Dept. of Computer Science, University of California, Davis
ECS60 Instructor: Rob Gysel 1/18/18

Programming Assignment #1*
Due date: 1/27/18 11:59pm

Programs are to be submitted to Kodethon by the due date. Refer to the syllabus for group work policies. The Kodethon autograder is still being developed, but it is possible to complete the assignment without it. You may use Rust, Java, or C++ for your code. However, I strongly recommend you use C++ for this program because I have provided skeleton C++ code for the project, and you do not need to do any object-oriented programming for this project.
Students that work alone for all programs a receive a B- or better on the programs will receive 1% extra credit at the end of the quarter. Programs submitted up to 24 hours late will still be accepted but incur a 10% grade penalty.

1 Overview & Learning Objectives

In this program you will study and experiment with InsertionSort, MergeSort, and QuickSort. There are multiple objectives of this assignment:

1. introduce the JSON (JavaScript Object Notation) and CSV (comma separated values) data formats, which are frequently used in professional settings,
2. examine the wallclock running time of InsertionSort, MergeSort, and QuickSort,
3. understand fine-grained details of algorithm analysis (wallclock vs. worst-case Big-O vs. average-case Big-O),
4. introduce automated testing, and
5. provide practice with open-ended project descriptions (this is an important job skill).

*Last updated January 18, 2018

2 Data Formats

JSON

We will call each array that is to be sorted a *sample*. You will be running your programs on one or two input files that contain a number of samples. These files will be formatted in *JavaScript Object Notation* (or JSON). This data format, and many like it, is frequently used in industry, and *every* computer science student should be exposed to it. These types of formats are frequently used for web requests and to pass data from one programming environment (ex. Objective-C) to another (ex. Swift).

JSON files for this program will be formatted like this (this is `SampleExample.json` in the files provided):

```
{
  "Sample1": [-319106570,811700988,1350081101,1602979228],
  "Sample2": [-319106570,811700988,797039,-1680733532],
  "metadata": {
    "arraySize":4,
    "numSamples":2
  }
}
```

`Sample1` and `Sample2` are arrays of integers. `metadata` is data about the data `Sample1` and `Sample2`: there are two samples and each is an array of size 4.
A JSON object consists of a collection of *key / value* pairs. It turns out you are already familiar with the concept of key / value pairs: an array can be viewed as a collection of key / value pairs where the keys are in the range $0, 1, \ldots, n-1$ for an array with n elements. For example, the array $A = [-3, 9, 4]$ has $A[1] = 9$, and here the "key" is 1 and the "value" is 9.
In the example above, the object has three keys: `Sample1`, `Sample2`, and `metadata`. The values of `Sample1` and `Sample2` are arrays with 4 integers. The value of `metadata` is a JSON object itself, and it has two keys: `arraySize` with value 4 and `numSamples` with value 2.

Reading (or *parsing*) a JSON object from a JSON file is a difficult task: do not write your own code for this. You must use a 3rd-party library for this task, and I have used https://github.com/nlohmann/json in the skeleton code, which is included in the project files as `include/json.hpp`. The github repo contains documentation, but for brevity, here are the operations with this library you will need to be familiar with:

Reading a JSON file: Use `ifstream` and the stream operator, like this:

```
std::ifstream file;
file.open(filename);
nlohmann::json jsonObject;
// Store the contents filename into jsonObject
if (file.is_open()) {
  file >> jsonObject;
}
```

Value access: To access a value associated with a key, use square braces along with the key. For example, if we read the file `SampleExample.json` to `jsonObject`, then `jsonObject["Sample1"]` returns the array

```
[-319106570,811700988,1350081101,1602979228]]
```

If we write `int n = jsonObject["metadata"]["arraySize"]`, then `n` is 4.

