****

**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**

**Țîganescu Anastasia**

**Group: FAF-231**

**Report**

**Laboratory Work No.2**

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

Checked:

**Burlacu Natalia,** *PhD, associate professor*

Department of Software and Automation Engineering,

FCIM Faculty, UTM

**Chisinau – 2024**

**Purpose of laboratory work:**

Solving problems with 1-D arrays. It aims to strengthen programming skills

by working with 1-D arrays, array manipulation, algoritmic thinking and

problem-solving.

**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:

* 1. with the use of the method of transmitting the parametric functions by value;
  2. 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).
  3. 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

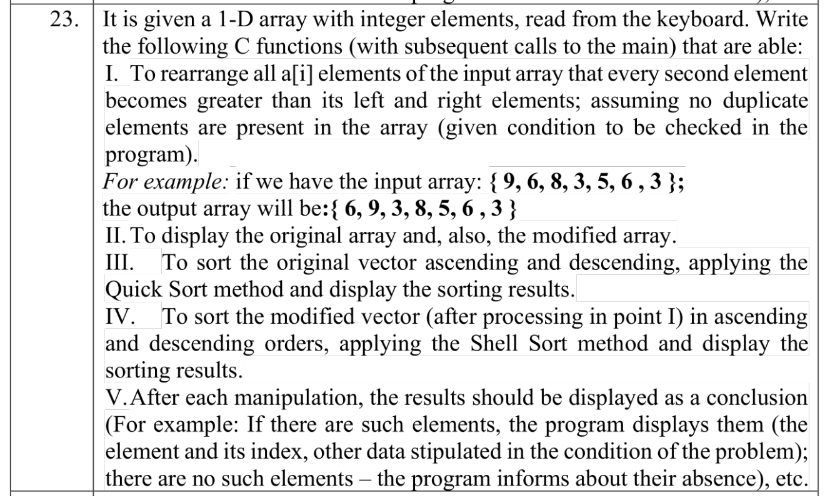
existing problem. Formulate and present in writing the modified condition; to

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 Counting Sort & Merge Sort.

**Condition of the problem:**



**Code of the program (having relevant comments in it):**

1. Version with the use of the method of transmitting the function parameters by value:

#include <stdio.h>

//the function which checks whether there are duplicate elements or not in the array

*int* check(*int* *A*[], *int* *n*){

*int* status = 1;

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

*int* check=*A*[i];

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

if (*A*[j] ==check){

status=0;

return status;

}

}

}

return status;

}

//the function which rearranges the elements of the array according to the condition:

//so that every second element is bigger than its neighbors

*void* rearrange(*int* *arr*[], *int* *n*){

*int* small[10], big[10], max,k,j,i, maxind;

*int* maxelements=*n*/2; //calculates the number second-positions in the array

//this block of code finds the maximum element of the array

j = 0;

max = *arr*[0];

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

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

max = *arr*[i];

maxind=i;

}

}

//here, in the array "big", the biggest elements of the array are stored

//the number of the biggest elements is equal to the maxelements variable

big[j]=max; //the first element of the "big" array is the maximum element of the original array

//the loop finds the other biggest elements and fills in the "big" array

for(j=1;j<maxelements;j++){

i=0;

max = *arr*[0];

while(max>=big[j-1]){

i++;

max=*arr*[i];

}

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

if(*arr*[i]>max && *arr*[i]<big[j-1]){

max = *arr*[i];

maxind=i;

}

}

big[j]=max;

}

*int* jbig=j; //jbig variable represents the number of elements of the "big" array

//here the "small" array is initialized, it contains all the other elements of the

//array, the ones smaller than the elements in the "big" array

j=0;

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

if(*arr*[i]<big[jbig-1]){

small[j]=*arr*[i];

j++;

}

}

*int* jsmall=j; //jsmall variable represents the number of elements of the "small" array

//finally, the array is updated: at every second position an element of the "big" array is inserted

j=0;

for(i=1;i<*n*;i+=2){

*arr*[i]=big[j];

j++;}

//at all the other positions left, an element of the "small" array is inserted

j=0;

for(i=0;i<*n*;i+=2){

*arr*[i]=small[j];

j++;}

}

//function which prints the array

*void* display(*int* *a*[], *int* *n*){

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

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

}

printf("\n");

}

//function which swaps values

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

*int* t=\**a*;

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

\**b*=t;

}

//function essential for the quick sort method, it specifies the pivot is at the end and

//traverses the array, compares values with the pivot and makes necessary swaps

*int* partition(*int* *arr*[], *int* *start*, *int* *end*, *int* *type*){

*int* pivot = *arr*[*end*];

*int* i=*start*-1;

*int* j,t;

if (*type*==1){ //type = 1 so the sorting will be ascending

for(j=*start*;j<*end*;j++){

if(*arr*[j]<=pivot){

i++;

swap(*arr*+i, *arr*+j);

}

}

swap(*arr*+*end*, *arr*+i+1);

return i+1;

}

else{ //type = 0 so the sorting will be descending

for(j=*start*;j<*end*;j++){

if(*arr*[j]>=pivot){

i++;

swap(*arr*+i, *arr*+j);

}

}

swap(*arr*+*end*, *arr*+i+1);

return i+1;

}

}

//function essential for the quick sort method, partitions the array according to the pivot and

