

**MINISTRY OF EDUCATION, CULTURE AND RESEARCH**

**OF THE REPUBLIC OF MOLDOVA**

**Technical University of Moldova**

**Faculty of Computers, Informatics and Microelectronics**

**Department of Software and Automation Engineering**

**Iamandii Ion student**

**Group: FAF-233**

**Report**

**Laboratory Work No.3**

***of the "Data Structures and Algorithms" course***

Checked:

Burlacu Natalia, PhD, Associate Professor

Department of Software and Automation Engineering,

FCIM Faculty, UTM

**Chisinau – 2024**

**Task:**

1. Solve the following problems in C, writing your own functions according to the given statements. Write the solution of the problem by procedural approach in two versions:

A. with the use of the method of transmitting the parametric functions by value;

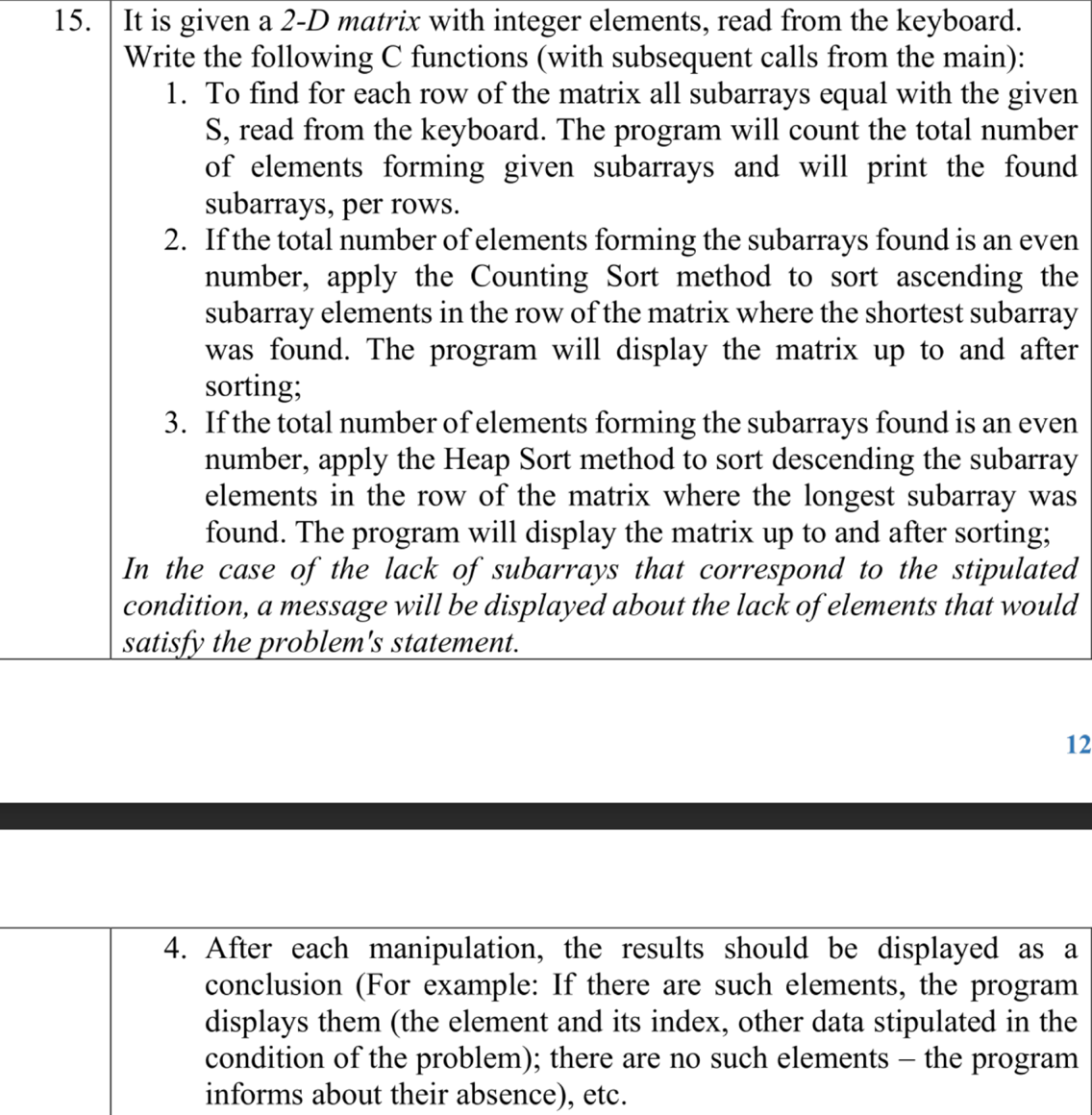
B. with the use of the method of passing parameters of functions by address/pointers (the formal parameter will be a pointer to the value of the corresponding object).

