Assignment 1 — Algorithm Analysis

Due: October 2nd

Analysis of algorithms concerns two broad concepts: *correctness* and *complexity*. Given that one or more algorithms all produce the correct output for any given input, we are interested in the resources they each require in different situations, and the most important resource is often time.

Deliverables:

GitHub Classroom: `https://classroom.github.com/a/VDX04KAT`

Required Files: `report.pdf`, `compile.sh`, `run.sh`

Optional Files: `*.c`, `*.h`, `*.py`, `*.java`, `*.js`, `*.json`, `*.ts`, `*.clj`, `*.kt`, `*.jl`, `*.rs`

Part 1: Ground Rules

Throughout this course, you may implement your assignments in any of the following languages:

- C17[1]
- Java 21.0.0
- Clojure 1.11.1
- Julia 1.6.7
- Python 3.10.12
- Node.js 18.17.0[2]
- Kotlin 1.9.0[3]
- Rust 1.66.1

To make automated grading possible, you will also need to submit two short shell scripts: one to compile your submission, one to run it. Further details, along with sample submissions in all supported languages, can be found here: `https://github.com/csc349fall23/asgn-samples`.

Furthermore:

- For record-keeping purposes, you must submit only original source files. Any directories, pre-compiled files, or compressed files will be ignored.
- You may not use *any* third party libraries, even if they are installed on Cal Poly's UNIX servers. Note that your submission will not have Internet access at runtime.

Part 2: GitHub Classroom

In this course, you'll submit your programs through GitHub, an online host for a separate piece of software named simply "git". Git tracks changes you make to files so that you can easily undo or combine changes while working on a project. You should already be familiar with the basic features of git from previous courses, such as CSC 202 and CSC 203. If not, the following resources might be helpful:

- `https://docs.github.com/en/get-started/quickstart/set-up-git`
- `https://docs.github.com/en/get-started/using-git/getting-changes-from-a-remote-repository`
- `https://docs.github.com/en/get-started/using-git/pushing-commits-to-a-remote-repository`
- `https://docs.github.com/en/github/managing-files-in-a-repository`

GitHub Classroom allows your instructors to accept electronic submissions through GitHub.

1. Enter your Cal Poly and GitHub usernames in this form: `https://goo.gl/jTK3Bx`. This allows us to determine who should get credit for your electronically submitted assignments.
2. Accept this GitHub Classroom assignment: `https://classroom.github.com/a/VDX04KAT`. This will create a new repository for you on GitHub, providing any starter code or sample tests. You will push your submissions to this repository, and automated grading feedback will appear as associated issues.

[1] The compiler will be GCC 11.4.0, which also supports the upcoming C2x standard.
[2] TypeScript 5.1.6 is also installed.
[3] This is JVM-backed Kotlin, not Kotlin/Native.

Part 3: Pseudocode

Recall that there exist several algorithms that sort a given sequence A of n elements. Arguably, the most naïvely intuitive among them is SelectionSort, which repeatedly *selects* the smallest element from among the as-yet-unsorted elements, having complexity $O(n^2)$:

SelectionSort$(A = (a_0, a_1, \ldots, a_{n-1}))$

Input: A finite, non-empty sequence A of n integers
Output: A permutation of A sorted in ascending order

1: **let** B be an empty sequence
2: **while** A is not empty **do**
3: **let** x be $\min\{A\}$
4: **let** A be $A - (x)$ and B be $B +\!\!+ (x)$
5: **return** B

Naïve intuition, however, is not necessarily efficient. MergeSort instead recursively sorts the two halves of a unsorted sequence, then *merges* the sorted halves back together. This is provably the best possible comparison-based sorting algorithm, and has complexity $O(n \log n)$:

MergeSort$(A = (a_0, a_1, \ldots, a_{n-1}))$

Input: A finite, non-empty sequence A of n integers
Output: A permutation of A sorted in ascending order

1: **let** mid be $\lfloor n/2 \rfloor$
2: **if** $n = 1$ **then**
3: **return** A
4: **else**
5: **return** Merge(MergeSort$((a_0, a_1, \ldots, a_{\text{mid}-1}))$,
MergeSort$((a_{\text{mid}}, a_{\text{mid}+1}, \ldots, a_{n-1}))$)