//recursively calls the quicksort function until the array is fully sorted

*void* quicksort(*int* *arr*[], *int* *start*, *int* *end*, *int* *type*){

if(*start*<*end*){

*int* pivot = partition(*arr*,*start*, *end*, *type*);

quicksort(*arr*,*start*, pivot-1, *type*);

quicksort(*arr*, pivot+1, *end*, *type*);

}

}

//function which performs shellsort

*void* shellsort(*int* *arr*[], *int* *n*, *int* *type*){

*int* t,i,j;

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

for( i=interval;i<*n*;i++){

t = *arr*[i];

if (*type*==1){ //type = 1 so the sorting will be ascending

for(j=i;j>=interval && *arr*[j-interval]>t;j-=interval){

*arr*[j]=*arr*[j-interval];

}}

else{ //type = 0 so the sorting will be descending

for(j=i;j>=interval && *arr*[j-interval]<t;j-=interval){

*arr*[j]=*arr*[j-interval];

}

}

*arr*[j]=t;

}

}

}

*int* main(){

//varianta 23

*int* A[50], n,i,j, status, copy[50];

status = 0;

//a loop which doesn't allow the user to proceed unless

//the array has distinct elements

while(status==0){

printf("Enter n:"); scanf("%d", &n); //reading the number of elements of the vector

for(i=0;i<n;i++){ //reading the vector elements

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

}

status = check(A,n); //here the condition which guides the iteration is checked

if (status == 1){

printf("There are no duplicate elements.");

}else{

printf("There are duplicate elements. Read elements again.");

}

}

for(i=0;i<n;i++){ //creating a copy of the original vector A

printf("%d ", A[i]);

copy[i]=A[i];

}

printf("\n");

//Task 1: the rearranging of the elements of the vector

rearrange(copy,n);

//Task 2: displaying the original array and the modified array

printf("Original array:");

display(A,n);

printf("Modified array:");

display(copy,n);

//Task 3: sorting (both ascendingly and descendingly) using Quick Sort and displaying the original vector

printf("Ascending sorting of original vector (QuickSort):");

*int* type = 1; //this variable indicates whether the sort will be ascending(1) or descending(0)

quicksort(A,0,n-1, type);

display(A,n);

printf("\n");

printf("Descending sorting of original vector (QuickSort):");

type = 0; //indicates the descending sort

quicksort(A,0,n-1, type);

display(A,n);

printf("\n");

//Task 4: sorting (both ascendingly and descendingly) using Shell Sort and displaying the modified vector

printf("Ascending sorting of modified vector (ShellSort):");

type = 1;

shellsort(copy,n, type);

display(copy,n);

printf("\n");

printf("Descending sorting of modified vector (ShellSort):");

type = 0;

shellsort(copy,n, type);

display(copy,n);

printf("\n");

return 0;

}

2. Version with the use of the method of passing the function parameters by pointers.

#include <stdio.h>

//the function which checks whether there are duplicate elements or not in the array

*int* check(*int* \**A*, *int* *n*){

*int* status = 1;

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

*int* check=\*(*A*+i);

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

if (\*(*A*+j) ==check){

status=0;

return status;

}

}

}

return status;

}

//the function which rearranges the elements of the array according to the condition:

//so that every second element is bigger than its neighbors

*void* rearrange(*int* \**arr*, *int* *n*){

*int* small[10], big[10], max,k,j,i, maxind;

*int* maxelements=*n*/2; //calculates the number second-positions in the array

//this block of code finds the maximum element of the array

j = 0;

max = \*(*arr*+0);

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

if (\*(*arr*+i)>max){

max = \*(*arr*+i);

maxind=i;

}

}

//here, in the array "big", the biggest elements of the array are stored

//the number of the biggest elements is equal to the maxelements variable

big[j]=max; //the first element of the "big" array is the maximum element of the original array

//the loop finds the other biggest elements and fills in the "big" array

for(j=1;j<maxelements;j++){

i=0;

max = \*(*arr*+0);

while(max>=big[j-1]){

i++;

max=\*(*arr*+i);

}

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

if(\*(*arr*+i)>max && \*(*arr*+i)<big[j-1]){

max = \*(*arr*+i);

maxind=i;

}

}

big[j]=max;

}

*int* jbig=j; //jbig variable represents the number of elements of the "big" array

//here the "small" array is initialized, it contains all the other elements of the

//array, the ones smaller than the elements in the "big" array

j=0;

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

if(\*(*arr*+i)<big[jbig-1]){

small[j]=\*(*arr*+i);

j++;

}

}

*int* jsmall=j; //jsmall variable represents the number of elements of the "small" array

//finally, the array is updated: at every second position an element of the "big" array is inserted

j=0;

for(i=1;i<*n*;i+=2){

\*(*arr*+i)=big[j];

j++;}

//at all the other positions left, an element of the "small" array is inserted

j=0;

for(i=0;i<*n*;i+=2){

\*(*arr*+i)=small[j];

j++;}

}

//function which prints the array

*void* display(*int* \**a*, *int* *n*){

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

printf("%d ", \*(*a*+i));

}

printf("\n");

}

//function which swaps values

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

*int* t=\**a*;

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

\**b*=t;

}

//function essential for the quick sort method, it specifies the pivot is at the end and

//traverses the array, compares values with the pivot and makes necessary swaps

*int* partition(*int* \**arr*, *int* *start*, *int* *end*, *int* *type*){

*int* pivot = \*(*arr*+*end*);

