**UNIVERSIDAD AUTÓNOMA DE MADRID**

**ESCUELA POLITÉCNICA SUPERIOR**



**REPORT ABOUT ASSIGNMENT 2**

**ANALYSIS OF ALGORITHMS**

**GROUP 1291**

**Degree in Bilingual Computer Engineering**

**PAULA BELTRÁN MARIANINI and LUCÍA GIL MAROTO**

**17/11/2020**

# INTRODUCTION

The files we have are the same as in the first assignment. They are:

* **Makefile**
* **.c filesà** divided in two groups, one corresponds to where we will write our code and the other one consists of 5 *exercise\*.c* that will help us later to check our code.
* **.h filesà** the headers of the first .C file group mentioned before, which contains all their declared functions, macros, etc.

Out work is going to be divided in two different sections:

**Section 1** is formed by two exercises. These are the MergeSort ‘s exercises where we will have to write its code and then adapt some of the exercises from last week.

In exercise 1 we need to implement MergeSort’s functions. Both are going to be written in the sorting.c file and declared in the sorting.h file. Algo we will have to make changes in exercise4.c from the first assignment to see if our functions work correctly.

In exercise 2 we need to adapt exercise5.c from the first assignment in order to make it work with MergeSort algorithm. We will plot these values and compare them with the theoretical expected.

**Section 2** is formed by three exercises. These are the ones related to QuickSort algortihm.

In exercise 3 we need to implement QuickSort’s functions. Both are going to be written in the sorting.c file and declared in the sorting.h file. Algo we will have to make changes in exercise4.c from the first assignment to see if our functions work correctly.

In exercise 4 we need to adapt exercise5.c from the first assignment in order to make it work with QuickSort algorithm. We will plot these values and compare them with the theoretical expected.

In exercise 5 we will have to implement a QuickSort function so that we avoid tail recursion. After this, we will have to adapt exercise5.c and compare its results with the ones obtained in the exercise5.c routine for QuickSort with tail recursion.

# OBJECTIVES

In addition, at the beginning of the practice we set some objectives in order to solve this task in the most effective, productive, and correct way.

|  |  |
| --- | --- |
| **DATE** | **OBJECTIVES:** |
| **WED 21/10** | **Read all the practice, set the objectives, and do ex.1** |
| **WED 28/10** | **Do ex.2 and ex.3** |
| **WED 04/11** | **Do ex.4 and ex.5** |
| **WED 11/11** | **Answer the final questions** |
| **WEEKEND** | **Check that all exercises are correct and write down the documentation.** |

Practice number two has as objective try to determine experimentally the execution times of algorithms which use divide and conquer approach. These algorithms are MergeSort and QuickSort. The second one we are going to study it in a recursive form and in a non-recursive form to analyze the differences in time of execution, number of operations, etc. On each algorithm we will analyze tables of different sizes and we will compare the results obtained with the theoretical study of each algorithm.

In order to achieve these goals, we are going to use part of the code of practice 1. We will use files permutations.c, times.c, the makefile and, with some modifications, exercise4.c and exercise5.c.

Here we have a summary of the inputs and outputs of each function we have to implement during the assignment.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **SECTION** | **EXERCISE** | **FUNCTION** | **INPUT** | **DEFINITION** | **OUTPUT** | **DEFINITION** |
| Section 1 | ONE | mergesort | int \*table | Table to be sorted | OB/ ERR | number of times the OB has been executed/ error |
| int ip | First element |
| int iu | Last element |
| merge | int \*table | Table to be sorted | ob/ ERR | number of times the OB has been executed/ error |
| int ip | First element |
| int iu | Last element |
| int imiddle | Index of the midpoint of the table |
| TWO | We did not need to implement any new function in this exercise. Only to change one that already existed in exercise 5 | - | - | - | - |
| Section 2 | THREE | quicksort | int \*table | Table to be sorted | cont/ ERR | number of times the OB has been executed/ error |
| int ip | First element |
| int iu | Last element |
| split | int \*table | Table to be sorted | cont/ ERR | number of times the OB has been executed/ error |
| int ip | First element |
| int iu | Last element |
| Int \*pos | Pointer to save pivot’s position |
| median | int \*table | Table to be sorted | OK/ERR | Ok in case of success/error |
| int ip | First element |
| int iu | Last element |
| Int \*pos | Pointer to save pivot’s position |
| intSwap | int \*x | First element to swap | OK/ERR | Ok in case of success/error |
| Int \*y | Second element to swap |
| FOUR | We did not need to implement any new function in this exercise. Only to change one that already existed in exercise 5 | - | - | - | - |
| FIVE | quicksort | int \*table | Table to be sorted | cont/ ERR | number of times the OB has been executed/ error |
| int ip | First element |
| int iu | Last element |

