

**Problem 1:** Design 3 algorithms based on binary min heaps that find the  $k$ th 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  $A[1 \dots \text{length}[A]]$ . For each  $A_i$  where  $i \in \{1 \dots \text{length}[A]\}$ , 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  $\text{length}[A]$  iterations. Once we've scanned through all the elements in  $A$ , the  $k$ th smallest element should be  $-1 * \text{smallest}(\text{heap})$ .

**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  $k$ th 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 buildheap on the array  $A$ . This ensures  $A$  is now an array that satisfies the heap property. Since we want the  $k$ th smallest element, call remove  $k - 1$  times. After  $k - 1$  removes, the  $k$ th 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  $j + 1^{\text{th}}$  smallest element is at the top of the heap. Thus, calling remove  $k - 1$  times extracts the  $(k - 1) + 1^{\text{th}}$  smallest element, which is the  $k$ th 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) Call buildheap on the array, ensuring the array is now a valid min-heap, call this heap  $A$ . Allocate another array of size  $k$  for our second heap. Let  $H$  be a heap containing a key-value pair, with its keys being the elements of  $A$  and the values being the index at which the element is found. Initially, insert the smallest of  $A$  along with the index of the smallest into  $H$ . At each iteration, remove the smallest value-index pair  $(A[i], i)$  from  $H$  and insert  $(A[2i], 2i)$  and  $(A[2i + 1], 2i + 1)$  into  $H$  (Note: if  $\text{length}[A] < 2i$ , do not insert either pair into  $H$ . If  $\text{length}[A] < 2i + 1$ , insert only  $(A[2i], 2i)$  into  $H$ ). After  $k - 1$  iterations, the  $k$ th smallest element is the smallest key in  $H$ . **Correctness for c: !!NEEDS TO BE COMPLETED!!**

**Runtime analysis for c:** If the original array has size  $n$ , buildheap on the original array is an  $O(n)$  operation. After buildheap, we execute  $k - 1$  operations of insert and remove on  $H$ , which has size  $k$ . In the worst case, one remove and two inserts will be called, which has a runtime of  $3 \log k$ . In general, the algorithm takes  $n + (k - 1) \cdot 3 \log k$  time to execute, which means the algorithm is  $O(n + k \log k)$ .

**Problem 2:** Consider the following sorting algorithm for an array of numbers (Assume the size  $n$  of the array is divisible by 3):

- Sort the initial 2/3 of the array.
- Sort the final 2/3 and then again the initial 2/3.

**Correctness:** Use induction on  $n$ , the size of the array, which we'll call  $A$ .

**Base Case:**  $n = 3$ . Sorting the initial 2/3 ensures  $A[1] \leq A[2]$ . Sorting the final 2/3 ensures  $A[2] \leq A[3]$ , thus ensuring that  $A[1] \leq A[3]$  and  $A[2] \leq A[3]$ . Sorting the initial 2/3 ensures again  $A[1] \leq A[2]$ , ensuring  $A[1] \leq A[2] \leq A[3]$ .

**Inductive Step:** Assume for all  $k$ ,  $3 \leq k < n$  that the algorithm properly sorts an array of size  $k$ . Then, for an array  $A$  of size  $n$ , we can partition up  $A$  into three segments  $B$ ,  $C$ , and  $D$ , with  $B$  containing  $A[1 \dots \frac{n}{3}]$ ,  $C$  containing  $A[\frac{n}{3} + 1 \dots \frac{2n}{3}]$ , and  $D$  containing  $A[\frac{2n}{3} + 1 \dots n]$ . Sorting the initial 2/3 of the array sorts  $B$  and  $C$ . By the inductive hypothesis,  $B$  and  $C$  are properly sorted and  $BC$  is a properly sorted segment with every element in  $B$  less than or equal to every element in  $C$ .

Sorting the final 2/3 of the array sorts  $C$  and  $D$ . By the inductive hypothesis,  $C$  and  $D$  are properly sorted and  $CD$  is a properly sorted segment with every element in  $C$  less than or equal to every element in  $D$ . This ensures that every element in  $D$  is greater than or equal to the elements in both  $B$  and  $C$ .

Sorting the initial 2/3 of the array sorts  $B$  and  $C$  again. In the process of sorting  $CD$ ,  $C$  contained the minimal values of  $D$ , which could be less than the elements found in  $B$ . Thus sorting the the initial 2/3 again ensures every element in  $B$  is less than or equal every element in  $C$ , by the inductive hypothesis.

Since  $B$ ,  $C$ ,  $D$  are properly sorted segments and  $\forall b \in B, \forall c \in C, \forall d \in D, b \leq c \leq d$ , the algorithm properly sorts the array.

**Runtime analysis:** The algorithm divides the problem into three sub-problems each of size  $\frac{2n}{3}$ . At each level, there is no other work done so we'll add an  $O(1)$  overhead. The recurrence can be written as  $T(n) = 3T(\frac{2n}{3}) + O(1)$  or  $T(n) = 3T(\frac{n}{1.5}) + O(1)$ .