*int* i=*start*-1;

*int* j,t;

if (*type*==1){ //type = 1 so the sorting will be ascending

for(j=*start*;j<*end*;j++){

if(\*(*arr*+j)<=pivot){

i++;

swap(*arr*+i, *arr*+j);

}

}

swap(*arr*+*end*, *arr*+i+1);

return i+1;

}

else{ //type = 0 so the sorting will be descending

for(j=*start*;j<*end*;j++){

if(\*(*arr*+j)>=pivot){

i++;

swap(*arr*+i, *arr*+j);

}

}

swap(*arr*+*end*, *arr*+i+1);

return i+1;

}

}

//function essential for the quick sort method, partitions the array according to the pivot and

//recursively calls the quicksort function until the array is fully sorted

*void* quicksort(*int* \**arr*, *int* *start*, *int* *end*, *int* *type*){

if(*start*<*end*){

*int* pivot = partition(*arr*,*start*, *end*, *type*);

quicksort(*arr*,*start*, pivot-1, *type*);

quicksort(*arr*, pivot+1, *end*, *type*);

}

}

//function which performs shellsort

*void* shellsort(*int* \**arr*, *int* *n*, *int* *type*){

*int* t,i,j;

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

for( i=interval;i<*n*;i++){

t = \*(*arr*+i);

if (*type*==1){

for(j=i;j>=interval && \*(*arr*+j-interval)>t;j-=interval){

\*(*arr*+j)=\*(*arr*+j-interval);

}}

else{

for(j=i;j>=interval && \*(*arr*+j-interval)<t;j-=interval){

\*(*arr*+j)=\*(*arr*+j-interval);

}

}

\*(*arr*+j)=t;

}

}

}

*int* main(){

//varianta 23

*int* A[50], n,i,j, status, copy[50];

status = 0;

//a loop which doesn't allow the user to proceed unless

//the array has distinct elements

while(status==0){

printf("Enter n:"); scanf("%d", &n); //reading the number of elements of the vector

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

scanf("%d", A+i); //reading the vector elements

}

status = check(A,n); //here the condition which guides the iteration is checked

if (status == 1){

printf("There are no duplicate elements.");

}else{

printf("There are duplicate elements. Read elements again.");

}

}

for(i=0;i<n;i++){ //creating a copy of the original vector A

printf("%d ", A[i]);

copy[i]=A[i];

}

printf("\n");

//Task 1: the rearranging of the elements of the vector

rearrange(copy,n);

//Task 2: displaying the original array and the modified array

printf("Original array:");

display(A,n);

printf("Modified array:");

display(copy,n);

//Task 3: sorting (both ascendingly and descendingly) using Quick Sort and displaying the original vector

printf("Ascending sorting of original vector (QuickSort):");

*int* type = 1; //this variable indicates whether the sort will be ascending(1) or descending(0)

quicksort(A,0,n-1, type);

display(A,n);

printf("\n");

printf("Descending sorting of original vector (QuickSort):");

type = 0; //indicates the descending sort

quicksort(A,0,n-1, type);

display(A,n);

printf("\n");

//Task 4: sorting (both ascendingly and descendingly) using Shell Sort and displaying the modified vector

printf("Ascending sorting of modified vector:");

type = 1;

shellsort(copy,n, type);

display(copy,n);

printf("\n");

printf("Descending sorting of modified vector:");

type = 0;

shellsort(copy,n, type);

display(copy,n);

printf("\n");

return 0;

}

3. Modified version

The condition for my new version was to check whether every 3 elements of the vector (the first, second and the third; the second, third and fourth, and so on) can represent lengths of a triangle. If the condition was met (if an element and its two neighbors can form a triangle), then the elements takes value 1 (and 0 if it is vice versa). Next I had to sort the original vector both ascendingly and descendingly, using Counting sort and then the results vector (both ascendingly and descendingly) using Merge Sort.

#include <stdio.h>

//function which checks whether all the elements of the vector are positive

*int* checkpositive(*int* *arr*[], *int* *n*){

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

if(*arr*[i]<=0){

return 0;

}

}

return 1;

}

//function which checks the triangle inequality

*int* condition(*int* *a*, *int* *b*, *int* *c*){

if (*a*+*b*>*c* && *a*+*c*>*b* && *b*+*c*>*a*){

return 1;

}

else{

return 0;

}

}

//function which checks if every 3 consecutive elements of the vector can represent lengths of a triangle

//the "condition" function is called here too

*void* checktriangles(*int* *arr*[],*int* *results*[], *int* *n*){

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

if (i==0 || i==*n*-1){

*results*[i]=0;

} else if (condition(*arr*[i-1], *arr*[i], *arr*[i+1])==1){

*results*[i]=1;

}

else{

*results*[i]=0;

}

}

}

//function which prints the array

*void* display(*int* *a*[], *int* *n*){

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

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

}

printf("\n");

}

//function which performs the counting sort of an array

*void* counting(*int* *arr*[], *int* *n*){

*int* i,max,count[50], final[50];

max=*arr*[0];

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

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

max=*arr*[i];

}

}

for(i=0;i<max+1;i++){

count[i]=0;

}

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

count[*arr*[i]]++;

}

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

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

}

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

final[count[*arr*[i]]-1]=*arr*[i];

count[*arr*[i]]--;

}

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

*arr*[i] = final[i];

