FALL 2007: **COT 5407** Intro. to Algorithms

[Homework 2; Due Sep 13 at start of class]

General submission guidelines and policies: Add a signed statement that you have adhered to the collaboration policy for this class and that what you are presenting is your own work. Without this statement, your homework will not be graded.

Problems

8. (**Exercise**) Solve these exercises (These will not be graded; it is enough to provide just final answers to show you have worked it out): Exercise 7.1-1, p148; Exercise 7.2-2, p153; Exercise 7.2-4, p153; Exercise 6.2-1, p132; Exercise 6.3-1, p135; Exercise 6.4-1, p136; Exercise 6.5-1, p140;

9. (**Regular**) Rewrite Partition (let's call the new algorithm Partition3) so that it takes as input an array A, and outputs a 3-partition of the array. In particular, it picks two pivots (for your implementation, choose them to be the last and the second-to-last items in the array) x and y, and outputs the array partitionaed into three pieces. In other words, assuming that $x \leq y$, the array output by the algorithm contains all items smaller than both x and y followed by item x, followed by all items between x and y, followed by item y, and followed by all items larger than both x and y. The partition must happen **in-place** meaning that no additional arrays ought to be used. Finally, the algorithm should return the locations of the two pivots. Analyze the time complexity of the algorithm and write down an invariant for the algorithm that would reflect its correctness.

10. (**Exercise**) Given the algorithm Partition3, show how the code for algorithm Quick-Sort should be changed to use it.

11. (**Regular**) Study Randomized-Partition and Randomized-Quicksort from page 154 of your text. Now modify it to implement two new features:

    (a) Implement the **median-of-3** method for choosing the pivot (described on page 162);

    (b) Implement R. Sedgewick's idea (1978) to avoid recursive calls when the size of the array is at most k. Set the value of k to be 8, which is the cutoff value used in the 1997 Microsoft C library implementation of Quicksort. Use InsertionSort for arrays of size at most k. There is no need to show code for InsertionSort.

12. (**Regular**) Sorting algorithms are not constrained in the way they treat "equal" items in the input list. Therefore, if all the items in a list are equal, any permutation of the inputs is a correct output. A sorting algorithm is said to be *stable* if the relative order

of any equal items in the input list is not changed in the output list. For example, if your list contained scores of students in an exam. Say that your input list has Adam with a score of 78 in location 7, and Alice with a score of 78 in location 11, Then after a *stable* sorting based on exam scores, the output list is guaranteed to have Adam appearing before Alice. An *unstable* sort provides no such guarantee. The output list (although it is still sorted) may or may not have Adam appearing before Alice.

In order to show that a sorting algorithm is stable, one would need a mathematical proof. However, to prove that it is not stable, all we need is a simple "counterexample". It should be obvious to you that InsertionSort, BubbleSort, and MergeSort are stable sorting algorithms (first convince yourself of this). For each of the following sorting algorithms, state which ones are **not** stable with the help of simple examples: Selection Sort, QuickSort, and HeapSort. Devise the smallest example you can build, if you claim the algorithm is not stable. Write down a brief argument if you think it is stable. You should consider a sorting algorithm to be stable if the algorithm given to you in the book or the one given to you in class (or a minor variant thereof) is stable.