CPSC-335 — Algorithms — Sort Race

Project #2 – Sort Race

Introduction

This project is to write (in HTML+JS+P5) program display the generational (i.e., **one row per 'pass'**) progress of several **different sorting algorithms (each in their own column) side-by-side**. Your program will run several algorithms in an interleaved fashion, where **each algorithm will get a turn** to perform a single Pass (approximately '**touching' all** the elements of its array to be sorted) in its Pass, and then show the newly updated array (the **Pass results**) in the next row of that algorithm's column.

Any Sort Array changes for each algorithm's Pass will be displayed on the program's HTML web page graphics **canvas** in a browser.

Algorithms

For this project, you will **run four** sorting algorithms: Insertion Sort, Gold's Poresort, Mergesort, and Quicksort. You should assume that the **setup for Mergesort** includes the "tree partitioning" into 1-element lists, as discussed in lecture (meaning that the **official first pass** for Mergesort is the merging of the 1-element sublists.

Input

The input for a Sort Race will be one of several 15-character hexadecimal strings (for example "5F7D8A1593**B**47**B**8"). You will be provided with a set of them for testing. Note, there **may be duplicate** characters in the input.

Input Ring

Once the first string has been completely sorted by the set of algorithms, then the original input hexadecimal string should be rotated rightward by one character (ie, the last string character is removed from the end and inserted in front of the first character. For example, "5F7D8A1593B47B8" would be rotated to become "85F7D8A1593B47B". Then with this new string as input, the algorithms should be reset and rerun.

This **rotate-reset-rerun** behavior should continue until the string has been rotated back into its original form. Then the entire program can stop. (I.e., for a 15-character string, there should be **15 sort runs**.)

Output

To show the race competition, you should display in an HTML page graphics canvas each of the racing algorithms, side-by-side, with each as a grid column with one cell wide by 40+ rows high. All four grid columns (one for each algo) can share the canvas if you put 1 or 2 cells between each algorithm column. Each grid cell should be **at least 20x20 pixels** wide, and as you will be putting **one hex digit in a cell**, you will have to ensure that the text digit character fits.

On the top (row 0) row, the **algorithm's name** will appear (or enough of it to fit within your 15-cell width).

On the next (row 1) the **input hex string will appear** in each sort algorithm's column. Thereafter, **for each pass of all** the algorithms, the updated order of the hexadecimal string should **be shown** in each column. This row-by-row display updating is intended to be similar to the cellular automata display of Project #1, but in several columns, one for each algorithm.

Note that you run all the algorithms for **one pass, and then display** all the single pass results in the row (for each of the columns) **BEFORE** you have the algorithm continue to run for the next pass.

Below are sample inputs that your program should be able to race. For a race, each sorting algorithm will start with the same input. You should be able to run any of the sample inputs.

Make sure that you have **enough rows to display** all the passes for the worst algorithm.

Output Delay

After each pass is displayed (for all the algorithms) you should **delay** the start of the **next pass** computations

for **between one half and one second**, in order for the audience to see and understand the results.

Also, if one algorithm finishes its sorting, then it should not further update its column display until after all the other algorithms have also finished.

Setup

Your program should select one of the sample inputs provided below at random (or you can ask the user for a hex "index" digit to pick one of them). Each is a list of 15 hex digits. Your program can include them as dedicated input data (ie, hardcoded).

Architecture

To simplify your code, and because you cannot run an algorithm through the entire sorting process at one time, we recommend that you create a **Pass object** for each algorithm, which keeps track of any between-Pass details (e.g., the array being sorted, or Mergesort might track a sublist size value) and is capable of **running one pass** of its algorithm if given the string, probably as an array, to work on.

It then becomes a bit simpler to run a single pass for each algorithm. You can do this for each algorithm by creating a **Row Mgr** (manager) object, which keeps track of what row needs to be updated, and can call each of the algorithms to get access to the pass-updated string for that algorithm. The Row Mgr could then get the results from a Pass object and display the updates to the algorithm's row.

You might also want a **Race Mgr** to setup the initial strings, and display, for each algorithm, and then call the Row Mgr enough times to get all the algorithms to finish sorting the string. The Race Mgr would then do the rotate-reset and kick off the Row Mgr again.

You **do not have to use** this architecture, but note that you cannot run an algorithm to completion and then display each of its passes, before running the next algorithm. The algorithms must each be "paused" in some fashion after each pass so that those pass results can be displayed.

Running Time

You should prepare a 1-page (at most) paper describing your analysis of the **rough running time** (not Big-O) of each algorithm as you have implemented it, not counting any GUI operations. Your basic operation for a Sort algorithm is the 2-item comparison operation. If you feel that other operations should also be included -- perhaps because they end up taking a significant (above 5% of the total) time -- then they should be included as well. Once you have an expression for running time in terms of the number of operations used (including the algorithm's setup), then show (briefly) how this running time is **converted to a Big-O** running time.

Sample Inputs

```
"05CA627BC2B6F03"
"065DE6671F040BA"
"0684FB893D5754E"
"07C9A2D183E4B65"
"09F48E7862D2616"
"1FAB3D47905C286"
```

```
"286E1D0342D7859"
"30E530C4786AF21"
"328DE4765C10BA9"
"34F2756F18E90BA"
"90BA34F0756F180"
"D7859286E2D0342"
```

Team

The team size is the same as before, but you can change team members (and team name) from the previous project if you wish.

Academic Rules

Correctly and properly attribute all third party material and references, if any, lest points be taken off.

Project Reporting Data, Readme File, Submission & Readme File, Grading

Same as for Project #1.