

Practice Set 3: Solutions

Problem 1:

Bubble-down(i)

```
if  $i \leq \lfloor A.\text{heapsize}/2 \rfloor$            #make sure this is not a leaf.
    if  $2i + 1 \leq A.\text{heapsize}$        #make sure that there are two children
        if  $A[2i] > A[2i+1]$  and  $A[2i] > A[i]$  #the left child larger than parent and right child.
            Swap( $A[i]$ ,  $A[2i]$ )
            Bubble-down( $2i$ )
        else if  $A[2i+1] > A[i]$          #right child larger than parent
            Swap( $A[i]$ ,  $A[2i+1]$ )
            Bubble-down( $2i+1$ )
    else                               # there is only one child
        if  $A[2i] > A[i]$ 
            Swap( $A[i]$ ,  $A[2i]$ )
```

Bubble-up(i)

```
if  $i > 1$ 
    if  $A[i] > A[\lfloor i/2 \rfloor]$ 
        Swap( $A[i]$ ,  $A[\lfloor i/2 \rfloor]$ )
        Bubble-up( $\lfloor i/2 \rfloor$ )
```

Problem 2

- Inserting into a heap is similar to the step carried out in the iterative heap building method. We simply add the new element to the array at the next available index, increase the heapsize, and perform a bubble-up. Since the bubble-up operation takes time $O(\log n)$, the insertion runs in time $O(\log n)$. Suppose the new element to be inserted is k :

```
A.heapsize ++
A[A.heapsize] = k
Bubble-up(A, A.heapsize)
```

- Below is the pseudocode for BiggerThan(A, x, i). The parameter i is necessary in order to efficiently implement a recursive algorithm. The solution works by examining the value at $A[i]$. If it is larger than x , we print out the value. Then we make a recursive call to the two children of the node at position i . When a node is discovered whose value is *smaller* than x , no recursive calls are made, because we know that all values in the subtree are out of range.

BiggerThan(A, x, i)

```
if  $i \leq A.\text{heapsize}$            #only continue if index  $i$  is in the heap  $A$ 
    if  $A[i] > x$ .                 #if key is smaller than  $x$ , no recursive calls are necessary
        print  $A[i]$ 
        BiggerThan( $A, x, 2i$ )     #repeat for left child
        BiggerThan( $A, x, 2i+1$ )   #repeat for right child
```

In the worst-case, the algorithm above searches through both subtrees at every node. The subtrees are not *exactly* the same size, but we will approximate them here as about $n/2$ nodes in each subtree. In this case, the recurrence for the runtime is $T(n) = 2T(n/2) + c$. The solution to this recurrence can be determined using Master method, where $k = \log_2 2 = 1$, $n^k = n$, and $f(n) = c$. Therefore $T(n)$ is $\Theta(n)$.

In the best-case, the algorithm may return in constant time. If the root node is smaller than x , then no recursive calls are made. Therefore $T(n) = O(1)$ in the best case.

Problem 3:

This question demonstrates that although max-heaps are excellent for finding the max and updating values in the tree, they are not very efficient when it comes to finding the *minimum*. A description of the algorithm is given below. The algorithm takes as input the heap A rooted at index i . It finds the minimum of the heaps roots at each child (recursively), and compares those two results with the value at index i .

Find-min(A, i)

```
if  $i > A.\text{heapsize}$ 
```

```

    return infinity
m1 = Find-min(A,2i)
m2 = Find-min(A,2i+1)
return min(m1,m2, A[i])

```

The runtime of this algorithm is $T(n) = 2T(n/2) + c$, in any case, which has solution $\Theta(n)$. This can be verified by the master method, where $n^k = n$ and $f(n) = c$, in which case the dominant function is n , and therefore the runtime is $\Theta(n)$. Notice that this is exactly the same runtime if we had just run a brute-force algorithm and looped through all the elements in the array looking for the minimum.

If we wanted to efficiently retrieve (and possibly delete) the maximum and the minimum, one option might be to maintain concurrently both a max heap and a min heap. This works well for retrieval and insertion, since the operations can be done in time $O(1)$ and time $O(\log n)$ respectively. However, if we would like to delete the maximum element, then it must be deleted from both heaps. The deletion from the max heap runs in time $O(\log n)$, but the deletion from the min-heap only runs in time $O(\log n)$ if we can locate the element efficiently. Normally it takes $O(n)$ to search for an element in an array. This seems like bad news. There are two possible solutions to this problem:

1. The elements of the heap are objects with have attributes that point to their corresponding element in the other heap.
2. A type of heap called a Max-Min heap. The nodes at **even** levels are smaller than all the descendants, and the nodes at **odd** levels are larger than all the descendants. It combines both types of heaps into one.