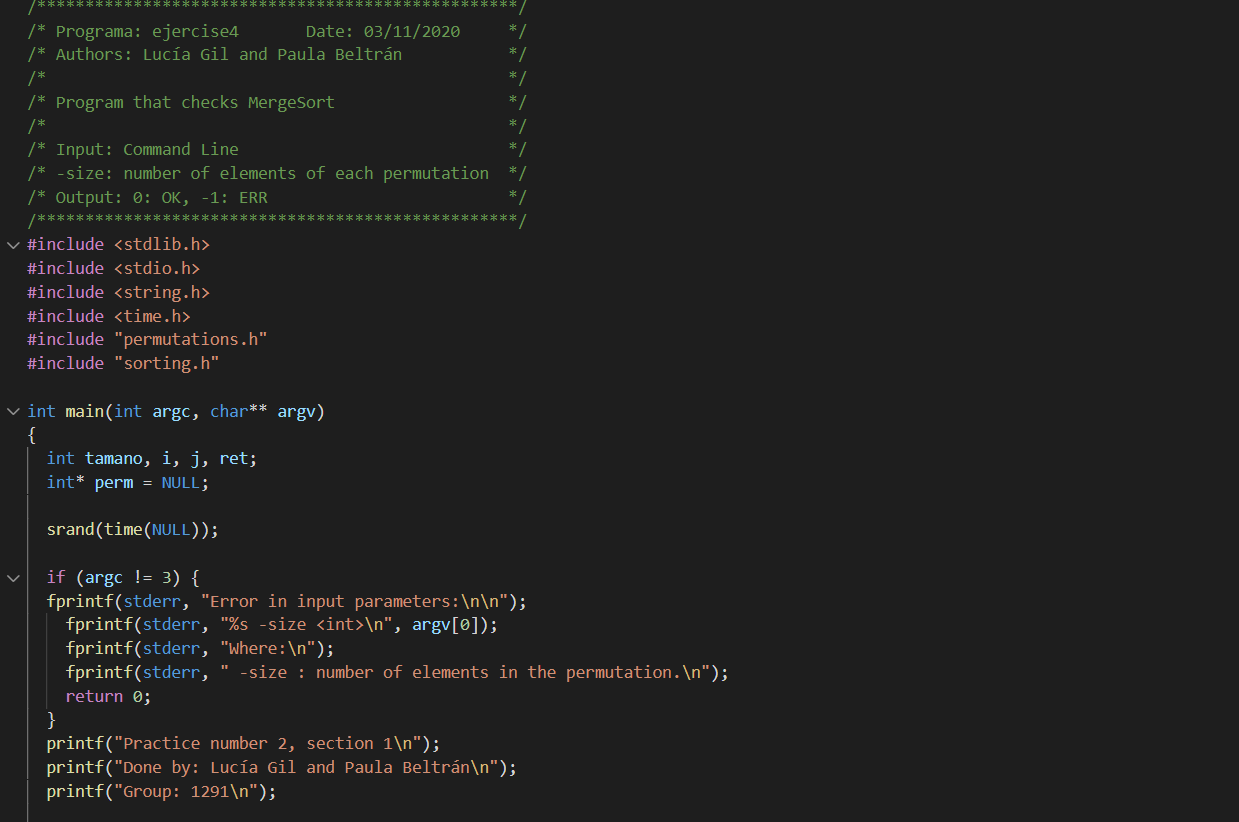
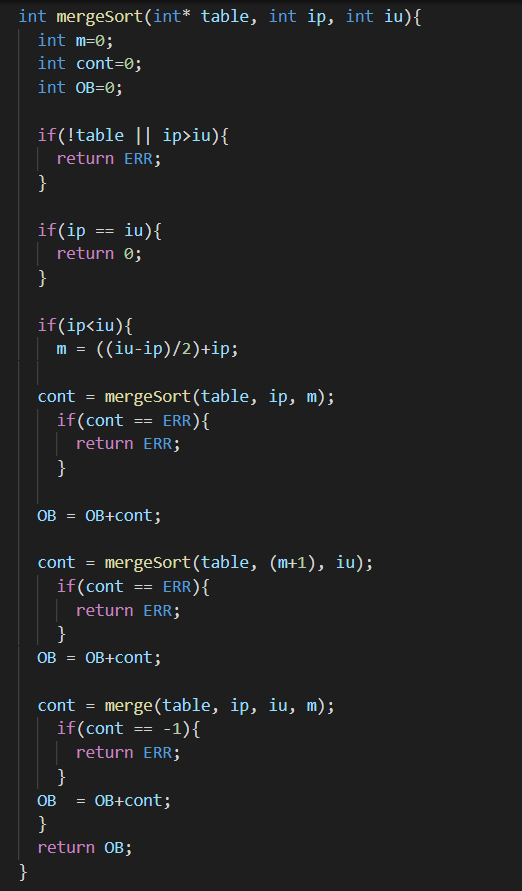
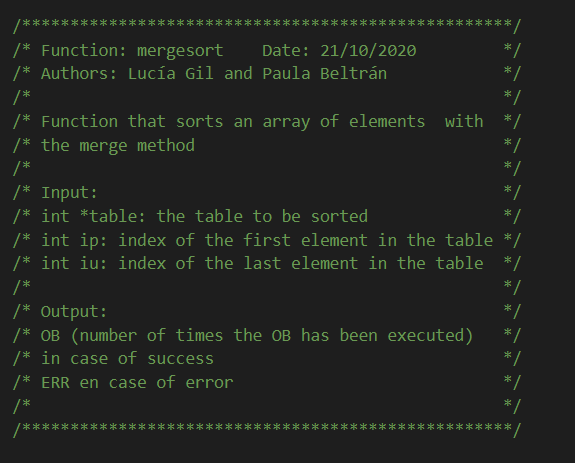