Iterating over keys: You can iterate over all key / value pairs at the *top level* by doing:

```
for (auto itr = jsonObect.begin(); itr != jsonObject.end(); ++itr) {
  std::cout << "key: " << itr.key() << " value: " << itr.value() << '\n';
}
```

This would print:

```
key: Sample1 value: [-319106570,811700988,1350081101,1602979228]
key: Sample2 value: [-319106570,811700988,797039,-1680733532]
key: metadata value: {"arraySize":4,"numSamples":2}
```

Note that the value of the `metadata` key is a JSON object itself, and this object has keys `arraySize` and `numSamples` that are not iterated over in the above for loop. The type of `itr.key()` is a string and the type of `itr.value()` varies. When the key is `Sample1`, the value is a JSON object representing an array of integers that you can iterate over like this:

```
for (auto arrayItr = jsonObect["Sample1"].begin();
     arrayItr != jsonObject.["Sample1"].end();
     ++arrayItr) {
  std::cout << arrayItr << " ";
}
```

This would print:

```
-319106570 811700988 1350081101 1602979228
```

CSV

Measurements of the sorting algorithms will be recorded in CSV files. This data format, and many like it, are frequently used in industry, and *every* computer science student should be exposed to it. There are many types of "CSV" files[1], and we describe one of the simplest versions here.

A *comma separated values* file, or CSV file, consists of a *header row* on the first line of the file, followed by *data records* on subsequent lines. The header row consists of a collection of *column names* separated by commas. The data records consists of data (this may be strings, numbers, etc.) separated by commas. For example, the contents of a CSV file for student information:

```
Name,ID,email,year
AlexGrothendieck,423518,alexg@myuni.edu,3
EmmyNoether,4245534,emmynoether@myuni.edu,2
JuliaRobinson,23634563,jrob@myuni.edu,2
MartinDavis,2359830,mdavis@myuni.edu,1
```

In the above example, the header row consists of four column names, `Name`, `ID`, `email`, and `year`. Each data record therefore has four data, and the order is significant: on line 2 (the first data record), the `Name` is `AlexGrothendieck`, the `ID` is `423518`, the `email` is `alexg@myuni.edu`, and the `year` is `3`.

3 Executables

Because multiple languages are allowed, your program must contain shell scripts to run your executables. I recommend writing bash scripts, and have provided examples of these[2]. The following is a list of executables and scripts you must turn in. I am listing them in the order I suggest you work on them.

Executable #1

Script: `sortedverification.sh`

Usage: `sortedverification.sh inputFile.json`

This script will call a C++, Java, or Rust program that takes a command-line argument `inputFile.json`, and verify that each sample is a sorted array. If a sample array is not sorted, there must be some position i such that the i^{th} element is equal to or larger than the $i+1^{\text{st}}$ element. We call this a *consecutive inversion.* For example, if $A = [-2, 0, 3, 2, 5]$ there is a consecutive inversion at location $i = 2$ because $A[2] = 3 \geq 2 = A[3]$. For example, the samples

[1]For example, there are CSV formats that allow tab delimiters instead of commas.
[2]My bash scripts are 2 lines long.

$$\texttt{Sample1} = [-1641818748, 1952682320, -195384256, -1702150187], \text{ and}$$
$$\texttt{Sample2} = [-683761375, -406924096, -362070867, -592214369]$$

are defined by the following input file `SampleExample.json`:

```
{
  "Sample1": [-319106570,811700988,1350081101,1602979228],
  "Sample2": [-319106570,811700988,797039,-1680733532],
  "metadata": {
    "arraySize":4,
    "numSamples":2
  }
}
```

`Sample2` has consecutive inversions at index 1 and 2, and running

```
./sortedverification.sh SampleExample.json
```

outputs a JSON file whose contents are printed to `stdout`:

```
{"Sample2":{"ConsecutiveInversions":{"1":[811700988,797039],"2":
[797039,-1680733532]},"sample":[-319106570,811700988,797039,-1680733532]},
"metadata":{"arraySize":4,"file":"SampleExample.json","numSamples":2,
"samplesWithInversions":1}}
```

The above text is printed on a single line to stdout (use output redirection to save to a file). To make it readable, use `https://jsonformatter.curiousconcept.com/` to obtain a *human readable* version of this string (i.e. whitespace is added for clarity):

```
{
   "Sample2":{
      "ConsecutiveInversions":{
         "1":[
            811700988,
            797039
         ],
         "2":[
            797039,
            -1680733532
         ]
      },
      "sample":[
         -319106570,
         811700988,
         797039,
         -1680733532
      ]
   },
   "metadata":{
      "arraySize":4,
      "file":"SampleExample.json",
      "numSamples":2,
      "samplesWithInversions":1
   }
}
```

You can also view this output in the file `SV-SampleExample.json`.
`Sample1` has no inversions so its data is not printed to the JSON output above. Notice that if the consecutive inversions of a sample are added to the JSON object, the sample data (the array) is also added to the JSON object.