Problem 4

- The second-largest element must be in one of the children of the root. Therefore, the second-largest element is at $A[2]$ or $A[3]$. The third-largest element has two values that are larger than it. Therefore, it could be a child of the root (in $A[2]$ or $A[3]$) or it could be a grand-child of the root (in either $A[4]$ or $A[5]$ or $A[6]$ or $A[7]$).
- In order to place the second-largest element in $A[2]$, we may have to swap it with $A[3]$. However, this could result in an invalid heap. In order to repair the heap after the swap, we simply bubble-down from $A[3]$ after the swap, since it may have a child that is larger than $A[3]$. Next, in order to place the third-largest element in $A[3]$, we may have to swap it with one of $A[4]$ or $A[5]$ or $A[6]$ or $A[7]$. After the swap, we need to bubble-down from the index $i = 4, 5, 6$ or 7 .

The pseudo-code below performs a constant number of operations in addition to Bubble-down, which runs in time $O(\log n)$ in the worst case. Therefore the overall runtime is $O(\log n)$

```

if A[2] < A[3]           #swap second largest element into place
    Swap A[2] and A[3]
    Bubble-down(A, 3)    #Repair subheap at index 3
k = 3
if A[4] > A[k]
    k = 4
if A[5] > A[k]
    k = 5
if A[6] > A[k]
    k = 6
if A[7] > A[k]
    k = 7
Swap A[3] and A[k]       #swap third-largest element into place
Bubble-down(A, k)        #Repair subheap at index k

```

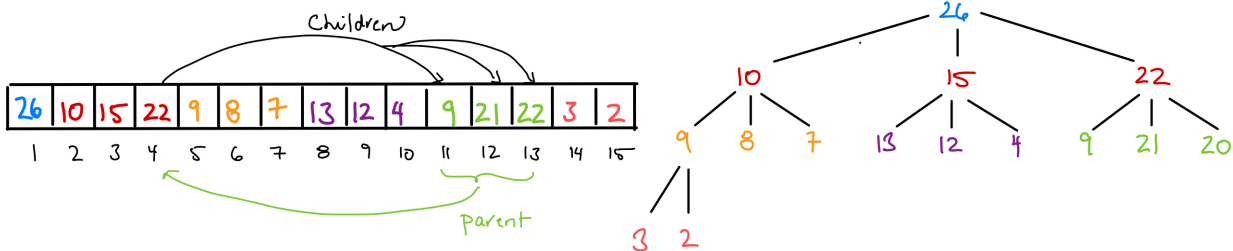
Problem 5

If we build a heap where each internal node has 3 children (instead of just 2) then the array would work as follows:

$A[1]$ contains the root

The parent of node i is at index $\lfloor (i+1)/3 \rfloor$.

The four children of the node at position i are $A[3i-1]$, $A[3i]$, $A[3i+1]$

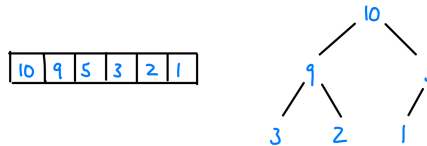


The height of this tree is $\Theta(\log_3(n))$, in fact it is exactly $\lfloor \log_3(2n) \rfloor$.

The bubble-up procedure works in the same way. We swap a node with its parent if and only if the parent is smaller. The bubble-down procedure would also be the same. We bubble-down as long as the current node has a child that is larger. We swap with the *largest* of the *four* children. The height of this new heap is still $O(\log n)$. Therefore both bubble up and bubble down still run in time $O(\log n)$. Using 3 children instead of 2 doesn't change the asymptotic runtime of the bubble-down or bubble-up procedures.