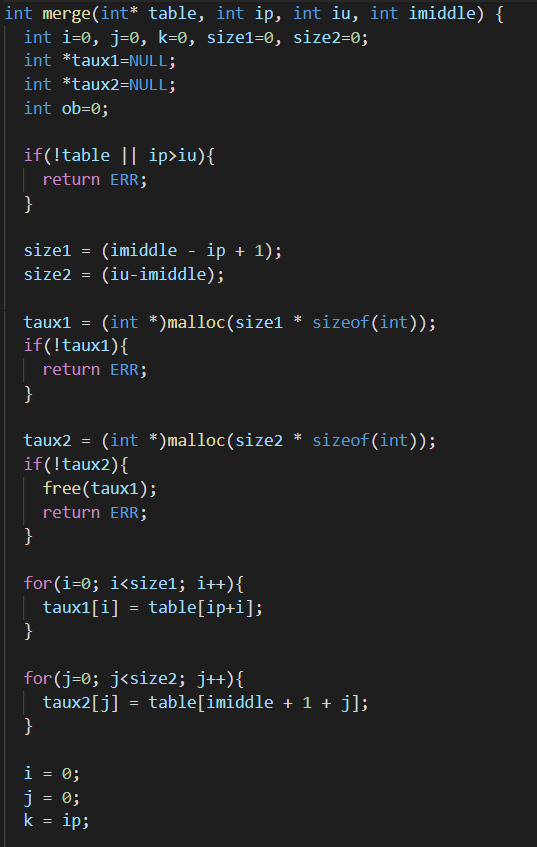
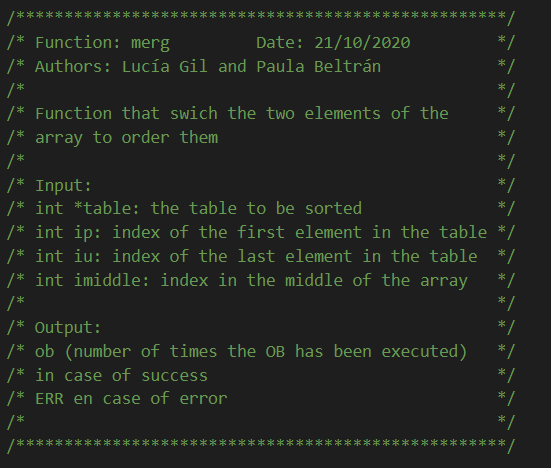
# TOOLS AND METHOLOGY

