

**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**

**Group: FAF-241**

**Report**

**Laboratory Work No.2**

*The variant No.9*

***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 betwgeen 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.

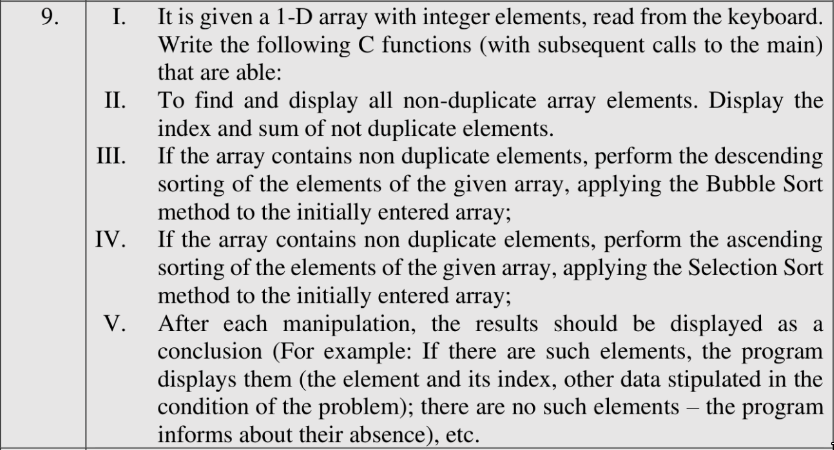
**The purpose** of the laboratory work was to solve a problem using C language by a procedural approach. The problem should be presented in two versions:

a) passing arguments by value

b) passing arguments by reference (using pointers)

Also, a block diagram which represents the solution to the problem should be created. After implementing the given variant, a modified version of the task should be written and solved in the same manner. The modified task should use Merge Sort and Insertion Sort, because the previous task used other sorting algorithms.

TASK 1 (Variant 9)



===================== Task 1 (Passing arguments by value) ==================

#include <stdio.h>

#include <stdint.h>

#include <time.h>

#include <malloc.h>

int printNonDuplicates(int\* arr, int size);

void bubbleSort(int\* arr, int size);

void selectionSort(int\* arr, int size);

double getTimeMicroseconds() {

struct timespec ts;

clock\_gettime(CLOCK\_MONOTONIC, &ts);

return ts.tv\_sec \* 1e6 + ts.tv\_nsec / 1e3;

}

int main() {

int n = 0;

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

scanf("%d", &n);

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

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

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

}

double start = getTimeMicroseconds();

printf("\n1) Find and display all non-duplicate array elements.\n");

int containsDup = printNonDuplicates(arr, n);

if (containsDup) { // bubble sort is stable that's why we do it if there are duplicates

printf("\n2) If the array contains duplicates perform a descending bubble sort. \n");

bubbleSort(arr, n);

printf("\nElements after descending bubble sort: \n");

for (int i = 0; i < n; i++) printf("%d ", arr[i]);

} else { // selection sort is not stable that's why we do it iff there are no duplicates

printf("\n2) If the array does not contains duplicates perform an ascending selection sort. \n");

selectionSort(arr, n);

printf("\nElements after ascending selection sort: \n");

for (int i = 0; i < n; i++) printf("%d ", arr[i]);

}

double end = getTimeMicroseconds();

printf("\n\nElapsed time: %f\n\n", end - start);

return 0;

}

int printNonDuplicates(int\* arr, int size) {

int sum = 0;

int containsDup = 0;

int tableHeadPrinted = 0;

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

int count = 0;

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

if (arr[i] == arr[j]) {

count++;

}

}

if (count == 1 && !tableHeadPrinted) {

printf("\n\n INDEX | VALUE \n");

tableHeadPrinted = 1;

}

if (count == 1) {

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

sum += arr[i];

continue;

}

containsDup = 1;

}

if (sum > 0) printf("The sum is: %d \n\n", sum);

else printf("There are only duplicates in the array!\n\n");

return containsDup;

}

void bubbleSort(int\* arr, int size) {

int swapped;