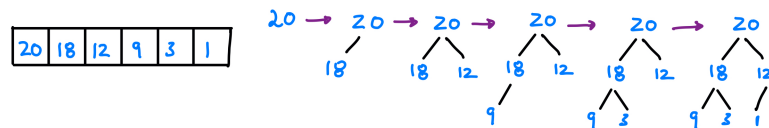
Problem 6:

- An array sorted in decreasing order represents a **max-heap**:

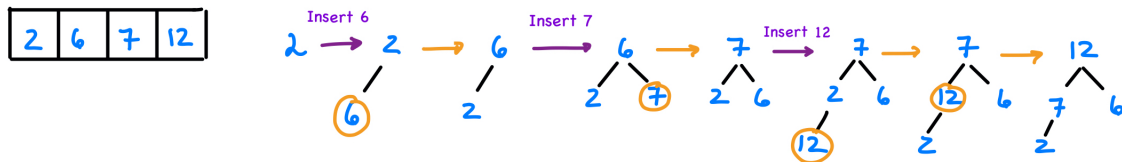


However, the converse is not true. A max-heap stored in an array is *not necessarily* in sorted order.

- Iterative method:** Suppose the original input array is sorted in *decreasing* order. As each new leaf is inserted, there is *no work* to be done. The leaf is already in its final position, and no swaps are made with the parent node. The heap is built in $O(n)$ time.



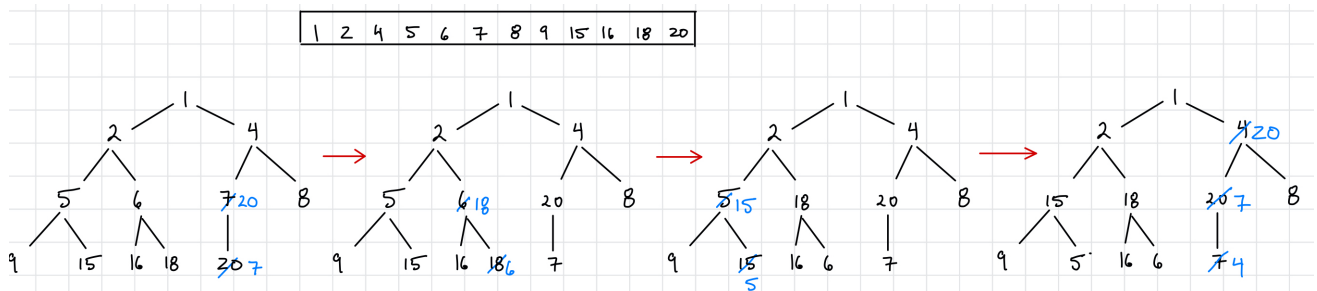
Suppose the original input array is sorted in *increasing* order. Then the iterative method will have to continue swapping each new leaf all the way up to the root position, taking time $O(\log n)$ for each insert. This represents the worst case of the iterative method. The overall run time is $O(n \log n)$, since there are a total of n nodes inserted to build the heap.



Bottom-up method:

Suppose the original input array is sorted in *decreasing* order. Then the array is already a heap. The bottom-up method would go through each node and verify that it is the max of its children. No swaps would be done. The algorithm would take $O(n)$ time.

Suppose the original input array is sorted in *increasing* order. Then the bottom-up method will need to carry out a *Bubble-down* operation *all the way down* to the leaf during each iteration of the bottom-up procedure. This represents the worst-case run time of the bottom-up method, and has runtime $O(n)$ as shown in class.



Problem 7:

The partitioning algorithm operates as shown in our class video, by using an external array. When the partitioning is complete, we call $\text{ArrayCopy}(B, A, s, f)$ which copies the elements from $B[1, \dots, n]$ into array $A[s, \dots, f]$.

Partition(A,s,f)

```

p = Random(s,f).      #A random number between s and f inclusively.
n = f - s + 1
initialize B[1.. n]
i = 1, j = n          #index i inserts to the left of B, index j inserts to the right
for count = s to f do: #loop through the elements of A and insert them into B
    if count ≠ p
        if A[count] < A[p]
            B[i] = A[count]
            i = i + 1
        else
            B[j] = A[count]
            j = j - 1
B[i] = A[p]            #insert the pivot element into the correct position
ArrayCopy(B,A,s,f)    #copy B into A
return j               #return the rank of the pivot

```

Problem 8:

Recall that the $\text{Partition}(A,s,f)$ algorithm from problem 6 returns the **rank** of the randomly chosen pivot (not the exact index).

