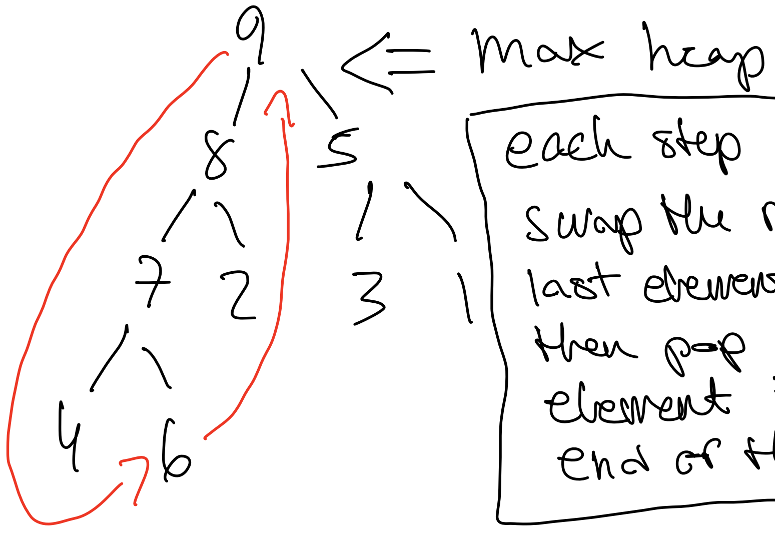
#1 [20 points]

In class we learnt how to apply the Heap sort algorithm to a set of numbers. Please refer to this algorithm ("Heap Sort") on Canvas. For a set of numbers, the steps in the first part involving Build Max heap are posted on Canvas: "Heap sort Part 1 Build Max heap".

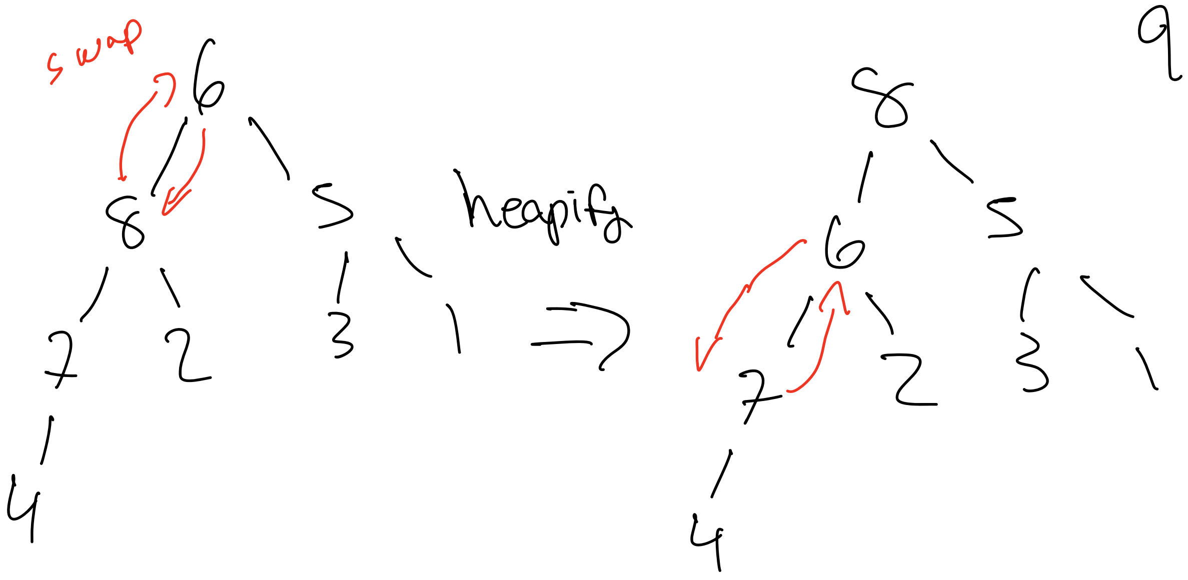
Working with the same set of numbers, now that the heap is built, sketch the next steps of the algorithm, which will eventually sort the array. Draw the tree and corresponding array at each step, just like the steps in the partial example solution.

9	8	5	7	2	3	1	4	6

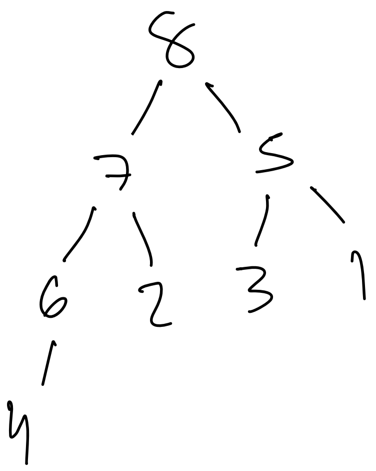
Swap 9,6
Pop 9



each step you swap the root and last element and then pop the last element into the end of the array



