Project 2, Part A
CSCI 303/503 Algorithms

Fall 2012

Due by: 8:00 a.m., Monday, October 29, 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 to to add four public methods to the class `GroupOfNumbers` and experiment with recording a rough estimate of the execution time. The four new public methods that you are to add to the class will allow a user to sort a `GroupOfNumbers` using either insertion sort, heapsort, mergesort, or quicksort. To implement these four new public methods, you also are to add several new protected methods to the class `GroupOfNumbers`.

Changes to `GroupOfNumbers`

On the Web page for the project you will find a revised copy of the file `group.h`. When you review the revised definition of `GroupOfNumbers`, you will see that you are to implement insertion sort, heapsort, mergesort, and quicksort.

Your implementation of insertion sort *must* follow the general strategy outline by Weiss in Section 7.2 of your text. Your implementation of heapsort *must* follow the general strategy outlined in Weiss in Section 7.5. Your implementation of mergesort *must* follow the general strategy outlined by Weiss in Section 7.6. Finally, your implementation of quicksort *must* follow the general strategy outlined by Weiss in Sections 7.7.2–7.7.4. In particular, for heapsort this means that you are *not* to use swaps in your implementation of `percolate_down`; note that in Weiss' *non-recursive* implementation of `percolate_down` he "picks up" the element to be moved and then "drops it in place" once its appropriate position has been identified. This same strategy is also used for insertion sort to avoid unnecessary data movement. You are *required* to adhere to this strategy (in your implementations of *both* `percolate_down` and `insertion_sort`) to avoid unnecessary data movement. Also, note that Weiss' implementation of heapsort is *not* recursive, so you too must implement a non-recursive version of heapsort. While Weiss' implementation of heapsort is not recursive, his implementation of mergesort *is* recursive, so you too must implement a recursive version of mergesort. You also are *required* to implement Weiss' version of quicksort, which means that your version must be recursive, it must use median-of-three partitioning, and it must include a reasonable cut-off that allows you to switch to insertion sort to finish the final ordering of the `GroupOfNumbers`.[1]

[1] One thing to keep in mind when looking at Weiss' implementations of sorting algorithms is that he explicitly passes an array `a` to be sorted. We, on the other hand, have made the sorting functions methods of the class

The instructions in the previous paragraph are important. Please read them carefully and make sure you understand the conditions in force for this project before you begin implementing your sorting procedures or you risk losing points.

Note that consistent with our use of C++ classes, the public methods for sorting take no arguments. However, to implement mergesort and quicksort recursively, and to make use of insertion sort within our implementation of quicksort (as does Weiss), we need to be able to pass indices to indicate the range over which the sorting is to take place. Thus the new definition of `GroupOfNumbers` contains protected methods that allow us to pass array indices and yet "hide" the details of the array-based implementation from the user.

Since you are to "percolate down" for heapsort (to restore the heap property) and you need a linear merge procedure for mergesort and you must use median-of-three partitioning for quicksort, there are other new protected methods to be implemented and then used.

In the file `group.h`, I include the definition and implementation of the protected method `left_child`, which plays a key role in the implementation. I also include a directive to the compiler to `inline` this method (with the goal of making execution time faster), though we have no way of forcing the compiler to do so.

Finally, since within the implementation of quicksort you are to switch to insertion sort to finish the final ordering of the `GroupOfNumbers`, you have one new static constant `CutOff`, which is used to determine when the switch should occur (and thus determines the base cases for the recursive quicksort method). I have assigned this static constant the value of 5, but you are free to experiment with other values during the course of your testing. (But remember that we grade using the file `group.h` that has been posted for the project.)

Timing the execution of the algorithms

For sufficiently large groups of numbers you should be able to get reasonably accurate approximations of the time spent executing insertion sort versus heapsort versus mergesort versus quicksort to sort the numbers in the group. While you will not be expected to report timing results for this project, you will be required to do so as part of later projects. So spend some time now experimenting to see how the timing measure works. At the very least, try testing your sorting methods with groups of size 10000 and 100000.[2] If your implementations of the sorting algorithms are working properly, then you should see appreciable differences between the execution times for insertion sort, heapsort, mergesort, and quicksort once you begin to sort large groups, particularly groups with more than 10000 entries. Feel free to test larger groups (with perhaps 1000000 entries) to really begin to see the differences between heapsort, mergesort, and quicksort (all of which are $O(n \log n)$ algorithms), but keep in mind that this will be "painful" for insertion sort and will consume a large

`GroupOfNumbers`. Thus there is no need to explicitly pass an array; the sorting methods must be attached to a `GroupOfNumbers` class object to be invoked by a user, and each `GroupOfNumbers` class object has an array of type `long` that we are interested in sorting.