}

}

//function which recursively divides the initial array into smaller subarrays so that it can be sorted using Merge Sort

*void* dividemerge(*int* *arr*[], *int* *a*, *int* *b*, *int* *type*){

*int* m;

if (*a*<*b*){

m=(*a*+*b*)/2;

dividemerge(*arr*, *a*, m, *type*);

dividemerge(*arr*,m+1,*b*, *type*);

mergesort(*arr*, *a*,m,*b*, *type*);

}

}

//function essential for the Merge Sort

*void* mergesort(*int* *arr*[], *int* *a*, *int* *m*, *int* *b*, *int* *type*){

*int* n1,n2,i,j,k;

n1=*m*-*a*+1;

n2=*b*-*m*;

*int* L[n1], R[n2];

for(i=0;i<n1;i++){

L[i]=*arr*[*a*+i];

}

for(j=0;j<n2;j++){

R[j]=*arr*[*m*+1+j];

}

i=0;j=0;k=*a*;

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

if(L[i]<R[j]){

if(*type*==1){

*arr*[k]=L[i];

i++;

}

else{

*arr*[k]=R[j];

j++;

}

}

else{

if(*type*==1){

*arr*[k]=R[j];

j++;

}

else{

*arr*[k]=L[i];

i++;

}

}

k++;

}

while(i<n1){

*arr*[k]=L[i];

k++;

i++;

}

while(j<n2){

*arr*[k]=R[j];

k++;

j++;

}

}

*int* main(){

//varianta 23 - modified version

*int* A[50], n,i,j, status, results[50];

status = 0;

//a loop which doesn't allow the user to proceed unless

//the array has positive elements

while(status==0){

printf("\nEnter n:"); scanf("%d", &n); //reading the number of elements of the vector

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

scanf("%d", &A[i]); //reading the vector elements

results[i]=A[i]; //creating a copy of the original vector A

}

status = checkpositive(A,n); //here the condition which guides the iteration is checked

if (status == 1){

printf("There are only positive elements.");

}else{

printf("There are nonpositive elements. Read elements again.");

}

}

//Task 1: the "checktriangles" function is called and the "results" array is filled

checktriangles(A, results,n);

//Task 2: displaying the original array and the modified array

printf("Original array:");

display(A,n);

printf("Modified array:");

display(results,n);

//Task 3: sorting the original vector using Counting Sort in an ascending manner

printf("Ascending sorting of original vector using counting sort:");

counting(A,n);

display(A,n);

//Task 4: Sorting the modified vector using Merge Sort, both ascendingly and descendingly

*int* type=1;

printf("Ascending sorting of modified vector using merge sort:");

dividemerge(results,0,n-1, type);

display(results,n);

type=0;

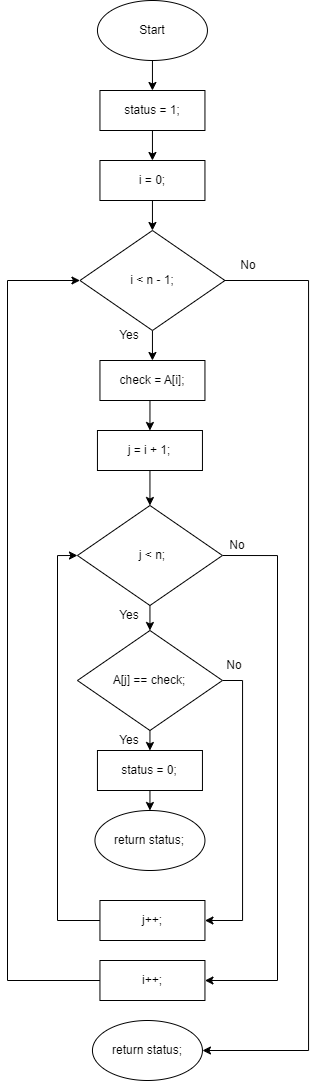
printf("Descending sorting of modified vector using merge sort:");

dividemerge(results, 0,n-1,type);