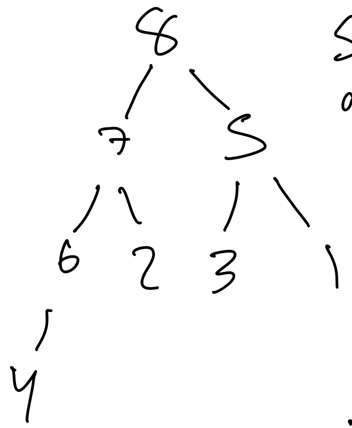
new heap



new array

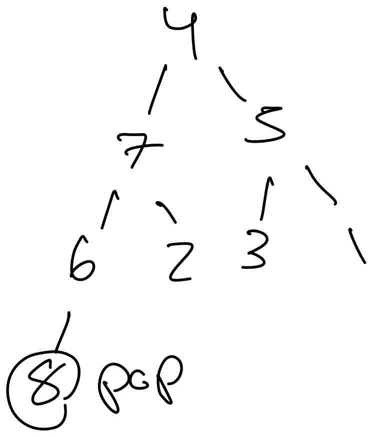
8	7	5	6	2	3	1	4	9

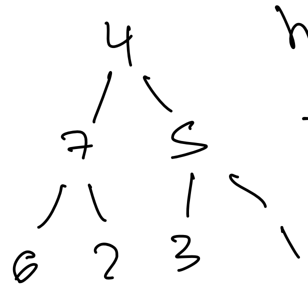
next step:



Swap 8 & 4
and pop 8

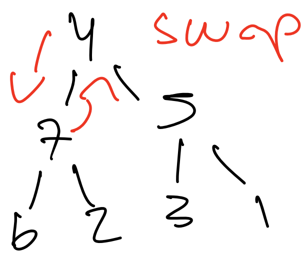
=>



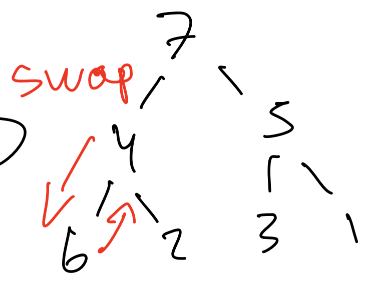


heapify

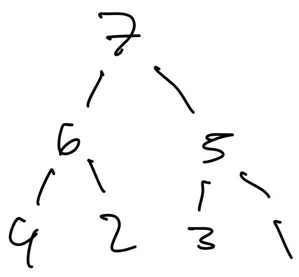
=>



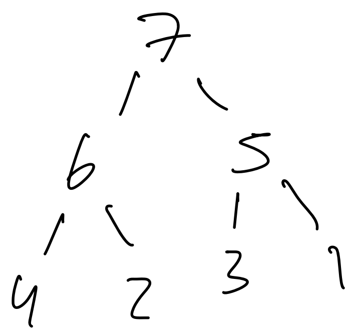
=>



=>



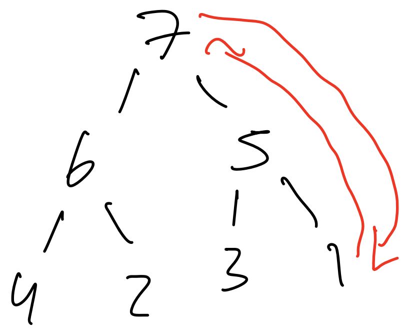
new heap



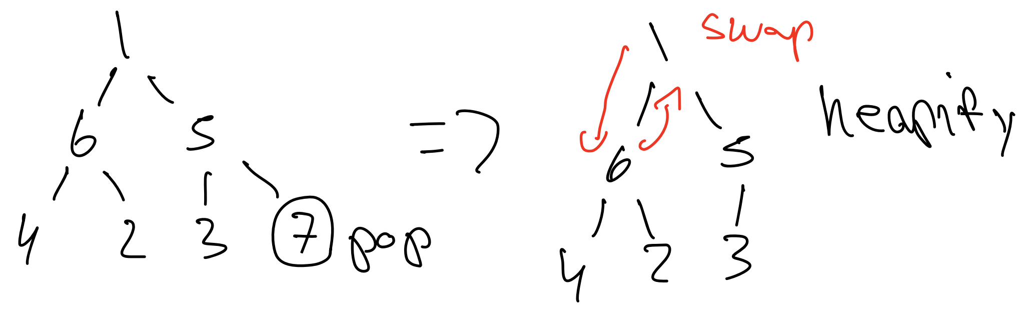
new Array

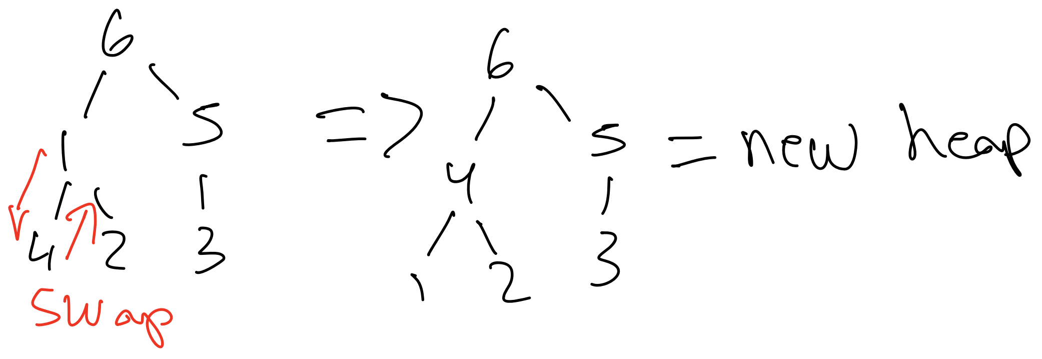
7	6	5	4	2	3	1	8	9

next step:



swap 7 & 1, pop 7

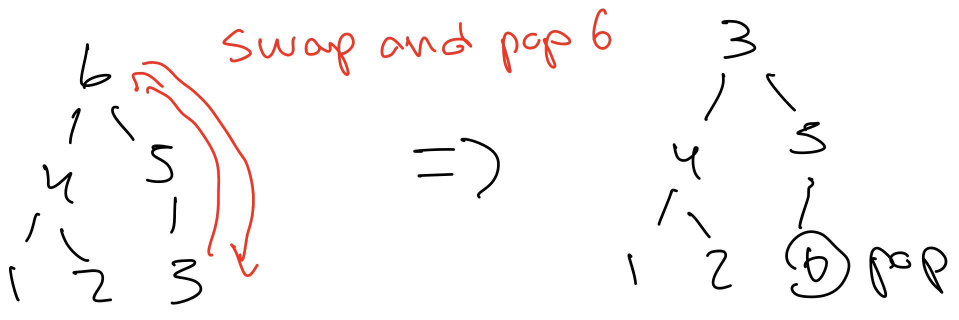


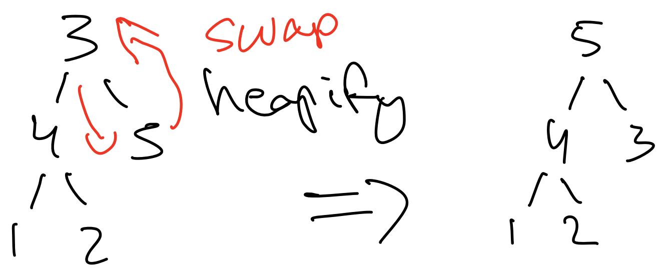


new Array

6	4	5	1	2	3	7	8	9

next step:

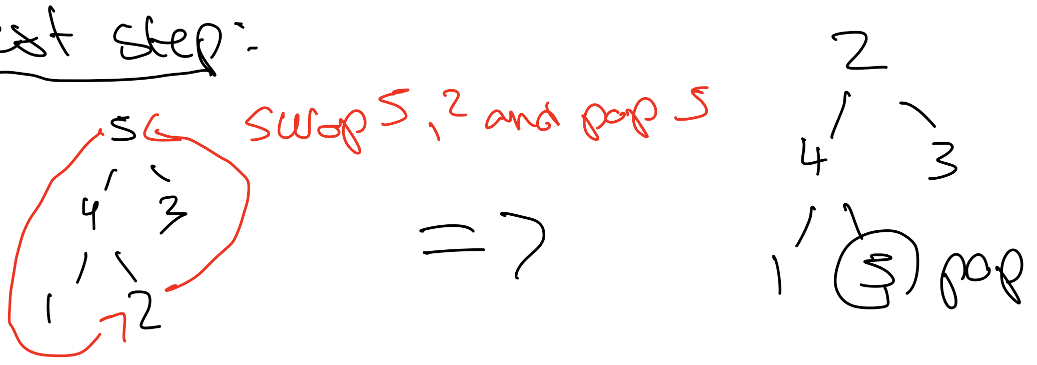


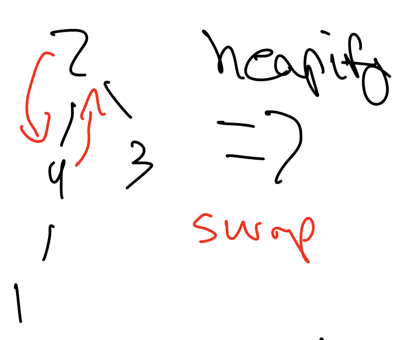


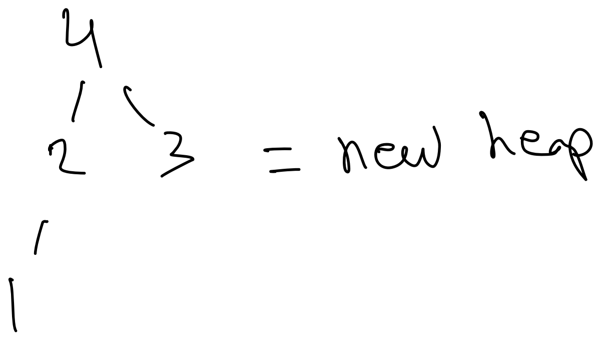
new Array

5	4	3	1	2	6	7	8	9

next step:



