Problems 1 and 2 are due at the start of class on Friday, February 15. Problem 3 can be turned in either on paper with Problems 1 and 2, or it can be added as a comment in the main program file for Problem 4. Your work for the programming assignment, Problem 4, should be submitted by copying it into your homework folder inside the directory /classes/cs327/homework. The program should be turned in by 3:00 PM on Friday, February 15. I will print out every .java file in your homework folder.

1. Do the following exercises from Chapter 6 of the textbook:

6.4-1: Using Figure 6.4 [shown on reverse] as a model, illustrate the operation of Heapsort on the array A = $\{5, 13, 2, 25, 7, 17, 20, 8, 4\}$.

6.4-5: Argue the correctness of Heapsort using the following loop invariant for the second *for* loop in the algorithm (after the array has been heapified): "At the start of each iteration of the *for* loop, the subarary $A[0 \ .. \ i\text{-}1]$ is a max-heap containing the i smallest elements of the original array, and the subarray $A[i \ .. \ n\text{-}1]$ contains the $n - i$ largest elements of the original array in sorted order.

6.4-3: What is the running time of Heapsort on an array A of length n that is already sorted into increasing order? What about decreasing order? (Why?)

6.5-2: Illustrate the operation of *max_heap_insert*(A,10) on the following max-heap: $A = \{15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1\}$. [We talked about how insertion is done, but I did not give the code in class. You don't need the code to do this exercise, but note that you will implement this routine as part of your programming assignment].

2. Here are two algorithm-development problems that can be solved using ideas from the partitioning algorithm used in Quicksort. Find algorithms that are **in-place** and that solve these problems with **linear run time**, $\Theta(n)$. The first problem is easy, the second is harder.

a) Sort an array containing only (many copies of) the numbers 1 and 2.

b) Sort an array containing only (many copies of) the numbers 1, 2, and 3.

3. Write up and turn in the results from the timing experiments that you do with the program that you turn in for Problem 4. Do **not** just list the times!! Give an analysis of the results. What do they say about the various sorting algorithms and their run times?

4. Write a class, *MaxHeap*, to implement a max-heap of values of type *double*. (You can either use an *ArrayList* to represent the heap, or use an array and be prepared to grow the array. The array implementation will probably be more efficient. (Maybe you should try both!))

Next, write three sorting methods: One should use the *heapsort* algorithm as covered in class. A second should sort the array by inserting all the elements from the array into a heap defined by your *MaxHeap* class, and then removing all the items from the heap and putting them back into the array in order. Finally, the third sorting method should be an optimized version of *quicksort* that is as efficient as you can make it. You will probably want to use randomized

quicksort. You might want to switch to insertion sort on very small subarrays. You might even consider using a stack instead of recursion.

Finally, you should write a main program that does timing experiments on your three sorting methods plus the built-in sorting method, *Arrays.sort*(A). It is probably best to apply all four algorithms to identical arrays. Remember that you can easily make a copy of an array, A, by calling A.*clone*(). Use arrays filled with random numbers. You will need to use fairly large arrays to get useful time measurements. Use *System.nanoTime*() to do the time measurements. Consider turing off the Java just-in-time compiler to get more accurate comparisons. To do that, run the program with the javac option -Xint.

Note that your grade for this assignment is based partly on your design of the timing experiments.

Possible term projects: **(1)** InsertionSort should be faster than Quicksort and other sorting methods on an "almost sorted" array. Investigate this idea and devise programming experiments to test it. Exactly how "deranged" does an array have to be, for Quicksort to be better than InsertionSort? **(2)** Java has a built-in *PriorityQueue* class. Investigate what it does and how it is used and, using timing experiments, how efficient it is for various types of heap elements.