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

// define a flag variable to optimize

swapped = 0;

for (int j = 0; j < size - 1; j++) {

if (arr[j] < arr[j+1]) {

int temp = arr[j];

arr[j] = arr[j + 1];

arr[j + 1] = temp;

swapped = 1;

}

}

// if the swap did not occur it means that the array

// is already sorted and no more operations are needed.

if (!swapped) break;

}

}

void selectionSort(int\* arr, int size) {

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

int min = i;

// for each element go through all the elements after it

// and then find the smallest. When the smallest is found

// swap it with the current element

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

if(arr[min] > arr[j]) {

min = j;

}

}

if (min != i) {

int temp = arr[i];

arr[i] = arr[min];

arr[min] = temp;

}

}

}

===================== Task 1 (Passing arguments by reference) ==================

#include <stdio.h>

#include <stdint.h>

#include <time.h>

#include <malloc.h>

void printNonDuplicates(int\* arr, int\* size, int\* containsDup);

void bubbleSort(int\* arr, int\* size);

void selectionSort(int\* arr, int\* size);

double getTimeMicroseconds() {

struct timespec ts;

clock\_gettime(CLOCK\_MONOTONIC, &ts);

return ts.tv\_sec \* 1e6 + ts.tv\_nsec / 1e3;

}

int main() {

int n = 0;

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

scanf("%d", &n);

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

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

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

}

double start = getTimeMicroseconds();

printf("\n1) Find and display all non-duplicate array elements.\n");

int containsDup = 0;

printNonDuplicates(arr, &n, &containsDup);

if (containsDup) { // bubble sort is stable that's why we do it if there are duplicates

printf("\n2) If the array contains duplicates perform a descending bubble sort. \n");

bubbleSort(arr, &n);

printf("\nElements after descending bubble sort: \n");

for (int i = 0; i < n; i++) printf("%d ", arr[i]);

} else { // selection sort is not stable that's why we do it iff there are no duplicates

printf("\n2) If the array does not contains duplicates perform an ascending selection sort. \n");

selectionSort(arr, &n);

printf("\nElements after ascending selection sort: \n");

for (int i = 0; i < n; i++) printf("%d ", arr[i]);

}

double end = getTimeMicroseconds();

printf("\n\nElapsed time: %f\n\n", end - start);

return 0;

}

void printNonDuplicates(int\* arr, int\* size, int\* containsDup) {

int sum = 0;

int tableHeadPrinted = 0;

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

int count = 0;

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

if (arr[i] == arr[j]) {

count++;

}

}

if (count == 1 && !tableHeadPrinted) {

printf("\n\n INDEX | VALUE \n");

tableHeadPrinted = 1;

}

if (count == 1) {

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

sum += arr[i];

continue;

}

\*containsDup = 1;

}

if (sum > 0) printf("The sum is: %d \n\n", sum);

else printf("There are only duplicates in the array!\n\n");

}

void bubbleSort(int\* arr, int\* size) {

int swapped;

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

// define a flag variable to optimize

swapped = 0;

for (int j = 0; j < \*size - 1; j++) {

if (arr[j] < arr[j+1]) {

int temp = arr[j];

arr[j] = arr[j + 1];

arr[j + 1] = temp;

swapped = 1;

}

}

// if the swap did not occur it means that the array

// is already sorted and no more operations are needed.

if (!swapped) break;

}

}

void selectionSort(int\* arr, int\* size) {

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

int min = i;

// for each element go through all the elements after it

// and then find the smallest. When the smallest is found

// swap it with the current element

for (int j = i; j < \*size; j++) {

if(arr[min] > arr[j]) {

min = j;

}

}

if (min != i) {

int temp = arr[i];

arr[i] = arr[min];

arr[min] = temp;

