Programming Assignment One: Sorting

Out: Sep. 22, 2016; Due: Oct. 8, 2016.

I. Motivation

1. To give you experience in implementing various sorting algorithms.
2. To empirically study the time efficiency of different sorting algorithms.

II. Programming Assignment

You are asked to implement six sorting algorithms: bubble sort, insertion sort, selection sort, merge sort, quick sort using extra array for partitioning, and quick sort with in-place partitioning. They sort integers in **ascending order**. They are combined into a single program. Based on the user specification, a corresponding sorting algorithm will be called.

1. Input format

You will read the sorting task from the **standard input**. (For the ease of testing, you can write each test case in a file and then use Linux file redirection function "<" to read from the file.)

The first line is an integer, which specifies the sorting algorithm you should call. The integer can take six values: 0 for bubble sort, 1 for insertion sort, 2 for selection sort, 3 for merge sort, 4 for quick sort using extra array for partitioning, and 5 for quick sort with in-place partitioning. Other values are illegal, but you can assume that the user will not input illegal values.

The second line specifies the number of integers you are asked to sort. Let that number be *N*. Then the following *N* lines list the *N* integers to be sorted.

An example of input is

```
3
5
-1
-3
2
```

```
0
4
```

This example calls merge sort to sort 5 integers -1, -3, 2, 0, and 4 in ascending order.

2. Output Specification

Your program writes the sorted result to the **standard output**. Each line lists one number. For the above example, the output looks like

```
-3
-1
0
2
4
```

III. Comparison of the Sorting Algorithms

We also ask you to study the performances of these six sorting algorithms. To do this, you should first generate different arrays of random integers with different sizes. Then you should apply your sorting algorithms to these arrays. Finally, you should plot a figure showing the runtime of each algorithm versus the array size. For comparison purpose, you should plot all six curves corresponding to the six sorting algorithms in the same figure. (You do not need to upload the source codes for this comparison program.)

Hint:

1. You may want to write another program that calls the core sorting procedures to do this study.
2. You can use mrand48() to generate integers uniformly distributed in the range [-2^{31}, $2^{31}-1$]. See http://linux.about.com/library/cmd/blcmdl3_mrand48.htm
3. You can use clock() function to get the runtime. See http://www.cplusplus.com/reference/ctime/clock/
4. Although the major factor that affects the runtime is the size of the input array, however, the runtime for an algorithm may also weakly depend on the detailed input array. Thus, for each size, you should generate a number of arrays of that size and obtain the mean runtime on all these arrays. Also, for fair comparison, the same set of arrays should be applied to all the algorithms.
5. You should try at least 5 representative sizes.

IV. Implementation Requirements and Restrictions

- You must make sure that your code compiles successfully on a Linux operating system.
- Your compiled program (i.e., the program described in Section II, not in Section III) should be named as `p1` exactly.
- You may not leak memory in any way. To help you see if you are leaking memory, you may wish to call `valgrind`, which can tell whether you have any memory leaks. (You need to install `valgrind` first if your system does not have this program.) The command to check memory leak is:

  ```
  valgrind --leak-check=full <COMMAND>
  ```

  You should replace `<COMMAND>` with the actual command you use to issue the program under testing. For example, if you want to check whether running program

  ```
  ./p1 < input.txt
  ```

  causes memory leak, then `<COMMAND>` should be "`./p1 < input.txt`". Thus, the command to check memory leak is

  ```
  valgrind --leak-check=full ./p1 < input.txt
  ```

- You may `#include <iostream>`, `<fstream>`, `<sstream>`, `<string>`, `<cstdlib>`, `<climits>`, `<ctime>`, and `<cassert>`. No other system header files may be included, and you may not make any call to any function in any other library.

V. Compiling

In order to let us test your code automatically, we ask you to provide a `Makefile` for us. This is a special file used for compiling code in Linux environment. **You should name this file exactly as "`Makefile`"**. A `Makefile` consists of a set of **rules** of how to compile a final executable program. Each rule has the following format:

```
<Target>: <Dependency>
[Tab] <Command>
```

`<Target>` is the target you want to "make", which depends on a list of files shown in `<Dependency>`. `<Command>` is the command to "make" the `<Target>`. **Note**: There must be a **tab** before the `<Command>`; otherwise, it is a syntax error.

We ask you to write your `Makefile` in the simplest form, that is, a `Makefile` containing just one rule, which generates the final executable program. For example, suppose that you want to compile a program named `helloworld` from two `.cpp` files, `hello.cpp` and `world.cpp`, and two `.h` files, `hello.h` and `world.h`. The target is `helloworld`, which depends on four files: `hello.cpp`, `world.cpp`, `hello.h`, and `world.h`. The command for compiling the program is:

```
     g++ -Wall -o helloworld hello.cpp world.cpp
```

Putting them together, you would write one rule in your `Makefile` like:

```
helloworld: hello.cpp world.cpp hello.h world.h
	g++ -Wall -o helloworld hello.cpp world.cpp
```

You should note that there is a tab before the command!

You should put your `Makefile` in your working directory. Once you have written your own `Makefile`, then you can **type "`make`"** in the terminal to compile the program.

A demo of `Makefile` is put in the Programming-Assignment-One-Related-Files.zip. Try it!

For more information about the Makefile, you can read some online tutorials. For example, http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/

VI. Testing

To make sure that the program works, you should test each sorting algorithm you write.

VII. Submitting and Due Date

You should submit all the source code files for the program described in Section II, a `Makefile`, and a report of the runtime comparison of different sorting algorithms. The

`Makefile` compiles a program named `p1`. The report should be in doc, docx, or pdf format. These files should be submitted via the online judgment system. See announcement from the TAs for details about submission. The submission deadline is 11:59 pm on Oct. 8th, 2016.

VIII. Grading

Your program will be graded along five criteria:

1. Functional Correctness
2. Implementation Constraints
3. General Style
4. Performance
5. Report on the performance study

Functional Correctness is determined by running a variety of test cases against your program, checking your solution using our automatic testing program. We will grade Implementation Constraints to see if you have met all of the implementation requirements and restrictions. In this project, we will also check whether your program has memory leak. For those programs that behave correctly but have memory leaks, we will deduct some points. General Style refers to the ease with which TAs can read and understand your program, and the cleanliness and elegance of your code. Part of your grade will also be determined by the performance of your algorithm. We will test your program with some large test cases. If your program is not able to finish within a reasonable amount of time, you will lose the performance score for those test cases. Finally, we will also read your report and grade it based on the quality of your performance study.