % radix]++;**

**// Calculul pozițiilor finale ale fiecărei cifre în vectorul de ieșire**

**for (int i = 1; i < radix; i++)**

**count[i] += count[i - 1];**

**// Construiește vectorul de ieșire**

**for (int i = n - 1; i >= 0; i--) {**

**output[count[(vector[i] / exp) % radix] - 1] = vector[i];**

**count[(vector[i] / exp) % radix]--;**

**}**

**// Copiază vectorul de ieșire în vectorul original**

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

**vector[i] = output[i];**

**}**

**// Funcție principală pentru a efectua sortarea radixsort pe un vector**

**void radixsort(int vector[], int n) {**

**// Găsește valoarea maximă pentru a determina numărul de cifre**

**int max = getMax(vector, n);**

**// Efectuează counting sort pentru fiecare cifră**

**for (int exp = 1; max / exp > 0; exp \*= 10)**

**countingSort(vector, n, exp);**

**}**

**void combsort(int vector[], int n) {**

**const float shrink\_factor = 1.3; // Factorul de contracție**

**int gap = n; // Setează spațiul inițial între elemente**

**int swapped = 1;**

**while (gap > 1 || swapped) {**

**// Actualizează spațiul între elemente**

**if (gap > 1) {**

**gap = (int)(gap / shrink\_factor);**

**}**

**// Resetăm variabila de schimbare a pozițiilor**

**swapped = 0;**

**// Efectuăm comparații și interschimbări**

**for (int i = 0; i < n - gap; i++) {**

**if (vector[i] > vector[i + gap]) {**

**// Realizăm interschimbarea**

**int temp = vector[i];**

**vector[i] = vector[i + gap];**

**vector[i + gap] = temp;**

**// Setăm variabila pentru a indica o schimbare**

**swapped = 1;**

**}**

**}**

**}**

**}**

**void merge(int vector[], int stanga, int mijloc, int dreapta) {**

**int n1 = mijloc - stanga + 1;**

**int n2 = dreapta - mijloc;**

**// Creează două sub-vectori temporari**

**int stanga\_vector[n1];**

**int dreapta\_vector[n2];**

**// Copiază datele în sub-vectorii temporari**

**for (int i = 0; i < n1; i++)**

**stanga\_vector[i] = vector[stanga + i];**

**for (int j = 0; j < n2; j++)**

**dreapta\_vector[j] = vector[mijloc + 1 + j];**

**// Combinați sub-vectorii înapoi în vectorul original**

**int i = 0; // Indexul pentru primul sub-vector**

**int j = 0; // Indexul pentru al doilea sub-vector**

**int k = stanga; // Indexul pentru vectorul combinat**

**while (i < n1 && j < n2) {**

**if (stanga\_vector[i] <= dreapta\_vector[j]) {**

**vector[k] = stanga\_vector[i];**

**i++;**

**} else {**

**vector[k] = dreapta\_vector[j];**

**j++;**

**}**

**k++;**

**}**

**// Copiază elementele rămase din stanga\_vector, dacă există**

**while (i < n1) {**

**vector[k] = stanga\_vector[i];**

**i++;**

**k++;**

**}**

**// Copiază elementele rămase din dreapta\_vector, dacă există**

**while (j < n2) {**

**vector[k] = dreapta\_vector[j];**

**j++;**

**k++;**

**}**

**}**

**// Funcție de sortare MergeSort pentru a sorta un vector unidimensional**

**void mergeSort(int vector[], int stanga, int dreapta) {**

**if (stanga < dreapta) {**

**// Găsește mijlocul vectorului**

**int mijloc = stanga + (dreapta - stanga) / 2;**

**// Sortează prima jumătate**

**mergeSort(vector, stanga, mijloc);**

**// Sortează a doua jumătate**

**mergeSort(vector, mijloc + 1, dreapta);**

**// Combinați cele două jumătăți sortate**

**merge(vector, stanga, mijloc, dreapta);**

**}**

**}**

**void sortare\_bubblesort(int \*vector, int n) {**

**for (int i = 0; i < n - 1; i++) {**

**for (int j = 0; j < n - i - 1; j++) {**

**if (vector[j] > vector[j + 1]) {**

**schimba(&vector[j], &vector[j + 1]);**

**}**

**}**

**}**

**}**

**void manipulare\_array(int \*vector, int n) {**

**int \*pozitii\_pare = (int \*)malloc(n \* sizeof(int));**

**int \*pozitii\_impare = (int \*)malloc(n \* sizeof(int));**

**int count\_pare = 0, count\_impare = 0;**

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

**if (i % 2 == 0) {**

**pozitii\_pare[count\_pare++] = i;**

**} else {**

**pozitii\_impare[count\_impare++] = i;**

**}**

**}**

**double media\_pare = media\_aritmetica\_pozitii(vector, n, pozitii\_pare);**

**double media\_impare = media\_aritmetica\_pozitii(vector, n, pozitii\_impare);**

**if (media\_pare > media\_impare) {**

**sortare\_heapsort(vector, n);**

**} else {**

**sortare\_countingsort(vector, n);**

**}**

**int are\_prime = 0;**

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

**if (vector[i] > 1) {**

**are\_prime = 1;**

**break;**

**}**

**}**

**if (are\_prime) {**

**radixsort(vector, n);**

**} else {**

**combsort(vector, n);**

**}**

**int produs\_negativ = 1;**

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

**if (vector[i] < 0) {**

**produs\_negativ \*= vector[i];**

**}**

**}**

**if (produs\_negativ < 0) {**

**mergeSort(vector, 0, n - 1);**

**} else {**

**sortare\_bubblesort(vector, n);**

**}**

**afisare(vector, n);**

**free(pozitii\_pare);**

**free(pozitii\_impare);**

**meniu(vector, n);**

**}**

# Codul:

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <time.h>**

**#define MAX 100**

**// Constantă globală pentru manipularea array-ului**

**const int k = 5; // Modifică valoarea conform cerințelor**

**// Prototipuri de funcții**

**int\*\* alocaMemorie(int n);**

**void elibereazaMemorie(int\*\* array, int n);**

**void afisareArray(int\*\* array, int n);**

**void quickSortDiagonalaSecundara(int\*\* array, int stanga, int dreapta, int n);**

**void shellSortPrimaColoana(int\*\* array, int n);**

**void selectionSortLinie(int\*\* array, int linie, int n);**

**void insertionSortColoane(int\*\* array, int coloana, int n);**

**void citire(int\*\* array, int n);**

**void randomize(int\*\* array, int n);**

**void manipulareCuArrayA(int\*\* array, int n);**

**void manipulareCuArrayB(int\*\* array, int n);**

**int main() {**

**int n;**

**printf("Introduceți dimensiunea array-ului (n): ");**

**scanf("%d", &n);**

**if (n <= 0 || n > MAX) {**

**printf("Dimensiune invalidă a array-ului. Ieșire din program.