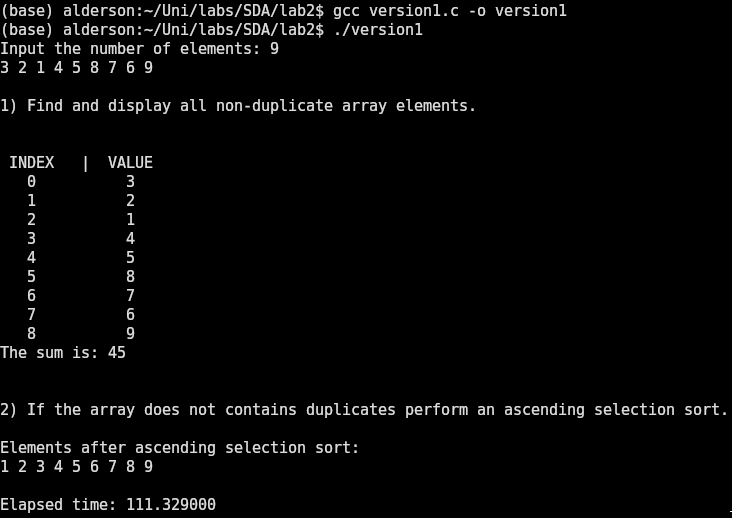
}

}

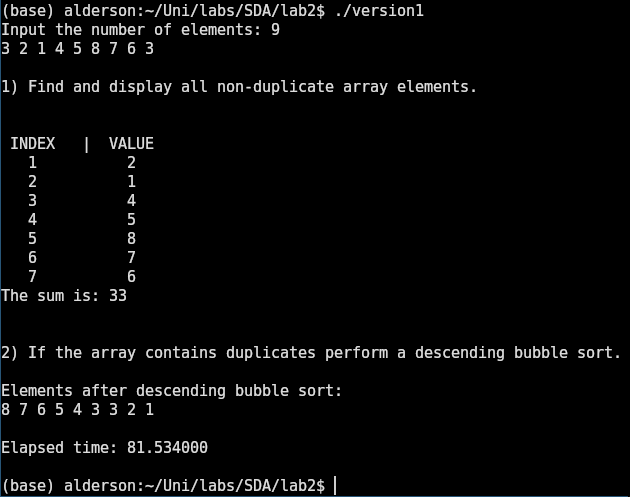
}

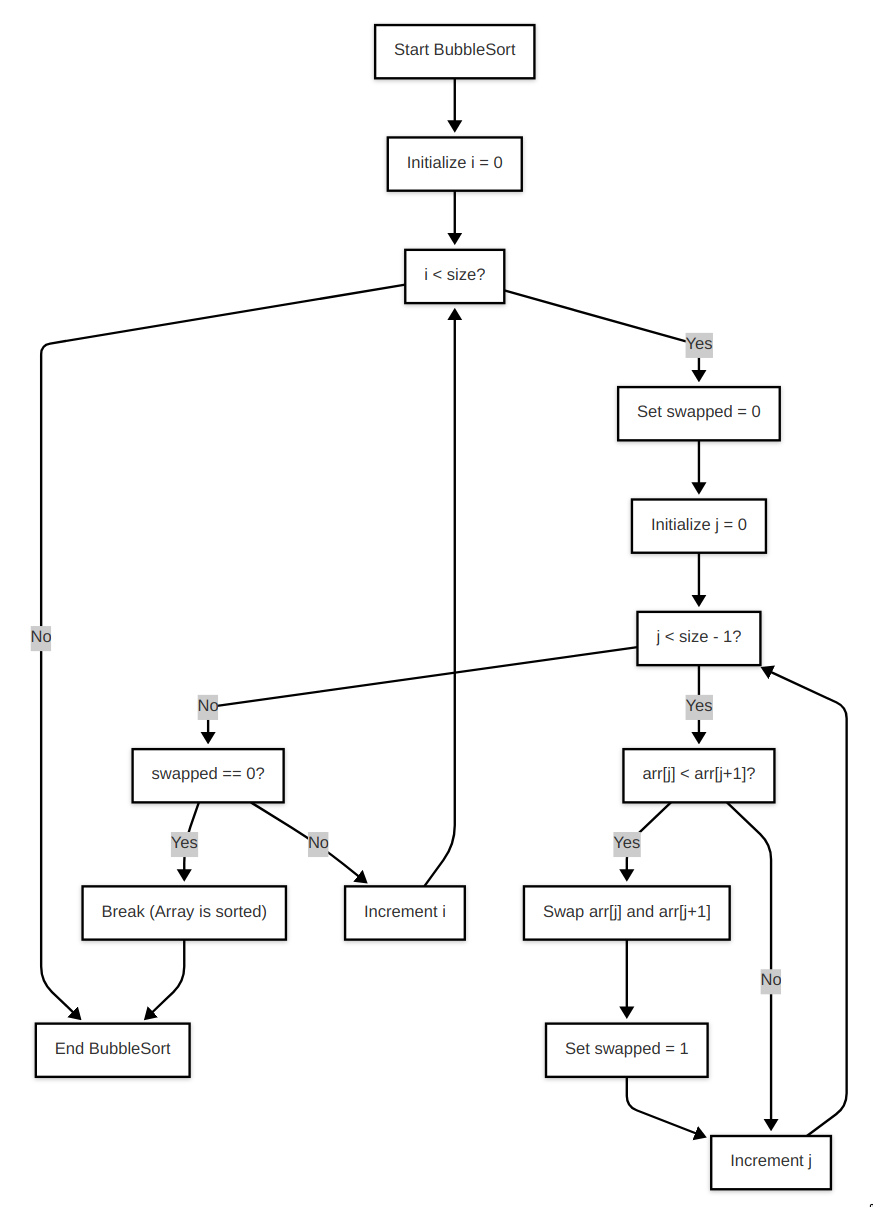
-----------------------------------------------------------------------------