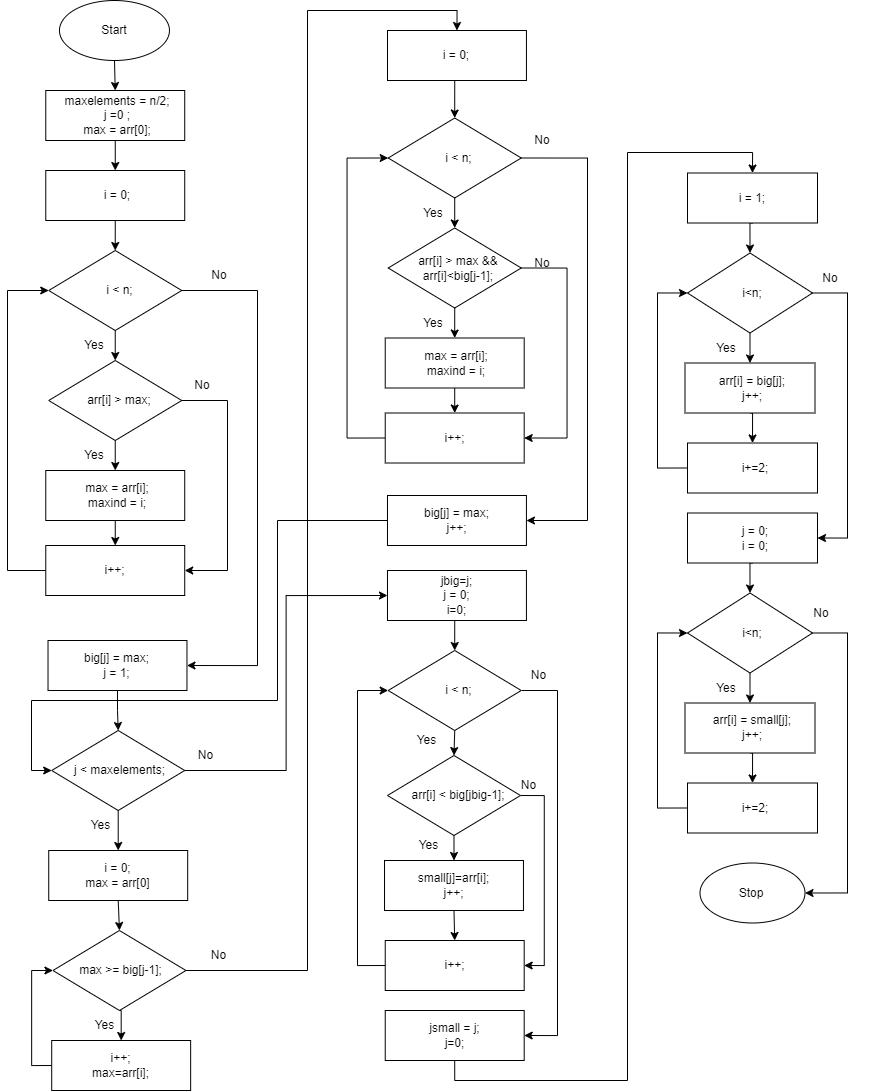
display(results, n);

}

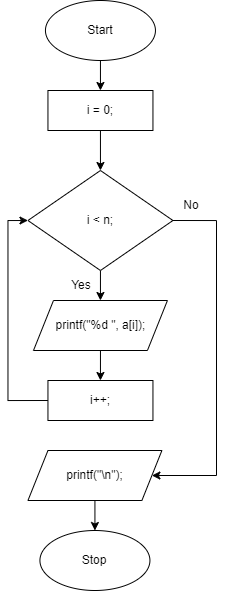
**Block diagrams (for the second version):**



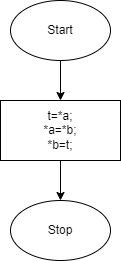
**Figure 1.1 -** *Function “check()”*



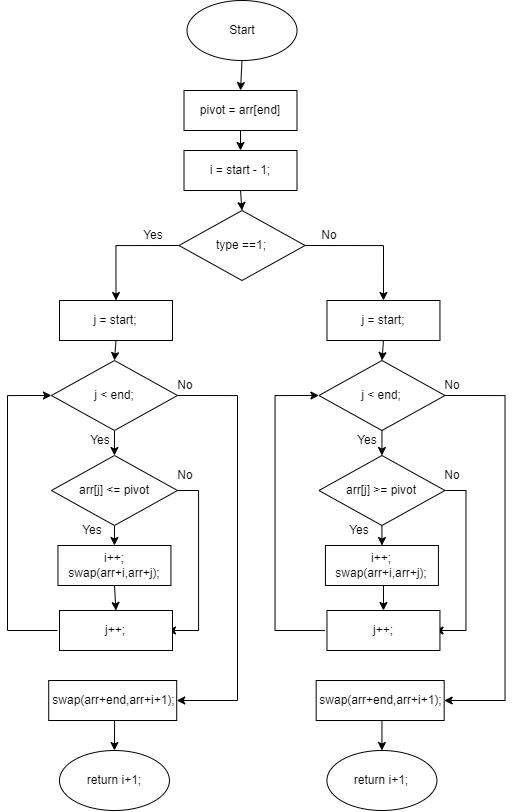
**Figure 1.2 -** *Function “rearrange()”*