C. To draw the block diagram corresponding to the solved problem.

2. Modify the content of your problems emerging from the possibilities that are missing, but which can be brought as added value in the condition of the existing problem. Formulate and present in writing the modified condition; to solve in C your problem in the modified version, using functions developed by you

Because of the fact that in every problem in version 1, you should use two specified sorting methods, in version 2, of the problem proposed (modified) by you, you should use the sorting methods as Combo Sort & Radix Sort.

**Condition of the problem:**



**Fig 1.1** – Condition of problem

**1. The code of the program, with relevant comments in it, and the Block diagram;**

**Code:**

**--------------** **The version with passing function parameters by pointers ---------------**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

void countingSort(int array[], int size) {

int output[10];

// Find the largest element of the array

int max = array[0];

for (int i = 1; i < size; i++) {

if (array[i] > max)

max = array[i];

}

// The size of count must be at least (max+1) but

// we cannot declare it as int count(max+1) in C as

// it does not support dynamic memory allocation.

// So, its size is provided statically.

int count[10];

// Initialize count array with all zeros.

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

count[i] = 0;

}

// Store the count of each element

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

count[array[i]]++;

}

// Store the cummulative count of each array

for (int i = 1; i <= max; i++) {

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

}

// Find the index of each element of the original array in count array, and

// place the elements in output array

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

output[count[array[i]] - 1] = array[i];

count[array[i]]--;

}

// Copy the sorted elements into original array

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

array[i] = output[i];

}

}

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

int temp = \*a;

\*a = \*b;

\*b = temp;

}

void heapify(int arr[], int n, int i) {

// Find largest among root, left child and right child

int largest = i;

int left = 2 \* i + 1;

int right = 2 \* i + 2;

if (left < n && arr[left] > arr[largest])

largest = left;

if (right < n && arr[right] > arr[largest])

largest = right;

// Swap and continue heapifying if root is not largest

if (largest != i) {

swap(&arr[i], &arr[largest]);

heapify(arr, n, largest);

}

}

void heapSort(int arr[], int n) {

// Build max heap

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

heapify(arr, n, i);

// Heap sort

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

swap(&arr[0], &arr[i]);

// Heapify root element to get highest element at root again

heapify(arr, i, 0);

}

}

void findCombinationsRecursive(int \*arr, int size, int row, int rows, int targetSum, int \*\*result, int \*resultSize, int currentCombination[size], int currentSize, int startIndex) {

if (currentSize > 0) {

int sum = 0;

for (int i = 0; i < currentSize; ++i) {

sum += currentCombination[i];

}

if (sum == targetSum) {

// If the sum is equal to the target, add the combination to the result array

result[\*resultSize][0] = currentSize;

for (int i = 0; i < currentSize; ++i) {

result[\*resultSize][i + 1] = currentCombination[i];

}

\*resultSize += 1;

}

}

// Continue building the combination

for (int i = startIndex; i < size; ++i) {

currentCombination[currentSize] = arr[i];

findCombinationsRecursive(arr, size, row, rows, targetSum, result, resultSize, currentCombination, currentSize + 1, i + 1);

// If a combination is found, stop further recursion

if (\*resultSize > 0) {

return;

}

}

}

void findCombinations(int \*arr, int size, int row, int rows, int targetSum, int \*\*result, int \*resultSize) {

\*resultSize = 0;

int currentCombination[size];

findCombinationsRecursive(arr, size, row, rows, targetSum, result, resultSize, currentCombination, 0, 0);

}

int main() {

int rows, size;

printf("Enter the number of rows: ");

scanf("%d", &rows);

printf("Enter the number of columns: ");

scanf("%d", &size);

int \*\*arr = (int \*\*)malloc(rows \* sizeof(int \*));

for (int i = 0; i < rows; ++i) {

arr[i] = (int \*)malloc(size \* sizeof(int));

