Programming Assignment #2

Due: 2015/10/01 at 11:59pm
Points: 20

Goal

To implement and compare different sorting strategies.

Background

We've learned five sorting algorithms (with one more in store), and we've talked about how different algorithms are suitable for different purposes. But the only way to know for sure which algorithm is the best is to measure it.

This is especially true when you consider factors like memory, disk, and network speed. For example, "big data" applications are those that involve more data than can fit on a single machine. A lot of these applications (e.g., those based on Hadoop) have at their core a routine for sorting an extremely large data structure.

In this assignment, you'll work on the problem of sorting a list of locations according to their distance from your current location. You'll experiment with different data structures and algorithms to get an idea of how they compare.

Distribution

Update your local Git repository (as described at `https://bitbucket.org/CSE-30331-FA15/cse-30331-fa15`) and change to the directory `PG2`. It contains the following files:

`pg2.pdf`	this file
`test2.sh`	test script
`test3.sh`	test script
`ndfood.txt`	places to eat on campus (small)
`osmpoi.txt`	points of interest in the United States (large)

Task

1 The files `ndfood.txt` and `osmpoi.txt` have the following format, one line per location:

```
41.699441 -86.241417 South Dining Hall
```

The first two whitespace delimited fields are latitude (y) and longitude (x), and the rest of the line is the name of the location.

- Write a simple class to represent one line of one of these files. Use `double`s for the coordinates.
- Write a function template to read the whole file into either a `list` or a `vector`. You'll be comparing the two, so it's critical to allow both. Use a template to allow the function to take an argument of either type.
- Write a program that takes a filename on the command line, like this:

  ```
  ./distance1 ndfood.txt
  ```

  It should just read the file in (using the class and function you just wrote) and then exit.

2 Write a program called `distance2` that takes three arguments on the command line, like this:

```
./distance2 ndfood.txt 41.7031036 -86.2391732
```

The first argument is the location database to use. The other arguments are the user's location as latitude and longitude, respectively (the values shown are for the Golden Dome). The program should compute the distance from the user's location to all the locations in `points.txt` in miles, using the following formula:

$$\text{distance} = \frac{24901}{2\pi}\text{acos}\left(\sin\left(\frac{\pi}{180}\phi_1\right)\sin\left(\frac{\pi}{180}\phi_2\right) + \cos\left(\frac{\pi}{180}\phi_1\right)\cos\left(\frac{\pi}{180}\phi_2\right)\cos\left(\frac{\pi}{180}(\lambda_1 - \lambda_2)\right)\right)$$

where ϕ_1, λ_1 are the latitude and longitude of the user's location and ϕ_2, λ_2 are the latitude and longitude of the point of interest (all in degrees). The value of π should be available as `M_PI` if you `#include <cmath>`. The output should look like this (just first few lines shown):

```
$ ./distance2 ndfood.txt 41.7031036 -86.2391732
0.407421 A la Descartes (Jordan Hall of Science)
0.466671 Au Bon Pain (Hesburgh Center for International Studies)
0.251821 Au Bon Pain (Hesburgh Library)
```

You should also write a `Makefile` that builds `distance2` (and all of the programs you will write below) automatically when you type `make`.

3 Modify your program into a new program called `distance3` that does the same thing, using a `std::vector` to store the locations, and sorts all the points by distance (smallest to largest) using `std::sort()`. If two locations have the exact same distance, the one whose name comes first alphabetically should be printed first. (See Appendix A for how to write your own comparison function.) The output should look like this (just the first few lines shown):

```
$ ./distance3 ndfood.txt 41.7031036 -86.2391732
0.112922 LaFortune Student Center
0.184448 North Dining Hall
0.185885 Cafe Poche (Bond Hall)
0.245472 Waddick's (O'Shaughnessy Hall)
0.251821 Au Bon Pain (Hesburgh Library)
0.278584 Reckers
0.278584 South Dining Hall
```

4 Do the same thing, but use a `std::list` to store the locations, and again sort all the points by distance (smallest to largest). You'll write two variants:

`distance4a`: sort using `std::list::sort()`.

`distance4b`: copy into a `std::vector`, sort using `std::sort()`, and copy back into a `std::list`.

5 Write one more variant, called `distance5`, that either:

- uses `vector` and sorts using your own implementation of quicksort; or
- uses `list` and sorts using your own implementation of mergesort.

You may **not** use any of these functions from the `<algorithm>` header in the standard library, or `std::list::merge()`.

6 Run all four variants using `osmpoi.txt` as the database, and measure the time and memory usage using `measure`. In order to get a more reliable measurement, run each variant five times and take the average.

To avoid overloading the student machines, try to run your tests elsewhere. As long as you time all four variants on the same (type of) computer, it doesn't matter where you run. For example, you can run on your own computer. Or, from the student machines, you can launch your tests on the Condor cluster, using the script `condor_measure.sh`, like this:

```
$ ./condor_measure.sh ./distance3 osmpoi.txt 41.7031036 -86.2391732
warning: this program overwrites the ./distance3.output file!
Staging files ... Ok!
Submitting Condor Job ...
Submitting job(s).
1 job(s) submitted to cluster 880156.
Waiting for Condor Job 880156 ...
880156.0   pbui            9/18 14:34   0+00:00:37 R  0   0.0  measure ./distance
Finished!
Copying results ... Ok!
Measured time and memory usage: 34.547749 seconds       115.062500 Mbytes
Cleaning up ... Ok!
```

For each variant, write (in your `README.md`) how it performed, and why (in one or two sentences) it ranked where it did. See Appendix B for a brief description of GCC's sorting algorithms.