\n");**

**return 1;**

**}**

**// Alocare dinamică a memoriei pentru array**

**int\*\* array = alocaMemorie(n);**

**int optiune;**

**do {**

**// meniul**

**printf("\n\_\_\_\_\_\_\_\_\_\_Meniu\_\_\_\_\_\_\_\_\_\n"**

**"\n 1.Citire de la Tastatura \n"**

**"\n 2.Random Valori\n"**

**"\n 3.Afisare Vector\n"**

**"\n 4.Efectuarea sarcinei A\n"**

**"\n 5.Efctuarea sarcinei B\n"**

**"\n 6.Esire din Program\n"**

**"\n Alegeti optiunea: ");**

**scanf("%d", &optiune);**

**printf("\n\n");**

**switch (optiune) {**

**case 1:**

**// citire de la tastatura a valorilor**

**citire(array, n);**

**break;**

**case 2:**

**// random completare**

**randomize(array, n);**

**break;**

**case 3:**

**// afisare a vector**

**afisareArray(array, n);**

**break;**

**case 4:**

**// Sarcina A**

**manipulareCuArrayA(array, n);**

**break;**

**case 5:**

**// Sarcina B**

**manipulareCuArrayB(array, n);**

**break;**

**case 6:**

**// Iesire din program**

**elibereazaMemorie(array, n);**

**exit(0);**

**default:**

**printf("Optiune invalida! Introduceti din nou.\n");**

**break;**

**}**

**} while (optiune != 6);**

**return 0;**

**}**

**// Funcție pentru alocarea dinamică a memoriei pentru array**

**int\*\* alocaMemorie(int n) {**

**int\*\* array = (int\*\*)malloc(n \* sizeof(int\*));**

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

**array[i] = (int\*)malloc(n \* sizeof(int));**

**}**

**return array;**

**}**

**// Funcție pentru eliberarea memoriei alocate dinamic**

**void elibereazaMemorie(int\*\* array, int n) {**

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

**free(array[i]);**

**}**

**free(array);**

**}**

**// Funcție pentru afișarea array-ului**

**void afisareArray(int\*\* array, int n) {**

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

**for (int j = 0; j < n; j++) {**

**printf("%d\t", array[i][j]);**

**}**

**printf("\n");**

**}**

**printf("\n");**

**}**

**// Funcție pentru QuickSort pe diagonala secundara**

**void quickSortDiagonalaSecundara(int\*\* array, int stanga, int dreapta, int n) {**

**if (stanga < dreapta) {**

**int pivot = array[dreapta][n - 1 - dreapta];**

**int i = stanga - 1;**

**for (int j = stanga; j < dreapta; j++) {**

**if (array[j][n - 1 - j] <= pivot) {**

**i++;**

**// Schimbarea elementelor**

**int temp = array[j][n - 1 - j];**

**array[j][n - 1 - j] = array[i][n - 1 - i];**

**array[i][n - 1 - i] = temp;**

**}**

**}**

**// Schimbarea elementelor**

**int temp = array[dreapta][n - 1 - dreapta];**

**array[dreapta][n - 1 - dreapta] = array[i + 1][n - 1 - i - 1];**

**array[i + 1][n - 1 - i - 1] = temp;**

**int pivotIndex = i + 1;**

**// Sortarea recursivă a părților stângă și dreaptă ale pivotului**

**quickSortDiagonalaSecundara(array, stanga, pivotIndex - 1, n);**

**quickSortDiagonalaSecundara(array, pivotIndex + 1, dreapta, n);**

**}**

**}**

**// Funcție pentru ShellSort pe