}

// input array

printf("Enter elements for the %dx%d array:\n", rows, size);

for (int i = 0; i < rows; ++i) {

for (int j = 0; j < size; ++j) {

printf("Enter element at position [%d][%d]: ", i, j);

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

}

}

// display entered array

printf("\nEntered %dx%d array:\n", rows, size);

for (int i = 0; i < rows; ++i) {

for (int j = 0; j < size; ++j) {

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

}

printf("\n");

}

int targetSum;

printf("\nEnter the sum of each subarray: ");

scanf("%d", &targetSum);

int \*\*result = (int \*\*)malloc(rows \* sizeof(int \*));

for (int i = 0; i < rows; ++i) {

result[i] = (int \*)malloc(size \* sizeof(int));

}

int \*\*combinations = (int \*\*)malloc(rows \* sizeof(int \*));

for (int i = 0; i < rows; ++i) {

combinations[i] = (int \*)malloc(size \* sizeof(int));

}

for (int r = 0; r < rows; r++) {

for (int j = 0; j < size; j++) {

combinations[r][j] = 0;

}

}

int nrElementsInSubarrays = 0;

for (int r = 0; r < rows; r++) {

int resultSize = 0;

findCombinations(arr[r], size, r, rows, targetSum, result, &resultSize);

if (resultSize > 0) {

for (int i = 0; i < result[0][0]; i++) {

combinations[r][i] = result[0][i + 1];

nrElementsInSubarrays++;

}

}

}

printf("\nSubarrays:\n");

for (int r = 0; r < rows; r++) {

printf("Row %d: ", r+1);

bool noCombinations = true;

for (int j = 0; j < size; j++) {

if (combinations[r][j] != 0) {

printf("%d ", combinations[r][j]);

noCombinations = false;

}

}

if (noCombinations) {

printf("No elements in this subarray");

}

printf("\n");

}

printf("\nThe total number of elements forming given subarrays: %d\n\n", nrElementsInSubarrays);

if (nrElementsInSubarrays % 2 == 0) { // if nrElementsInSubarrays is even

// counting sort ascendingly the shortest subarray

int indexAndLength[rows][1];

for (int i = 0; i < rows; i++) {

indexAndLength[i][0] = 0;

}

// record the length of each subarray

for (int i = 0; i < rows; i++) {

int length = 0;

for (int j = 0; j < size; j++) {

if (combinations[i][j] != 0) {

length++;

}

}

indexAndLength[i][0] = length;

}

// find the index of the row where the shortest subarray is

int index = 0;

for (int i = 0; i < rows; i++) {

if ((indexAndLength[i][0] != 0) && (indexAndLength[i][0] <= indexAndLength[index][0])) {

index = i;

}

}

int length = indexAndLength[index][0];

int subarray[2][length];

for (int i = 0; i < length; i++) {

subarray[0][i] = combinations[index][i];

subarray[1][i] = combinations[index][i];

}

countingSort(subarray[1], length);

printf("Matrix with resorted rows:\n");

for (int row = 0; row < rows; row++) {

for (int col = 0; col < size; col++) {

if (row != index) {

printf("%d\t", arr[row][col]);

} else {

bool check = true;

for (int i = 0; i < length; i++) {

if (arr[row][col] == subarray[0][i]) {

printf("%d\t", subarray[1][i]);

check = false;

break;

}

}

if (check) {

printf("%d\t", arr[row][col]);

}

}

}

printf("\n");

}

} else { // if nrElementsInSubarrays is odd:

// heap sort descendingly the longest subarray

int indexAndLength[rows][1];

for (int i = 0; i < rows; i++) {

indexAndLength[i][0] = 0;

}

// record the length of each subarray

for (int i = 0; i < rows; i++) {

int length = 0;

for (int j = 0; j < size; j++) {

if (combinations[i][j] != 0) {

length++;

}

}

indexAndLength[i][0] = length;

}

// find the index of the row where the shortest subarray is

int index = 0;

for (int i = 0; i < rows; i++) {

if ((indexAndLength[i][0] != 0) && (indexAndLength[i][0] >= indexAndLength[index][0])) {

index = i;

}

}

int length = indexAndLength[index][0];

int subarray[2][length];

int tempArr[length];

for (int i = 0; i < length; i++) {

subarray[0][i] = combinations[index][i];

tempArr[i] = combinations[index][i];

