

**MINISTRY OF EDUCATION AND RESEARCH OF THE REPUBLIC OF MOLDOVA**

**Technical University of Moldova**

**Faculty of Computers, Informatics and Microelectronics**

**Department of Software and Automation Engineering**

**Caț Chiril**

**Report**

**Laboratory Work No.3**

*The variant No.4*

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

Checked:

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

Department of Software and Automation Engineering,

Facultatea FCIM, UTM

**Chisinau – 2025**

**The structure of the report for the laboratory work in the "Data Structures and Algorithms" course will contain:**

1. The purpose of the laboratory work (formulated by the student according to the problem to be solved);
2. For each task should be written the condition/conditions of the problems;
3. **The program code, having relevant comments in it will be present for each given task;**
4. For each task should be shown the screenshot of the code execution (in all aspects of the code run);
5. The student's conclusions regarding the content of the laboratory work with personal reflections on what was achieved.
6. *The name and surname of the student/teacher and no. the laboratory work should be modified according to didactical requirements.*

**Note:**

* The report pages should be numbered in the footer, center area;
* The text from items 1 & 2; 4 & 5 have to be written in Times New Roman, font size 14 pt;
* The space between the lines will be set at 1,5 lines.
* Item 3 of this list (the developed program code should be written in relation to Courier New, font size 10 pt; the space between code lines being 1.15 lines).
* The report should be uploaded for checking by the lab teacher in the right Report section (numbered in the same mode as your task) according to the deadline terms specified by your teacher.

**1.** **The purpose of the laboratory work**

The purpose of the laboratory work was to solve the following problems in C, writing my own functions according to the given statements.

Write the solution of the problem by procedural approach in two versions:

1A. with the use of the method of passing parameters of functions using standard formal parameters;

1B. where appropriate, formal/real parameters will be expressed either by the dereferencing operator (\*) or by the address operator (&).

------------------------------------------------------------------------------

**2. The problem statements and solutions**

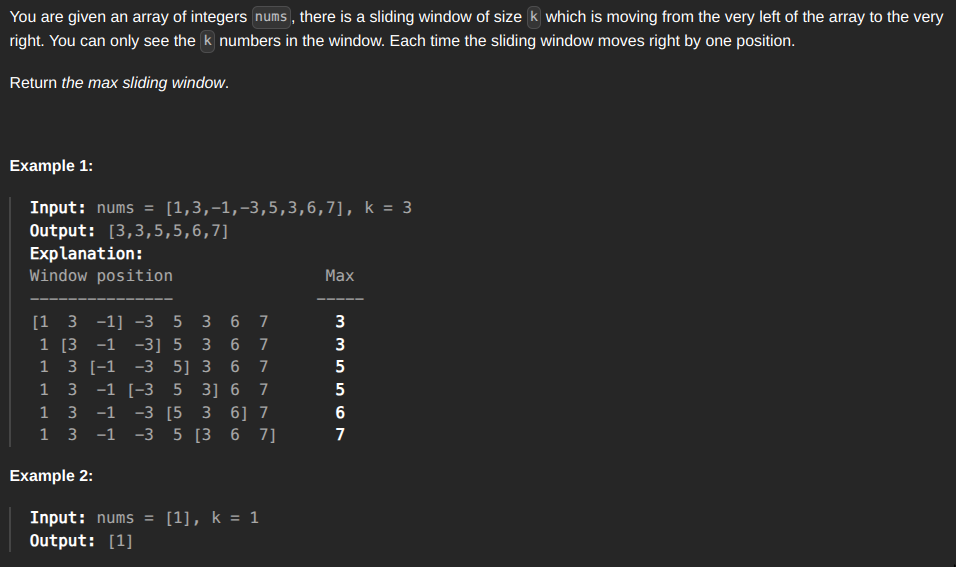
1. General Condition

2A. For de input array (-s) will be also perfomed the descending Merge Sort;

2B. For de output array (-s) will be also perfomed the descending Counting Sort. All results should be displayed.

2C. To bring clarity to the developed and run code for each functionality made into the presented solution, all results should be displayed after their manipulation.

