1.Heaps.

(a) What are the minimum and maximum number of elements in a heap of height h?

Assuming height index starts on 1, otherwise when height index starts at 0 answers are (2^{h+1} - 1 and 2^h):

A heap of height h has a maximum of $2^h - 1$ elements when each non-leaf node has two child nodes.

A heap of height h has a minimum of 2^{h-1} elements when the lowest level of the heap has only one node positioned on far left as per the heap property.

(b) Show that an n-element heap has height ⌊log n⌋.

A heap is a complete binary tree, meaning it shares the characteristic of having a height of ⌊log n⌋ where n = total count of elements in the heap.

Going back to my answer (a), we know the maximum and minimum size of a heap with height h.

$2^{h-1} <= n < 2^h - 1$

$h-1 <= \log(n) < h$

Because the height is an integer, we have h = floor(lg(n))

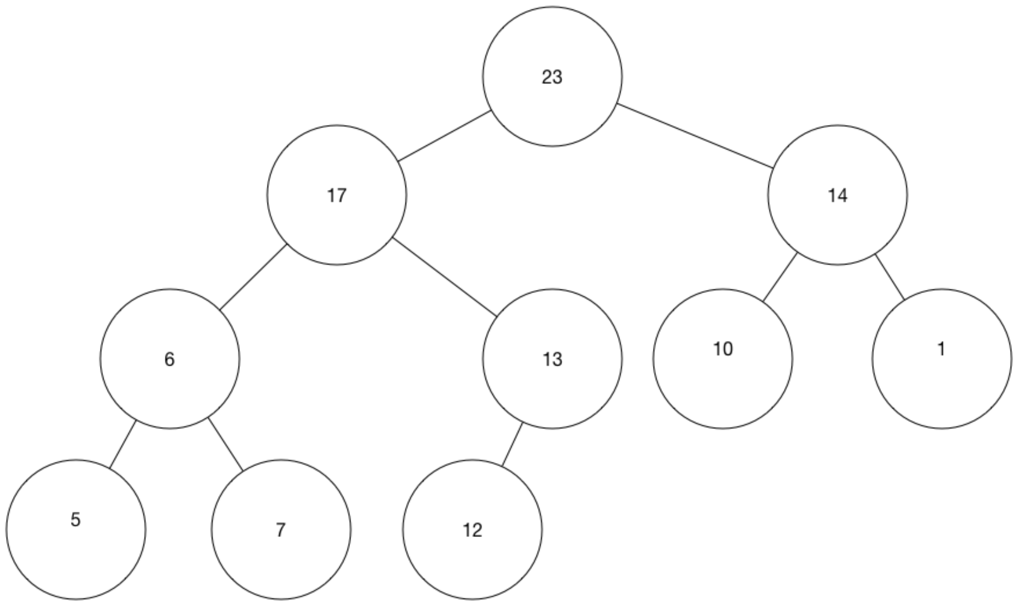
(c) Show that in any subtree of a max-heap, the root of the subtree contains the largest value occurring anywhere in that subtree.

This remains true as per the heap property, stating that in a max-heap the largest element must remain as the root, where the left and right children must be <= root.

If an element n is the root of the subtree, than both of n's children nodes are <= to n. Since this transitive property remains true for the children nodes, all of the descendants will be <= the root, making the root node n the largest value.