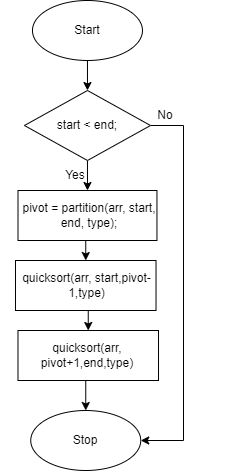
**Figure 1.3 -** *Function “display()”*



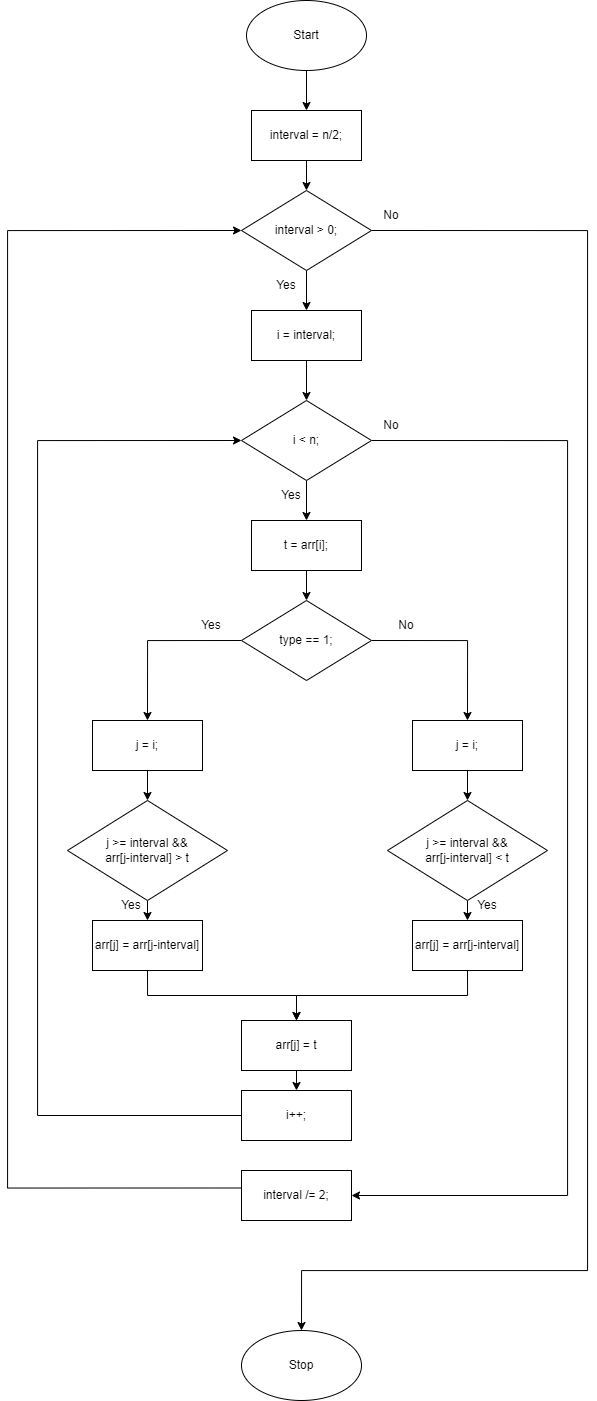
**Figure 1.4 -** *Function “swap()”*



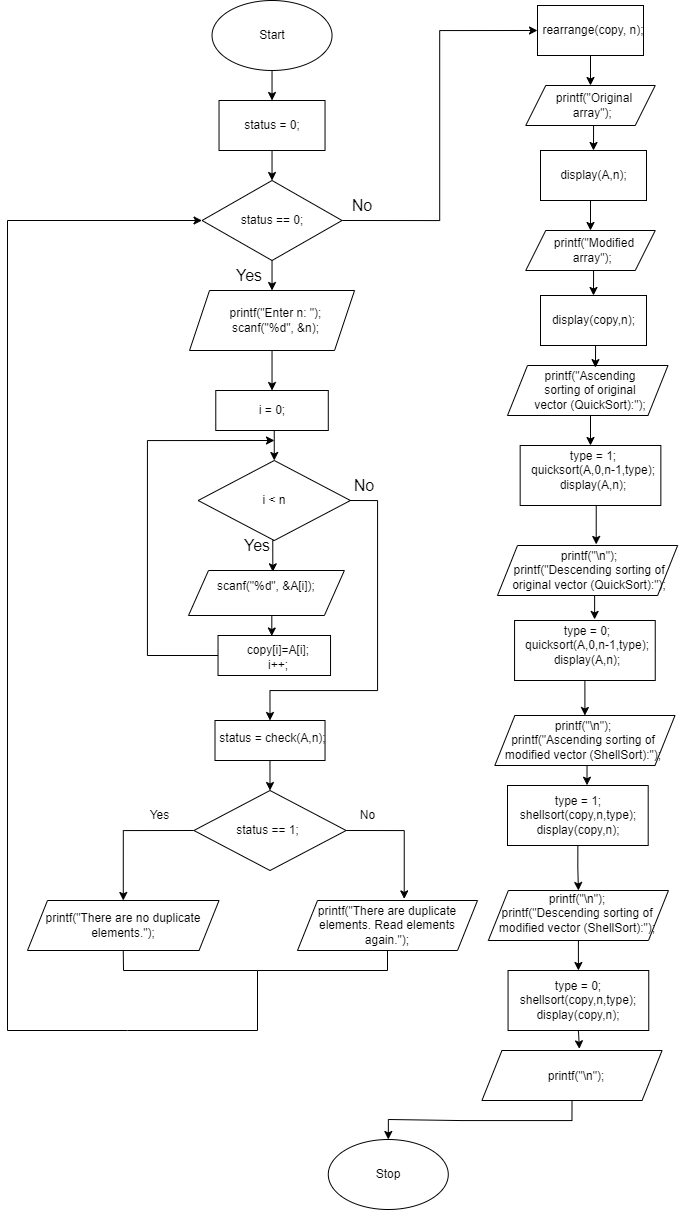
**Figure 1.5 -** *Function “partition()”*