Executable #2

Script: `consistentresultverification.sh`

Usage: `sortedverification.sh inputFile.json`.

This script will call a C++, Java, or Rust program that takes two command-line arguments `inputFile1.json` and `inputFile2.json`, and verify that these files represent the same samples.
I have copied `SampleExample.json` to `AlmostSampleExample.json` and modified the second and third entries of `Sample1` in `AlmostSampleExample.json`. These differences are output when I run

```
./consistentresultverification.sh SampleExample.json AlmostSampleExample.json
```

The program outputs the following to stdout (to save to a file, use output redirection): and the output, after making it human readable using the above link and doing some hand-modification to place arrays on the same line is:

```
{
  "Sample1": {
    "Mismatches": {
      "1": [ 811700988, 8117009 ],
      "2":[ 1350081101, 13500811 ]
    },
    "sample1": [ -319106570, 811700988, 1350081101, 1602979228 ],
    "sample2": [ -319106570, 8117009, 13500811, 1602979228 ]
  },
  "metadata": {
    "samplesWithConflictingResults": 1
  },
  "sample1": {
    "metadata": {
      "arraySize": 4,
      "file": "SampleExample.json",
      "numSamples": 2
    }
  },
  "sample2": {
    "metadata": {
      "arraySize": 4,
      "file": "AlmostSampleExample.json",
      "numSamples": 2
    }
  }
}
```

You can also view this output in the file `CRV-Output-SampleExample-AlmostSampleExample.json`.

Executable #3

Script: `timealgorithms.sh`
Usage: `timealgorithms.sh inputFile.json`

This script will call a C++, Java, or Rust program that takes a command-line argument `inputFile.json`, runs InsertionSort, MergeSort, and QuickSort on all samples in the file, measures various statistics, and prints the to a CSV file.
Do not implement your own versions of the algorithm. Use the zyBooks code from Ch. 1.7-1.9 for your code. Slight variations of these algorithms will not work with the autograder, as you will be gathering specific statistics about how the algorithms behave. I've implemented the sorting algorithms in C++ for you, so I recommend that you start with my implementations (e.g. `insertionsort.cpp`, `mergesort.cpp`, and `quicksort.cpp`)
Collect the following statistics:

Running Time: i.e. wallclock time. I used `time` from the `<ctime>` library for this.

Number of Comparisons: A count of how often an algorithm compares at least one element from the array it is sorting to something else. The following lines of code both count as a single comparison:

```
(*numbers)[i] < (*numbers)[j]
(*numbers)[i] < a
```

You will need to add lines of code to the sorting algorithms to achieve this.

Number of memory accesses: A count of how often an algorithm accesses the array it is sorting. In the above example, the first line counts as *two* memory accesses while the second line counts as *one*.

These statistics are then printed to the screen in CSV format (to save to a file, use output redirection). Your header row for your CSV file must have the following column names (see `TimeOutputExample.csv` for an example):

Sample: The name of the sample that pertains to this row's statistics (e.g. `Sample1`)

InsertionSortTime: The wallclock time of running `InsertionSort` on this row's sample

InsertionSortCompares: The number of compares used when running `InsertionSort` on this row's sample

InsertionSortMemaccess: The number of memory accesses when running `InsertionSort` on this row's sample

MergeSortTime: The wallclock time of running `MergeSort` on this row's sample

MergeSortCompares: The number of compares used when running `MergeSort` on this row's sample

MergeSortMemaccess: The number of memory accesses when running `MergeSort` on this row's sample

QuickSortTime: The wallclock time of running `QuickSort` on this row's sample

QuickSortCompares: The number of compares used when running `QuickSort` on this row's sample

QuickSortMemaccess: The number of memory accesses when running `QuickSort` on this row's sample

4 Compilation

Submit a script called `compile.sh` that will compile all of your source code once it is run. For example, in the skeleton code I have provided, I have used `compile.sh` which uses a `Makefile` to build my code.

5 Take the Program Quiz

A quiz will be released on Canvas (forthcoming) that you are to take after completing the project. You must take your own quiz, even if you have a programming partner. You will have unlimited attempts but will not see your score until after the due date (to allow for changing your answers without allowing gamification of the quiz). This quiz will be worth 15% of your programming grade.