prima coloană**

**void shellSortPrimaColoana(int\*\* array, int n) {**

**for (int gap = n / 2; gap > 0; gap /= 2) {**

**for (int i = gap; i < n; i++) {**

**for (int j = i - gap; j >= 0 && array[j][0] < array[j + gap][0]; j -= gap) {**

**// Schimbarea elementelor**

**for (int k = 0; k < n; k++) {**

**int temp = array[j][k];**

**array[j][k] = array[j + gap][k];**

**array[j + gap][k] = temp;**

**}**

**}**

**}**

**}**

**}**

**// Funcție pentru SelectionSort pe o linie specificată**

**void selectionSortLinie(int\*\* array, int linie, int n) {**

**for (int i = 0; i < n - 1; i++) {**

**int minIndex = i;**

**for (int j = i + 1; j < n; j++) {**

**if (array[linie][j] < array[linie][minIndex]) {**

**minIndex = j;**

**}**

**}**

**// Schimbarea elementelor**

**int temp = array[linie][i];**

**array[linie][i] = array[linie][minIndex];**

**array[linie][minIndex] = temp;**

**}**

**}**

**// Funcție pentru InsertionSort pe o coloană specificată**

**void insertionSortColoane(int\*\* array, int coloana, int n) {**

**for (int i = 1; i < n; i++) {**

**int key = array[i][coloana];**

**int j = i - 1;**

**// Mișcarea elementelor mai mari la dreapta**

**while (j >= 0 && array[j][coloana] < key) {**

**array[j + 1][coloana] = array[j][coloana];**

**j--;**

**}**

**array[j + 1][coloana] = key;**

**}**

**}**

**// Funcție pentru citirea array-ului de la tastatură**

**void citire(int\*\* array, int n) {**

**printf("Introduceți elementele array-ului:\n");**

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

**for (int j = 0; j < n; j++) {**

**printf("array[%d][%d]: ", i, j);**

**scanf("%d", &array[i][j]);**

**}**

**}**

**}**

**// Funcție pentru completarea aleatoare a array-ului**

**void randomize(int\*\* array, int n) {**

**srand(time(NULL));**

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

**for (int j = 0; j < n; j++) {**

**array[i][j] = rand() % 100;**

**}**

**}**

**}**

**// Funcție pentru manipularea array-ului conform sarcinii A**

**void manipulareCuArrayA(int\*\* array, int n) {**

**// A. Manipularea array-ului în funcție de condițiile date**

**int contorElementeDeasupraDiagonaleiPrincipale = 0;**

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

**for (int j = i + 1; j < n; j++) {**

**if (array[i][j] > k) {**

**contorElementeDeasupraDiagonaleiPrincipale++;**

**}**

**}**

**}**

**if (contorElementeDeasupraDiagonaleiPrincipale > k) {**

**// Sortare crescătoare elemente diagonala secundara folosind QuickSort**

**quickSortDiagonalaSecundara(array, 0, n - 1, n);**

**printf("Sortare crescatoare elemente diagonala secundara:\n");**

**} else {**

**// Sortare descrescatoare elemente prima coloana folosind ShellSort**

**shellSortPrimaColoana(array, n);**

**printf("Sortare descrescatoare elemente prima coloana:\n");**

**}**

**// Afisarea array-ului dupa manipulare**

**afisareArray(array, n);**

**}**

**// Funcție pentru manipularea array-ului conform sarcinii B**

**void manipulareCuArrayB(int\*\* array, int n) {**

**// B. Manipularea array-ului în funcție de condițiile date**

**int maxElement = array[0][0];**

**int maxLinie = 0;**

**int maxColoane[MAX];**

**int contorMaxColoane = 0;**

**// Găsirea elementului maxim și identificarea liniilor și coloanelor asociate**

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

