CS14 Summer 2016
Programming Assignment 4

Due Monday July 20 at 11:30 PM

Updated 07-19-16*

Synopsis - In the assignment you will implement two fundamental *Divide-and-Conquer* sorting algorithms, *Quicksort* and *Mergesort*, and measure their performance on data of various size and consistency. In addition to your implementations of each, you will submit the findings of your experimentation in a written report.

1 Quicksort

Implement a template function `quicksort`. Do not associate it with a class, rather implement it as utility function in a file named `Quicksort.H`. Then `#include Quicksort.H` in your `main.cc`. For debugging purposes initially use a random pivot. Once you are convinced the function sorts correctly, extend the functionality to allow for the following pivots: First, Median Of Three, Random Quick Select, and Deterministic Quick Select.

We must be able to call your function in the following manner:
`vector<string> result = quicksort(v, pivot);` – where `v` is an already defined vector of strings and `pivot`, indicating which pivot is used, is a string that can have one of the following values: `first`, `random`, `median_three`, `random_qselect`, or `deterministic_qsort`. Similarly, your function must also accept and return vectors of type `int`. We now describe each of the four pivots in more detail.

1.1 First

Pick the first element from the vector to be sorted. This element is the pivot.

1.2 Random

Randomly pick one element from the vector to be sorted. This element is the pivot.

1.3 Median Of Three

Randomly pick three elements from the vector to be sorted. The median of the three is the pivot.

1.4 Selection

For any vector `V` of comparable values, let `select(V,k)` denote the k^{th} smallest value in `V`, with `select(V,1)` being the smallest. Obviously, `select(V,k)` is the same as `V[k-1]` if and only if the

values in `V` are in sorted order, which normally requires $O(n\,log(n))$ comparisons to achieve, where n denotes `V.size()`. We will present two versions of the QuickSelect algorithm. The randomized version has linear average-case complexity but requires n^2 comparisons in the worst case. The deterministic version has linear worst-case complexity but much higher overhead.

Median. The *median* of an odd number of values is the one in the middle. But, if there are an even number of values, there are two in the middle. Statisticians take the average of those two values, but computer scientists prefer to select one. When `V.size()` is even:

- `select( V, 1+V.size()/2 )` selects the larger of the two.[1]
- `select( V, (1+V.size())/2 )` selects the smaller of the two.

But when `V.size()` is odd both expressions select the middle value. Both are equally valid, but the latter the more common definition:

```
template<T>
T median( vector<T>& V ) { return select( V, (1+V.size())/2 ); }
```

1.4.1 Random Quick Select

To find `select(V,k)` via the randomized Quick Select algorithm:

- Assert that `V` is not empty and that `k` is between `1` and `V.size()`, inclusively.
- Pick a so-called "pivot" value, `p`, at random from `V`.[2]
- Construct three new vectors `L`, `E`, and `G` containing, respectively, all entries of `V` that are **L**ess than `p`, all entries that are **E**qual to `p`, and all entries that are **G**reater than `p`.
- If `k <= L.size()`, then `select(V,k)` must be in `L`, so return `select(L,k)`, which, of course, involves a recursive call to `select`.
- Otherwise, if `k <= L.size()+E.size()`, then `select(V,k)` is surely in `E`, so simply return `p`.
- Otherwise, `select(V,k)` is surely in `G`, so decrement `k` by `L.size()+E.size()` and return the `k-th` smallest member of `G`, which again can be found via a recursive call to `select`, i.e., return `select(G,k-(L.size()+E.size())`.

1.4.2 Deterministic Quick Select (Blum, Floyd, Rivest, Pratt, Tarjan)

Unfortunately, if on each recursion we happen to randomly select the smallest member of the vector of values, then all values other than that pivot will be in `G`, so the recursions will go n deep requiring $O(n^2)$ comparisons.

To force, rather than simply expect, the number of comparisons to be linear in the size of the input, we proceed as above, but, to select our pivot, we partition `V` into quintuples, find the median of each of those quintuple, and select the median of those medians, i.e.:

[1] Note that, if `V` is sorted, `V[i]` is equal to `select(V, i+1)`, in general, and in particular, `V[ V.size()/2 ]` is equal to `select( V, 1+V.size()/2 )`.

[2] The closer the pivot is to the median, the sooner the algorithm is likely to terminate.

Clearly, at least half of the medians will now be in `G` along with their respective above-median quintuple-mates. So at least 30% of the members of `V` are in the union of `G` and `E`, i.e., at most 70% of the members of `V` are in `L`. Similarly, at most 70% of the members of `V` are in `G`.

To aid conceptualization, imagine an array whose columns are those quintuples each order in decreasing order, top to bottom. And imagine that those quintuples are arranged in order of increasing median, left to right.

SORTED QUINTUPLES ORDERED BY MEDIANS

```
ABOVE    ->      *      *     *  g  g     g     g
ABOVE    ->      *      *     *  g  g     g     g
MEDIANS  ->      l ...  l ... l  p  g ... g ... g
BELOW    ->      l      l     l  l  *     *     *
BELOW    ->      l      l     l  l  *     *     *
```

2 Mergesort

Implement the template functions `mergesort` and `merge` as presented in the *Weiss* text. Do not associate them with a class, implement them as utility functions in a file named `Mergesort.H`. Then `#include Mergesort.H` in your `main.cc`.

We must be able to call your function in the following manner:
`vector<string> result = mergesort(v);` – where `v` is an already defined vector of strings. Similarly, your function must also accept and return vectors of type `int`.

3 Testing

Produce several test files for use in measuring the performance of the algorithms. Your test files should be of at least two primitive data types, `string` and `int`. You can generate random sequences of strings and integers at http://www.random.org/strings/ and http://www.random.org/integers/, repectively. You will also need to add functionality for counting the number of comparisons and the number of data movements for a particular run of the algorithm being tested. Another statistic you will measure is the number of recursive calls made for the execution of a particular function. Test input sizes of increasing and large `N`. A general approach would be to have input sequences that are *mostly-ordered, mostly-random, random, monotonically increasing,* and *monotonically decreasing.*

4 Report

For each sorting algorithm answer the following questions. Justify your answers with Graphs (see below) and references to specific test cases.

1. What is the *worst-case* running time of the sorting algorithm? Explain.
2. What is the *average-case* running time of the sorting algorithm? Explain.
3. What is the *best-case* running time of the sorting algorithm? Explain.

4. Is the sorting algorithm *stable*? If not, why?

Graphs - You should produce plots of input-size v. comparisons, input-size v. data movements, input-size v. number of recursive calls for each algorithm and pivot combination. Include your plots and their associated data in table format in a single excel file named `plots.xls`.

5 Submission

You must turn-in a tar archive through iLearn named `assn4.tgz` which contains the following files: `main.cc`, `Quicksort.H`, `Mergesort.H`, `assn4.pdf`, and `plots.xls`. Your main function must adequately and automatically test your functions, and display statistics about the sorting that occurred. **Do not forget to put a class header on every file. Files lacking a header will not be graded**. *Re-download and test your submission. Programs that do not compile with the appropriate flags or segfault will receive a zero.*