22C:44 Homework 5
Due October 12, 2000

Problem 1 is worth 20 points and the other two are worth 10 points each.

1. This problem involves implementing versions of `QuickSort` and comparing their performance. You can use any platform and any programming language for your implementation. However, you are responsible for knowing enough about the programming environment of your choice to complete this assignment.

   (a) Consider the algorithm:

   ```
   RandomPermutation (A[p..q]) {
        for i ← p to q-1 do {
             j ← Random(i, q);
             swap(A, i, j);
        }
   }
   ```

   In the above code, `Random(i, q)` returns a random integer in the range `i` through `q`. Assuming that the elements in `A[p..q]` are distinct, show that each permutation of `A[p..q]` is produced with the same probability. Implement this function.

   (b) Suppose we want to generate large "almost sorted" (as defined in Problem 4, Homework 4) arrays of numbers that have some amount of randomness. One way to do this is the following:

   Step 1 Fill the array `A[1..n]` with the numbers $1, 2, \ldots, n$ in that order.

   Step 2 Randomly permute (using the function in part (a)) disjoint blocks of `A` of size $c + 1$, that is, `A[1..c+1]`, `A[c+2..2c+2]` and so on.

   Implement this function.

   (c) Modify the `QuickSort` function given in our textbook so that it keeps track of the number of comparisons made. Implement this modified `QuickSort`. Run the following experiment: for each size s generate 10 almost sorted permutations (as in part (b)) with $c = 10$; run `QuickSort` on each of these permutations and compute the average number of comparisons made. Do this for sizes $s = 500, 1000, 1500, 2000, \ldots, 10000$. At the end of this experiment you should have 20 numbers, each representing the average number of comparisons made by `QuickSort` for permutations of a particular size.

   (d) Implement your solution for Problem 4 (b) in Homework 4. Recall that in this solution, you designed a linear time algorithm for `QuickSort` on almost sorted arrays.

   (e) Examine the results of the two experiments and comment on whether the results are in keeping with asymptotic running times we expect these functions to have.

   Part (a) requires a written answer as well as code. Part (e) requires a written answer. Make sure that you answer part (e) clearly and completely because in a sense this is the most important part of the problem. You should turn in printouts of code you have written for all the parts. Use comments liberally in your code and highlight key changes you have made to existing code (for example, you should highlight your modification to `QuickSort` that keeps track of the number of comparisons being made).

2. This problem involves heaps.

   (a) Describe a algorithm that merges k sorted arrays into a single sorted array in $O(n \lg k)$ time, where n is the total number of elements in all the arrays.
   Hint: Use a binary heap of size k.

(b) A *run* in an array A is a contiguous block of elements in increasing order. Use your solution in (a) to devise an algorithm that sorts an array A in $O(n \lg k)$ time where k is the fewest number of runs that A is partitioned into.

3. In the worst case linear time selection algorithm discussed in class, we partitioned the array into blocks of 5. What happens when you partition the array into blocks of 3? In particular, calculate the number of elements in that array that are guaranteed to be smaller than the median of the medians. Using this, write down the recurrence relation for the running time of the selection algorithm, assuming that blocks of 3 instead of blocks of 5 are used.