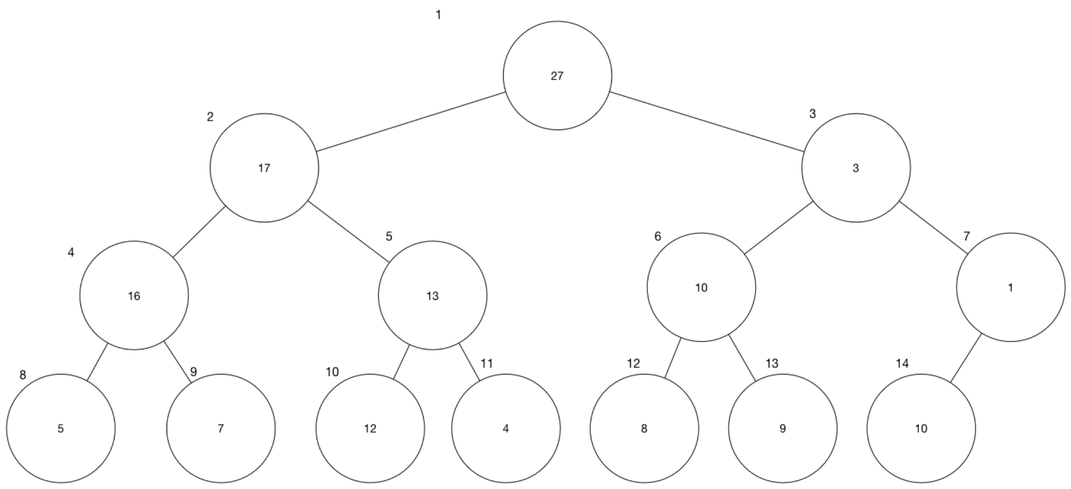
(d) Is the sequence ⟨23, 17, 14, 6, 13, 10, 1, 5, 7, 12⟩ a max-heap? Why or why not?

Sequence Visualized: This sequence is not a max-heap because the leaf element (7) does not satisfy the heap property. It is larger than its parent node.

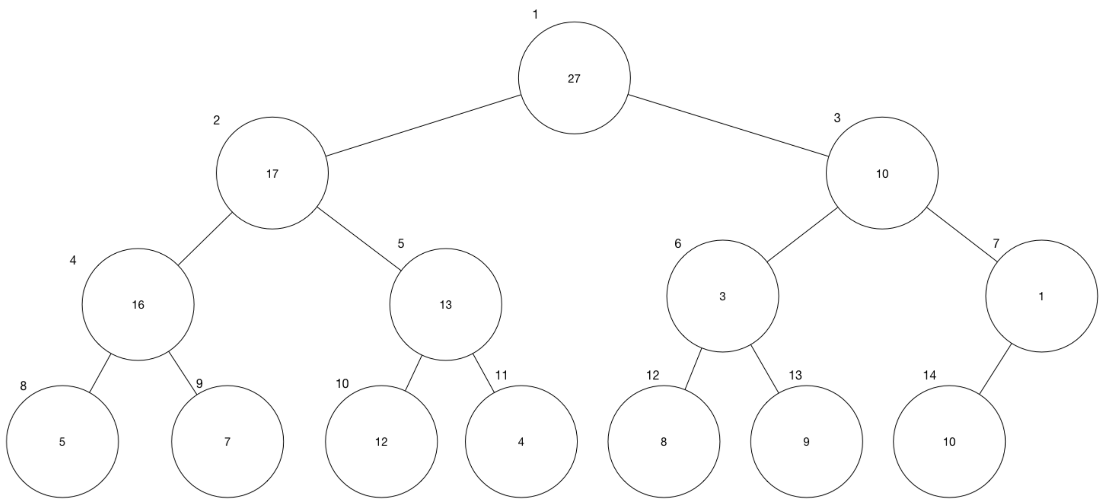


2. Heaps.

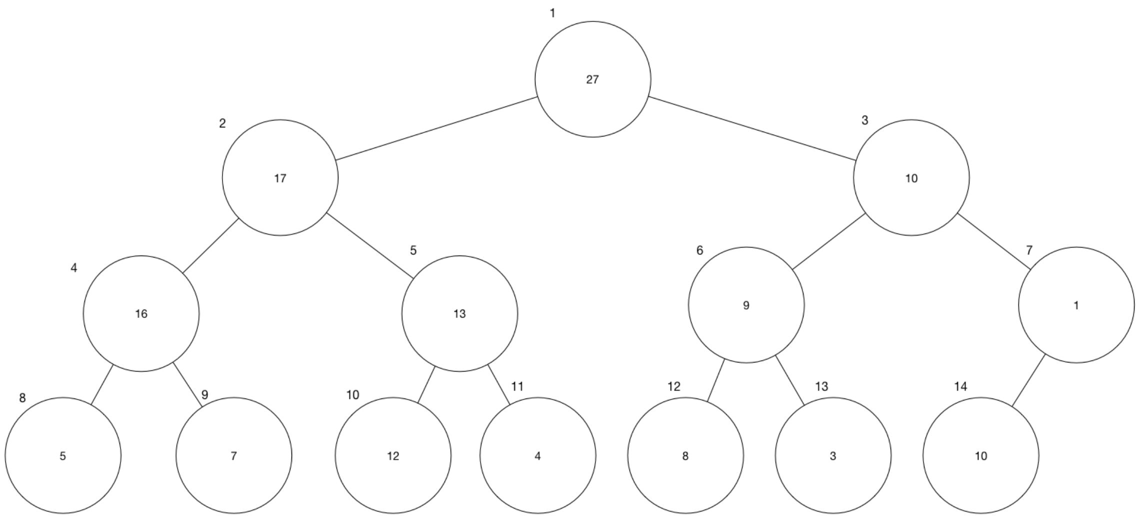
(a) Illustrate the operation of Max-Heapify(A, 3) on the array $A = \langle 27,17,3,16,13,10,1,5,7,12,4,8,9,10 \rangle$.