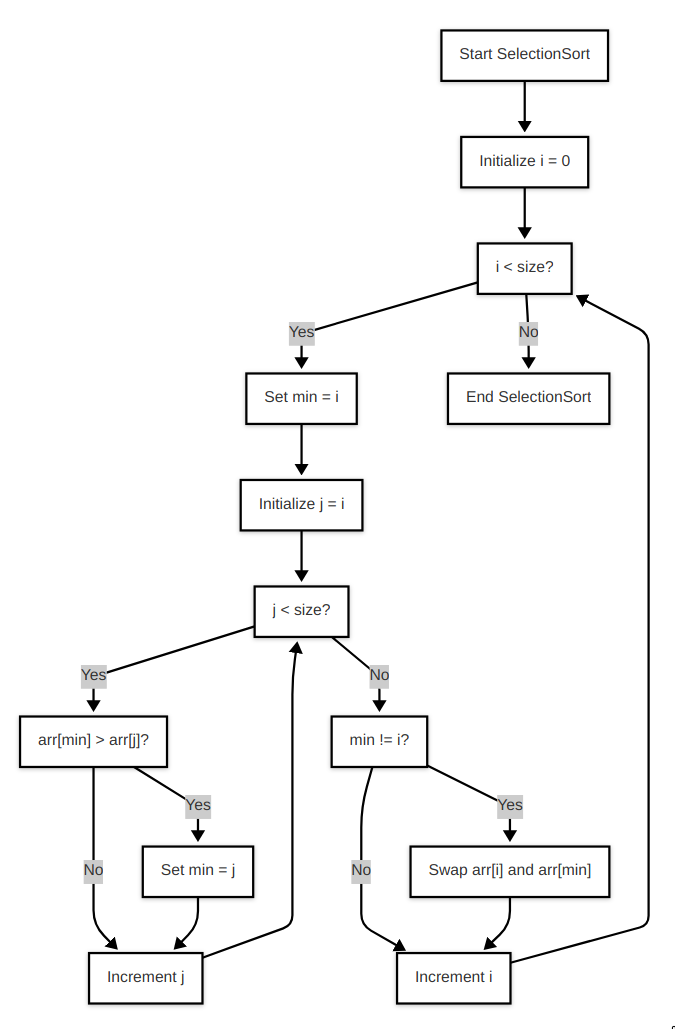
The results of the code are shown below.

