SENG201 Data and Game Structures Programming Assignment 2

Report for Part 2

Barış Cem Bayburtlu
202228009

Introduction

In our homework we were given **sorting methods in a jar** file with *censored* names (from `sort1` to `sort5`) and we were told to **perform certain tests** using them. We were told that it would be useful to use parameters that can distinguish between these methods (using arrays with **ascending**, **descending** and **random values**) as parameters while performing certain tests. By doing these tests, we were asked to **understand which sorting method is which.**

Experimental Setup

We only **needed functions** for **creating random integer arrays** and **benchmarking the methods**. Firstly, I will explain **how our random integer arrays work**:

- With **size parameter,** method **creates an array of that specified length**.
- If our function is for **random numbers**, it generates a random **3 digit random integer** and puts it into our array.
- If our function is of **ascending or descending type**, it puts it into the array starting from **1 and up to the length in descending or ascending order**.
- When it is done, **it returns us the array**.

…and it's time for the **benchmark function**, how did I do that? Let me explain:

- Our method **gets an array as a parameter** and we will use that to make our **methods** to **solve it.**
- Firstly, we need to create a **dummy array** to create a **warm up session** for our code. Because without it, our **first sorting result would be wrong**.
- After **solving** our **dummy array**, our method creates **5 different copied arrays** to make them **solve with 5 different solving algorithms**.
- `sortAndCompare` method gets the start time, **solves the array** with **specified method** and **gets the end time**. If we **extract start time from end time**, we can find **how much time we needed** to use the **command**. Our method **gets that time** and **prints it** so we can see it as well.

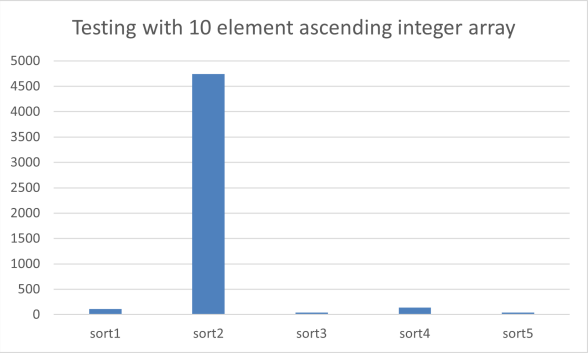
Procedure

Our testing program **begins with the generation** of **random**, **ascending**, and **descending** integer arrays in sizes **ranging** from **10 to 100,000 elements**, the program systematically **applies sorting algorithms to each combination of array type and size**. The `sortAndCompare` method serves as the core evaluator, **generating reference** and **test arrays for each scenario** within **a nested loop structure**. **The sorting algorithms**, labeled `sort1` through `sort5`, are dynamically **invoked and timed to measure execution times**. The results, which include the names **of the sorting algorithms** and **their corresponding**

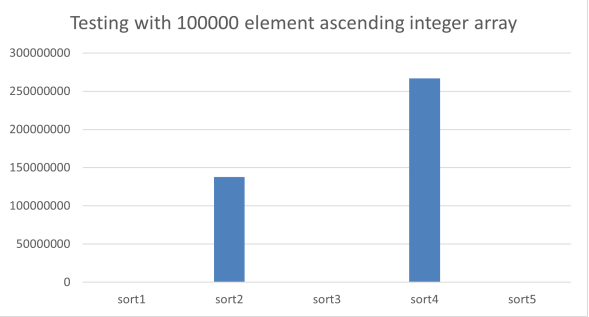
time metrics, are carefully **printed** for each array type and size, **providing detailed insight** into the **efficiency of sorting algorithms under various conditions**.

Experimental Results

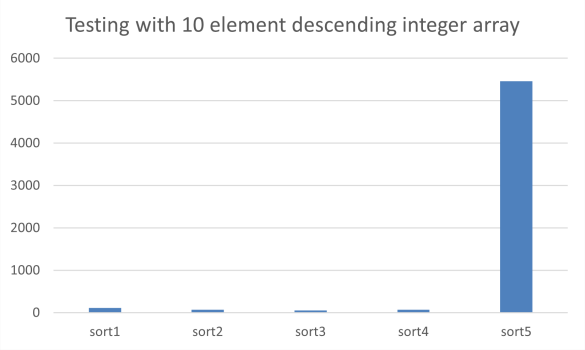
Results showed us that **some sorting algorithms** are **good** for some events **and bad** for some events. You can see our **graphs** from **our experiment below**:



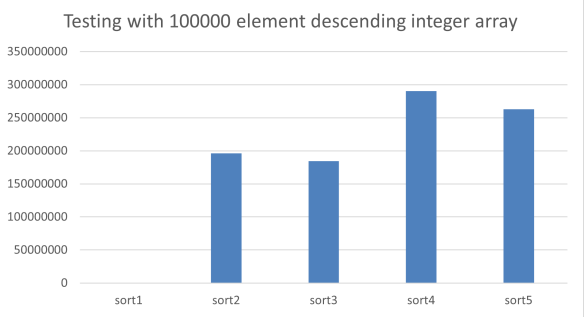
10 element ascending array test



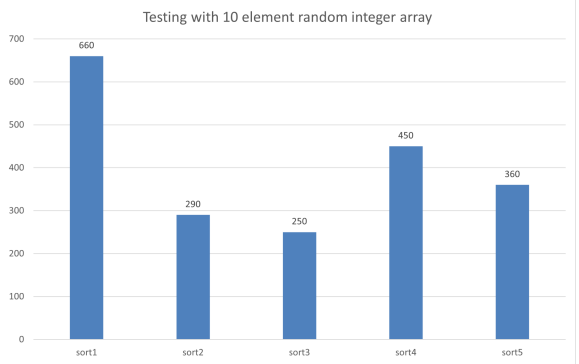
100000 element ascending array test



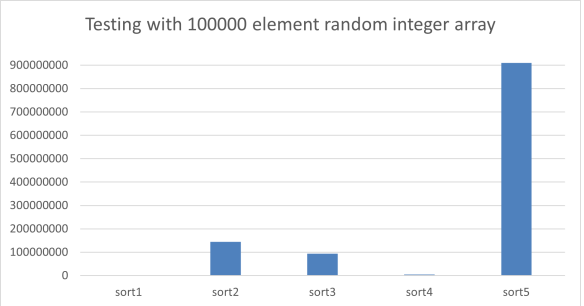
10 element descending array test



100000 element descending array test



10 element random array test



100000 element random array test

Answer

After **comparing** all 5 sorting algorithms, I get the data from my program that gives me the time, and using the **Big-O notation values of the algorithms**, I guess which **sorting algorithm is which** as follows:

- `sort1`: **Merge Sort**
- `sort2`: **Selection Sort**
- `sort3`: **Insertion Sort**
- `sort4`: **Quick Sort**
- `sort5`: **Bubble Sort**

You can come to a similar conclusion by looking at the graphs and big-o notations of sorting one by one.