Course: Data Structures and Algorithms
Project: Empirical Comparison of the execution time of the sorting algorithms Insertion Sort, Selection Sort, Bubble Sort, MergeSort, HeapSort, and QuickSort.

1 Instructions

- Implement all algorithms in C++, where each one must be a method taking an array of integers, and n as the number of elements in the array.

1.1 Methodology

- Each sorting algorithm must be executed for arrays of integers for the following values of n: $10,000$; $30,000$; $90,000$; $270,000$; $810,000$; $2,430,000$; $7,290,000$; $21,870,000$; and $65,610,000$.

- Integer inside the input arrays must be randomly generated with uniform distribution, using the function `rand()` from the library `stdlib.h`. The command `v[i]=rand();` generates an integer between 0 and $2,147,483,647$.

- For each size of array, the program must run each sorting method with 6 different random sequences. This way, the number of executions you must run is: (6 algorithms) $\times$ (9 sizes) $\times$ (6 sequences) $= 324$.

- To assure a fair comparison, in each "round" of executions, all algorithms must sort the same random sequence. To generate the same random sequence in different places in the code, you must use the same seed:

```
// create a sequence
int seed = 4;
srand(seed);
for(int i=0; i<n; i++) V[i] = rand();

// create a sequence that is equal to the previous one
srand(seed);
for(int i=0; i<n; i++) V[i] = rand();
```

- For this project, you must create 6 pseudorandom sequences using the same seeds:

```
seed[0] = 4;
seed[1] = 81;
seed[2] = 151;
seed[3] = 1601;
seed[4] = 2307;
seed[5] = 4207;
```

- Also, run your program for a round of executions with a sequence of $50,000$ elements in descending order.

- **Attention:** Quicksort may fail to process large sequences with a stack overflow error. In order to prevent that, modify the method `partition` to a random version, as discussed in class. This version of QuickSort must be used for all its executions.

- For each execution of a method, the program must measure the execution time in seconds. That is, each pair (method, size of array) will have 6 measurements of time: one for each pseudorandom sequence. How to measure the time?

```
time_t seconds = time(NULL);
// processing that will have its time measured
int runned_time = time(NULL) - seconds;
cout << runned_time << "\n";
```

- After finishing the program, it's time to execute it and go sleep. Just let the program run; don't do anything in the computer while the experiments are running (don't even touch the mouse).

- The program must return all data that you need to analyze. You may need to save the data in a file:

```
// write output file
void writeResults(int n, int time){
    // open file and points to next position to write an element
    ofstream myfile;
    // append the text at the final position of the file
    myfile.open("sorting.txt",ios::app);
    myfile << "Method: " << method_name << "\n";
    myfile << "Time: " << time << "\n \n";
    myfile.close();
}
```

1.2 Results

- With all data, you must now elaborate a report containing:
  - graphics that allow an easy and quick comparison among the methods;
  - a text analyzing the graphics;
  - a graphic with the average of the execution time for each method and for all the 6 different random sequences.
- It's likely that the inefficient (quadratic) methods (Bubble, Selection and Insertion) will unbalance the comparison. You must decide if these methods should be in different graphics.
- You must also decide a timeout for each experiment (like 20 minutes). This way, if some method exceeds 20 minutes trying to sort an array, you do not need to run the experiment for other arrays of the same size or bigger.
- For each pair (method, size of array) that was not completely executed, estimate how much it would take to run. Explain your estimation.