2. Variant 4 Condition



------------------------ Passing arguments by value ---------------------------------  
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

void countingSort(int\* arr, size\_t n) {

int max = arr[0];

// find the maximum elelment

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

if(arr[i] > max) {

max = arr[i];

}

}

// create an array with the size of the maximum element

int\* count = (int\*)calloc(max + 1, sizeof(int));

if (count == NULL) {

perror("Failed to allocate memory!");

exit(1);

}

// use the element value as ints index and count

// how many times it was found

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

// skip negative numbers because negative index is not defined

if (arr[i] <= 0) continue;

count[arr[i]]++;

}

// reconstruct the array by using the info from the previous step

int index = 0;

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

// add count[i] same elements to the array

while(count[i] > 0) {

arr[index] = i;

// next element in the array

index++;

// decrease the numbo of elements in count

count[i]--;

}

}

// free allocated memry to avoid leaks

free(count);

}

void merge(int\* data, int left, int mid, int right, int\* temp) {

// i is for the left array, j is for the right array and k is for the temp array

int i = left, j = mid, k = left;

// while the index for the left array is inside its bounds

// adn the index of the right array is inside its bounds

while (i < mid && j < right) {

// we want to compare the corresponding elements from each array

// if the value of the right array is bigger we want to copy

// the value of the left array to our temp

if (data[i] >= data[j]) {

temp[k] = data[i];

k++; i++;

}

// otherwise if the left array has a bigger value, we copy the

// value of the right array into our temp

else {

temp[k] = data[j];

k++; j++;

}

}

// copy the remaining elements from left

while(i < mid) {

temp[k] = data[i];

k++; i++;

}

// copy the remaining elements from right

while(j < right) {

temp[k] = data[j];

k++; j++;

}

// copy sorted temp array back into original array

for (i = left; i < right; i++) {

data[i] = temp[i];

}

}

void mergeSort(size\_t n, int\* data) {

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

if (!temp) {

perror("Failed to allocate memory!");

exit(1);

}

// width of the subarrays, doubling at each iteration

for (int width = 1; width < n; width \*= 2) {

// here we have 2 \* width, because we are mergin 2 subarrays

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

int left = i;

// these conditions are needed so that mid and right don't exceed the

// array length

int mid = (i + width < n) ? i + width : n;

int right = (i + 2 \* width < n) ? i + 2 \* width : n;

// prefetch next segment of memory for optimization

\_\_builtin\_prefetch(&data[right], 1, 3);

merge(data, left, mid, right, temp);

}

}

// free temp buffer since it's no longer needed

free(temp);

}

void pop\_left(int\* arr, size\_t\* size) {

if (\*size == 0) return;

// shift all elements to left by one position

for (size\_t i = 0; i < \*size - 1; i++) {

arr[i] = arr[i + 1];

}

(\*size)--;

}

void pop(int\* arr, size\_t\* size) {

// set last element to zero and reduce array size

arr[\*size - 1] = 0;

(\*size)--;

}

void push(int\* arr, size\_t\* size, int element) {

// add new element at the end and increase array size

arr[\*size] = element;

(\*size)++;

}

int\* max\_sliding\_window(int k, int n, int\* arr, size\_t\* resLen) {

int\* res = (int\*)malloc(k \* sizeof(int));

// deque for storing indexs of useful elements

int\* deque = (int\*)malloc(k \* sizeof(int));

size\_t dequeLen = 0;

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

// remove all elements smaller than arr[i]

while (dequeLen > 0 && deque[dequeLen - 1] < arr[i]) {

pop(deque, &dequeLen);

}

push(deque, &dequeLen, i);

}

push(res, resLen, arr[deque[0]]);

int l = 1;

int r = k;

while (r <= n - 1) {

// remove elements that are out of the window

if (dequeLen > 0 && deque[0] == l - 1) {

pop\_left(deque, &dequeLen);

}

// remove elements smaller than arr[r]

while (dequeLen > 0 && arr[deque[dequeLen - 1]] < arr[r]) {

pop(deque, &dequeLen);

}

// push new element and update the result

push(deque, &dequeLen, r);

push(res, resLen, arr[deque[0]]);

l++; r++;

}

// free deque since we no longer need it

free(deque);

return res;

}

int main() {

int n;

printf("Input the number of elements: ");

scanf("%d", &n);

printf("\n");