Our environment has been Windows, in where we have used the Visual Studio Code with the terminal associated, gcc and Valgrind to compile and check all the memory errors. Then, to create the different histograms we have used GNUPlot.

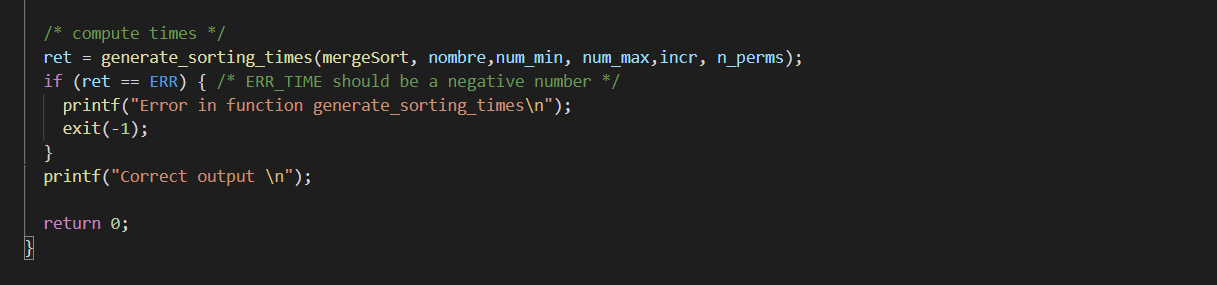
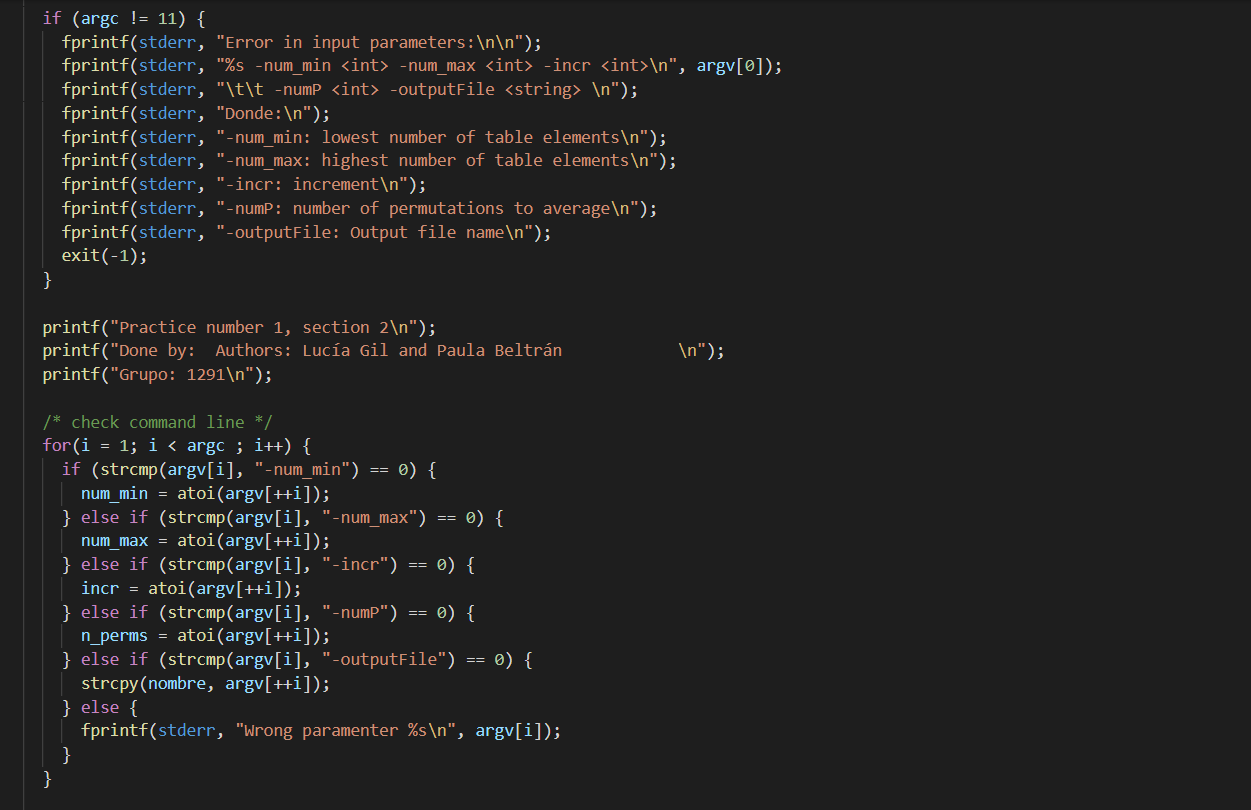
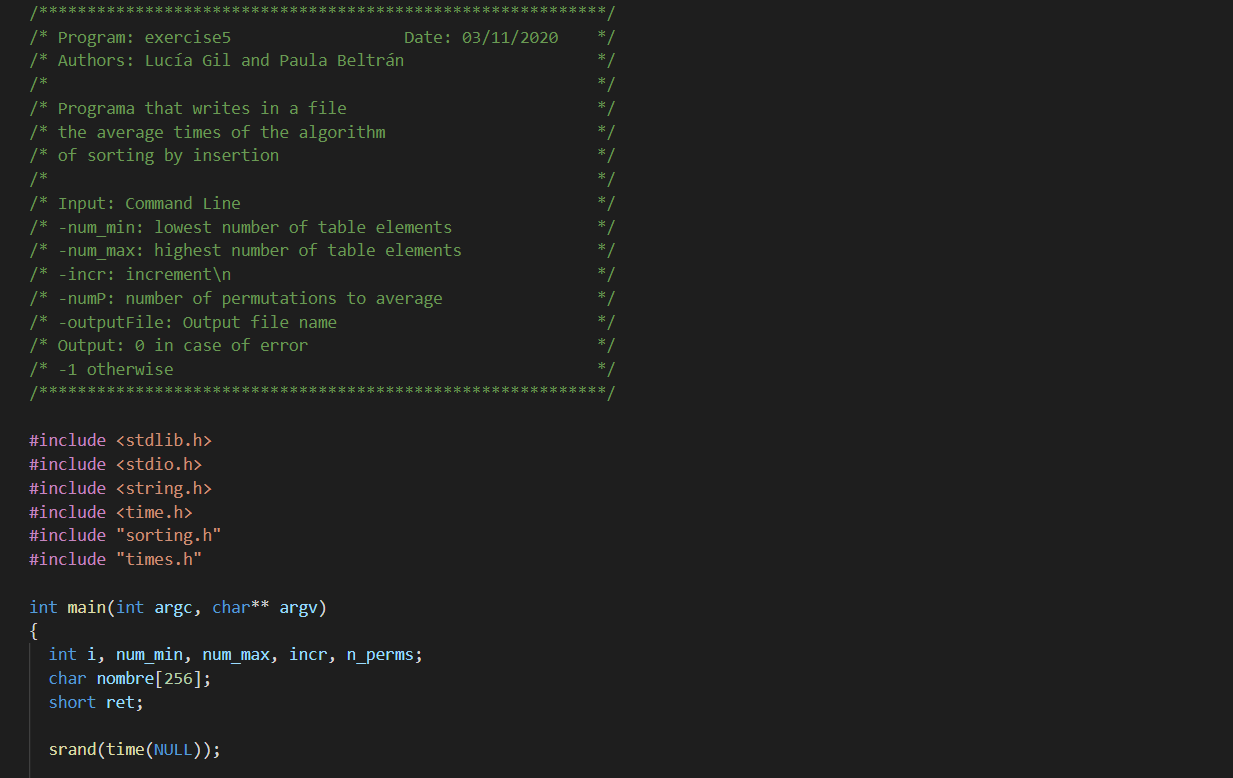
# PRINTED CODE