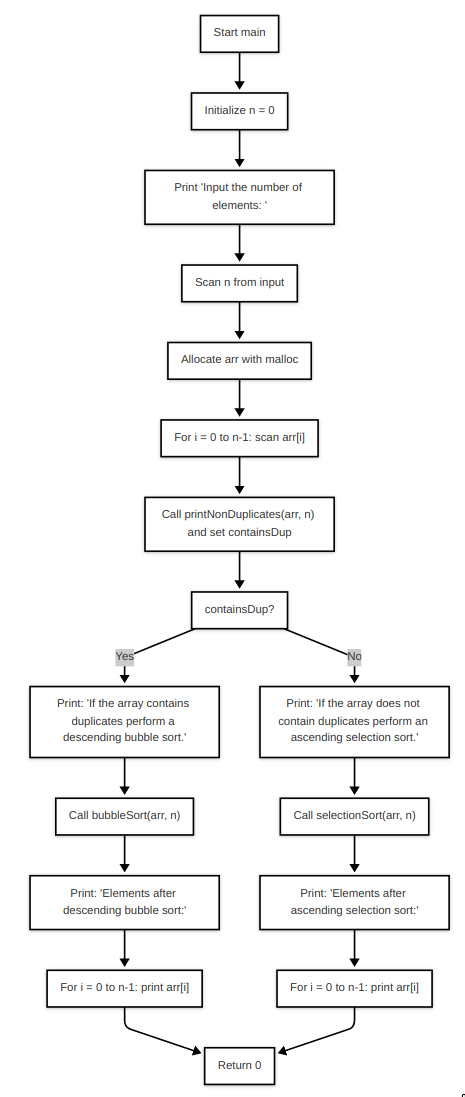
a) The array does not contain duplicates so selection sort is performed in ascending order.

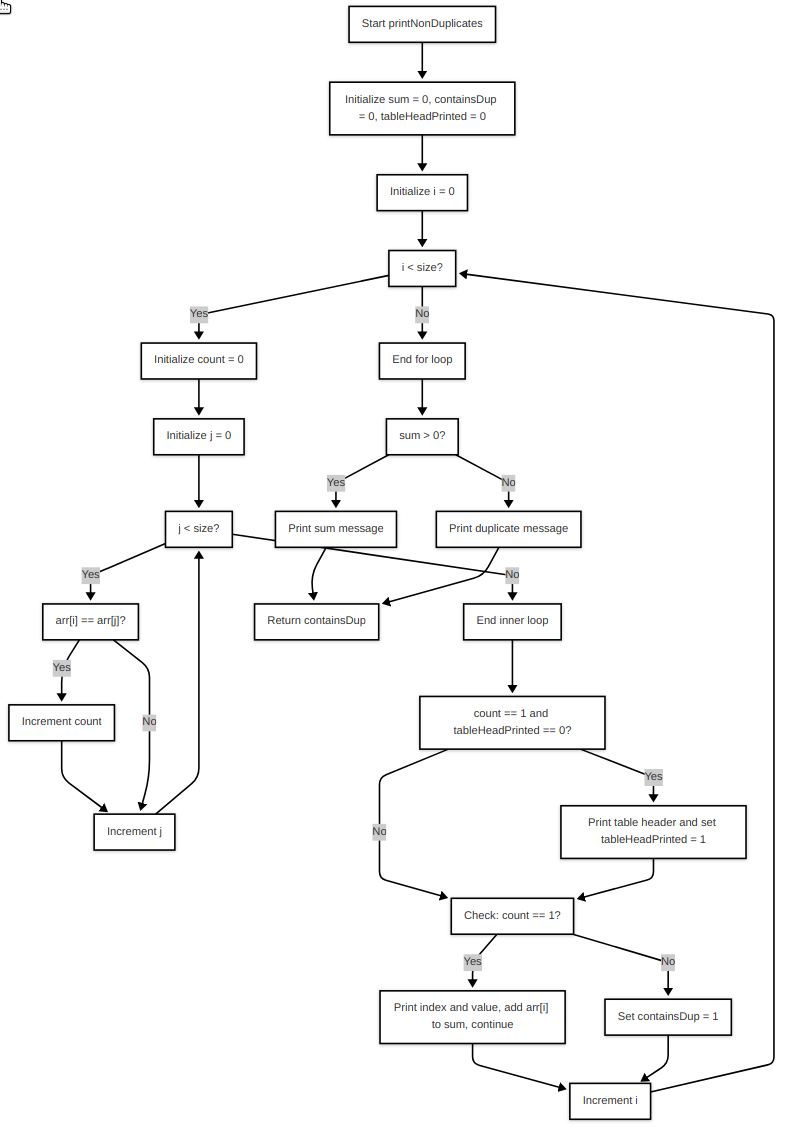
b) The array contains duplicates so bubble sort is performed in descending order.