Randomized-Select(A,s,f,k)

```

if s = f and k = 1.    #If the array has size 1, and the rank is 1, then return the first element
    return A[s]
else
    p = Partition(A,s,f) #p is the RANK of the pivot element (not the array index, the actual rank)
    if p = k
        return A[p + s - 1] #return the pivot element by adjusting the rank p to the start index
    else if k < p
        return Randomized-Select(A, s, s + p - 2, k) #Recursive call to the left subarray
    else return Randomized-Select(A, s + p, f, k - p) #Recursive call to the right subarray

```

Problem 9:

- Given an array of size n , we can use the Select algorithm to determine which element represents the median. Once we have this element, we can loop through the array and output all elements that are larger than the median. The Select algorithm runs in $\Theta(n)$ time. Therefore the entire algorithm runs in $O(n)$.

```

k = Select((n - 1)/2)
for i = 1 to n
    if A[i] > k
        Print A[i]

```

- In order to print out the largest 10 elements, we can run the Select algorithm to determine the element of rank $n - 9$. Once we have this element, we can loop through the array and output all elements that are larger than that element.

```

k = Select(n - 9)
for i = 1 to n
    if A[i] > k
        Print A[i]

```

The two main steps are the same as in the previous algorithm, and therefore the runtime is the same.

- In order to print out the largest $n/4$ elements, we can run the Select algorithm to determine the element of rank $n - n/4 + 1$. Once we have this element, we can loop through the array and output all elements that are larger than that element.

```

k = Select(3n/4 + 1)
for i = 1 to n
    if A[i] > k
        Print A[i]

```

The two main steps are the same as in the previous algorithm, and therefore the runtime is the same.

Note that all three options have the same runtime. However, option 2 is the only one that could be solved in time $O(n)$ using brute-force.

The runtime of the Select algorithm is $\Theta(n)$, and the for-loop runs in time $O(n)$, therefore the entire algorithm runs in time $O(n)$. Therefore the whole algorithm runs in time $\Theta(n)$.

Problem 10:

We demonstrate the three different approaches using $k = 3$ and the array:

$\{3, 6, 7, 15, 8, 1, 9, 2, 18, 13\}$

The three different approaches to solving this problem are:

Option 1: If we sort the entire array, using Mergesort for example, this takes time $\Theta(n \log n)$.

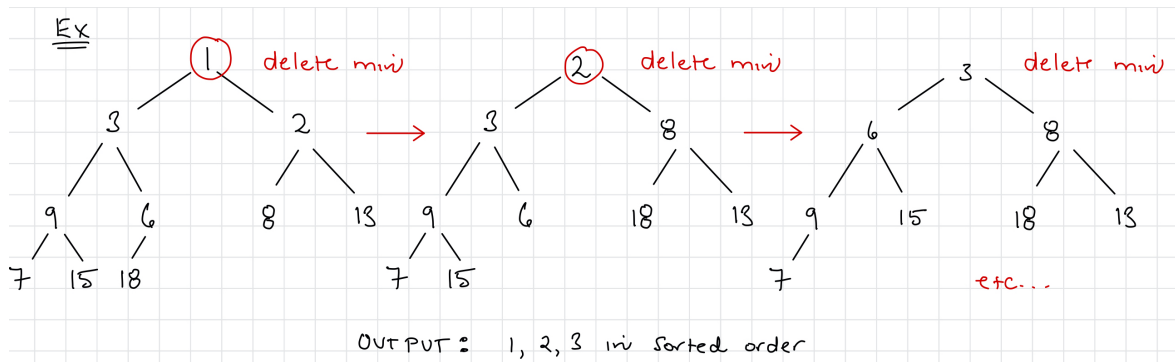
$1, 2, 3, 6, 7, 8, 9, 13, 15, 18$

Then we could simply pass through the sorted list from back to front and output the first k items from the list.

$1, 2, 3, 6, 7, 8, 9, 13, 15, 18$

The step would take time $\Theta(k)$. The total runtime would then be $\Theta(n \log n)$.

Option 2: We could build a min-heap in time $O(n)$. Then we could call *Delete-Min* exactly k times. Each call to Delete-min takes time $O(\log n)$, and so deleting the min k times takes $O(k \log n)$. The entire procedure (building the heap and deleting the min's) takes time $O(k \log n + n)$. Depending on the value of k , this results in a variety of runtimes. For example, if $k = 3$, then the runtime is $O(\log n + n) = O(n)$, but if $k = \sqrt{n}$, then the runtime is $O(\sqrt{n} \log n + n) = O(n)$.