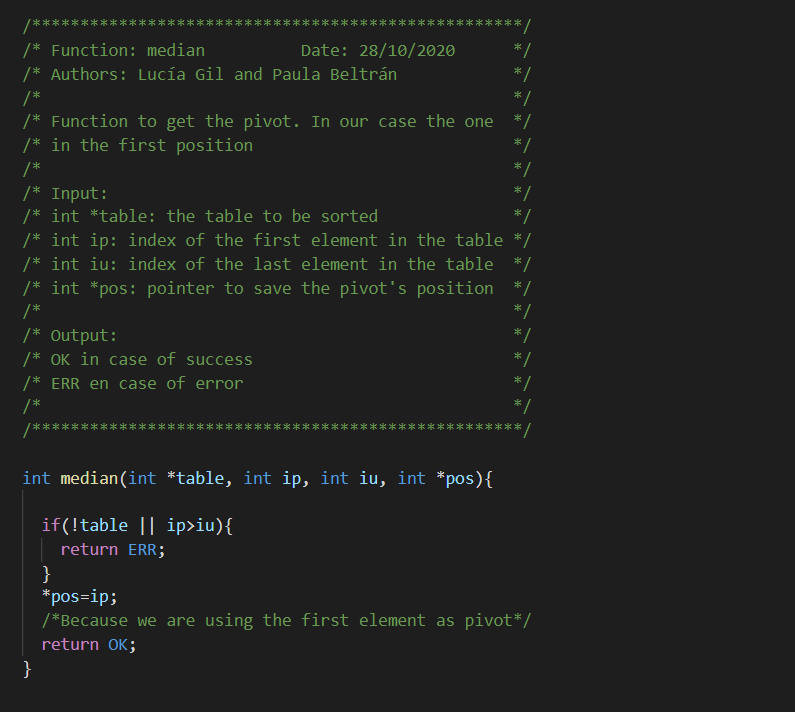
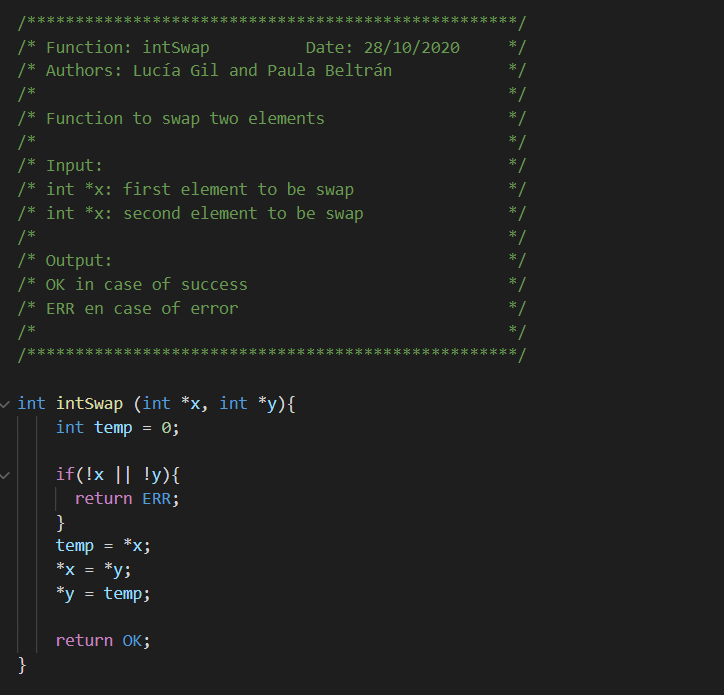
Routine code for Section 2:

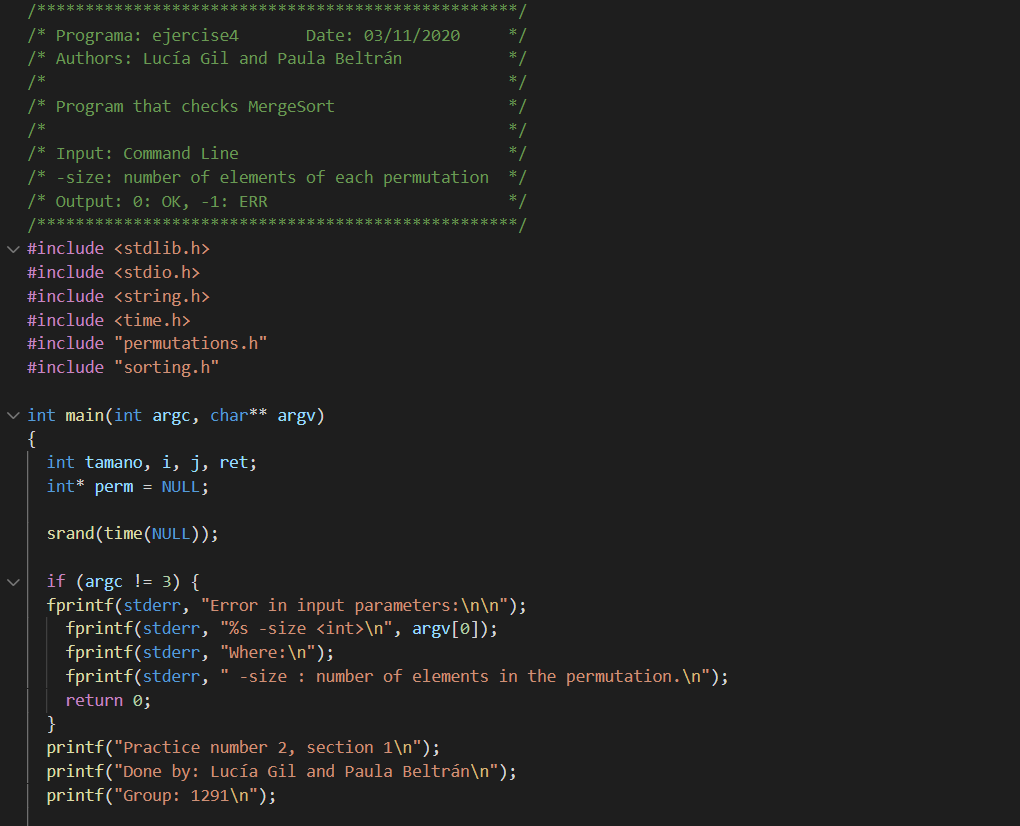
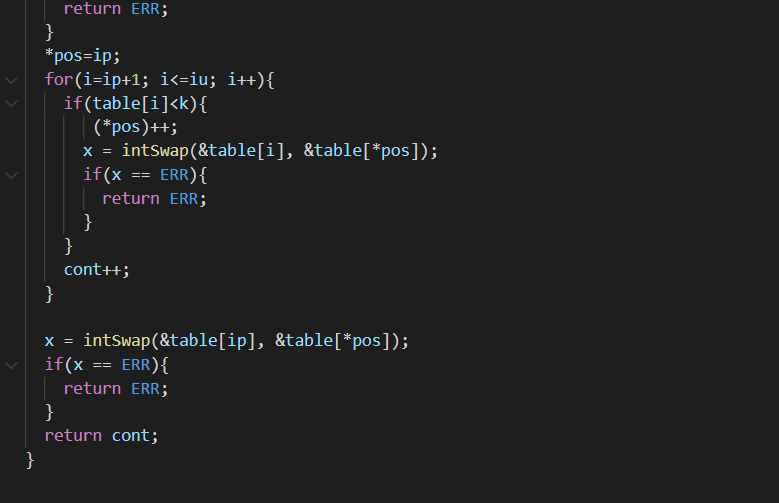
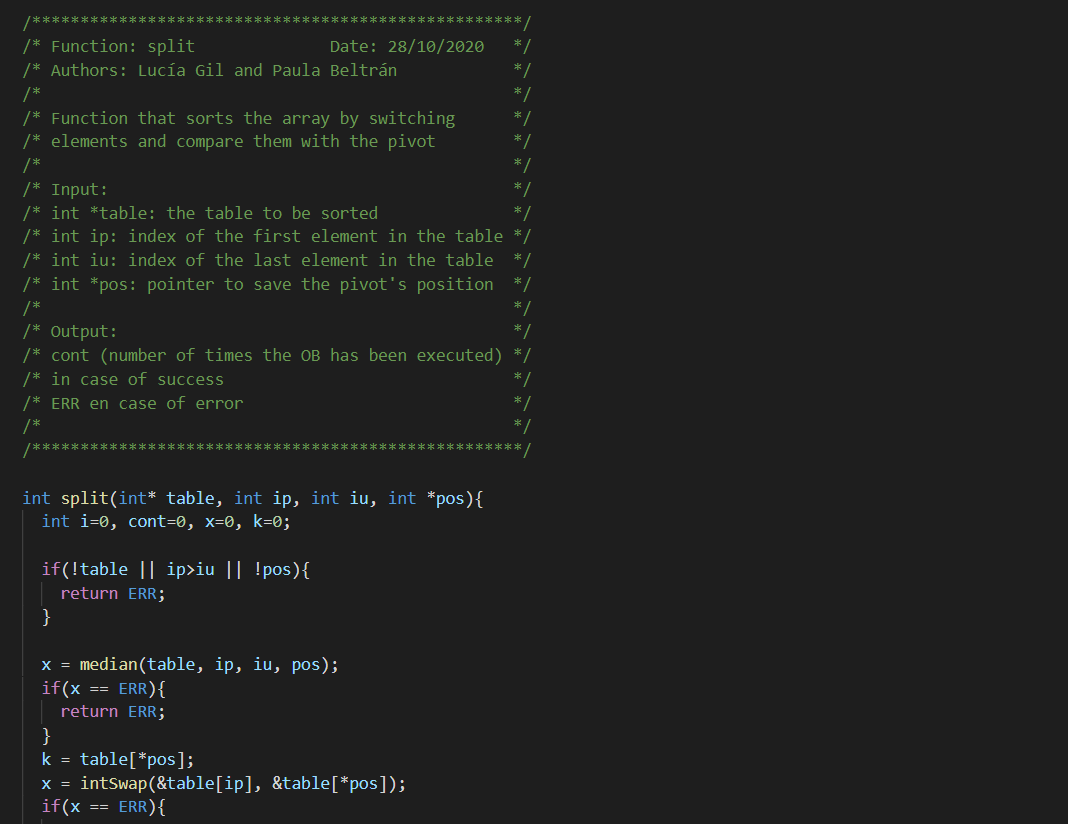
