David Sun
CMPS 102
Homework 2

Problem 1: Design 3 algorithms based on binary min heaps that find the kth smallest # out of a set of n #'s in time:

a) $O(n \log k)$

b) $O(n + k \log n)$

c) $O(n + k \log k)$

Note: For all problems, an array will be used to represent the heap.
a) For part a, we initially allocate an array of size k for our heap with all the elements initialized to $-\infty$ *(according to CLRS, a heap containing all duplicate keys can still be classified as a valid min-heap).* Then, we will make a pass through our array A from $A[1 \dots length[A]]$. For each A_i where $i \in \{1 \dots length[A]\}$, check if $-A_i$ is greater than the minimum of the heap. If such is the case, call remove on the heap, and insert $-A_i$ into the heap. We apply this operation for $length[A]$ iterations. Once we've scanned through all the elements in A, the kth smallest element should be $-1 * smallest(heap)$.
Correctness for a: The constraint in part a is that only the smallest element can be removed. In this case, we construct a heap using the negatives of the array elements. Thus if the *negative* of an array element is *greater than* the minimum heap value, then replace the minimum heap value with the negative of the array element. Since $\forall a, b \in \mathbb{R}$, if $a > b$ then $-b > -a$, replacing the minimum value in the heap containing negatives is equivalent of replacing the maximums in a max-heap. By the end of one pass through the array, the min-heap would contain the negatives of the k-smallest values with the k-th smallest of the array as the smallest value of the heap. To obtain the minimum, multiply the smallest value of the heap by -1 and that value would be the original kth smallest element of the array.
Runtime analysis for a: One pass through the array is an O(n) operation. Each array element comparison with the smallest heap element is an $O(1)$ operation. In the worst case, we would need to replace the minimum heap at element at every compare, which would cost us $2 \log k$ for a remove and insert operation. Thus, in the worst case, the algorithm generally runs in $n \cdot 2 \log k$ time, which suggests the algorithm is $O(n \log k)$

b) Call buidheap on the array A. This ensures A is now an array that satisfies the heap property. Since we want the kth smallest element, call remove $k - 1$ times. After $k - 1$ removes, the kth smallest element would be the smallest element on the heap.
Correctness for b: If A is a proper min-heap containing the original array elements, then a call to remove ensures the next smallest element is at the top of the heap. For j removes on the heap, the $j + 1^{th}$ smallest element is at the top of the heap. Thus, calling remove $k - 1$ times extracts the $(k - 1) + 1^{th}$ smallest element, which is the kth smallest.
Runtime analysis for b: Buildheap on an array of size n is an $O(n)$ operation. After buildheap executes, calling remove on a heap of size n $k - 1$ times costs $(k - 1) \log n$ time. In general, the algorithm runs in $n + (k - 1) \log n$ time, which means that the algorithm is $O(n + k \log n)$.

c)