Program #2

Programs must be written in C++ and are to be submitted using handin on the CSIF by the due date using the command:

```
handin cs36c P2 file1 file2 ... fileN
```

Your programs must compile and run on the CSIF. Use `handin` to submit *all* files that are required to compile (even if they come from the prompt). Programs that do not compile with `make` on the CSIF will lose points and possibly get a 0.
There are 100 example inputs along with their expected outputs in the directory as well as the autogragrader to download.
The `Example*-TimedAlgorithmsSolution.csv` have 0's for the running time fields (since this is variable, you do not need to match it), and your solutions should have the actual running time in these fields.

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. use your automated tests to detect code with bugs.

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. In this class, we will use the third-party library https://github.com/nlohmann/json. See `JSON.pdf` on Canvas for a tutorial on JSON and this library.

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

Executable #1

Executable Name: `sortedverification`
Source: `sortedverification.cxx`
Usage: `sortedverification file.json`

This program takes the name of a JSON file as a command-line argument `file.json` that represents the output of a sorting algorithm and verifies that

[1] For example, there are CSV formats that allow tab delimiters instead of commas.

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 $A[2] = 3 > 2 = A[3]$. For example, the samples

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

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 SampleExample.json
```

prints the contents of a JSON object to the screen (i.e. 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
   }
}
```

`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

Executable Name: `consistentresultverification`
Source: `consistentresultverification.cxx`
Usage: `consistentresultverification file1.json file2.json`.

This program takes two command-line arguments `file1.json` and `file2.json` that contain JSON objects representing the output of two sorting algorithms, and verifies that these files represent the same samples or reports their differences.
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:

```
{
  "Sample1": {
    "AlmostSampleExample.json": [
      -319106570,
      8117009,
      13500811,
      1602979228
    ],
    "Mismatches": {
      "1": [
        811700988,
        8117009
      ],
      "2": [
        1350081101,
        13500811
      ]
    },
```

```
    "SampleExample.json": [
      -319106570,
      811700988,
      1350081101,
      1602979228
    ]
  },
  "metadata": {
    "File1": {
      "arraySize": 4,
      "name": "SampleExample.json",
      "numSamples": 2
    },
    "File2": {
      "arraySize": 4,
      "name": "AlmostSampleExample.json",
      "numSamples": 2
    },
    "samplesWithConflictingResults": 1
  }
}
```

The metadata field now contains information about the files being read in. The key `Sample1` has information because it differs between `SampleExample.json` and `AlmostSampleExample.json`. Its value contains the sample from each file along with the differences between the asmples. Differences are listed in the `Mismatches` key, which contains a list of positions that mismatch and their contents. Note that the key-value pair `"1": [ 811700988, 8117009 ]` exists because the second entry of `Sample1` in `SampleExample.json` is `811700988` and `AlmostSampleExample.json` is `8117009`.

Executable #3

Executable Name: `timealgorithms` **Source:** `timealgorithms.cxx`
Usage: `timealgorithms file.json`

This program takes the name of a JSON file as a command-line argument (`file.json`) that represents a collection of arrays to be sorted (an input file for the sorting algorithms) and runs InsertionSort, MergeSort, and QuickSort on all samples in the file, measures various statistics, and prints these statistics to a CSV file.
Do not implement your own versions of the algorithm. Use the code given in `inerstionsort.cpp`, `mergesort.cpp`, and `quicksort.cpp`. Slight variations of these algorithms will not work with the autograder, as you will be gathering specific statistics about how the algorithms behave.
Collect the following statistics:

Running Time: i.e. wallclock time. I used `clock` and `CLOCKS_PER_SEC` from the `<ctime>` library for this. The autograder won't check this field; so the only important part about this field is your ability to get a sense of which algorithm is fastest on a given input.

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. If necessary, take lazy evaluation into account.

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*. If necessary, take lazy evaluation into account.

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 Files To Submit

Submit the following files for your program: `Makefile`, `mergesort.cpp`, `mergesort.h`, `consistentresultverification.cxx`, `quicksort.cpp`, `createdata.cxx`, `quicksort.h`, `insertionsort.cpp`, `sortedverification.cxx`, `insertionsort.h`, `timealgorithms.cxx`. Your program must compile on the CSIF using `make` without warnings. Code that compiles with warnings will lose 5%.