(see lecture slides for the definition of Merge, omitted for brevity)

That being said, depending on the elements being sorted, it may not be necessary to compare them to one another at all. CountingSort, assuming that elements are integers[4], *counts* the occurrences of each unsorted element, then reproduces that many elements in sorted order. It has complexity $O(n + (\max\{A\} - \min\{A\}))$:

CountingSort$(A = (a_0, a_1, \ldots, a_{n-1}))$

Input: A finite, non-empty sequence A of n integers
Output: A permutation of A sorted in ascending order

1: **let** T be an empty dictionary and B be an empty sequence
2: **for** i **from** $\min\{A\}$ **to** $\max\{A\}$ **do**
3: **let** $T(i)$ be 0
4: **for** all $a \in A$ **do**
5: **let** $T(a)$ be $T(a) + 1$
6: **for** i **from** $\min\{A\}$ **to** $\max\{A\}$ **do**
7: **for** j **from** 1 to $T(i)$ **do**
8: **let** B be $B +\!\!+ (i)$
9: **return** B

In your programming language of choice (see Part 1), implement these three algorithms.

[4] CountingSort requires that the unsorted elements fall within some well-defined range, so that it can iterate through all possible elements when reconstructing the sorted sequence.

Part 4: Correctness

You may assume that all sequences will contain one or more comma-separated integers. For example:

```
2, -1, 9, 8, 5, -3, 0, 8
```

Your program must accept as a command line argument the name of a file containing a sequence as described above, then print to `stdout` the running times and results of applying the three algorithms. For example:

```
>$ ./compile.sh
>$ ./run.sh in1.txt
Selection Sort (0.01 ms): -3, -1, 0, 2, 5, 8, 8, 9
Merge Sort     (0.02 ms): -3, -1, 0, 2, 5, 8, 8, 9
Counting Sort  (0.01 ms): -3, -1, 0, 2, 5, 8, 8, 9
```

Your running times will likely vary from those above. Aside from that, your program will be tested using `diff`, so its printed output must match *exactly*.

Part 5: Complexity

Record the running times of each algorithm for sequences between 1000 and 50000 elements in length, in increments of 1000, containing random integers from the range $[-10000000, 10000000]$. That is to say:

1. Randomly generate a sequence of 1000 integers. Run the three sorts; record their running times.
2. Randomly generate a sequence of 2000 integers. Run the three sorts; record their running times.
3. Randomly generate a sequence of 3000 integers. Run the three sorts; record their running times.

⋮

50. Randomly generate a sequence of 50000 integers. Run the three sorts; record their running times.

Plot these timing results, with the length of the sequence on the horizontal axis and the running time on the vertical axis — you should have a single scatter plot with 150 data points. You may certainly automate this process[5], however, note that this should not be a part of the program specified in Part 4.

Assuming efficient implementations, you should observe that the plots of SelectionSort and MergeSort curve upwards, the former growing noticeably faster than the latter[6]. In contrast, the plot of CountingSort should be close to a horizontal line — its bottleneck is the integers' range, not the sequences' lengths.

Part 6: Submission

The following files are required and must be pushed to your GitHub Classroom repository by the deadline:

- `report.pdf` — Plots of the three sorting algorithms' running times for varying lengths of randomized sequences, as described in Part 5.
- `compile.sh` — A Bash script to compile your submission (even if it does nothing), as specified.
- `run.sh` — A Bash script to run your submission, as specified.

The following files are optional:

- `*.c`, `*.h`, `*.py`, `*.java`, `*.js`, `*.json`, `*.ts`, `*.clj`, `*.kt`, `*.rs` — The source code of a working program to compare three sorting algorithms, as specified.

Any files other than these will be ignored.

[5] One possibility is to have a script write the running times to a CSV file, which can then be plotted within spreadsheet programs such as Microsoft Excel or Google Sheets.

[6] This assumes a recursive implementation of MergeSort and Merge; depending on your chosen programming language, iterative implementations will likely be substantially faster, which is not wrong, but unfortunately less illustrative.