Iteration at index <i = 3>, swap element 3 with element 6



Recursively iterate through child nodes to ensure heap property is maintained
Starting at node 6, max-heapify subtree. Swap larger child with element 6

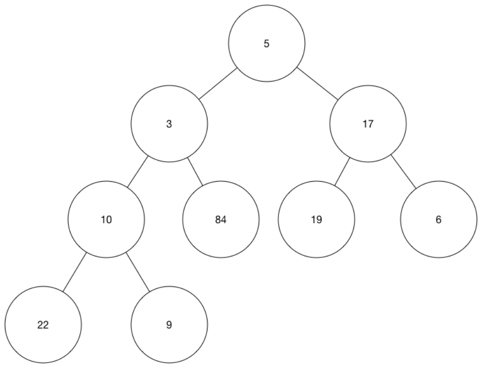


Heap Property is maintained. The Max-Heapify procedure is complete.

(b) Illustrate the operation of Build-Max-Heap(A) on the array A = ⟨5,3,17,10,84,19,6,22,9⟩.

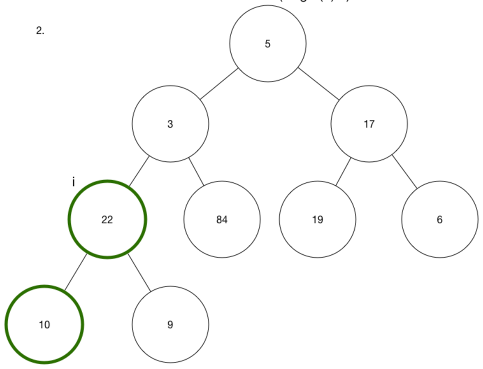
Build Heap

1.



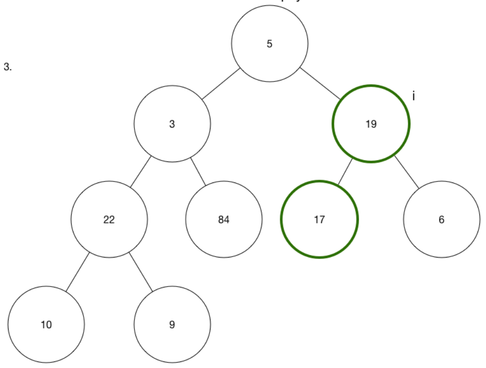
Perform Max-Heapify from index i to 1 (i--)
where i = floor(length(A)/2)

2.



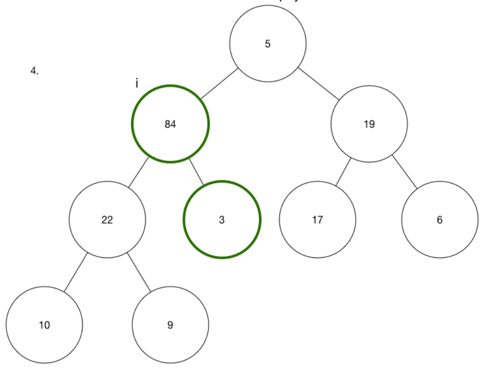
i--
Perform Max-Heapify on index 3

3.