The **modified task** which uses Merge Sort and Insertion Sort is formulated in the following way:

Write a program in C using the procedural paradigm, which:

a) Can read a 1-D array from the keyboard

b) Can read a 1-D array from a .bin file if it is specified in the command line arguments (argv).

The program should be performing insertion sort if the number of elements passed to it is less than 32 and it should be performing merge sort if the number of elements is greater than 32. The program should be running reasonably fast.

================= Task 2 (passing arguments by value) ===============

#include <stddef.h>

#include <stdio.h>

#include <time.h>

#include <malloc.h>

#include <stdlib.h>

void readFromKeyboard(size\_t\* numberOfElements, int\*\* data);

void readFromFile(size\_t\* numberOfElements, int\*\* data, char\* fileName);

void insertSort(size\_t numberOfElements, int\* data);

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

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

double getTimeMicroseconds() {

struct timespec ts;

clock\_gettime(CLOCK\_MONOTONIC, &ts);

return ts.tv\_sec \* 1e6 + ts.tv\_nsec / 1e3;

}

int main(int argc, char \*argv[]) {

size\_t numberOfElements = 0;

int\* data = NULL;

if (argc == 1) {

readFromKeyboard(&numberOfElements, &data);

} else {

readFromFile(&numberOfElements, &data, argv[1]);

}

double start = getTimeMicroseconds();

if (numberOfElements <= 32) {

insertSort(numberOfElements, data);

} else {

mergeSort(numberOfElements, data);

}

double end = getTimeMicroseconds();

numberOfElements <= 32 ?

printf("Used insertion sort to sort all the elements!") :

printf("Used merge sort to sort all the elements!");

printf("\nINDEX | VALUE\n");

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

printf(" %d %d\n", i, data[i]);

}

printf("\n\nExecution in microseconds: %f\n\n", end - start);

free(data);

return 0;

}

void insertSort(size\_t numberOfElements, int\* data) {

// check for edge case

if (numberOfElements <= 1) return;

for (size\_t i = 1; i < numberOfElements; i++) {

int val = data[i];

// j is the element which comes before the current one

int j = i - 1;

// while j is greater than 0 and the value at j is greater than our value

while (j >= 0 && data[j] > val) {

// we move each element to the right of the array.

data[j + 1] = data[j];

// and then decrease our j

j--;

}

// after we found the correct place for our value

// we insert it there.

data[j + 1] = val;

}

}

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

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;

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

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

}

}

free(temp);

}

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++;

}

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

data[i] = temp[i];

}

}

void readFromKeyboard(size\_t\* numberOfElements, int\*\* data) {

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

scanf("%ld", numberOfElements);

\*data = malloc(\*numberOfElements \* sizeof(int));

printf("\nInput the elements separated by space: \n");

if (\*data == NULL) exit(1);

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

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

}

}

void readFromFile(size\_t\* numberOfElements, int\*\* data, char\* fileName) {

FILE \*file = fopen(fileName, "rb");

if (!file) {

perror("Error opening file");

exit(1);

}

fseek(file, 0, SEEK\_END);

long file\_size = ftell(file);

if (file\_size < 0) {

perror("Error getting file size");

fclose(file);

exit(1);

}

rewind(file);

\*numberOfElements = file\_size / sizeof(int);

\*data = (int\*)malloc(\*numberOfElements \* sizeof(int));

if (!data) {

perror("Memory allocation failed");

fclose(file);

exit(1);

