**Question 1)**

Input: A is an array of real numbers, indexed from 1 to n

1: n ← length(A)

2: for j ← 1 to n/2 do

3: k = n + 1 − j

4: A[j] ← A[j] + A[k]

5: A[k] ← A[j] − A[k]

6 :A[j] ← A[j] − A[k]

The following pseudocode is simply changing the order of the array in the reverse order. With more detail, every iteration changes the element of j and k. For example, if there is an array [1,2,3,4,5,6,7,8], after running the code above, it will change into [8,7,6,5,4,3,2,1].

Here is the detail of how it will run.

n = 8(the length of the array, which is 8)

for j = 1 to 4,

k = 8

A[1] = A[1] + A[8], which is (1+8=9) so A[1] will have the value of 9 stored

A[8] = A[1] – A[8], which is (9-8=1) so A[8] will have the value of 1 stored

A[1] = A[1] – A[8], which is (9-1=8) so A[1] will have the value of 8 stored

At the end of first iteration, array will be [8,2,3,4,5,6,7,1]

k = 7

A[2] = A[2] + A[7], which is (2+7=9) so A[2] will have the value of 9 stored

A[7] = A[2] – A[7], which is (9-7=2) so A[7] will have the value of 2 stored

A[2] = A[2] – A[7], which is (9-2=7) so A[2] will have the value of 7 stored

At the end of second iteration, array will be [8,7,3,4,5,6,2,1]

k = 6

A[3] = A[3] + A[6], which is (3+6=9) so A[3] will have the value of 9 stored

A[6] = A[3] – A[6], which is (9-6=3) so A[6] will have the value of 3 stored

A[3] = A[3] – A[6], which is (9-3=6) so A[3] will have the value of 6 stored

At the end of third iteration, array will be [8,7,6,4,5,3,2,1]

k = 5

A[4] = A[4] + A[5], which is (4+5=9) so A[4] will have the value of 9 stored

A[5] = A[4] – A[5], which is (9-5=4) so A[5] will have the value of 4 stored

A[4] = A[4] – A[5], which is (9-4=5) so A[4] will have the value of 5 stored

At the end of fourth (last) iteration, array will be [8,7,6,5,4,3,2,1]

End of loop

[1,2,3,4,5,6,7,8] changed into reverse order [8,7,6,5,4,3,2,1]

1. The sub-array A[1 . . . j] contains its original contents in their original order.

The following is **False** because in the example that I provided, A[1..j] (which was A[1…4] in my example) was [8,7,6,5], and not [1,2,3,4]. This happens because you reverse the order of the first element and the last element, second element and the second last element, and etc. Since the value changes, the original content does not exist in the original order.

1. The sub-array A[1 . . . j] contains the original contents of sub-array A[(n + 1 − j). . . n] in reverse order.

The following is **True** because as stated above, every iteration simply swaps the element in A[j] and A[k] where j starts from 1 and goes up until it reaches the total length of array divided by 2 and k starts from the length of the array and goes down until j reaches the length of array divided by 2. Therefore, the sub-array A[1…j] contains the contents of A[(n+1-j)…n] in the reverse order. In the example above that I provided, A[1…4] was changed into the reverse order of A[5…8] (A[1,2,3,4] changed into A[8,7,6,5]).

1. The sub-array A[1 . . . j] contains its original contents in reverse order.

The following is **False** because the sub-array A[1..j] does contains the reverse order but not the reverse order of it but gets the reverse order of A[(n + 1 − j). . . n]. In my example above, A[1,2,3,4] does not contain A[4,3,2,1] but A[8,7,6,5].

1. The sub-array A[(n + 1 − j). . . n] contains its original contents in their original order.

The following is **False** because the sub-array A[(n + 1 − j). . . n] does not contain the original content. Rather, it contains the sub-array A[1…j] in the reverse order. In my example, A[8,7,6,5] contains the element of A[1,2,3,4], not A[8,7,6,5].

1. The sub-array A[(n + 1 − j). . . n] contains the original contents of sub-array A[1 . . . j] in reverse order.

The following is **True** because the as explained in part 2 of the same question, every iteration swaps the element in A[j] and A[k] where j starts from 1 and goes up until it reaches the total length of array divided by 2 and k starts from the length of the array and goes down until j reaches the length of array divided by 2. Therefore, the sub-array A[(n + 1 − j). . . n] contains the contents of A[1…j] in their reverse order. In my example, A[5,6,7,8] contains A[4,3,2,1].

1. The sub-array A[(n + 1 − j). . . n] contains its original contents in reverse order.

The following is **False** because the sub-array A[(n + 1 − j). . . n] contains the reverse order of array A[1..j] not the reverse order of A[(n + 1 − j). . . n]. In my example, A[5,6,7,8] contains A[4,3,2,1] not A[8,7,6,5].

