Project 2, Part B
CS 303/503 Algorithms

Fall 2012

Due by: 8:00 a.m., Monday, November 12, 2012

This programming project is subject to the CSCI 303/503: Regulations for Programming Projects found at `www.cs.wm.edu/~va/CS303/project_regs.html`.

For this project you are required to implement three new public methods, along with three new protected methods, for the class `GroupOfNumbers`. The purpose will be to enable you to do some experiments to better understand how the quicksort algorithm/implementation works with different pivot selection rules.

Choosing the pivot for quicksort using other rules

You are to investigate the effect of using rules other than "median of three" to choose the pivot used by `quicksort`. You are to implement three new public methods that allow you to investigate the three simplest rules:

1. choose the first element in the partition as the pivot,
2. choose the middle element in the partition as the pivot, and
3. choose the last element in the partition as the pivot.

It is easy enough to find each of these pivots since they are defined exactly by their positions, using the usual formula for finding the middle:

$$\texttt{middle} = \left\lfloor \frac{\texttt{left} + \texttt{right}}{2} \right\rfloor .$$

The challenge here is to deal with the fact that changing the strategy for choosing the pivot has an effect on the way you scan the current partition to set up the new, smaller, partitions for the recursive calls. For this reason, you have three new protected methods, corresponding to each of the three new public methods, that allow you to make the necessary changes to ensure that the final ordering is correct and that in the process of sorting the group you do not generate any memory access errors. *Read the last sentence again, carefully.* Not only must you obtain the correct ordering for any input, you must make sure that during the course of identifying the two new partitions you do not wind up "running off the end" of the array `_group`. You also must make sure that every problem you hand to quicksort is well-defined.

When you are done implementing these three new pivoting rules within quicksort, you should have a better appreciation of the symbiosis between Weiss' implementation of the median of three rule for choosing the pivot and his implementation of quicksort.

Timing the sorting algorithms

Once they are working correctly, take your implementations of insertion sort, heapsort, mergesort, and quicksort (using four different pivot selection rules) and time how long it takes to sort a `GroupOfNumbers` object when the numbers in the group are randomly generated, sorted in order, sorted in reverse, and identical.

Assume in all cases that the `GroupOfNumbers` objects contains $250,000$ numbers. You know how to initialize `GroupOfNumbers` objects with randomly generated entries; to ensure reasonably representative timing results, make sure that the range for the pseudo random number generator is from 0 to 250000. It is easy to generate files with $250,000$ entries that are either sorted in order, sorted in reverse, or identical and you have a constructor for `GroupOfNumbers` objects that allows you to initialize a group using numbers read in from a file.

In effect, you are to fill out Table 1. As you are filling in the table, ask yourself if the timings you are reporting in Table 1 are consistent with the the analytical time complexities for each algorithm applied to that particular type of input. If not, you need to check your implementation!

Sorting algorithm (including pivot rules for quicksort)	randomly generated	sorted in order	sorted in reverse	identical elements
insertion sort				
heapsort				
mergesort				
quicksort with pivot rule: choose median of 3				
quicksort with pivot rule: choose first element				
quicksort with pivot rule: choose middle element				
quicksort with pivot rule: choose last element				

Table 1: Timing results for the sorting algorithms with $n = 250000$.

To give you some idea of what to expect, I ran all these timing tests on one of the `bg`'s using my implementation. I compiled my program using the `-O` option so that my executable would run more efficiently. My timings for insertion sort (in the order in which you are to fill in the row in Table 1) were 18.11, 0, 36.26, and 0 By way of comparison, my timings for Weiss' quicksort algorithm, using median of 3 to select the pivot, were 0.02, 0, 0.01, and 0

Your timings will undoubtedly vary from mine, depending on:

- whether you compile with optimization turned on,
- the machine you use to run your experiments,
- whether there are other jobs executing on that machine when you run your experiments, and

- decisions you made regarding the specifics of implementing the sorting algorithms,

but you should expect to see roughly the same results for your timing tests if your implementation is reasonably efficient and working correctly.