}

size\_t elements\_read = fread(\*data, sizeof(int), \*numberOfElements, file);

if (elements\_read != \*numberOfElements) {

perror("Error reading file");

free(data);

fclose(file);

exit(1);

}

fclose(file);

}

================ Task 2 (passing arguments by reference) =============

#include <stddef.h>

#include <stdio.h>

#include <time.h>

#include <malloc.h>

#include <stdlib.h>

void readFromKeyboard(size\_t\* numberOfElements, int\*\* data);

void readFromFile(size\_t\* numberOfElements, int\*\* data, char\* fileName);

void insertSort(size\_t\* numberOfElements, int\* data);

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

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

double getTimeMicroseconds() {

struct timespec ts;

clock\_gettime(CLOCK\_MONOTONIC, &ts);

return ts.tv\_sec \* 1e6 + ts.tv\_nsec / 1e3;

}

int main(int argc, char \*argv[]) {

size\_t numberOfElements = 0;

int\* data = NULL;

if (argc == 1) {

readFromKeyboard(&numberOfElements, &data);

} else {

readFromFile(&numberOfElements, &data, argv[1]);

}

double start = getTimeMicroseconds();

if (numberOfElements <= 32) {

insertSort(&numberOfElements, data);

} else {

mergeSort(&numberOfElements, data);

}

double end = getTimeMicroseconds();

numberOfElements <= 32 ?

printf("Used insertion sort to sort all the elements!") :

printf("Used merge sort to sort all the elements!");

printf("\nINDEX | VALUE\n");

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

printf(" %d %d\n", i, data[i]);

}

printf("\n\nExecution in microseconds: %f\n\n", end - start);

free(data);

return 0;

}

void insertSort(size\_t\* numberOfElements, int\* data) {

// check for edge case

if (\*numberOfElements <= 1) return;

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

int val = data[i];

// j is the element which comes before the current one

int j = i - 1;

// while j is greater than 0 and the value at j is greater than our value

while (j >= 0 && data[j] > val) {

// we move each element to the right of the array.

data[j + 1] = data[j];

// and then decrease our j

j--;

}

// after we found the correct place for our value

// we insert it there.

data[j + 1] = val;

}

}

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

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;

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

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

}

}

free(temp);

}

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++;

}

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

data[i] = temp[i];

}

}

void readFromKeyboard(size\_t\* numberOfElements, int\*\* data) {

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

scanf("%ld", numberOfElements);

\*data = malloc(\*numberOfElements \* sizeof(int));

printf("\nInput the elements separated by space: \n");

if (\*data == NULL) exit(1);

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

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

}

}

void readFromFile(size\_t\* numberOfElements, int\*\* data, char\* fileName) {

FILE \*file = fopen(fileName, "rb");

if (!file) {

perror("Error opening file");

exit(1);

}

fseek(file, 0, SEEK\_END);

long file\_size = ftell(file);

if (file\_size < 0) {

perror("Error getting file size");

fclose(file);

exit(1);

}

rewind(file);

\*numberOfElements = file\_size / sizeof(int);

\*data = (int\*)malloc(\*numberOfElements \* sizeof(int));

if (!data) {

perror("Memory allocation failed");

fclose(file);

exit(1);

}

size\_t elements\_read = fread(\*data, sizeof(int), \*numberOfElements, file);

if (elements\_read != \*numberOfElements) {

perror("Error reading file");

free(data);

fclose(file);

exit(1);

