Practice Problem Set 3

.

Problem 1 (****not possible for student presentation*)
We discussed the both bubble-down and bubble-up in class. Write the pseudocode for bubble-down(i) for index i where the heap is stored in array A. Assume that A.heapsize refers to the number of elements in the heap. Repeat for bubble-up(i).

Problem 2

- How do you insert into a heap? Assume that the array that contains the elements is not completely full. Write the pseudo-code and justify the runtime.
- Provide a recursive algorithm called BiggerThan(A,x,i) that takes as input a max-heap stored in A and a number x. The parameter i is used to designate the index of root of a sub-heap. The algorithm prints out all numbers from the subheap rooted at index i that are larger than x. Determine the recurrence for the runtime of your algorithm (in the worst-case) and determine the runtime in big-oh notation.

Problem 3

- Given a max-heap, suppose we would like to output the *minimum* element. Describe a recursive algorithm that solves this problem and provide the pseudo-code. Write a recurrence for the worst-case runtime $T(n)$ and determine it's big-oh notation. Is your recursive algorithm asymptotically faster than just a brute-force algorithm?
- Suppose you wish to store your data so that you can efficiently remove either the maximum element or the minimum element. What kind of data structure could you use? It is possible to perform both types of deletions in time $O(\log n)$?

Problem 4

- Suppose we are given a max-heap stored in array A. Recall that the maximum element is located at $A[1]$. Where can we find the second-largest element? What about the third-largest element?
- Suppose we wish to update our heap so that the largest, second-largest and third-largest elements are stored in positions $A[1], A[2], A[3]$ respectively. Describe an algorithm that takes as input a max-heap in array A and performs the necessary operations meet the requirement and include the pseudo-code. Justify why it runs in time $O(\log n)$.

Problem 5
A heap can be built using 3 children for each node in the tree instead of 2. How would you represent this heap an array? Describe where to find the children and the parent of node i. How would you update the bubble-down and bubble-up procedures?

Problem 6 (***not possible for student presentation*)

- Array A contains elements that are sorted in decreasing order. Is this a heap?
- In class we saw two algorithms for building a heap. Describe the runtime of each of these algorithms when the input array is already sorted in decreasing order.

Problem 7 (***not possible for student presentation*)

The first step of the algorithm Randomized-Select algorithm from class is to partition the elements about a randomly selected element in the array. Write the pseudo-code for Partition(A,s,f) which partitions the elements of array A about a randomly selected pivot, and *returns* the *rank* of the pivot. You may assume that you have access to a function that selects a random number is some range, for example Random(a,b) returns a random integer between the integers a and b inclusively. You may use additional space during the execution of your algorithm if needed.

Problem 8: (***not possible for student presentation*)

Write the pseudo-code for the Randomized-Select algorithm from class. Assume the algorithm takes as input the array A, start and finish indices s and f and the rank k. You may use the parition algorithm from question 7: Partition(A,s,f).

Problem 9:

Design an algorithm that takes as input an array of n numbers (where n is odd), and outputs all values that are larger than the median . You may make use of the **Select** algorithm from class. Justify why the runtime of your algorithm is $O(n)$.

Suppose we only want to print out the largest 10 elements from the array. Does this problem have the same asymptotic runtime?

Finally, what if we want to print out the top $n/4$ elements of the array. Does this problem have the same asymptotic runtime?

Problem 10: (***not possible for student presentation*)

Given as input n distinct numbers in an array, describe an algorithm that outputs the k smallest elements in *sorted order*. For example, $\{3, 2, 4, 5, 6, 10, 7, 1, 8, 9\}$ and $k = 4$ would result in $\{1, 2, 3, 4\}$. There are many options for how to approach this. Consider the following 3 approaches and compare the the runtime of each:

1. Sorting the entire array first
2. Building a heap
3. Using the **Select** algorithm to find the kth smallest element and the sorting only a subset of the elements.

Problem 11:

Suppose we carry out a version of the **Select** algorithm where we divide into groups of size $\sqrt{n}$ instead of groups of size 5. Does this alter the complexity of the algorithm? Justify your answer. What if we use groups of size 3?

Problem 12:

Consider the Randomized-Select algorithm that picks a pivot randomly. Use the substitution method to derive the worst-case runtime for Randomized-Select. What is the best-case runtime in big-oh notation?

Suppose a student writes a new magical Partition algorithm that is *guaranteed* to pick a pivot that is in the middle 75% of the sorted array. If the Randomized-Select algorithm from class now uses this new Partition algorithm, what is the worst-case runtime? Write a recurrence for the worst-case runtime, $T(n)$, and determine the big-oh value of $T(n)$. Why is this better than the original Randomized-Select algorithm?