This homework is due next Friday, March 2. You can work with a group of up to four people on this assignment, and you can turn in one set of solutions for the group. Everyone should do problem number 1, which is important for understanding how priority queues can be used to speed up some algorithms. If you are working alone, you should do at least ***one*** *extra problem chosen from problems 2 through 8. If you are working in a group, you should do at least one extra problem* ***for each person in the group****. Some of these problems ask you to devise explicit algorithms. You can write out the algorithms in clear and complete pseudocode or as Java subroutines. You should consider testing your algorithms by implementing them in actual Java programs, but that is not a requirement except for problem 7, which asks for an actual program. (If you do write programs and want to submit them, you can turn them into the homework folder and include a note about the program with the written work that you turn in.) Grading might be based partly on ambition. I might ask some people to present their solutions in class.*

1. (*Based on Exercise 4-14.*) Suppose that you have K lists that are **already** sorted into non-decreasing order. The total number of items in all the lists combined is N. You can assume that the lists are in an array of lists. (For example, an *ArrayList* of *ArrayLists* in Java). The goal is to merge the lists into a single sorted list.

a) There is an obvious algorithm for merging the lists with run time $\Theta(N * K)$. Describe that algorithm, without writing it out in detail, and explain why its run time is $\Theta(N * K)$.

b) Now, write an algorithm with run time $\Theta(N * \log(K))$. Use a heap to speed up the obvious algorithm from part **a**. Explain why the run time is $\Theta(N * \log(K))$. (If you implement this algorithm, you can use a Java *PriorityQueue* for the heap. You might even use a *PriorityQueue<ArrayList>*.)

You will not do all of the following problems!
Please read the note at the top of this page!

2. Write an algorithm for solving the "Dutch National Flag" problem with run time $\Theta(n)$ and only using a few variables for extra memory, as discussed in class, and explain how a variation of the algorithm can be used as a substitute for the usual partitioning algorithm in Quicksort. (The problem is to sort an array where every item in the array is *blue*, *white*, or *red*, so that all the *blues* come before the *whites*, and all the *whites* come before all the *reds*. You could, instead, sort an array containing only the numbers 1, 2, and 3.)

3. (*Based on Exercise 4-24.*) Let $A[0], A[1], \ldots, A[N-1]$ be an array such that the first $N - \sqrt{N}$ elements are **already** sorted into non-decreasing order. Nothing is known about the last $\sqrt{N}$ elements.

a) Insertion sort has better performance on an array that is almost sorted. If you apply *Insertion Sort* to the array, what will its worst-case run time be? (Why?)

b) Devise an algorithm that sorts the array with run time $\Theta(N)$. (Hint: You can use extra space. You might encounter an unusual run-time function.)

4. (*Based on Exercise 4-9*) You are given an array of N integers and an integer x. You want an algorithm to determine whether two elements exist in the array whose sum is exactly x.

- **a)** Assume that the array is already sorted. Give an algorithm that solves the problem in guaranteed time $\Theta(N)$. (Note that there is an easy hash-table-based algorithm that answers the question in expected time $\Theta(N)$. Of course that also uses extra memory proportional to N. Don't use a hash table for this problem, and use only $\Theta(1)$ extra memory.)
- **b)** Assume that the array is not sorted. Give an algorithm that solves the problem in time $\Theta(N * \log N)$. (This is **really** easy after doing part **a**.)

5. (*A standard extension of Exercise 4-16*) Use the partitioning idea of Quicksort to give an algorithm that finds the k-th smallest element in an array of N items, with expected running time $\Theta(N)$. (The k-th smallest element is the one that would be $A[k]$ if the array is sorted. Of course, it is possible to find this element by sorting the array and returning $A[k]$, but that would have run time $\Theta(N * \log(N))$. Hint: Must you look at both sides of the partition?)

6. (*Sorting by counting*) Suppose that you have an array, A, of N integers that contains only integers from the range 0 through $K - 1$, where you know the value K. For this problem to make sense, K should be $\mathcal{O}(N)$. Consider the following idea for sorting the array: "Use an array of size K to count the number of times that each of the values 0 through $K - 1$ occurs in the array A. Use that to make another array, S, of size K, where $S[i]$ is the index where the first occurence of i should occur in the sorted array. Finally, copy items from A into their correct sorted positions in another array, B, of the same size." Flesh out this outline into a complete algorithm (or program). What is the run time of this algorithm?

7. Some background on disk drives: A disk is divided into *blocks* of a fixed size, such as 8 kilobytes. When data is read from or written to a disk, full blocks are transferred between main memory and the disk. You should think of reading and writing blocks of data from the disk as being very slow, so the goal of algorithms that do a lot of disk access is often to minimize the number of blocks read or written.

Suppose that you have a huge file on disk that you would like to sort. It is much too big to fit in the computer's memory all at once. Describe a reasonably efficient algorithm for sorting the file. You can use extra space—that is, you can create more files—on the disk. Estimate the number of blocks that are accessed by the algorithm, in terms of the number of blocks in the file. That is, try to at least approximately express the number of blocks read and/or written as a function $f(b)$, where b is the number of blocks in the original file.

8. Write a version of Quicksort for sorting an array of integers that uses as many speed-up tricks as you can. Write a program to compare the run time of your version to Java's *Arrays.sort* on identical arrays. (*Arrays.sort* uses Quicksort for arrays of integers.) Include a report on your results, on paper, when you turn in the homework. You might want to look at several sample arrays with different characteristics (random, already sorted, constant, random with large numbers of duplicates).