CS 15 Lab 11: Sorting

Getting Started

1. Copy over the starter materials from the usual place.
2. Look at the table in the `RESULTS` file. You will fill this out as you go.
3. Run, observer, learn:
   - make
   - ./sort
   - ./sort -?
   - ./sort -r 10 selectionPlace -debug
   - ./sort -r 10 selectionPlace
   - ./ntimes 2 5
   - ./count 2 10
   - ./count 10 2
   - ./count 10 0 — ./sort -f - selectionPlace
   - -debug
   - ./sort -r 1000 selectionPlace
   - ./sort -r 10000 selectionPlace
   - Pause. Explain the last two runs.
   - Predict the result of sorting 100,000 random elements using in-place selection sort. Write down your estimate.
   - ./sort -r 100000 selectionPlace
   - How close was our estimate?
   - ./sort2 -r 100000 selectionPlace

- sort2 is an optimized executable based on our solution (equivalent of make faster)

4. Read the file comments at the top of `sorting.cpp`.

Submitting Your Lab

You will modify and submit your code and timing results as discussed in class.

The Assignment

Does an in-place sorting algorithm run faster than one that uses an auxiliary array? Does Quicksort, despite having an `O(n2)` worst-case run-time complexity, actually perform better than the `O(n log n)` mergesort?

The goal of this lab is to answer these questions. There is a table in `RESULTS` (and illustrated at the bottom of this document) for you to fill in with experimental results. Fill in the `RESULTS` table as you go, and discuss your results with the TAs.

You'll need to write code for:

1. In-place insertion sort
2. Quicksort, both the function and the partition function

If you have time, you may also write:

1. Insertion sort with an auxiliary array.
2. The merge function for mergesort.

Be sure your sorts work on 0- and 1-element lists as well as already sorted, reverse sorted, and random lists.

Evaluating Sorting Algorithms

If you finish in-place insertion sort, quicksort as well as insertion sort with an auxiliary array and mergesort, you'll have a total of 6 algorithms to compare. (Selection sort, both in-place and with an auxiliary list, is written for you.)

Now we can compare and contrast the sorting algorithms! As you saw at the beginning, the sort can run and time different algorithms on different

inputs: randomly generated lists of a given size, input from a file, input from standard input. You can use this capability to debug your code and then to measure it's performance for different data sets.

Fill in and extend the table in your `RESULTS` file as you go through the lab. The column headers should be input size/type. For example, 1000/ran means 1,000 randomly generated numbers; 1000/inc means 1,000 numbers increasing (already sorted); 1000/dec means 1,000 decreasing (reverse sorted) numbers; 1000/dup means 1,000 copies of the same element (duplicates). You can make up other test cases. If you do, describe them in your `RESULTS` file. You might also see that some input sizes, although we have enough memory, cannot be sorted in a reasonable amount of time by the `O(n2)` sorts.

As you go, make predictions. Given how long quick sort or merge sort run on 1,000 elements, how long with they run on 10,000 or 100,000 elements? Similarly for insertion sort.

Try to get quick sort to be slow. What inputs could you pick that would evoke slower times? Compare to merge sort.

Talk to each other and the TAs about what you're seeing.

If you want to go further, you can explore the role of compiler optimizations on performance. These optimizations can make a significant difference, but they will not affect the relative performance of different data sets; i. e., Big-O still applies. `make fast`, `make faster`, and `make fastest` will compile the program with increasingly aggressive optimizations applied. Not required, but if you are interested and have time, you may explore the effects on your program.