Project 3
CS 303/503 Algorithms

Fall 2012

Due by: 8:00 a.m., Wednesday, November 21, 2012

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

For this project you have five new public methods to implement for the class `GroupOfNumbers`

- `bool sort_select(long k, long & kth_smallest) const`,
- `bool partial_sort_select(long k, long & kth_smallest) const`,
- `bool min_heap_select(long k, long & kth_smallest) const`,
- `bool partial_max_heap_select(long k, long & kth_smallest) const`, and
- `bool quick_select(long k, long & kth_smallest) const`

that will allow a user to select the kth smallest number in a `GroupOfNumbers` object using one of five algorithms represented by these new methods. All five are algorithms we have covered in class. The specific requirements for each are detailed below. Before covering the specifics for the individual methods, note the features shared by all five methods:

- Each method takes as an input argument `k`, the position order selected by the user.
- Each method returns as an output argument `kth_smallest`, the *value* of the kth smallest number in the `GroupOfNumbers` object.
- The return type of each method is `bool`. The return value should be `true` if the value of k is valid, i.e., $1 \leq k \leq n$, where n is the size of the input (i.e., the number of numbers in the group). Otherwise, the return value should be `false`.

The new `public` methods

`bool sort_select(long k, long & kth_smallest) const`

Requirements for the algorithm:

- Sort all n numbers in the group in increasing order.
- Return the kth smallest number in the group.

You are free to choose the sorting algorithm.

`bool partial_sort_select(long k, long & kth_smallest) const`

Requirements for the algorithm:

- Sort the first k numbers in the group in increasing order.
- For each of the remaining $n - k$ numbers a_j, $j \in \{k, \ldots, n-1\}$, in the group
  - if a_j is less than a_{k-1},
    then "bump" a_{k-1} and insert a_j in order with respect to $a_0, \ldots, a_{k-2}$.
- Return the kth smallest number in the group.

You are free to choose the sorting algorithm.

`bool min_heap_select(long k, long & kth_smallest) const`

Requirements for the algorithm:

- "Heapify" the n numbers in the group *in linear time.*
- Execute `delete_min(x)` k times.
- Return the kth smallest number in the group.

`bool partial_max_heap_select(long k, long & kth_smallest) const`

Requirements for the algorithm:

- "Heapify" the first k numbers in the group *in linear time.*
- For each of the remaining $n - k$ numbers a_j, $j \in \{k, \ldots, n-1\}$, in the group
  - "Peek" at the maximum number in the heap.
  - If a_j is less than the maximum number in the heap,
    then "`delete_max(x)`" and "`insert(`a_j`)`."
- "Execute `delete_max(x)`."
- Return the kth smallest number in the group.

The decisions on how to manage the implementation of the various pieces within this algorithm is left up to you. The trade-off in efficiency is between effective code reuse, at the expense of higher constants in the overall time complexity, versus developing more efficient methods, which requires programming time. The call is up to you, so long as you justify your decision in the comments you include in your implementation.

`bool quick_select(long k, long & kth_smallest) const`

Requirements for the algorithm:

- Your implementation of `quick_select` *must* follow the general strategy outlined by Weiss in Section 7.7.6., 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 partition.
- Return the kth smallest number in the group.

A comment on the fact that the five new methods are accessors

Notice that all five of the new public methods include `const` in their prototype. The rationale here is that asking for the kth smallest element in the group is a request for information regarding the object, not a request that suggests the object should be modified (i.e., mutated) in any way. Thus your implementation of these methods must preserve this feature of the specification.

Some comments on `group.h`

The copy of the file `group.h` contains the `public` interfaces for each of the five new methods. I have outlined above the specifications you are required to meet when you implement the five new methods, but many of the details regarding the specifics of the implementation are left up to you.

You should need to use at least one new `protected` method to implement at least one of the new `public` methods. You may want to add additional `protected` methods to help implement some of the other new `public` methods.

With this in mind, for this project you are allowed to *add* new methods to the `protected` section of the class `GroupOfNumbers`.

You are *not* allowed to remove any of the existing methods of the class `GroupOfNumbers`. Nor may you in any way modify the prototypes of any of the methods of the class `GroupOfNumbers` that have already been defined in the copy of `group.h` that you download from the Web page for the course. In other words, you are allowed to *augment* the `protected` section of the class by adding new methods, but you are not allowed any other changes to the class definition.

The previous requirement is important. Read it again carefully to be sure you understand it.

In particular, if you modify the prototypes for any of the `public` methods, expect a 0 for the project since modifying the public interface makes it impossible for us to use a test program for grading.

As a final note, since you may augment `group.h` during the course of your development, you need to submit a copy of your file `group.h` along with a copy of your file `group.cc`.

Timing the selection algorithms

For the final project, you will be required to submit timing results for all six of the selection algorithms you implement. Since appropriate timing results will figure prominently in the grading of Project 4, it would be advisable to go ahead and time your implementations of sort select, partial sort select, minimum binary heap select, partial maximum binary heap select, and quickselect with the median-of-three rule for choosing the pivot (once you think you have them working correctly) to see if the timing results you obtain are consistent with what would be expected for these methods.

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 used randomly generated entries and gave the pseudo random number generator the range from -10000000 to 10000000. I compiled my program using the -O option so that my executable would run more efficiently. My timing results for $k = 1$ (in the order in which the selection methods are specified) were 1.11, 0.04, 0.13, 0.05, 0.09.

Once again, 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 selection algorithms,

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

Final instructions

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

```
~gpeng/bin/submit cs303 project3
```

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

Please be sure to keep copies of your implementation of `GroupOfNumbers`.

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:

20 points (each) for implementing:

```
bool min_heap_select(long k, long & kth_smallest) const
bool partial_max_heap_select(long k, long & kth_smallest) const
bool quick_select(long k, long & kth_smallest) const
```

15 points for implementing:

```
bool partial_sort_select(long k, long & kth_smallest) const
```

5 points for implementing:

```
bool sort_select(long k, long & kth_smallest) const
```

10 points either for "past sins" from Projects 1A, 1B, 2A, and 2B 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.

Total: 90 points