}

heapSort(tempArr, length);

for (int i = 0; i < length; i++) {

subarray[1][i] = tempArr[length-i-1];

}

printf("Matrix with resorted rows:\n");

for (int row = 0; row < rows; row++) {

for (int col = 0; col < size; col++) {

if (row != index) {

printf("%d\t", arr[row][col]);

} else {

bool check = true;

for (int i = 0; i < length; i++) {

if (arr[row][col] == subarray[0][i]) {

printf("%d\t", subarray[1][i]);

check = false;

break;

}

}

if (check) {

printf("%d\t", arr[row][col]);

}

}

}

printf("\n");

}

}

return 0;

}

**-----------------------------------Modified version--------------------------------------**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#include <math.h>

int findMax(int arr[], int n)

{

int mx = arr[0];

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

if (arr[i] > mx)

mx = arr[i];

return mx;

}

int findDigit(int num)

{

int cNum = num;

int count = 0;

while(cNum>0)

{

cNum /= 10;

count++;

}

return count;

}

int getDigit(int num, int pos)

{

return abs(num) / (int)pow(10, pos) % 10;

}

void radixSort(int arr[], int size)

{

// Separate the array into positive and negative parts

int \*positiveArr = (int \*)malloc(size \* sizeof(int));

int \*negativeArr = (int \*)malloc(size \* sizeof(int));

int posCount = 0, negCount = 0;

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

{

if (arr[i] < 0)

negativeArr[negCount++] = -arr[i];

else

positiveArr[posCount++] = arr[i];

}

// Sort negative numbers descendingly using radix sort

if (negCount > 0)

{

int negDigits = findDigit(findMax(negativeArr, negCount));

for (int j = 0; j < negDigits; j++)

{

int count[10] = {0};

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

{

int digit = getDigit(negativeArr[i], j);

count[digit]++;

}

for (int k = 1; k < 10; k++)

count[k] += count[k - 1];

int \*output = (int \*)malloc(negCount \* sizeof(int));

for (int i = negCount - 1; i >= 0; i--)

{

int digit = getDigit(negativeArr[i], j);

output[count[digit] - 1] = negativeArr[i];

count[digit]--;

}

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

negativeArr[i] = output[i];

free(output);

}

}

// Sort positive numbers ascendingly using radix sort

if (posCount > 0) {

int posDigits = findDigit(findMax(positiveArr, posCount));

for (int j = 0; j < posDigits; j++) {

int count[10] = {0};

for (int i = 0; i < posCount; i++) {

int digit = getDigit(positiveArr[i], j);

count[digit]++;

}

for (int k = 1; k < 10; k++)

count[k] += count[k - 1];

int \*output = (int \*)malloc(posCount \* sizeof(int));

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

int digit = getDigit(positiveArr[i], j);

output[count[digit] - 1] = positiveArr[i];

count[digit]--;

}

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

positiveArr[i] = output[i];

free(output);

}

}

// Combine the positive and negative parts

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

arr[i] = -negativeArr[negCount - i - 1];

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

arr[i + negCount] = positiveArr[i];

// Free allocated memory

free(negativeArr);

free(positiveArr);

}

int newgap(int gap) {

gap = (gap \* 10) / 13;

if (gap == 9 || gap == 10)

gap = 11;

if (gap < 1)

gap = 1;

return gap;

}

void combSort(int a[], int aSize) {

int gap = aSize;

int temp, i;

for (;;)

{

gap = newgap(gap);

int swapped = 0;

for (i = 0; i < aSize - gap; i++)

{

int j = i + gap;

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

{

temp = a[i];

a[i] = a[j];

a[j] = temp;

swapped = 1;

}

}

if (gap == 1 && !swapped)

break;

}

}

void findCombinationsRecursive(int \*arr, int size, int row, int rows, int targetSum, int \*\*result, int \*resultSize, int currentCombination[size], int currentSize, int startIndex) {

if (currentSize > 0) {

int sum = 0;

for (int i = 0; i < currentSize; ++i) {

sum += currentCombination[i];

}

if (sum == targetSum) {

// If the sum is equal to the target, add the combination to the result array

result[\*resultSize][0] = currentSize;

for (int i = 0; i < currentSize; ++i) {

result[\*resultSize][i + 1] = currentCombination[i];