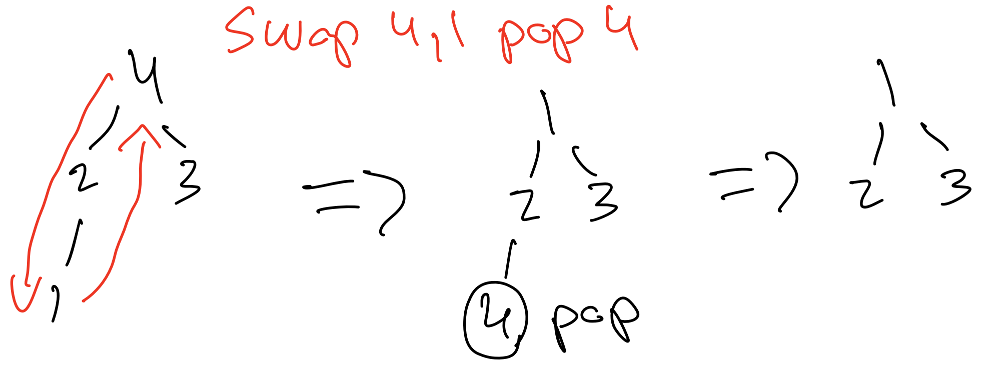




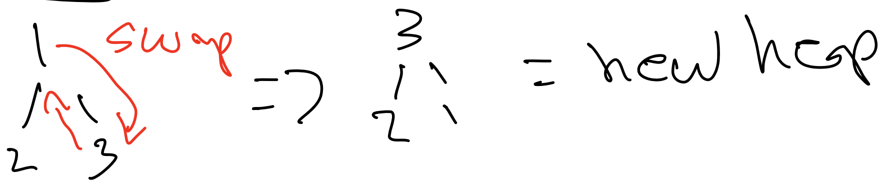
new Array

4	2	3	1	5	6	7	8	9

next step:



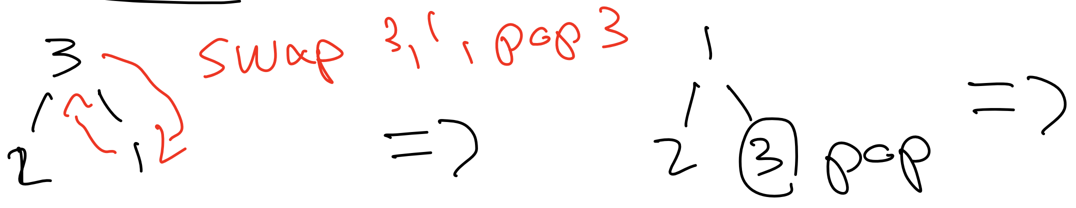
heapify

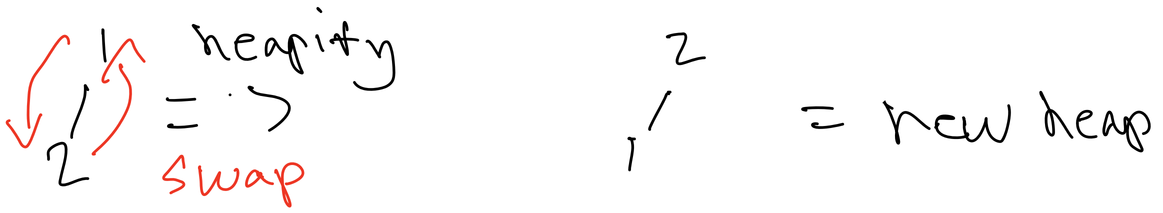


new Array

3	2	1	4	5	6	7	8	9

next step:



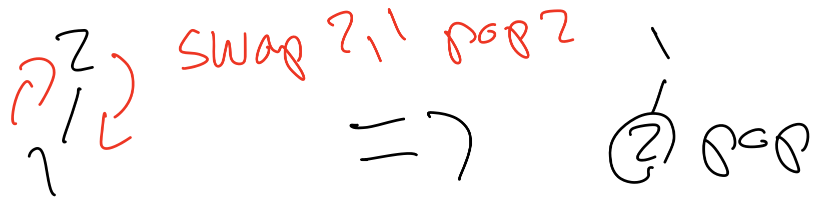


?

new Array

2	1	3	4	5	6	7	8	9

next step:



1 = new heap

new Array

1	2	3	4	5	6	7	8	9

last step:

1 pop 1

Final Array

1	2	3	4	5	6	7	8	9

#2 [10 points]