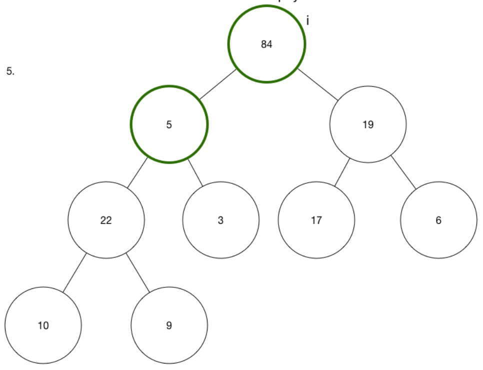
i--
Perform Max-Heapify on index 2

4.



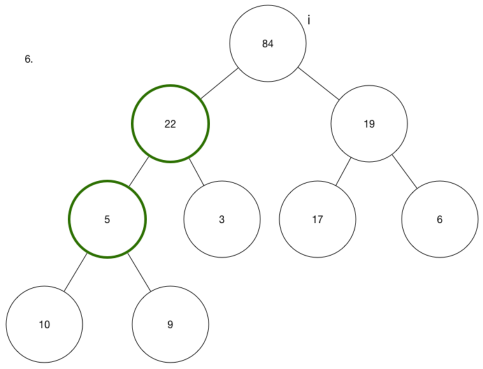
i--
Perform Max-Heapify on index 1

5.



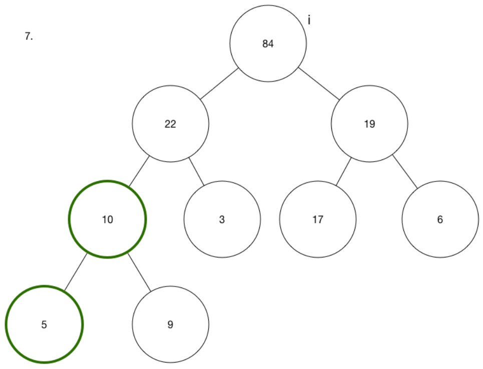
Recursively perform max-heapify on branch (index 2)

6.



Recursively perform max-heapify on branch (index 2)

7.



Build-Max-Heap(A) process is complete

(c) You are given a list of numbers for which you need to construct a min-heap. How would you use an algorithm for constructing a max-heap to construct a min-heap?

To create a min-heap using the build-max-heap algorithm, you would use HeapSort. HeapSort takes in an array <A>, passes it into Build-Max-Heap(A), and sorts it in ascending order. This results in a min-heap array <A>.

3. Heapsort.

(a) Argue the correctness of Heapsort using the following loop invariant.

At the start of each iteration of the for loop, the subarray A[1 . . . i] is a max-heap containing the i smallest elements of A[1...n], and the subarray A[i +1...n] contains the n–i largest elements of A[1 . . . n].

Initialization: The subarray A[1…i] is empty, thus the loop invariant stands correct.

Maintenance: In A[1…i], a subarray of sorted elements, A[1] is the smallest element and is smaller than its parent elements at positions A[(i+1)…n]. As the loop iteration completes, the loop invariant holds true for the following iteration or termination phase.

Termination: After the loop condition returns false, it remains by our loop invariant maintenance that A[2…n] is sorted. Also, A[1] was also proven to be the smallest element in the array, which makes the array sorted.

4. Quicksort.

(a) Argue that Tail-Recursive-Quicksort(A, 1, n), where n is the number of elements in array A, correctly sorts the array A.

Both QuickSort (QS) and Tail-Recursive-QuickSort (TRQS) call the same Partition method, where each partition (left and right) call QS on themselves. The difference between QS and TRQS is that in TRQS, when the first recursive call returns, the original arguments (p, r) are used again, rather than splitting and calling half like QS. Since p = q + 1, the next loop iteration executes with arguments q + 1 and r, which is identical to having another recursive call with these parameters. This maintains loop invariance.

(b) Describe a scenario in which Tail-Recursive-Quicksort's stack depth is Θ(n) on an n element input array.

With Tail-Recursive-Quicksort, all recursive calls on the left partition (not right, because this is TRQS) will give Θ(n) stack depth. The stack depth is n when the array is reverse sorted.

There is a special occurrence where we will have a stack depth of 1 when the array is sorted. This happens because further recurrences will not be called, since the array is already sorted. Only one iteration happens.