[2] When you define the `GroupOfNumbers` object to be sorted, use the constructor that allows you to initialize the active entries in the array using the pseudo-random number generator. To get a reasonably accurate sense of the performance of the various sorting algorithms, give the constructor a reasonable range over which to produce entries. So, for instance, if `total_requested` is 100000, I suggest setting `lower` to 0 and `upper` to 100000. If you restrict the random number generator to too small a range, relative to the total number of entries requested, then you will wind up with an appreciable number of duplicate entries and that tends to lead to an inaccurate assessment of how the various sorting algorithms are likely to perform on a wide range of inputs.

number of computing cycles. If you are going to run such a large experiment with insertion sort (or a particularly large example with heapsort, mergesort, or quicksort), do not run your experiment on a machine on which others are trying to work. Either save such a run for "off-peak" hours or use a dedicated machine (i.e., your own personal computer, if you have one).

Recall that the Department of Computer Science maintains eight machines (the "`bg`'s") for running computationally expensive programs. I suggest you log in to one of these machines if you want to run larger/longer experiments on the Computer Science system. Their specifications are as follows:

Name	**Processor**	**RAM**	**Processor Model**
`bg1`, `bg2`, `bg3`	8 cores at 2.93GHz	48GB	X5570
`bg4`, `bg5`	4 cores at 3.00GHz	32GB	L5240
`bg7`	20 cores at 2.13GHz	192GB	E7-L8867
`bg8`, `bg9`	8 cores at 3.2GHz	192GB	X5672

Note that the machine `bg6` is reserved for the exclusive use of students in CSCI 321; ignore this restriction at your own peril.

An optional (but instructive) exercise

If you want to see how interactions with the compiler may further improve the performance, try compiling with optimization "on" and rerunning your tests. In my limited experiments, the execution time for large data sets decreases by anywhere from 5–30% once I turned on the compiler optimization.

So how do you turn on the compiler optimization? When you compile, add the option `-O` (that's an "Oh" for "Optimize" rather than a "zero") when you compile, as in:

```
g++ -O -Wall ....
```

If you look at the Linux `info` pages for `g++` (execute: `info g++`) and search on `-O` you will find a discussion involving the `-O` option and the various levels of compiler optimization available to you, though the discussion is not necessarily illuminating if you do not already know something about the subject. Nonetheless, the compiler optimization option is easy for you to try, even if you do not know how or why it works, and it could save you an appreciable amount of time when you have developed and debugged your program and are ready to run some substantive timing tests.

Final instructions

To submit your file for grading, go to the directory containing your copy of `group.cc` and at the prompt in that directory enter:

```
~gpeng/bin/submit cs303 project2a
```

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

Note that `submit` will only accept your file `group.cc` for grading. We will use the posted copies of `group.h`, `rng.h`, `rng.c`, and `timing.c`, along with our own test program, to grade your implementation of the methods for `GroupOfNumbers`.

Your file `group.cc` *must not* include any other `.c` or `.cc` files. When we compile our program for testing we will do so using the following command on the Computer Science system:

```
g++ -Wall test.cc group.cc rng.c timing.c
```

(assuming that our test program is in a file called `test.cc`).

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" from Projects 1A and 1B that had been detected by the graders, but remain uncorrected by you, or for earlier errors the graders are finding only now as they revisit your implementation.

6 points for implementing:

```
void quicksort(long left, long right);
```

5 points (each) for implementing:

```
void insertion_sort(long left, long right);
void heapsort();
void percolate_down(long i, long n);
void mergesort(long *temp_group, long left, long right);
void merge(long *temp_group, long leftPos, long rightPos, long rightEnd);
long median_of_3(long left, long right);
```

2 points for implementing:

```
void mergesort();
```

1 point (each) for implementing:

```
void insertion_sort();
void quicksort();
```

Total: 50 points