A warning

On the `bg`'s, it is probable that you will have a default limit on the size of the runtime stack you are allowed during execution. Generally, this is a good idea (it can help terminate a program that is stuck an infinite loop). But for the purposes of the timing tests you are *required* to run, it will be a hindrance. To get around this, execute the Linux command

```
ulimit -a
```

When I did so on `bg5`, here's what I got:

```
bg5$ ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 0
file size               (blocks, -f) unlimited
pending signals                 (-i) 257783
max locked memory       (kbytes, -l) 64
max memory size         (kbytes, -m) unlimited
open files                      (-n) 1024
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority              (-r) 0
stack size              (kbytes, -s) 8192
cpu time               (seconds, -t) unlimited
max user processes              (-u) 257783
virtual memory          (kbytes, -v) unlimited
file locks                      (-x) unlimited
```

Note that there is, indeed, a limit on `stack size`.

To remove the limit on the stack size, I execute:

```
ulimit -s unlimited
```

When I then execute `ulimit -a` again, I see the following:

```
bg5$ ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 0
file size               (blocks, -f) unlimited
pending signals                 (-i) 257783
max locked memory       (kbytes, -l) 64
```

```
max memory size         (kbytes, -m) unlimited
open files                      (-n) 1024
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority              (-r) 0
stack size              (kbytes, -s) unlimited
cpu time               (seconds, -t) unlimited
max user processes              (-u) 257783
virtual memory          (kbytes, -v) unlimited
file locks                      (-x) unlimited
```

Note that `stack size` now is `unlimited` so that it is possible for me to execute the timing tests you also are required to run.

Some comments on running the timing tests

Tests on large data sets can be resource hogs, so if you are going to run your experiments on the Department of Computer Science system, I suggest that you do so on one of the `bg`'s, since these machines are dedicated to running computationally expensive experiments. It is an interesting exercise to see how much your timings on one of these machines do – or do not – change if you compile with the optimization flag "`-O`".

Ideally, you should run your tests for each algorithm multiple times and average the timings you obtain because the numbers will vary depending upon what other jobs are running on the machine at the same time. That said, given the scheduling constraints, I do not expect you to do multiple runs and average the timings.

Some comments on submitting your timing results

Since all projects submissions are handled electronically using `submit`, and since we do not want to have to deal with different electronic document formats, you are required to report your timing results in a text (`.txt`) file, `Table1.txt`, which you are to submit along with your file `group.cc`. *We do not want – and will not accept – paper copies of your timing results.*

In an attempt to keep the formatting consistent, I have posted a template, `Table1.txt`, on the Web page for the project. Please download a copy of this file and report your timing results in the appropriate positions in it. In addition, where further explanation/discussion is required, please do so in the positions indicated in the file.

Final instructions

Give yourself ample time to implement and test your new methods for `GroupOfNumbers`. The requirements imposed by the project increase the possibilities for errors that can take time to track down.

Also allow yourself sufficient time to obtain this preliminary round of timing results. A good strategy would be to run the timing test for each sorting method as soon as you are confident that

your implementation is working correctly. That way you can space out the tests over a period of time—and you will be sure to have at least some timing results to submit for the project.

To submit your files for grading, go to the directory containing your copies of `group.cc` and `Table1.txt` and at the prompt in that directory enter:

```
~gpeng/bin/submit cs303 project2B
```

In spelling file names and identifiers which appear in these specifications (or in attached files) you must use the exact case and spelling.

Point Breakdown for the Project

To give you some idea of how to allocate your time and effort on this project, here is the point breakdown for the project:

10 points for "past sins" that remain uncorrected

14 points (each) for implementing:

```
void quick_choose_first(int left, int right);
void quick_choose_middle(int left, int right);
void quick_choose_last(int left, int right);
```

7 points for filling out the table with timing results

1 point (total) for implementing:

```
void quick_choose_first();
void quick_choose_middle();
void quick_choose_last();
```

Total: 60 points