Rubric

`distance1` correctly uses a template:	1
`distance2` passes `test2.sh`:	2
`distance3` passes `test3.sh`:	3
`distance4a`, `distance4b` pass `test3.sh`:	2
`distance5` passes `test3.sh`:	5
Question 6:	4
Correct memory management according to Valgrind:	1
Good style:	1
Good comments:	1
Total:	20

Submission

1. Provide a `README.md` that includes your answer to Question 6 and anything else you'd like to say.
2. Provide a `Makefile` that automatically builds all of your programs when you type `make`.
3. To submit your work, upload it to your repository on Bitbucket:
   (a) `git add` *`filename`* for every file that you created or modified
   (b) `git commit -m` *`message`*
   (c) `git push`

   The last commit made at or before 11:59pm will be the one graded.
4. Check your submission: clone your repository to a new directory, run `make`, and run all tests.

A Custom comparators

There are a few ways to customize how functions like `std::sort` compare objects.

- Define the less-than operator (`operator<`) for your location class. That is, if your class is called `location`, you will define a function (outside your class):

  ```
  bool operator< (const location &x, const location &y);
  ```

  that returns `true` if `x` should come before `y`, and `false` otherwise. Alternatively, you can define a member function inside your class:

  ```
  bool operator< (const location &y) const;
  ```

- Define a function that acts like the less-than operator and pass it as the second argument to `std::sort` (or the only argument to `std::list::sort`).

- Define a class with a member function `operator()` that acts like the less-than operator for locations.

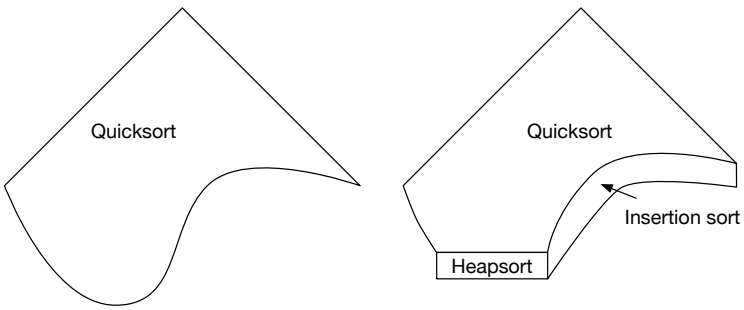
  ```
  struct compare_location {
    bool operator() (const location &x, const location &y);
  };
  ```

  Then pass an instance of this class as an argument to `std::sort` or `std::list::sort`. This trick allows you to store extra information (like your current location) that you might want to use in making the comparison.

B GCC's sorting algorithms

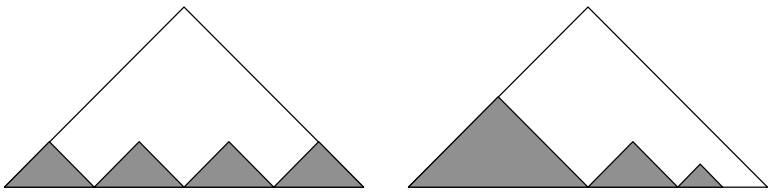
You don't need to fully understand these algorithms or their source code, but the following information is provided to help you to answer the question about why these algorithms perform well.

Arrays The source for `std::sort` can be found online (`http://j.mp/gccsort`). The algorithm is called introsort, and it's a hybrid of quicksort, heapsort, and insertion sort. It starts out as quicksort, but the recursion stops in two cases. If the recursion depth exceeds $2\lfloor \log_2 n \rfloor$, then switch to heapsort (`std::__partial_sort()`). If the length of a subarray is `_S_threshold` or less, then switch to insertion sort. (More precisely: If the length is `_S_threshold` or less, stop the recursion; then after all the recursion is done, do one big insertion sort.)



We might picture the call tree of quicksort as the figure on the left. At the root is the call to the sorting function on the whole list, and below are the recursive calls. Because partitioning can't in general find the true median, the tree is ragged on the bottom. Then, introsort looks something like the figure on the right: below a certain depth, the recursion is cut off and the algorithm switches to heapsort; also, the bottom fringe of the tree is replaced with insertion sort.

Linked lists The source for `std::list::sort` can be found online (`http://j.mp/gcclistsort`) and is a marvel of compactness – mergesort in 36 lines. If you think of mergesort as a tree of merges, standard mergesort goes bottom-up and this mergesort goes kind of left-to-right:



The figure on the left is standard mergesort. The elements have all been merged into four sorted sublists of equal size. The figure on the right is the GCC mergesort. This algorithm maintains a list of sorted sublists (called `__tmp`). The algorithm works by inserting the elements one at a time, left to right, into this data structure, as a sublist of length one. As soon as there are two sublists with the same length, they are merged. Hence, the sublists' lengths are all different powers of two. (This data structure has some similarity to a kind of priority queue, the *binomial heap*.) When all elements are inserted, all the remaining sublists are merged into a single sorted list. The operations are basically the same as a standard mergesort; only the order of operations is different!