Introduction to Algorithms (CS430):

Homework - 3

(1) Given a list of n numbers,

let, $\boxed{k = n/4}$

plm: Determine the k^{th} element in a list of n numbers, using randomized partition method.

In Quick sort, we pick a pivot element and partition the list around it; by moving the pivot element to its correct position. This implies that all elements which are on left side of pivot are less than pivot and which are on right side of pivot are greater than the pivot. Quick sort generally chooses first or last element as pivot.

Randomized partition → In Randomized partition method, we randomly pick a pivot element; using a random number generator function. Swap the element at randomly generated index with the last element and call the partitioning process. The Difference is minimal, in this procedure, we implement swap before actually partitioning:

ALGORITHM:

STEP 1 : Use randomized partition method to select a pivot and put that in it's correct position.

STEP 2 : if (pivot == K^{th} position)
return pivot position.

STEP 3 : if (pivot < K^{th} position)
repeat step 1 and 2 on right sub-array.

STEP 4 : if (pivot > K^{th} position)
repeat step 1 and 2 on left subarray.

ANALYSIS:

In worst case, randomized partition method would return a pivot that is either rightmost or leftmost element of the list.

So probability of worst case = $2/n$

" average case = $(n-2)/n$

[n → no. of elements in list]

So Time complexity would be $O(n^2)$ for $2/n$ and $O(n \log n)$ for $(n-2)/n$ times.

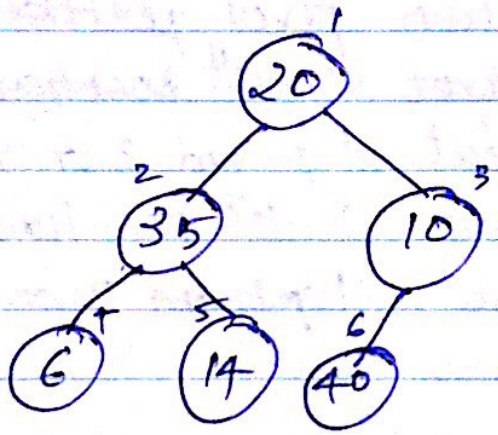
② HEAP SORT:

Eg: Consider a list:

1	2	3	4	5	6
20	35	10	6	14	40

A ⇒

Constructing a nearly complete binary tree,



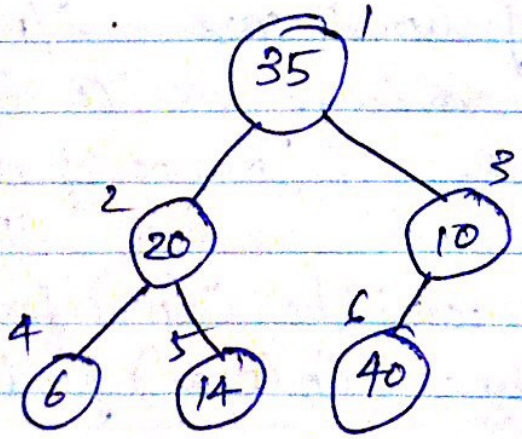
Build a max-heap using max-heapify property:

heap_size(A) = 6

i) A[1] < A[2]

Max-heapify (A, 1):

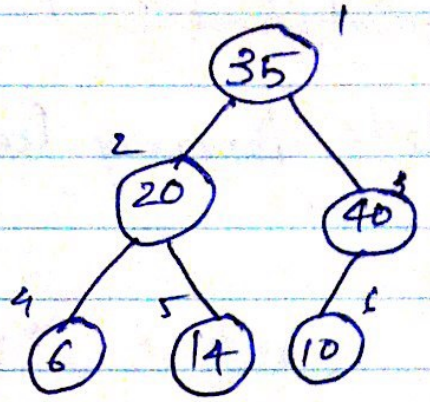
Exchange A[1] with A[2]



ii) A[3] < A[6]

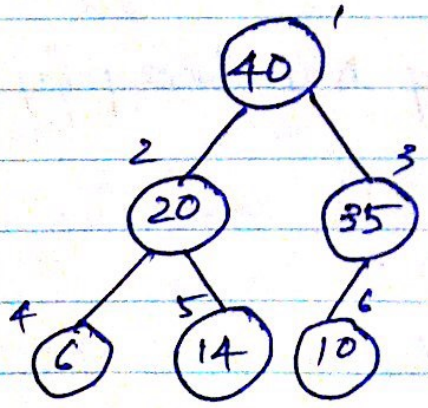
Max_heapify (A, 3)

Exchange A[3] with A[6]



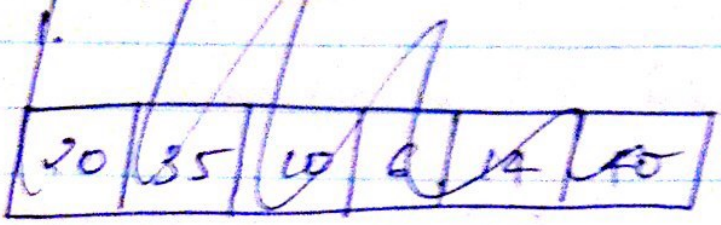
A[1] < A[3]

Exchange A[1] with A[3]



find max element and swap A[n] with i-1. Discard node n from data structure and repeat max heapify since root may violate max heap property.