**Figure 1.6 -** *Function “quicksort()”*

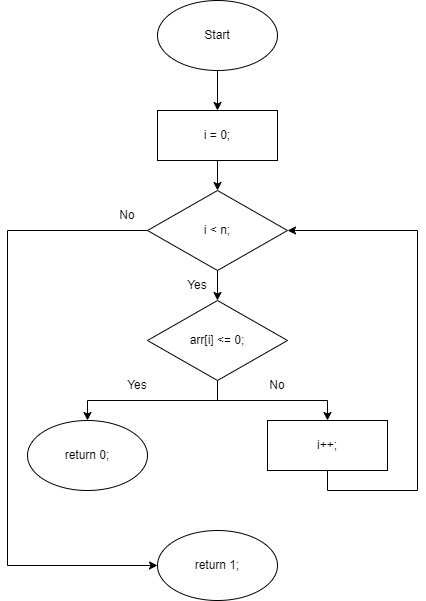


**Figure 1.7 -** *Function “shellsort()”*

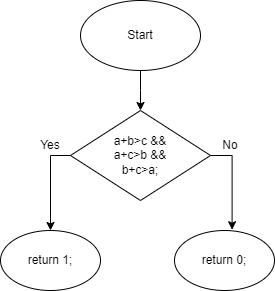


**Figure 1.8 -** *Function “main()”*

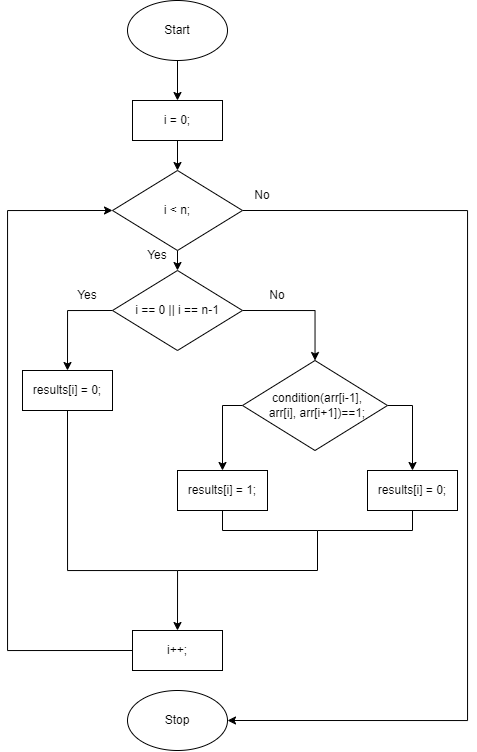
**Block diagrams (for the second version):**



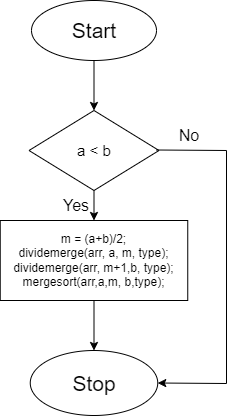
**Figure 2.1 -** *Function “checkpositive()”*



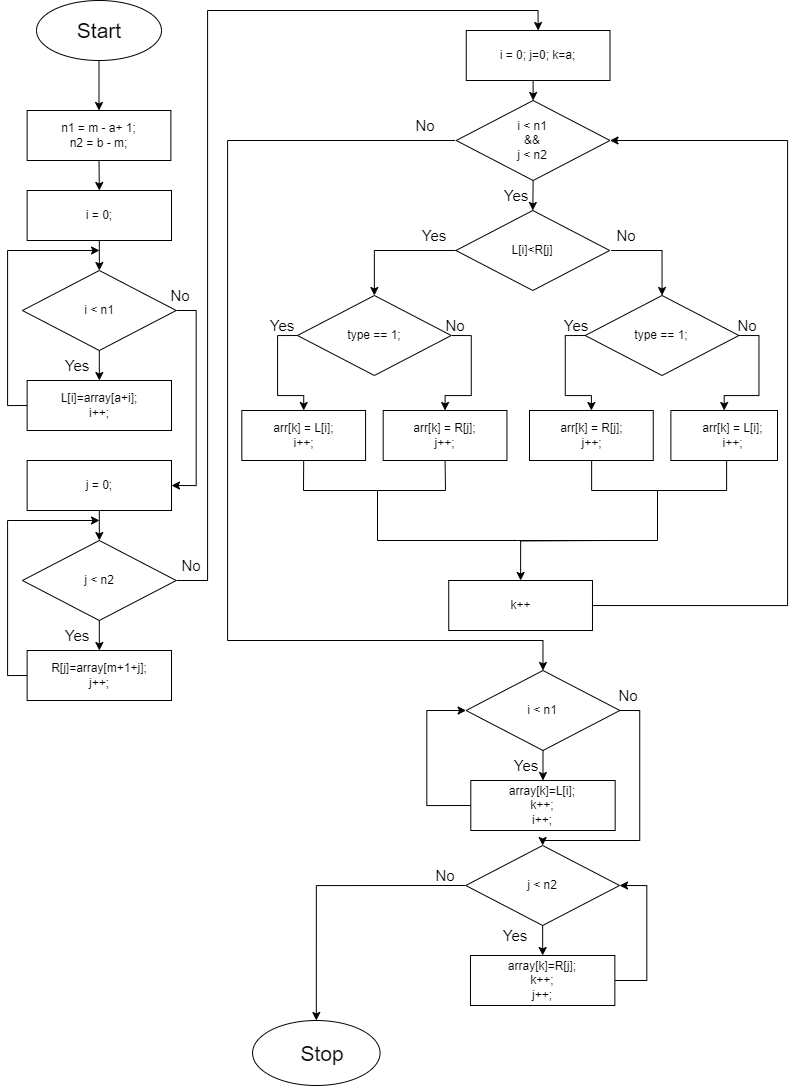
**Figure 2.2 -** *Function “condition()”*