5. Partition.
The colors of the Dutch national flag are red (R), white (W), and blue (B). The Dutch national flag problem is to rearrange an array of n characters R, W, and B so that all the Rs come first, the Ws come next, and the Bs come last. Design a linear in-place algorithm for this problem and show that your algorithm is O(n). Hint: Use the idea of the Partition algorithm of Quicksort.

At first sight, it may seem trivial on how to sort these three color (or character) values; however, after assigning an integer value of 0, 1 and 2 respectively, the solution becomes much clearer. We can use the standard Partition algorithm after reassigning R, W, and B to 0, 1, 2 respectively.

Because we only have 3 element values, if we select our median element value (1) as our pivot, the array will be sorted in O(n). This is because everything to the left of our pivot (0) will be in the correct position, and everything to the right of our pivot (2) will be in its correct position.

```
int dutchSort(A[], left, right)
{
 // Pivot point, should be last element with value of 1
 int pivot;
 // Reassign colors to integer values, pick out the pivot
 for i = 0 to A.length {
  if (A[i]) == 'R' {A[i] = 0}
  if (A[i]) == 'W' {A[i] = 1; pivot = i}
  if (A[i]) == 'B' {A[i] = 2}
 }
 swap (A[pivot], A[right])
 temp = left
 for i = left to (right-1) {
  if (A[i] <= A[right]) {
   swap(A[i], A[temp])
   temp++
  }
 }
 swap(A[temp], A[right])
 return temp
}
```

6. Select.
In the algorithm Select, the input elements are divided into groups of 5.

(a) Will the algorithm work in linear time if they are divided into groups of 7?

If the Select algorithm divides input elements into groups of 7, as opposed to groups of 5, it will still work in linear time. This can be proven by setting up a proof of induction where our group size is 2k+1 when k is an arbitrary integer greater than 1.

Because this induction holds true when k is equal to 2, which was stated and proven by the traditional Select algorithm, we have thus shown that while k is equal to 3 [2(3)+1], our induction remains true.

By setting up a recurrence, you can use substitution to show that the runtime is O(n).

This is indeed true for any number of elements greater than 1.

(b) Argue that Select does not run in linear time if groups of 3 are used.

For groups of 3, the algorithm does not run in linear time. This is shown by setting up a recurrence, which shows that there is no linear solution to the recurrence, thus the algorithm does not run in linear time.

More than half of the groups of 3 would have elements that are greater than x, exclusive of the group which contains x. Only one group would have less than 3 elements, when n%3!=0.

Only odd group sizes greater than or equal to 5 run in linear time.

7. Largest i numbers in sorted order.
Given a set of n numbers, we wish to find the i largest in sorted order using a comparison-based algorithm. Find the algorithm that implements each of the following methods with the best asymptotic worst-case running time, and analyze the running times of the algorithms in terms of n and i.

(a) Sort the numbers, and list the i largest.
In order to sort a list of numbers, stating the i largest, we can use any sorting algorithm. One we have learned so far, which will work for this case, is MergeSort.

MergeSort can solve this problem in $\Theta(n \log n)$ runtime.

(b) Build a max-priority-queue from the numbers, and call Extract-Max i times.

In order to build a max-priority-queue from an array of numbers, we can use the Build-Max-Heap algorithm. This has a runtime of $\Theta(n)$.

To fulfill our second request, calling Extract-Max i times, we will have a runtime of Θ(i log(i)).

To conclude with our total runtime, we simply add up our two runtimes, resulting in Θ(n + i log i).

(c) Use an order-statistic algorithm to find the ith largest number, partition around that number, and sort the i largest numbers.

In order to use an order-statistic algorithm to find the ith largest number, partitioning and sorting around that number, we will have to use a few different algorithms.

For the first part, finding the ith largest number, we can use the Select algorithm. The Select algorithm has a runtime of Θ(i).

For the second part, partitioning around that number, we simply use the Partition algorithm, passing our ith largest number as the argument. The Partition algorithm has a runtime of Θ(i) also.

Lastly, to sort the ith largest numbers we will use a sorting algorithm like MergeSort, as stated in part (b) of this question. The MergeSort algorithm has a runtime of Θ(i log(i)).

To conclude with our total runtime, we simply add up our three runtimes, resulting in Θ(i + i log(i)).