The analysis of recursive algorithms gives rise to recurrence relations. For Merge Sort, when n is a power of 2, the recurrence relation is:

T(n) = 1 if n = 1

T(n) = 2T(n/2) + n if n > 1

Using the recursion tree for merge sort, we were able to solve this recurrence relation into its closed form solution: n log n + n

Prove by induction that T(n) = n log n + n and hence is O(n log n)

Base Case: $T(1) = (1)\log(1) + (1) = 1$

$T(1) = 1$: True

Induction hypothesis: For any number $n > 1$
$n \log n + n$ is true:

Inductive step: $T(n) = (n)\log(n) + (n)$

$T(n) = n \log n + n$

$T(n) = 2T\left(\frac{n}{2}\right) + n$

$= 2\left(\left(\frac{n}{2}\right)\log\left(\frac{n}{2}\right) + \left(\frac{n}{2}\right)\right) + n$

$= n \log \frac{n}{2} + n + n$

$= n \log n - n + n + n$

$= \underline{n \log n + n}$

#3 [15 points]

Is there a heap T storing seven distinct elements such that a preorder traversal of T yields the elements of T in sorted order? How about an inorder traversal? How about a postorder traversal?

Let these distinct elements be: 1, 2, 3, 4, 5, 6, 7

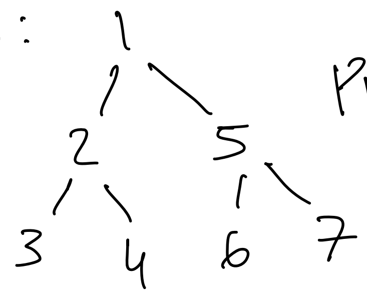
There is a min heap storing T elements that is sorted after a preorder traversal:
[1,2,5,3,4,6,7] = [1,2,3,4,5,6,7] after preorder traversal.

There is a min heap storing T elements that is sorted after postorder traversal:
[1,5,2,7,6,4,3] = [7,6,5,4,3,2,1] after postorder traversal.

Also a max heap is sorted after postorder traversal:
[7,3,6,1,2,4,5] = [1,2,3,4,5,6,7] after postorder traversal.

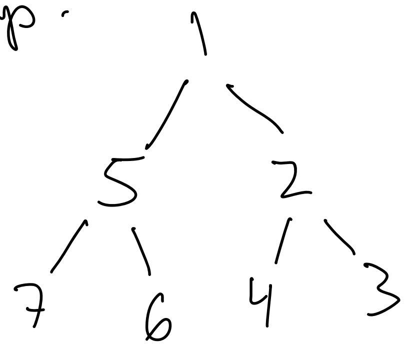
There is not a max or min heap that is sorted after inorder traversal.

min heap:



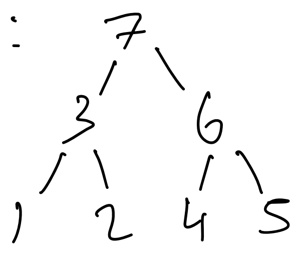
Preorder traversal =[1,2,3,4,5,6,7]

min heap:



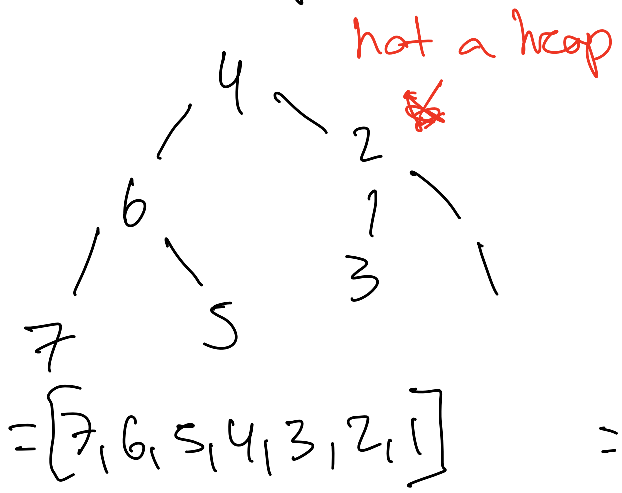
Postorder traversal =[7,6,5,4,3,2,1]

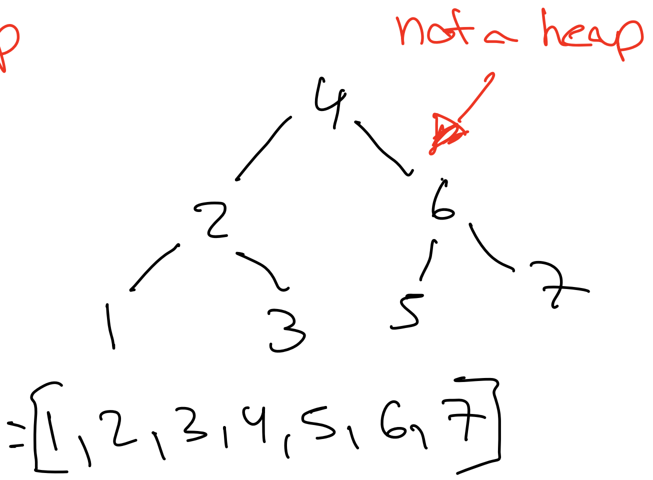
max heap:



Postorder traversal =[1,2,3,4,5,6]

Max heap





These are the only 2 lists that are sorted after inorder traversal, but neither list is a max or min heap.