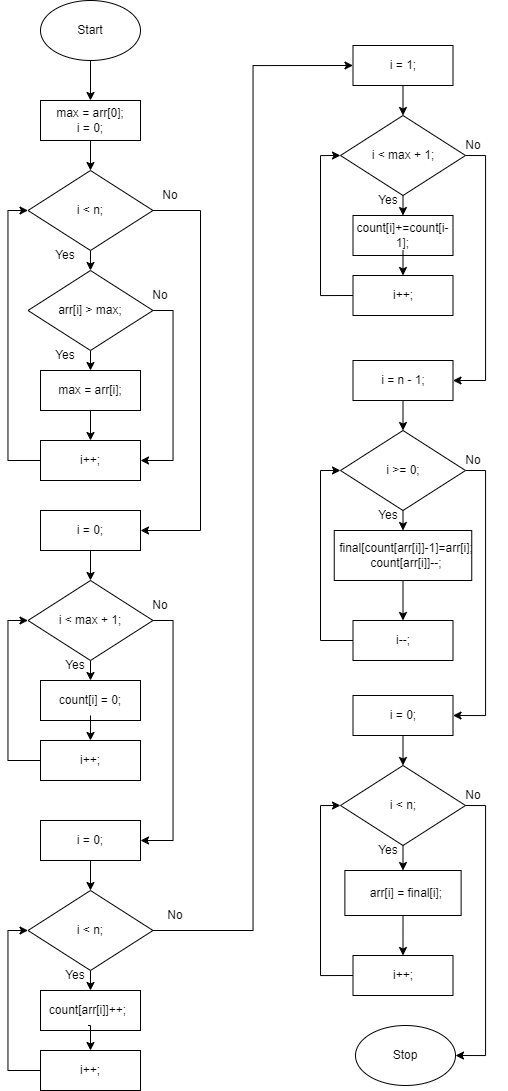
**Figure 2.3 -** *Function “checktriangles()”*



**Figure 2.4 -** *Function “dividemerge()”*

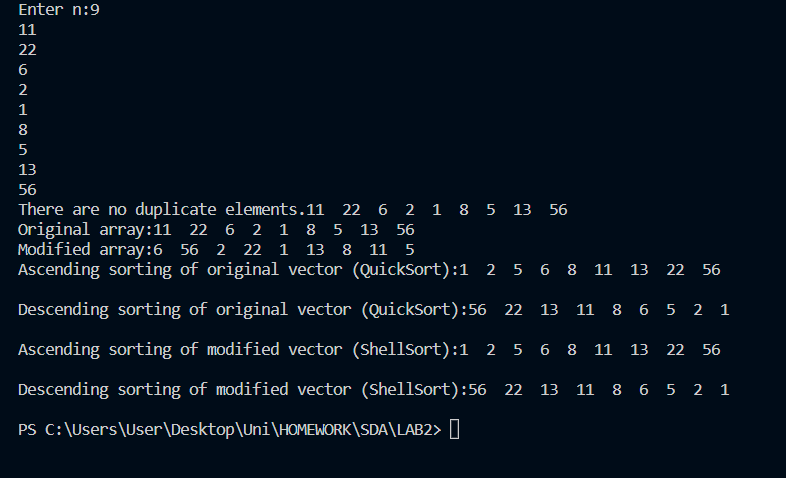


**Figure 2.5 -** *Function “mergesort()”*

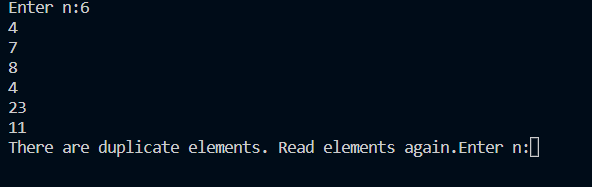


**Figure 2.6 -** *Function “counting()”*

**Output (first version):**

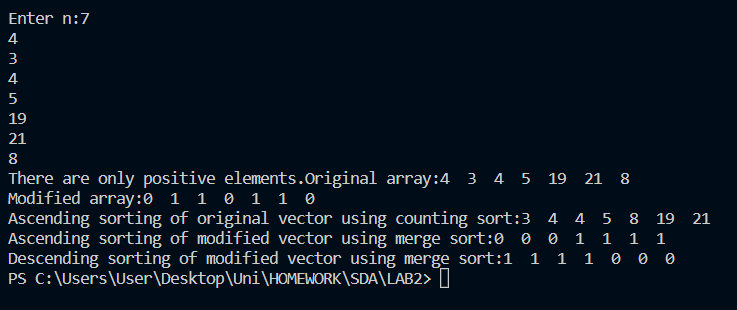


**Figure 3.1 -** *Output when all elements are distinct.*



**Figure 3.2 -** *Output when not all elements are distinct.*

**Output (modified version):**



**Figure 3.3 -** *Output.*

**Conclusion:**

In this laboratory work, I dealt with vectors and sorting algorithms. I had to rearrange a vector in a specific way and then use both Quick Sort and Shell Sort (ascendingly and descendingly). My modified version consisted of checking whether every 3 elements of the vector (the first, second and the third; the second, third and fourth, and so on) can represent lengths of a triangle. Then, I performed Counting sort and Merge Sort.

I managed to put to use the knowledge I gained during the lectures and the seminars and I was also provided with insights regarding efficiency of various sorting algorithms.

I took notice of how efficient all these algorithms are but, in spite of this, some are limited as the input size increases. So, I learned the importance of choosing sorting algorithms based on diverse scenarios and also learned to implement them both ascendingly and descendingly.