}

\*resultSize += 1;

}

}

// Continue building the combination

for (int i = startIndex; i < size; ++i) {

currentCombination[currentSize] = arr[i];

findCombinationsRecursive(arr, size, row, rows, targetSum, result, resultSize, currentCombination, currentSize + 1, i + 1);

// If a combination is found, stop further recursion

if (\*resultSize > 0) {

return;

}

}

}

void findCombinations(int \*arr, int size, int row, int rows, int targetSum, int \*\*result, int \*resultSize) {

\*resultSize = 0;

int currentCombination[size];

findCombinationsRecursive(arr, size, row, rows, targetSum, result, resultSize, currentCombination, 0, 0);

}

int main() {

int rows, size;

printf("Enter the number of rows: ");

scanf("%d", &rows);

printf("Enter the number of columns: ");

scanf("%d", &size);

int \*\*arr = (int \*\*)malloc(rows \* sizeof(int \*));

for (int i = 0; i < rows; ++i) {

arr[i] = (int \*)malloc(size \* sizeof(int));

}

// input array

printf("Enter elements for the %dx%d array:\n", rows, size);

for (int i = 0; i < rows; ++i) {

for (int j = 0; j < size; ++j) {

printf("Enter element at position [%d][%d]: ", i, j);

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

}

}

// display entered array

printf("\nEntered %dx%d array:\n", rows, size);

for (int i = 0; i < rows; ++i) {

for (int j = 0; j < size; ++j) {

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

}

printf("\n");

}

int targetSum;

printf("\nEnter the sum of each subarray: ");

scanf("%d", &targetSum);

int \*\*result = (int \*\*)malloc(rows \* sizeof(int \*));

for (int i = 0; i < rows; ++i) {

result[i] = (int \*)malloc(size \* sizeof(int));

}

int \*\*combinations = (int \*\*)malloc(rows \* sizeof(int \*));

for (int i = 0; i < rows; ++i) {

combinations[i] = (int \*)malloc(size \* sizeof(int));

}

for (int r = 0; r < rows; r++) {

for (int j = 0; j < size; j++) {

combinations[r][j] = 0;

}

}

int nrElementsInSubarrays = 0;

for (int r = 0; r < rows; r++) {

int resultSize = 0;

findCombinations(arr[r], size, r, rows, targetSum, result, &resultSize);

if (resultSize > 0) {

for (int i = 0; i < result[0][0]; i++) {

combinations[r][i] = result[0][i + 1];

nrElementsInSubarrays++;

}

}

}

printf("\nSubarrays:\n");

for (int r = 0; r < rows; r++) {

printf("Row %d: ", r+1);

bool noCombinations = true;

for (int j = 0; j < size; j++) {

if (combinations[r][j] != 0) {

printf("%d ", combinations[r][j]);

noCombinations = false;

}

}

if (noCombinations) {

printf("No elements in this subarray");

}

printf("\n");

}

printf("\nThe total number of elements forming given subarrays: %d\n\n", nrElementsInSubarrays);

if (nrElementsInSubarrays % 2 == 0) { // if nrElementsInSubarrays is even

// comb sort ascendingly the shortest subarray

int indexAndLength[rows][1];

for (int i = 0; i < rows; i++) {

indexAndLength[i][0] = 0;

}

// record the length of each subarray

for (int i = 0; i < rows; i++) {

int length = 0;

for (int j = 0; j < size; j++) {

if (combinations[i][j] != 0) {

length++;

}

}

indexAndLength[i][0] = length;

}

// find the index of the row where the shortest subarray is

int index = 0;

for (int i = 0; i < rows; i++) {

if ((indexAndLength[i][0] != 0) && (indexAndLength[i][0] <= indexAndLength[index][0])) {

index = i;

}

}

int length = indexAndLength[index][0];

int subarray[2][length];

for (int i = 0; i < length; i++) {

subarray[0][i] = combinations[index][i];

subarray[1][i] = combinations[index][i];

}

combSort(subarray[1], length);

printf("Matrix with resorted rows:\n");

for (int row = 0; row < rows; row++) {

for (int col = 0; col < size; col++) {

if (row != index) {

printf("%d\t", arr[row][col]);

} else {

bool check = true;

for (int i = 0; i < length; i++) {

if (arr[row][col] == subarray[0][i]) {

printf("%d\t", subarray[1][i]);

check = false;

break;