**Question 2)**

**Pseudocode**:

findKSmallestItems(k, heapArray):

newArray 🡨 first k elements of the heapArray

totalLength 🡨 len(heapArray)

quickSort(newArray)

for k to totalLength:

If heapArray[k] < newArray[last element]:

newArray[last element] 🡨 heapArray[k]

QuickSort(newArray)

return newArray

**How It works with example:**

In my pseudocode, we first construct an array of elements that contains the first k elements from the heap array that was one of the inputs. By making this array, we temporarily have the k elements that are small. Then, we construct quicksort to the newArray. The reason we use quicksort is because quicksort, in average case, runs in n\*log(n) because we are sorting the sub-array that is already somewhat sorted and is never in the worst case due to the function of the heap array. By doing this, we can keep the biggest element of the first k elements at the end and make it easy to compare. If we do not use this, we would have to search the biggest element every time within the newArray, which is wasting time. For the remaining elements that were not selected (for remaining n-k elements later in the heap array that we got as input), we simply compare that to the last element of the newArray, which is believed to be the biggest number within the array. If the original heap array contains smaller elements than the last element of the newArray, we simply substitute that value to the last element of the newArray and do quicksort again to make the values easy to compare again. When the loop ends, it means that there does not exist any elements that are smaller than the newArray, so we return the newArray.

**Ex)**

1. From textbook, there is a heap array of [1,2,5,10,3,7,11,15,17,20,9,15,8,16] and let k = 6.
2. So, we make an array of first 6 elements from the heap array, which is [1,2,5,10,3,7] and do quicksort, which is [1,2,3,5,7,10].
3. Starting from index k=6, which is 11, we compare to the last element of the array, which is 10.
4. Since 11,15,17, and 20 are greater than 10, we skip them.
5. 9 is less than 10, so we substitute that value to the end of newArray and do quicksort again, which is [1,2,3,5,7,9].
6. Skip 15 since it is greater than 9.
7. Since 8 is less than 9, we substitute that value with 9 and do quicksort again, which makes [1,2,3,5,7,8].
8. Skip 16 since it is greater than 8.
9. Return the array [1,2,3,5,7,8].

The numbers 1,2,3,5,7,8 are the six smallest numbers from the original heap array.

**Proof**:

This pseudocode always works because after getting the first k elements from the original heap array, we simply compare the greatest number from the first k elements to the remaining elements that are left in the original heap array and substitute the value that is less than the greatest number in the array that we constructed.

**Run-time:**

The run-time of this pseudocode is O(k logk) because creating a new array with first k elements and setting few variable is constant time c. Then, quicksort will take k\*log(k) time. Then, comparing the elements within the rest of the array requires log(k) for each step and therefore takes (n(total length of array)-k)\*log(k) since the comparison requires just the comparing between the last element of quicksort that we constructed.

Therefore, k\*log(k) + c + (n-k)\*log (k) is simply runtime of k\*log(k).

**Question 3)**

numberOfLeaves(v):

if(v == null):

return 0

if(v.leftChild == null and v.rightChild == null):

return 1

else:

return numberOfLeaves(v.leftChild) + numberOfLeaves(v.rightChild)

**Proof:**

The input of the function is the root node of the binary tree. This function returns the total number of leaves of the binary tree. If the node is null itself, there cannot exist any left and right child so there does not exist any leaves, where we return 0. However, for the remaining cases, we look at the case where leftChild and rightChild are both null for a node. If both child are null, then the node is a leaf node and we simply found 1 leaf node so we return 1. However, in other cases where there is both left and right child or there is one left child or right child, we have to continue to check until the leaf node comes out, where we will call two functions itself by having input of leftChild and rightChild to find it for us.

numberOfLeavesWithoutRecursion(v):

Queue<Node> q = new Queue <>

Add the Node v to Queue q

totalLeaves 🡨 0

if(v == null):

return 0

else:

while(queue is not Empty):

Node x 🡨 Take the first element from Queue (poll)

if (x.leftChild and x.rightChild are both null):

totalLeaves ++

else if(x.rightChild does not equal null and x.leftChild does not equal null):

add x.rightChild to Queue q

add x.leftChild to Queue q

else if(x.rightChild does not equal null):

add x.rightChild to Queue q

else:

add x.leftChild to Queue q

return totalLeaves

**Proof:**

The input of the function, just like part(a), is the root node of the binary Tree and this function calculates the number of leaves without using recursion. Similar to the part (a), if the node that we called is already empty, we return 0 since if the node is already null, there cannot exist any left or right child. If it is not, we will enter that node into Queue (which is first come first out). While the queue is not empty, which means that if there are nodes that have left node or right node exist, we run the loop of the following. We take the first node out of the queue and check if there are no left and right child. If there is none, it means that the node is a leaf node and add 1 to the totalLeaves. Since left and right child are empty, we do not need to add them to the queue since adding it will cause error. If at least one of right or left child exists, that means the node is not a leaf node and therefore we need to add them to queue for further search of the leaf node. We first need to check if both of left and right child are not null in order to check. If they are both not null, we add both the right child and left child of the node to the queue. If only one of them are not null, we insert that node to the queue. I did it as setting checking right child first. If right child of the node is not null, we put the right child to the queue for further check. If right child is null, it means that left child is not null, so we put the left child to queue, again, for further search for leaf node. The order I put is important because you first need to check whether both are not null. If you only check whether the right node is not null and put it into queue, it will skip the other else if statements and continue even if the left child might not be equal to null. Therefore, you need to check if both are not null first. Once you find every leaf node, there will be no more elements in the Queue anymore since everytime you find a leaf node, you do not insert anything to the queue. When the queue is empty, we can simply return the total number of leaves.