CPSC 327, 12 April 2013 Second Test, Take-home Part

This is a test. You should not work on the test or discuss it with other people in the class. You can use your textbook and notes, but you should not use other books or resources, including the Internet. You can ask the Professor for hints or clarification. There will be time to do that in class on Monday so that everyone can hear the answers. The test is due in class next Wednesday, April 17.

1. (*5 points*) Quicksort is not an in-place algorithm. The textbook notes, "It requires a stack to store parameters of subarrays that are yet to be sorted. [...] the size of this stack can be made to be in $\mathcal{O}(\log n)$ by always sorting first the smaller of two subarrays obtained by partitioning." What is the space efficiency of the usual Quicksort algorithm, without the optimization mentioned by the textbook? Explain your answer. The usual algorithm is given by

```
static void quicksort(double[] A, int low, int high) {
  if (high <= low)
    return;
  int mid = partition(A, low, high);
  quicksort(A, low, mid - 1 );
  quicksort(A, mid + 1, high );
}
```

2. (*7 points*) When applying binary search to a list of n items, you look at the $\frac{n}{2}$-th item to decide whether to continue the search in the first half or in the second half of the list. Consider a *ternary search* in which you look at the $\frac{n}{3}$-th and (possibly) the $\frac{2n}{3}$-th items to decide whether to continue the search in the first third, second third, or third third of the list.

a) Write a clear and complete algorithm for ternary search of an array. You can write either a recursive or a non-recursive version.

b) Do you think that ternary search is an improvement on binary search? Why or why not?

3. (*9 points*) The operations on a priority queue are to *add* an item to the queue and to *remove* the largest item. Sometimes, it is required to be able to *change the priority* of an item in the queue. (This does not really apply to a priority queue of numbers. It applies when the queue holds complex objects and the priority is only one property of the objects. For example, consider a priority queue of jobs waiting for run time on a computer. The priority of a job might change while it is in the queue.)

a) Suppose that the priority queue is implemented as an unsorted list. How would a *changePriority* operation be implemented in that case? (This is trivial.)

b) Suppose that the priority queue is implemented as a sorted list, sorted by priority. How would a *changePriority* operation be implemented in that case?

c) Now suppose that the priority queue is implemented by a heap—which it should be, of course, for efficiency. If you simply change the priority of an item in the heap, the resulting data structure might not satisfy the heap property. Explain how to implement a *changePriority* operation for a priority queue implemented as a heap. Give a clear and complete explanation!

4. (*9 points*) The algorithms that we looked at for inserting items into a heap and deleting items from a heap involved repeatedly swapping a child with its parent. An improved algorithm for insertion would go something like: increase the size of the heap by one. Starting with the new node, move items down the tree until you have a space where the new item can be placed where it will satisfy the heap property. A similar algorithm can be used for deletion.

a) Write Java subroutines to implement improved algorithms that follow this plan. Assume that the heap contains values of type *double* stored in an array of type *double*`[]`.

b) What sort of improvement in run time do you expect to see from this change? Why?

5. (*9 points*) Develop a dynamic programming algorithm that will find the length of the *longest* path in a weighted directed acyclic graph. You can present the algorithm in clearly written and fully commented pseudocode. Hint: Do a topological sort and then work backwards from the end.

(Note: This problem is applicable in the following context. The graph represents some task that must be performed. Each edge in the graph represents a subtask. The weight of an edge is the time that it takes to perform the subtask. Vertices in the graph represent prerequisites; that is, before the task represented by an edge leaving the vertex can be started, all the tasks represented by edges entering the vertex must be completed. Then the length of the longest path in the graph is the minimum time in which the overall task can be completed.)