}

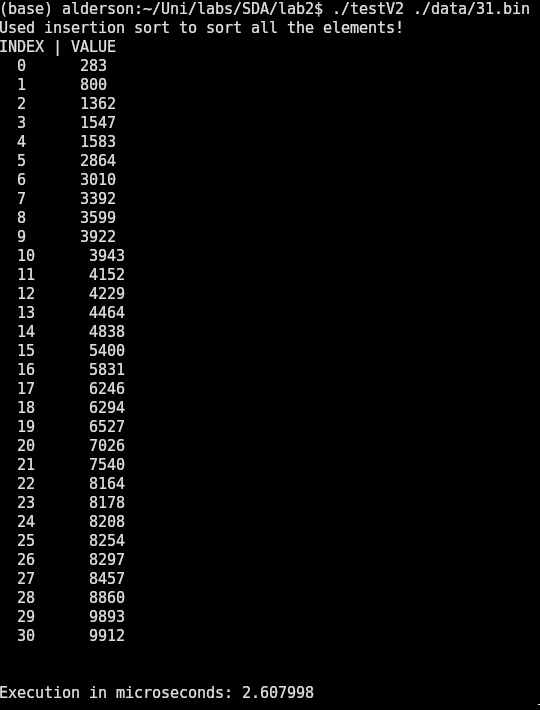
fclose(file);

}

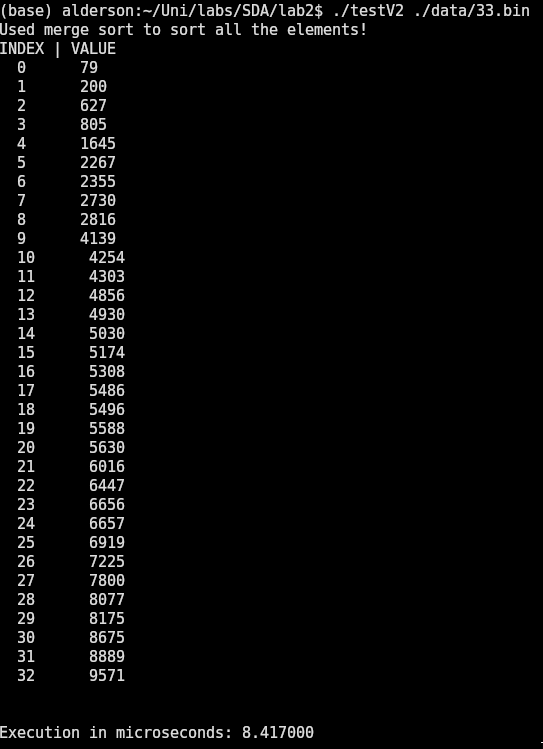
-----------------------------------------------------------------------------

**The result of the code** can be seen below

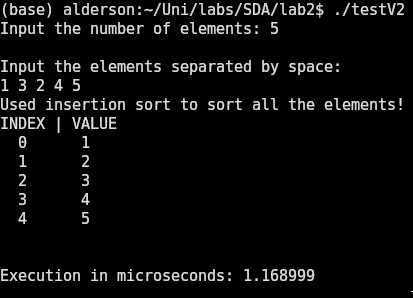
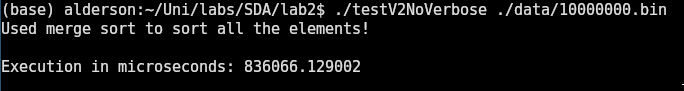
a) The number of the elements in the .bin file is less than 32



b) The number of elements in the .bin file is greater than 32



c) The numbers can be introduced from the keyboard

  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
d) The program should be running reasonably fast. 836ms for 10 million random entries between 0 and 2147483646.  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
**In conclusion,** we can see that different sorting algorithms are well suited for different tasks. For instance although bubble sort and selection sort have the same O(n^2) complexity, bubble sort is a stable algorithm, which does not change the order of elements with the same value, unlike selection sort. This can be very valuable in some cases where one sorting is applied after the other, like in databases.

Also, a very important aspect is the way the algorithm accesses memory. As can be seen in the Task 2, insertion sort performs the sorting in around 2 microseconds, while merge sort does it in 8 microseconds with a difference of 2 elements. This is because insertion sort performs a sequential read of the array, which reduces cache misses, and merge sort doing more “unpredictable” reads from the CPU perspective.

A way to improve the performance of the algorithm in Task 2, is to use quick sort, if the stability is not crucial. Also, another way to sort a large number of numbers is by using Bitonic Sort, which is a sorting algorithm without branches, which makes it suitable for running on GPUs. Although, a downside is the latency introduced by the move of large data chunks from RAM to VRAM and its inefficiency for small datasets.