}

}

if (check) {

printf("%d\t", arr[row][col]);

}

}

}

printf("\n");

}

} else { // if nrElementsInSubarrays is odd:

// radix sort descendingly the longest subarray

int indexAndLength[rows][1];

for (int i = 0; i < rows; i++) {

indexAndLength[i][0] = 0;

}

// record the length of each subarray

for (int i = 0; i < rows; i++) {

int length = 0;

for (int j = 0; j < size; j++) {

if (combinations[i][j] != 0) {

length++;

}

}

indexAndLength[i][0] = length;

}

// find the index of the row where the shortest subarray is

int index = 0;

for (int i = 0; i < rows; i++) {

if ((indexAndLength[i][0] != 0) && (indexAndLength[i][0] >= indexAndLength[index][0])) {

index = i;

}

}

int length = indexAndLength[index][0];

int subarray[2][length];

int tempArr[length];

for (int i = 0; i < length; i++) {

subarray[0][i] = combinations[index][i];

tempArr[i] = combinations[index][i];

}

radixSort(tempArr, length);

for (int i = 0; i < length; i++) {

subarray[1][i] = tempArr[length-i-1];

}

printf("Matrix with resorted rows:\n");

for (int row = 0; row < rows; row++) {

for (int col = 0; col < size; col++) {

if (row != index) {

printf("%d\t", arr[row][col]);

} else {

bool check = true;

for (int i = 0; i < length; i++) {

if (arr[row][col] == subarray[0][i]) {

printf("%d\t", subarray[1][i]);

check = false;

break;

}

}

if (check) {

printf("%d\t", arr[row][col]);

}

}

}

printf("\n");

}

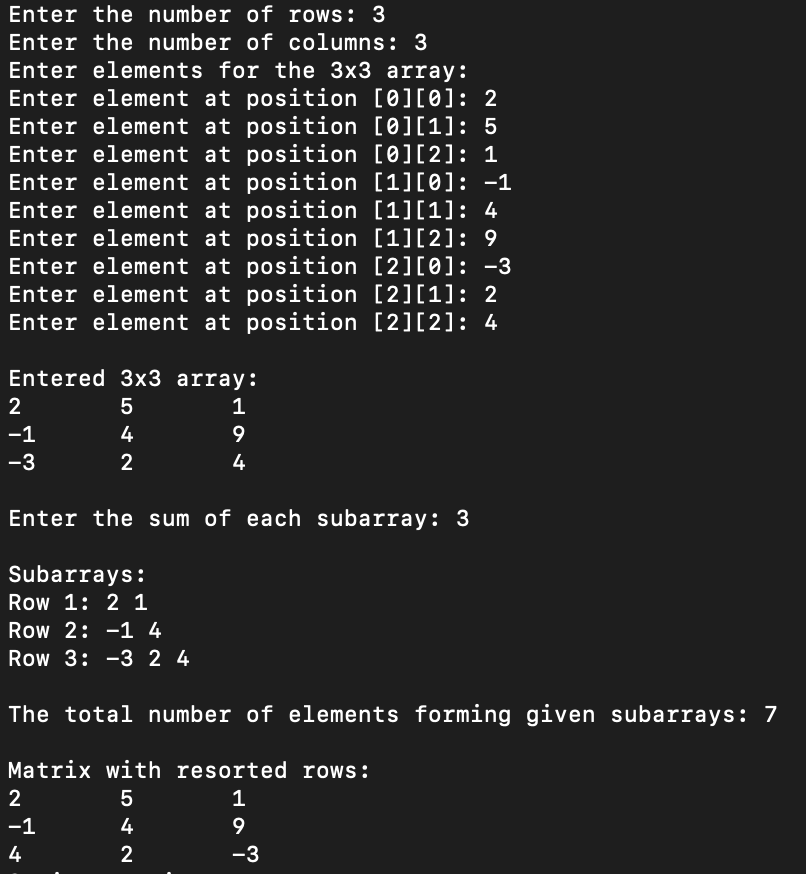
}

return 0;