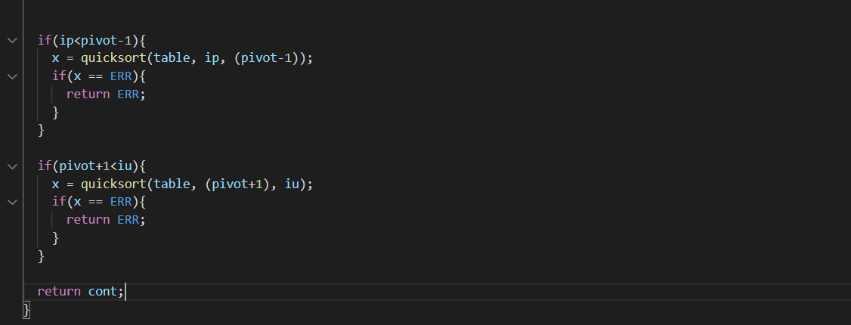
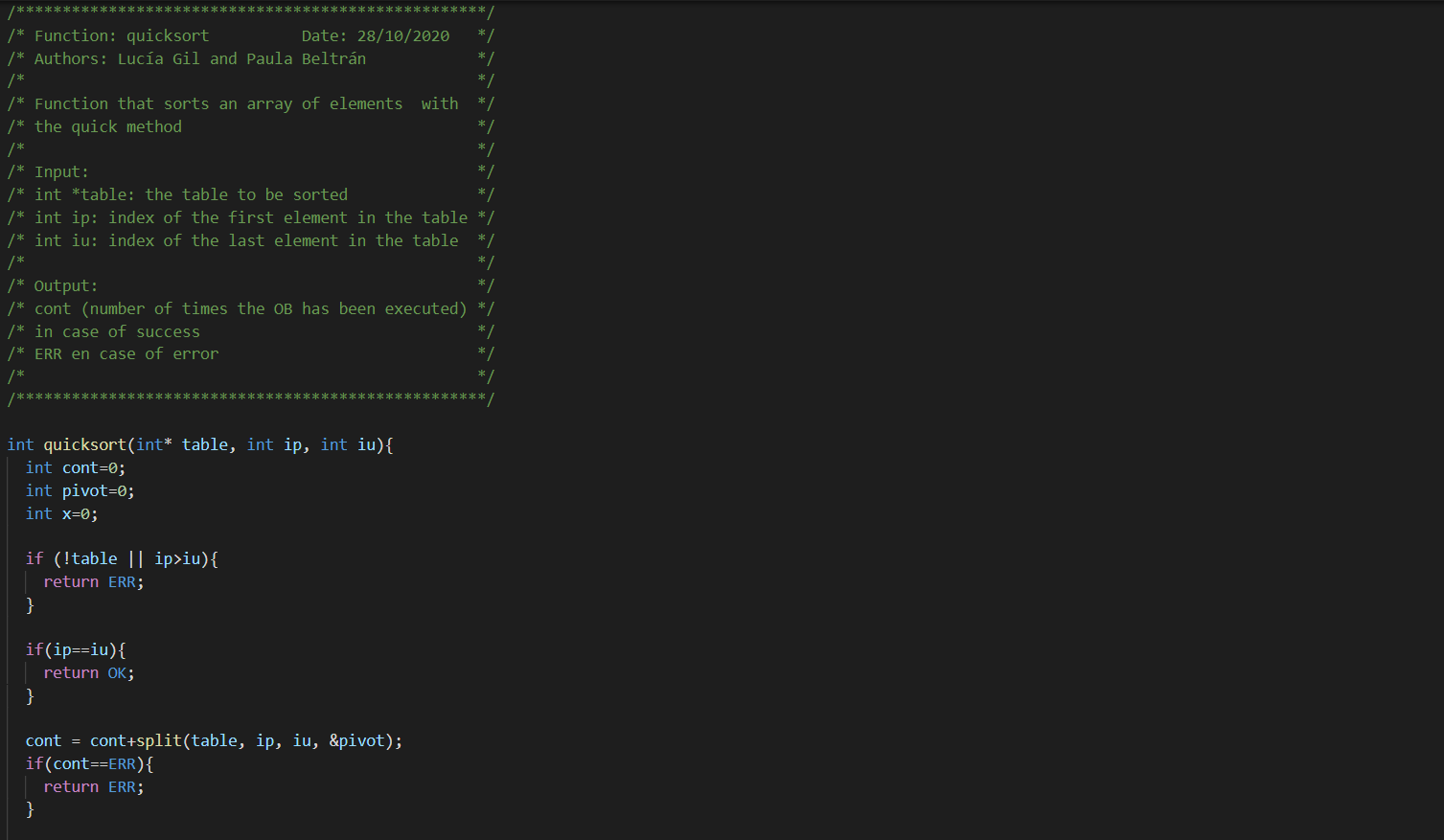
* **Exercise 1**