// allocate memory for array

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

if(!arr) {

printf("\nFailed to allocate memory!\n");

return 0;

}

printf("Input the array elements: \n");

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

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

}

int k;

printf("\n");

printf("Input k: ");

scanf("%d", &k);

printf("\n");

// check if k is valid

if (k > n) {

printf("k cannot be bigger than n");

return 0;

}

size\_t resLen = 0;

int\* res = max\_sliding\_window(k, n, arr, &resLen);

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

memcpy(copyMergeSort, arr, n \* sizeof(int));

printf("\n\n1)Input array in descending order using merge sort: ");

mergeSort(n, copyMergeSort);

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

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

}

printf("\n\n2)The result for version 4: ");

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

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

}

printf("\n\n3)Output array in ascending order using counting sort: ");

countingSort(res, n);

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

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

}

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

// free allocated memory to avoid leaks

free(copyMergeSort);

free(arr);

free(res);

return 0;

}

----------------------------Passing arguments by reference ---------------------------------

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

void countingSort(int\* arr, size\_t\* n) {

int max = arr[0];

// find the maximum elelment

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

if(arr[i] > max) {

max = arr[i];

}

}

// create an array with the size of the maximum element

int\* count = (int\*)calloc(max + 1, sizeof(int));

if (count == NULL) {

perror("Failed to allocate memory!");

exit(1);

}

// use the element value as ints index and count

// how many times it was found

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

// skip negative numbers because negative index is not defined

if (arr[i] <= 0) continue;

count[arr[i]]++;

}

// reconstruct the array by using the info from the previous step

int index = 0;

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

// add count[i] same elements to the array

while(count[i] > 0) {

arr[index] = i;

// next element in the array

index++;

// decrease the numbo of elements in count

count[i]--;

}

}

// free allocated memry to avoid leaks

free(count);

}

void merge(int\* data, int\* left, int\* mid, int\* right, int\* temp) {

// i is for the left array, j is for the right array and k is for the temp array

int i = \*left, j = \*mid, k = \*left;

// while the index for the left array is inside its bounds

// adn the index of the right array is inside its bounds

while (i < \*mid && j < \*right) {

// we want to compare the corresponding elements from each array

// if the value of the right array is bigger we want to copy

// the value of the left array to our temp

if (data[i] >= data[j]) {

temp[k] = data[i];

k++; i++;

}

// otherwise if the left array has a bigger value, we copy the

// value of the right array into our temp

else {

temp[k] = data[j];

k++; j++;

}

}

// copy the remaining elements from left

while(i < \*mid) {

temp[k] = data[i];

k++; i++;

}

// copy the remaining elements from right

while(j < \*right) {

temp[k] = data[j];

k++; j++;

}

// copy sorted temp array back into original array

for (i = \*left; i < \*right; i++) {

data[i] = temp[i];

}

}

void mergeSort(size\_t\* n, int\* data) {

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

if (!temp) {

perror("Failed to allocate memory!");

exit(1);

}

// width of the subarrays, doubling at each iteration

for (int width = 1; width < \*n; width \*= 2) {

// here we have 2 \* width, because we are mergin 2 subarrays

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

int left = i;

// these conditions are needed so that mid and right don't exceed the

// array length

int mid = (i + width < \*n) ? i + width : \*n;

int right = (i + 2 \* width < \*n) ? i + 2 \* width : \*n;

// prefetch next segment of memory for optimization

\_\_builtin\_prefetch(&data[right], 1, 3);

merge(data, &left, &mid, &right, temp);

}

}

// free temp buffer since it's no longer needed

free(temp);

}

void pop\_left(int\* arr, size\_t\* size) {

if (\*size == 0) return;

// shift all elements to left by one position

for (size\_t i = 0; i < \*size - 1; i++) {

arr[i] = arr[i + 1];

}

(\*size)--;

}

void pop(int\* arr, size\_t\* size) {

// set last element to zero and reduce array size

arr[\*size - 1] = 0;

(\*size)--;

}

void push(int\* arr, size\_t\* size, int\* element) {

// add new element at the end and increase array size

arr[\*size] = \*element;

(\*size)++;

}

void max\_sliding\_window(int\* k, size\_t\* n, int\* arr, size\_t\* resLen, int\* res) {