Option 3: We could use the Select algorithm to find the element of rank k . This would take time $O(n)$. For $k = 3$ for example, the result is element 3. Next, we would loop through the array to find all numbers that are *at most* 3. This takes time $O(n)$ and results in the set:

$\{3, 1, 2\}$

Finally we sort this set using Merge-sort, resulting in the final output $\{1, 2, 3\}$. Since there are only k items in this set, the time is $O(k \log k)$. Thus the total runtime of this third option is $O(n + k \log k)$. Again, depending on the value of k , this results in different runtimes. If $k = 3$, the runtime is $O(n)$. If $k = \sqrt{n}$, the runtime is $O(n + \sqrt{n} \log n) = O(n)$.

Problem 11:

- If we divide our numbers into groups of size \sqrt{n} , then there would be exactly \sqrt{n} groups. Recall that the first step of the Select algorithm is to find the median of each group. If we find the median of each group using insertion sort (as in the original algorithm), then this takes time $O((\sqrt{n})^2) = O(n)$, and since there are a total of \sqrt{n} groups, then sorting all of them takes $\sqrt{n} \cdot O(n) = O(n^{3/2})$. If we find the median of each group using mergesort, then sorting each group takes time $\Theta(\sqrt{n} \log \sqrt{n})$, but again since there are \sqrt{n} groups, the total time is $\sqrt{n} \cdot \Theta(\sqrt{n} \log n) = \Theta(n \log n)$. Unfortunately, regardless of what sorting algorithm we use, we cannot find the median of the medians in $O(n)$ time. Therefore this algorithm is not as efficient as the Select algorithm from class, which runs in time $O(n)$.

- If we used groups of 3 instead of groups of 5:

1. We can find the median of each mini-group using Insertion-Sort, in constant time per column. We repeat this for all $n/3$ groups. Therefore all medians are found in time $O(n)$:
2. Next we make a recursive call to find the median of medians (called x), which takes time $T(n/3)$.
3. Next, we partition the elements into those that are larger than x and those that are smaller than x . This takes time $\Theta(n)$
4. Finally we make a recursive call to one of the above sets. The best-case scenario would be a recursive call on input of size at least

$$\frac{1}{2} \cdot \frac{n}{3} \cdot 2 = \frac{n}{3}$$

and thus the worst-case scenario is to a set of size at most $2n/3$. So the recursive call takes time $T(2n/3)$ in the worst case.

The worst-case runtime of this recursive algorithm is :

$$T(n) = T(n/3) + T(2n/3) + cn$$

We can show by substitution method that this is not $O(n)$ because it is in fact $\Omega(n \log n)$

Problem 12

The worst-case runtime of the Randomized-Select is when the algorithm picks the maximum element as the random pivot. In this case the recursive call is made to an array of size $n - 1$. The runtime recurrence is $T(n) = T(n - 1) + cn$. We can show this has solution $O(n^2)$: *Goal*: Show $T(n) \leq dn^2$.

Assume: $T(n - 1) \leq d(n - 1)^2$

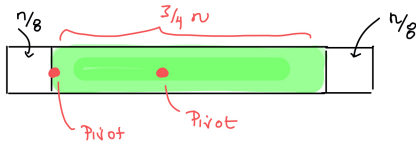
Substitute:

$$\begin{aligned} T(n) &= T(n - 1) + cn \\ &\leq d(n - 1)^2 + cn \\ &= d(n^2 - 2n + 1) + cn \\ &\leq dn^2 - 2dn + d + cn \\ &\leq dn^2 - 2dn + dn + cn \\ &\leq dn^2 - n(d - c) \\ &\leq dn^2 \end{aligned}$$

where the last line is true as long as $d \geq c$.

In the best-case scenario, the random pivot is actually the element k . In this case, we return the element of rank k after the partitioning step. Therefore the algorithm runs in time $O(n)$.

If the Randomized-select algorithm was always able to select a pivot from the middle 75% of the array, then the recursive call would be made to an array of size at most $7n/8$.



The recurrence for the runtime is $T(n) = T(7n/8) + cn$, which can be solved using the master method: $k = \log_{8/7} 1 = 0$, and so $n^k = 1$. Therefore $T(n)$ is $\Theta(n)$.