ECE368 Fall 2016 Homework 7

IMPORTANT:

- Do NOT leave your name or Purdue ID on this homework.
- Write your homework security number at the TOP of EACH page.

Read and sign the ***Academic Honesty Statement*** that follows:

"In signing this statement, I hereby certify that the work on this exercise is my own and that I have not copied the work of any other student while completing it. I understand that, if I fail to honor this agreement, I will receive a score of zero for this exercise and will be subject to further disciplinary action."

`Homework security number:`

Please acknowledge any people who have helped you with this homework.

Question	Credits
1	
2	
3	

1. (30 points) A *k-ary* max-heap is a generalization of a binary max-heap in which each node in the heap has at most *k* children instead of two. In other words, it is an almost complete k-ary tree in which every node satisfies the max-heap property.

(a) Given the array `A[12] = [S, O, R, T, I, N, G, I, S, F, U, N]`, show how you will build the max-heap for k = 3 by inserting the elements of array A in succession. Show the snapshot of the max-heap after each element has been inserted and any heapify operations have been completed.

Solution: Solve similar to example in class... :)

(b) If the max-heap is represented by an array *A*, describe how to find the parent and the *k* children (at most) of element `A[i]`. Assume the array starts at `A[0]`.

Solution: Assuming that the first element of the array is `A[0]`, the `k` children of `A[i]` are given by `A[(i*k)+1]` to `A[(i*k)+k]`. The parent of `A[i]` is given by `A[floor((i-1)/k )]`. You can draw a simple example and verify that this is indeed correct.

(c) What familiar sorting algorithm is k-ary HeapSort really performing for `k = 1`? Justify your answer.

Solution: For `k=1`, the heap is really just a linked list. Building the heap consists of essentially inserting each element in the correct place in the list. This is exactly what *Insertion Sort* does.

2. (30 points) Bob, the builder, has a set N of n nuts and a set B of n bolts, such that each nut in N has a unique matching bolt in B. Unfortunately, the nuts in N all look the same, and the bolts in B all look the same as well. The only kind of comparison that Bob can make is to take a nut-bolt pair (a,b), such that $a \in N$ and $b \in B$, and test it to see if the threads of a are larger, smaller, or a perfect match with the threads of b. Describe an efficient algorithm for Bob to match up all the nuts in N with the corresponding bolts in B. What is the average running time of this algorithm in terms of nut-bolt comparisons that Bob must do?

Solution: We will assume that the sets N and B are represented as arrays. We will use the following randomized algorithm which is similar to randomized quicksort. Pick a random nut, say `N[i]` and by comparing it with every bolt we can partition the bolts into three groups: bolts whose threads are smaller than that of `N[i]`, a bolt that matches `N[i]` and the bolts whose threads are larger than those of `N[i]`. Next, using the bolt that matches the nut `N[i]` we perform a similar partition of the array N of nuts. After this partitioning step the nut `N[i]` is matched, the nuts and bolts smaller than `N[i]` are on one side and the ones larger than `N[i]` are on the other. By recursively applying itself to both the groups, our algorithm matches all the nuts and bolts. As with randomized quicksort, the average running time of this algorithm is O(`n*log(n)`).

3. (40 points) As we discussed in class, stable sorting algorithms maintain the relative order of records with equal values. That is, a sorting algorithm is stable if whenever there are two records R and S with the same key and with R appearing before S in the original list, R will appear before S in the sorted list as well. Give a simple scheme to make any comparison-based sorting algorithm stable. How much extra memory will your scheme need?

Solution: You need an extra field associated with each element in the array (alternatively, we can use a seperate array) that stores its position (*i.e.,* index) in the original unsorted array. When performing sorting using your favorite comparison-based sorting algorithm, break any ties according to the value in this additional field. This ensures that the sorting algorithm is stable. You need to store `n` such elements, so the space complexity of this enhancement is O(`n`).