UNBC Laboratory Assignment
Computer Science 200—Fall 2018

Sorting and Timing II

Purpose:

To gain experience with coding sorting algorithms and testing their behaviour. verify their asymptotic timing characteristics.

Due Date:

This assignment is due ***29 November 2018***.

Assignment:

Your goal is to implement heap sort, quick sort, and a robust sort.

The robust sort uses a mixed strategy. It starts out as quick sort, but if it finds itself failing to make sufficient progress, it switches to heap sort.

Implementing heap sort

⇒ Implement heap sort, quick sort, and a robust sort.

Implement each of these sorting algorithms in a manner suitable for inclusion in a library of routines to be supplied to another user. In particular:

- Make sure that your algorithm works when the array supplied by the user is very small, even 0 or 1 elements.
- Code your algorithms with generic public static function interfaces.
- Write two versions of each algorithm: one that has a signature like

```
public static <E extends Comparable<E>> void sort(E [] data) ...
```

  and one that has a signature like

```
public static <E> void sort(E [] data, Comparator<E> pred) ...
```

- Code quick sort to use your previously coded insertion sort as its cleanup phase.
- Code a version of heap sort with signature like

```
public static <E> void sort(E [] data, int begin, int end,
                            Comparator<E> pred) ...
```

that you can use as a subroutine of a robust sort.

Here are some details to watch relating to particular sorts.

Heap sort Check the heap-building algorithm carefully. I believe that there may be an error in my lecture notes.

When removing items from the heap, the textbook algorithm places the last element in the array in the first position, then bubbles it down. This is slower than bubbling the hole at the top down to the bottom and then moving the last element into the hole and bubbling it up. The latter ideas are in my notes, and are suggested by the exercies at the end of Chapter 7.

Quick sort

- Try to code quick sort so that it cannot overflow the stack even in the presence of bad luck when partitioning.
  Means to do this are in my lecture notes. Another approach is to keep a priority queue (or stack!) of regions to partition arranged so that you are always partitioning the smallest region first.
- Use a cutoff for partitioning that is large enough that you don't need to worry too too much about the details of pivot selection ((*i.e.*, you can assume that there are at least three distinct elements). After all of the recursion is done, finish by using insertion sort.
- If you have time, explore various cutoffs to see how they affect the running time of quick-sort.

a robust sort Create a sorting routine with the average case speed of quick sort with the worst-case time characteristics of heap-sort by using the following strategy.

1. Create a pessimistic estimate h of the depth of the recursive calls required: say $h = 10 + 3\log_2 n$.
2. Pass h as a parameter to your initial recursive quick sort call, and have it pass $h - 1$ to its callees and so on.
3. If the recursive quick sort algorithm finds itself called with $h = 0$, switch to heap sort on the given region.

Almost all of the time the algorithm will run the same as unmodified quick sort, but it is now impossible to get $\Theta(n^2)$ behaviour.

Testing and Measuring

⇒ Test your sorting algorithms for *correctness*. That is, write code that verifies that your sorting algorithm produces (a) a sorted array, and (b) an array that is a permutation of the original. You may want to code some test routines to generate data other than permutations. For instance, an array randomly filled with 1's and 2's provides a good test for how your algorithms handle equal elements.

⇒ Time your sorting procedures for various different sized data collections.

All of your tests should be automated. That is, they should produce timing data files that are easy to use with your plotting software.

You must hand-in plots of running times versus size of data collection!

⇒ Show that your actual running times are consistent with the $\Theta()$-behavior for the sorting algorithm. For instance, for sorting functions that are $\Theta(n \log n)$, find the best value of c_1 so that $c_1 n \log n$ passes smoothly through your data points.