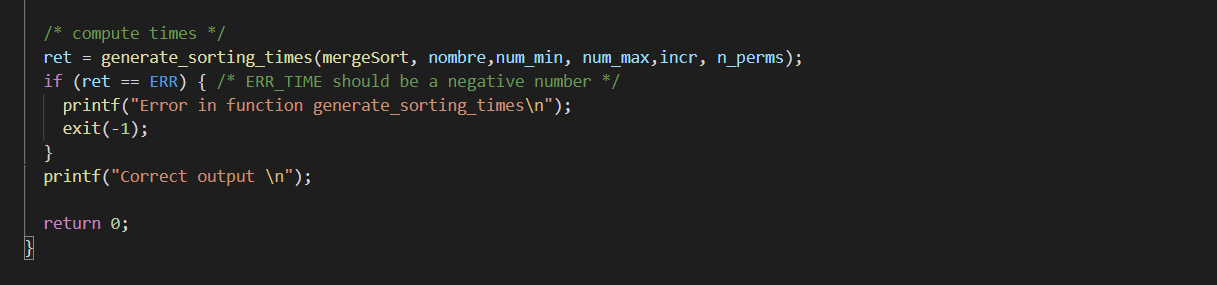
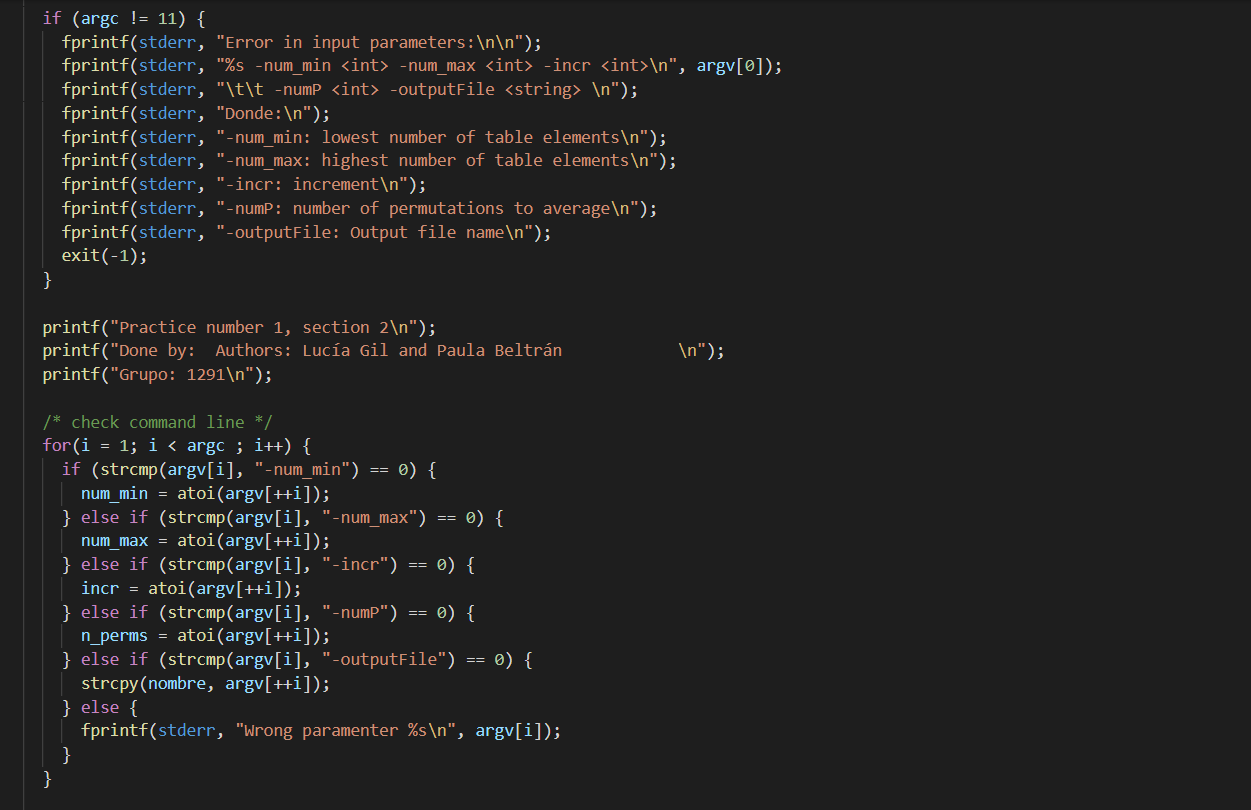
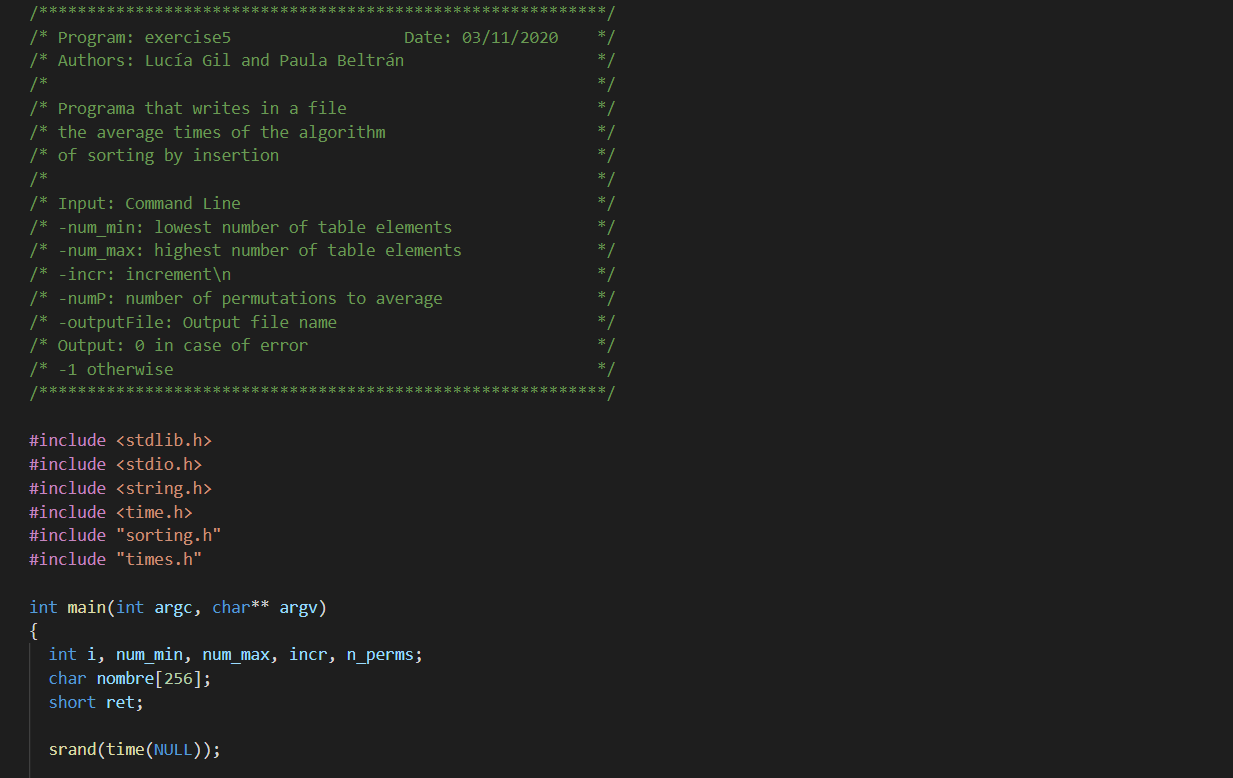
* **Exercise 2**



* **Exercise 3**



* **Exercise 4**



* **Exercise 5**