(a) [15 points]

Briefly describe an algorithm for merging *k* sorted lists, each of length *n/k,* whose worst-case running time is O(nk). Clearly describe the algorithm in words, don't write the pseudo code. Explain mathematically that runtime is in fact O(nk).

Bird's eye view:

Merge k Sorted lists

An algorithm for merging k sorted lists of length n/k, is to merge the first and second list first and then merge that list (the merged list of list 1 and 2) with the third list and continue that step recursively until there is only 1 list remaining. This method will run in O(n) * O(k), it will make (n) operations for each (k) list.

Step-by-step:

An algorithm for merging k sorted lists, each with a length of n/k where n is the total number of elements and k is the total amount of lists would involve creating an empty "result" list, and using a function for merging 2 sorted (mergeTwoLists()) lists and appending that list into the empty "result" list and then merging the current sorted "result" list with the next list available, and appending each value from the next "k" list being merged into the "result" list and continue until there are no more k lists remaining and return the result list that is one sorted list of all k lists.

1. First is creating an empty list, result= [].

2. Create a for loop that would start at i = 0, and goes for as long as i < klists.size() and incrementing 1, i++ (klists is a list of k number of lists).

3. Use mergeTwoLists() function comparing empty list "result" and klist[i], and append sorted list into "result" and increment i.

4. The algorithm should keep calling mergeTwoLists(result, klist[i]) until i is not less than klists.size().

5. After the last klists list is merged into the "result" list, return the "result" list as the answer.

This algorithm is O(nk) because the mergeTwoLists(arg1, arg2) function operates in O(n) time with the number of iterations being dependent on the number of elements (N), and the for-loop that calls mergeTwoLists(arg1, arg2) until i >= klists.size() which is equal to k and the number of iterations the for-loop completes is based on k and therefore operates at O(k). With the mergeTwoLists() function operating at O(n) and the for-loop operating at O(k) the full algorithm is O(n) * O(k) = O(nk).

Merge 2 sorted lists algorithm that is used as a helper function:

An algorithm for merging 2 sorted lists, mergeTwoLists(), would compare the first index of list1 and list2 and if the value for list1[0] is smaller or equal to the value of list2[0], then that value is appended to empty list S, and then you increment the pointer of list1 to the next index, list1[1], and vice versa if the value of list2[0] is smaller than list1[0]. The algorithm can continue the process until the lists are empty, and all the values are appended to the empty S list, sorted. If one list is empty before the other, then can append the remaining values of the non-empty list into the S list. This algorithm runs in O(n) time.

Sum of arithmetic sequence formula:

$$S_n = \frac{n}{2}\left[a_1 + a_n\right]$$

$$\text{Time} = \frac{n}{k} + \frac{n}{k} + \frac{n}{k} + \dots + \frac{n}{k}$$

$$= 2\left(\frac{n}{k}\right) + 3\left(\frac{n}{k}\right) + 4\left(\frac{n}{k}\right) + \dots + k\left(\frac{n}{k}\right)$$

$$= \sum_{i=2}^{k} i\left(\frac{n}{k}\right)$$

$i = 2$ because don't Start merge until 2 lists

$$= \frac{n}{k}\sum_{i=2}^{k} i$$

# of elements

1st + last element

$$= \left(\frac{n}{k}\right) \cdot \frac{(k-i+1)(2+k)}{2}$$

$$= \left(\frac{n}{k}\right) \cdot \frac{(k-2+1)(2+k)}{2}$$

$$= \left(\frac{n}{k}\right) \cdot \frac{(k-1)(k+2)}{2}$$

$$= O(nk)$$

(b) [25 points]

Briefly describe an algorithm for merging *k* sorted lists, each of length *n/k*, whose worst-case running time is O(n log k). Clearly describe the algorithm in words, how you will implement it; don't write the pseudo code. Justify the running time. Explanation should be clear: doesn't have to be a mathematical derivation.

Birds eye view Explanation:

Merge k sorted lists

One way for an algorithm to merge k sorted lists of length n/k in O(n log k) time is to utilize a divide and conquer method. You can take the list that contains k lists of n/k length and divide klists into 2 halves. And then recursively split the lists in half until there are only 2 lists. The bottom layer of the tree of klists is only the leave lists. From there you would merge each pair of leaves (merge lists 1 and 2, 3 and 4, 5 and 6, etc...) recursively, until you only have one list remaining, you can do this using the function for merging two lists, mergeTwoLists(), as a helper function. The mergeTwoLists() function will merge each pair of leaf klists and because it is operating on each pair recursively, it is cutting the amount of elements to be compared in half. At this point the algorithm will recursively merge the lists until there is only one sorted list remaining. This algorithm runs in O(n log k) time because the process of the mergeTwoLists() function operates in O(n) time, and when you divide and conquer the list of klists by separating them, you cut the amount of comparisons of the elements of klists in half which operates in O (log k). Therefore, for merging each list in O(n) time and separating the lists so that the elements compared happens log k times equals an O(n log k) time complexity for the algorithm.