}

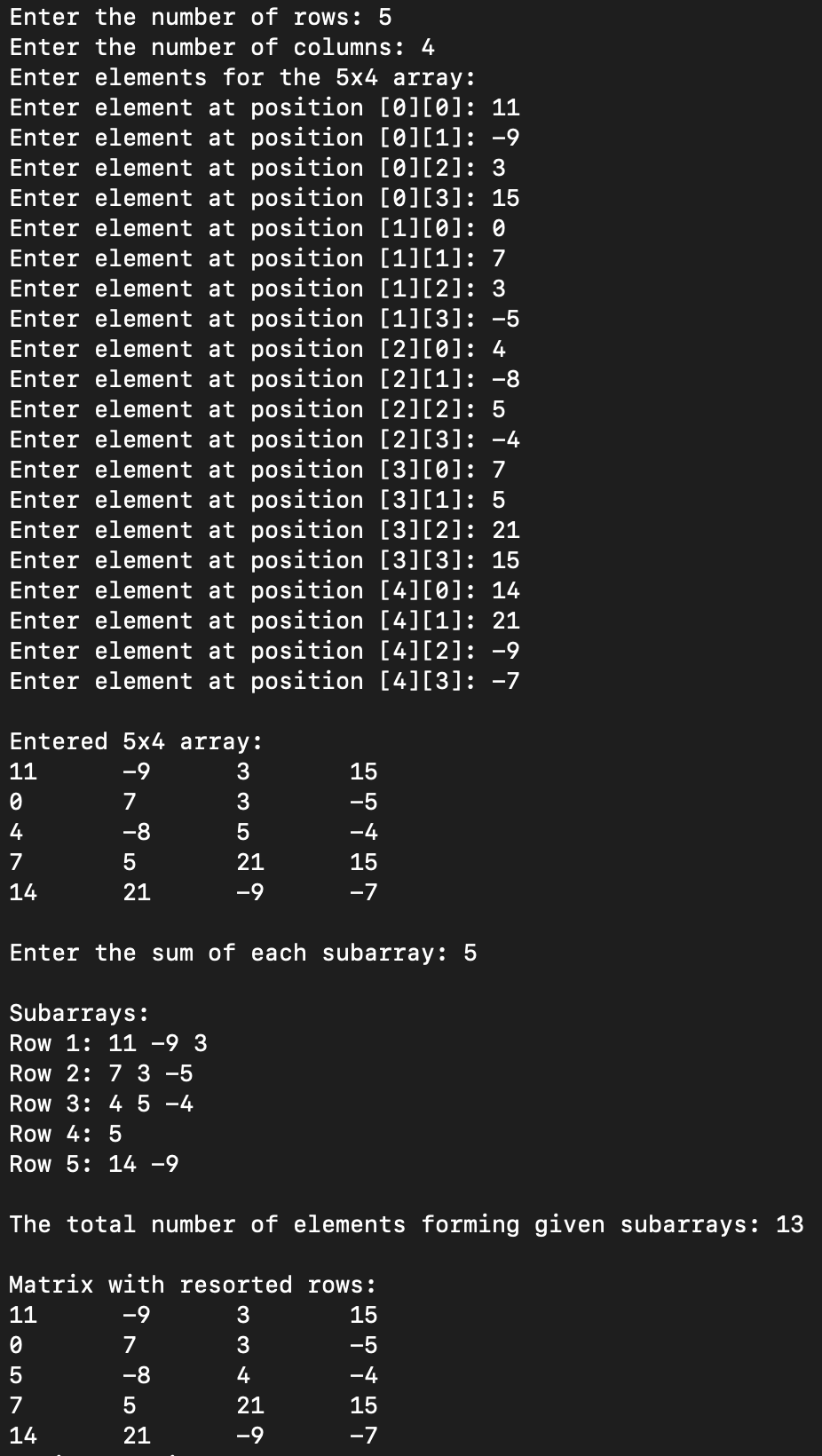
In this version I used the comb sort and the radix sort

**Output first version:**



**Fig 1**

**Output modified version:**

****

**Fig 2**

**Conclusion:**

During the laboratory work, I applied the knowledge acquired in courses and seminars, employing various sorting methods to arrange a vector. Starting with simpler ones like Counting Sort and Heap Sort, I progressed to more complex ones like Comb and Radix Sort. Overall, considering the work spans over 20 pages, I consider this a good starting point.

As a result of the project, I gained skills in working with custom functions in the C language and structures. I incorporated pointers within structures and implemented multiple sorting methods. I discovered the pros and cons of pointers, particularly in the realm of memory allocation.