// deque for storing indexs of useful elements

int\* deque = (int\*)malloc(\*k \* sizeof(int));

size\_t dequeLen = 0;

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

// remove all elements smaller than arr[i]

while (dequeLen > 0 && deque[dequeLen - 1] < arr[i]) {

pop(deque, &dequeLen);

}

push(deque, &dequeLen, &i);

}

push(res, resLen, &arr[deque[0]]);

int l = 1;

int r = \*k;

while (r <= \*n - 1) {

// remove elements that are out of the window

if (dequeLen > 0 && deque[0] == l - 1) {

pop\_left(deque, &dequeLen);

}

// remove elements smaller than arr[r]

while (dequeLen > 0 && arr[deque[dequeLen - 1]] < arr[r]) {

pop(deque, &dequeLen);

}

// push new element and update the result

push(deque, &dequeLen, &r);

push(res, resLen, &arr[deque[0]]);

l++; r++;

}

// free deque since we no longer need it

free(deque);

}

int main() {

size\_t n;

printf("Input the number of elements: ");

scanf("%ld", &n);

printf("\n");

// allocate memory for array

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

if(!arr) {

printf("\nFailed to allocate memory!\n");

return 0;

}

printf("Input the array elements: \n");

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

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

}

int k;

printf("\n");

printf("Input k: ");

scanf("%d", &k);

printf("\n");

// check if k is valid

if (k > n) {

printf("k cannot be bigger than n");

return 0;

}

size\_t resLen = 0;

int\* res = (int\*)malloc(k \* sizeof(int));

max\_sliding\_window(&k, &n, arr, &resLen, res);

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

memcpy(copyMergeSort, arr, n \* sizeof(int));

printf("\n\n1)Input array in descending order using merge sort: ");

mergeSort(&n, copyMergeSort);

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

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

}

printf("\n\n2)The result for version 4: ");

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

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

}

printf("\n\n3)Output array in ascending order using counting sort: ");

countingSort(res, &n);

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

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

}

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

// free allocated memory to avoid leaks

free(copyMergeSort);

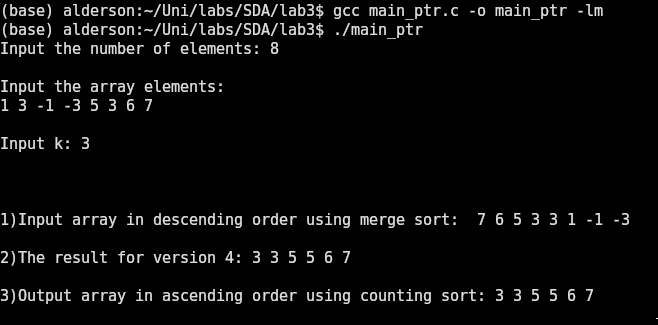
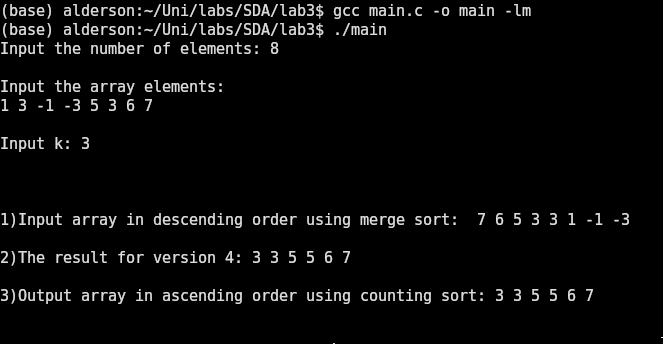
free(arr);

free(res);

return 0;

}

**3.** **Screenshots of the outputs**

 **4. Conclusion**

The solution presented in this report was chosen to be the best among other solutions in the team, because it was the only solution approaching a linear time complexity, with a time complexity of O(n\*k). It is possible to make the solution O(n), but this would require a different implementation of the double ended queue than the one presented in the solution. The downside of an O(n) solution would be a worse memory layout, because a double linked list would have to be used, which itself would degrade the performance. So in order to keep the code cleaner and more efficient with inputs of reasonable size the O(n\*k) solution was chosen.