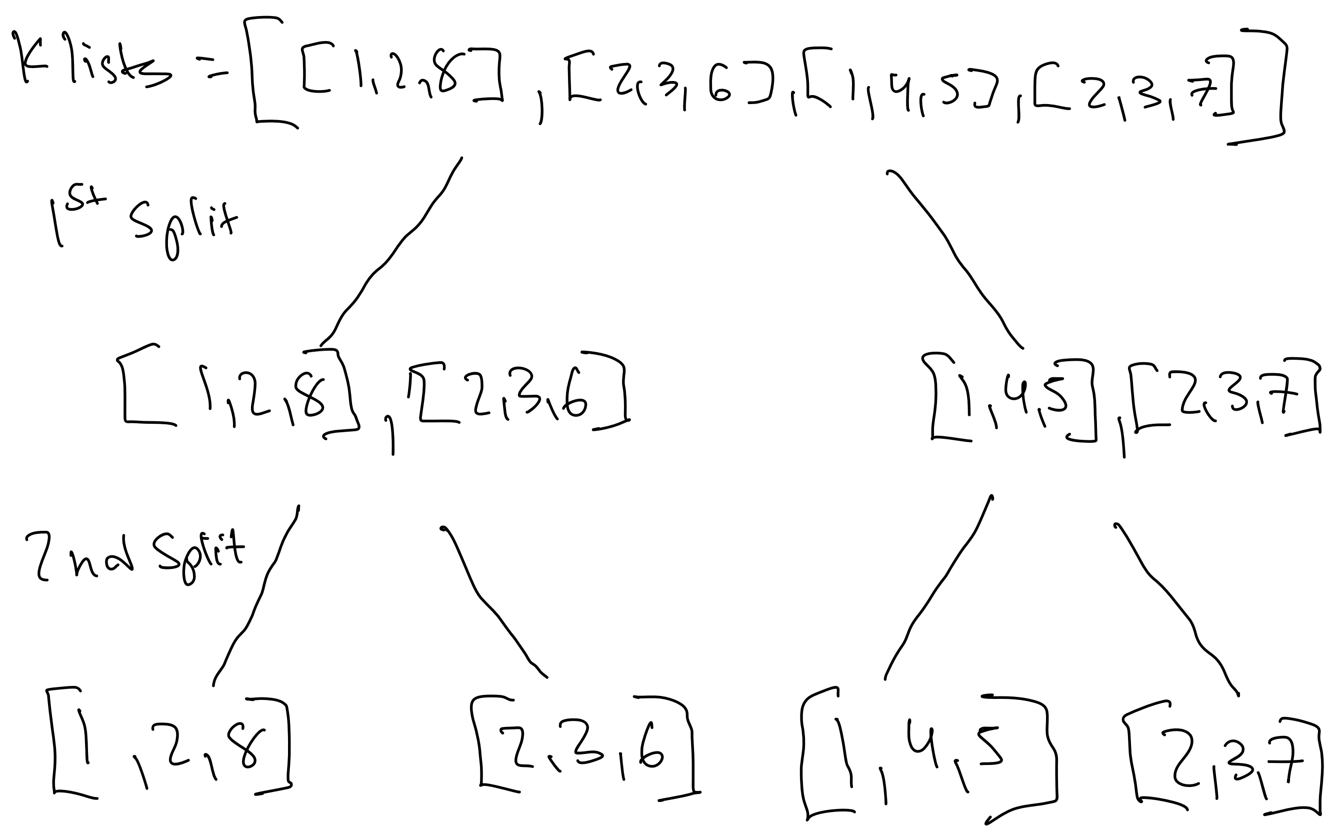
Breaking down the Algorithm:

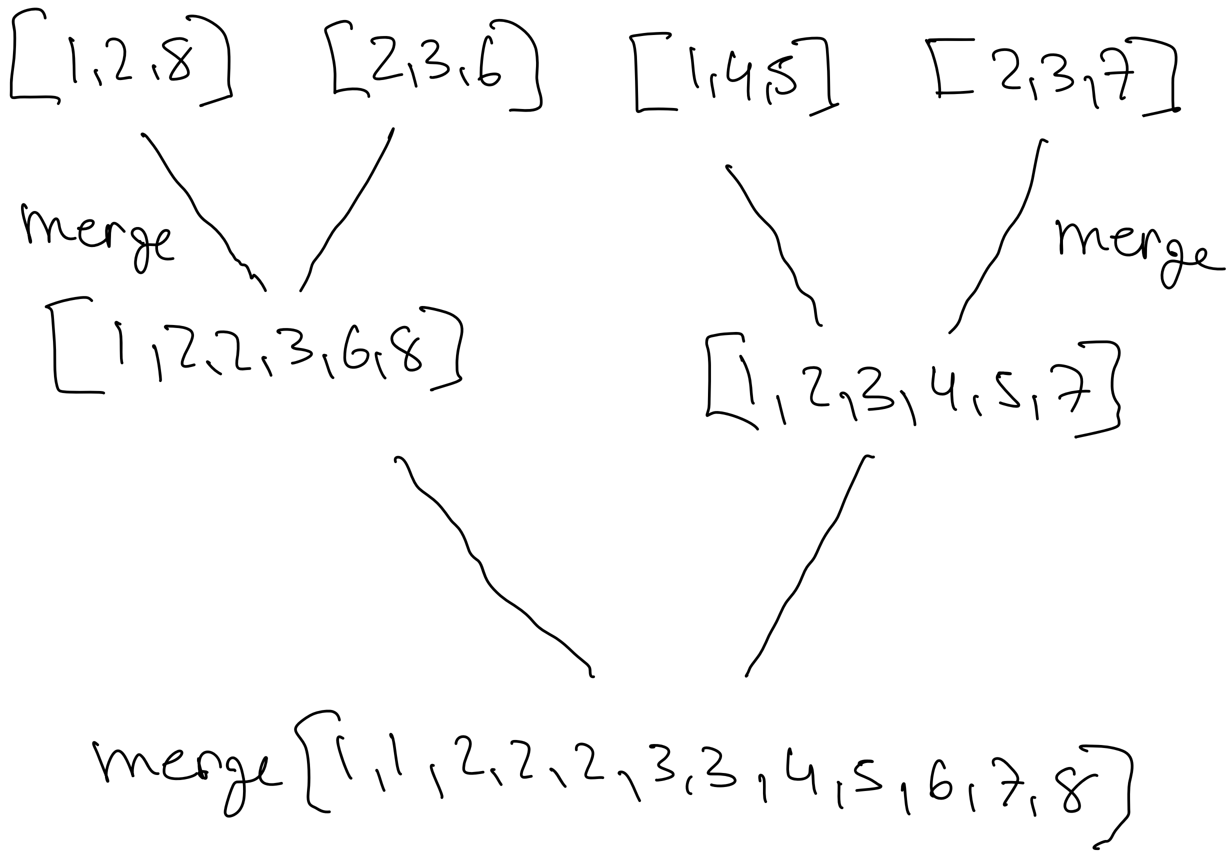
For an algorithm that is merge k sorted lists with length of n/k you would created an empty list "result" and have a while loop that would continues while the size of klists is greater than 1, and a for-loop that continues as for as long as the initial index i = 0 is less than klists.size() and after each iteration incrementing the index by 2. Inside the for-loop you would have 2 variables that represent the lists to be merged list1 and list2. list1 is set to klists[i], and list2 is set to klists[i +1], if the list2 [i+1] is greater than the size of klists then set list2 to a NULL list. After the values for list1 and list2 are set, you can merge the 2 lists with the mergeTwoLists() helper function and add the value into an empty list "result". Outside of the for-loop set klists to result. The number of klists will decrement, and then reenter the for-loop until klists = 1. At that point the algorithm will exit the while-loop and that point you can return the value of lists at index 0 (lists[0]).

Step 1:
    Test for edge cases if k = 0 then return NULL or if k = 1 return the klists[0]
Step 2:
    Create a while-loop that continues while klists size > 1
Step 3:
    Create an empty list (newList)
Step 4:
    Create a for-loop the starts at index i = 0, while i < klists.size(), and i increments by 2
Step 5:

Within the for-loop set the first list (variable can be named list1), to merge, to lists[i] and the second list (list2), to merge, to lists[i + 1], if the second list is out-of-bounds then set the list to NULL, If the first list is in-bounds and the second list is out-of-bounds, then the first list will just merge merge with a NULL list which gives you just list1.
Call the mergeTwoList() function merging list1 and list2 (mergeSort(list1, list2)) and add the list into the empty list (newList).
Once the for-loop ends, set klists to newList, if klists size is still larger than 1 after the being set to newList, then the the function will enter the for-loop again and will repeat the process until newList = 1 and klists = newList, then return the first list in klists.

Example:





For the O(nk) approach to merging k sorted lists you would compare the elements of the first list [1,2,8] 3 times (compare with the 2nd list, then 3rd list, then 4th list). In the O(nlogk) approach you compare the elements in the klist [1,2,8] only 2 times (compare with list 2 and the compare with the merge list of 3 and 4) which is how you get the O(n) * O(logk) instead of O(n) * O(k), giving you O(nlogk) complexity.