**for (int j = 0; j < n; j++) {**

**if (array[i][j] > maxElement) {**

**maxElement = array[i][j];**

**maxLinie = i;**

**contorMaxColoane = 0;**

**} else if (array[i][j] == maxElement) {**

**maxColoane[contorMaxColoane++] = j;**

**}**

**}**

**}**

**if (contorMaxColoane == 1) {**

**// Sortare crescatoare elemente liniei in care se afla elementul maxim folosind SelectionSort**

**selectionSortLinie(array, maxLinie, n);**

**printf("Sortare crescatoare elemente liniei cu elementul maxim:\n");**

**} else {**

**// Sortare descrescatoare elemente de pe coloanele in care se afla elementul maxim folosind InsertionSort**

**for (int i = 0; i < contorMaxColoane; i++) {**

**insertionSortColoane(array, maxColoane[i], n);**

**}**

**printf("Sortare descrescatoare elemente de pe coloanele cu elementul maxim:\n");**

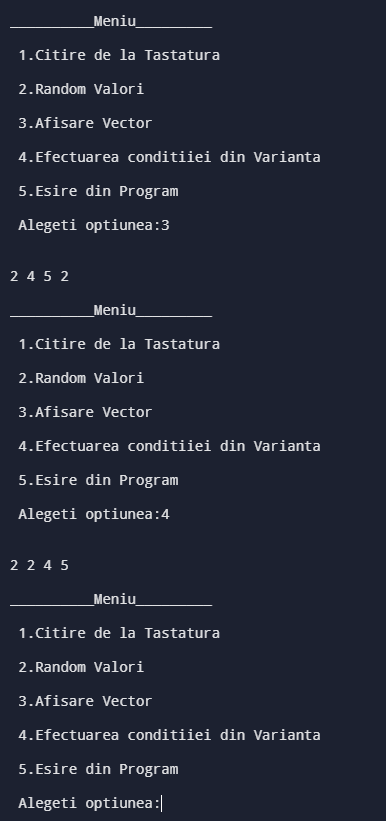
**}**

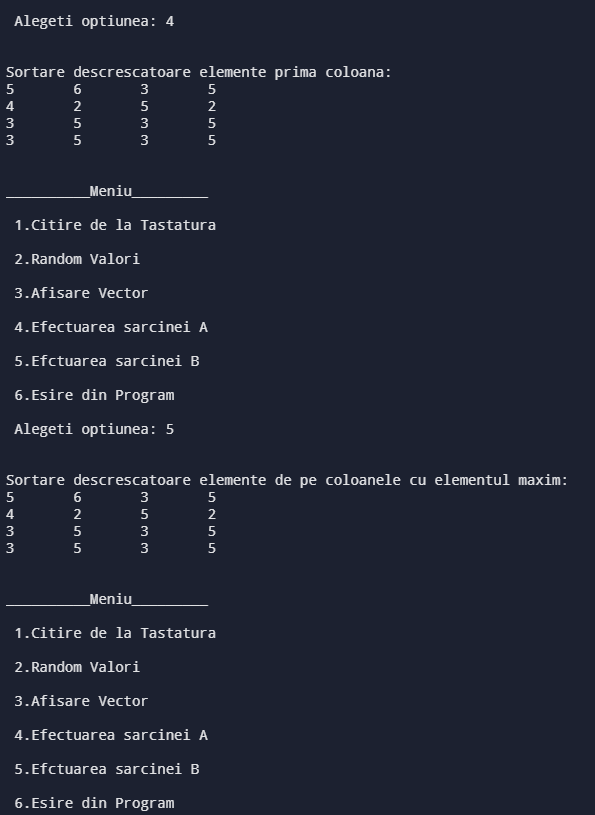
**// Afisarea array-ului dupa manipulare**

**afisareArray(array, n);**

**}**

# Rezultatele:

****

****

# Concluzie:

Scopul lucrării a fost să implementeze algoritmi eficienți pentru prelucrarea tablourilor în limbajul de programare C, folosind diverse tehnici și metode de sortare, cum ar fi QuickSort, ShellSort, SelectionSort și InsertionSort. Acești algoritmi au fost aplicați asupra unui tablou bidimensional, unde utilizatorul poate alege să introducă manual valorile sau să genereze aleatoriu datele.

În cadrul lucrării, am implementat un meniu interactiv pentru a oferi utilizatorului opțiuni diverse, precum citirea manuală a datelor, generarea aleatoare a acestora, afișarea tabloului, și aplicarea celor două sarcini specificate (A și B) în funcție de condițiile date. Am folosit funcții, pointeri și alocare dinamică a memoriei pentru a asigura o implementare corectă și eficientă.

Concluziile desprinse din această lucrare arată că programarea algoritmilor de sortare în limbajul C poate fi realizată într-un mod modular, clar și eficient. Utilizarea tehnicilor de sortare precum QuickSort sau ShellSort poate îmbunătăți semnificativ performanța în funcție de natura datelor de intrare și cerințele specifice ale problemei.

Alocarea dinamică a memoriei și manipularea tablourilor bidimensionale necesită atenție sporită pentru a evita scurgerile de memorie și erorile de acces. Prin intermediul acestui proiect, s-a evidențiat importanța înțelegerii și aplicării corecte a conceptelor legate de pointeri, alocare dinamică și funcții în limbajul C pentru a dezvolta soluții robuste și eficiente la problemele de prelucrare a datelor.