Now situation :

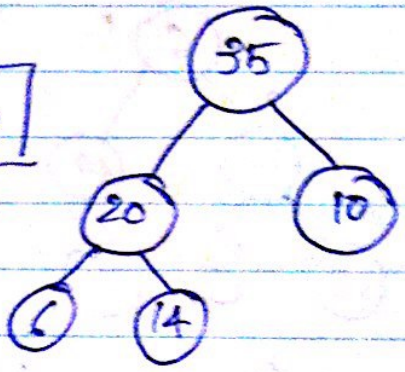


Now situation is

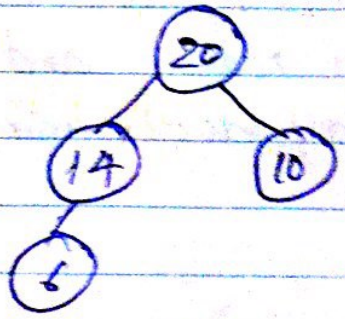
40	20	35	6	14	10

Swap Max element with A[n], and maxheapify

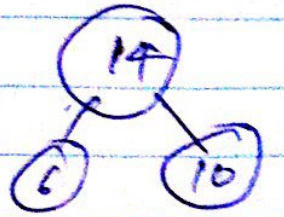
35	20	10	6	14	40

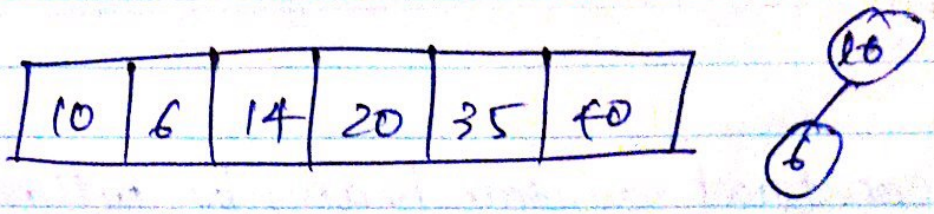


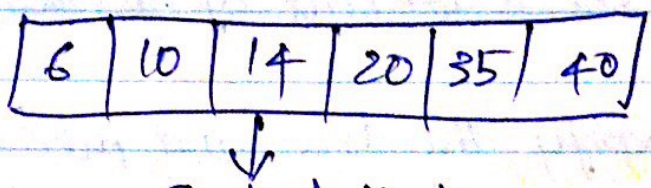
20	14	10	6	35	40



14	6	10	20	35	40







Sorted list

Since it is a nearly complete binary tree, height of the tree is bounded by $(\log n)$.

∴ Time complexity of max-heapify → $O(\log n)$

Time complexity for building max heap tree → $O(n)$

Time complexity for finding max-element & swapping with $A[n]$ → $O(1)$

By above analysis, time complexity for heap sort → $O(n \log n)$

(3) ALGORITHM:

STEP 1: Construct a min heap, by pulling out of first elements of k sorted arrays.

STEP 2: Min Heapify the tree and pull out the minimum element and add it to the resultant array of size n.

STEP 3: pull out the second element from array and add to the heap. Repeat STEP 2 until all element of k arrays are consumed and kept in the proper position in the resultant array.

No. of elements in heap = k
Time to heapify = $O(\log k)$

No. of times this heapify process repeats = Total no. of elements in resultant array = n

∴ Time complexity of the algorithm = $O(n \log k)$

④ Radix sort sorts the elements digit by digit. The least significant digit is sorted first followed by next higher digit until the most significant digit. At each iteration, the numbers whose digits are all consumed appear in the sorted order.

For Eg consider this list of numbers.

12, 301, 415, 26, 710

1st Iteration.

710	301	12			415	16
0	1	2	3	4	5	6

↳ 710, 301, 12, 415, 26

2nd Iteration.

415
12

301	710	26
0	1	2

↳ ~~710~~, 301, 710, 12, 415, 26

Here all digits in 12 and 26 are consumed and they appear in sorted order in the bucket. Hence we can output 12 and 26.

Algorithm:

STEP 1: Sort by least significant bit (1st place)

STEP 2: The numbers whose digits are all consumed appear in sorted order, so output them to the resultant array.

STEP 3: Sort by other digits until the most significant bit.
And repeat STEP 2.

STEP 4: output the remaining element to the resultant array once all digits of all numbers are consumed.

Analysis:

The time complexity depends on the number of digits (K) and on length of the array (n)
At each step we have to make an array of start indices of the new buckets and move cell data into right bucket. Also output the number if all digits are consumed, which takes O(n+k) steps.

(5) Each subsequence has $(\log n)$ elements. Therefore, number of unique ways to arrange the numbers in any subsequence is $(\log n)!$

We have $\frac{n}{\log n}$ subsequences, each of which can be arranged in $(\log n)!$ ways.
The total no. of ways in which such n ordering is possible is $\Rightarrow [(\log n)!]^{n/\log n}$

Let $S(n)$ denote minimum no. of comparisons.

A binary tree of height h can have at most 2^h leaves. So,

$$2^{S(n)} \geq [(\log n)!]^{n/\log n}$$

$$S(n) \geq \log \left[[(\log n)!]^{n/\log n} \right]$$

$$\geq \frac{n}{\log n} \log (\log n)!$$

$$\geq \frac{n}{\log n} \log n \log (\log n)$$

$$\boxed{S(n) \geq \Omega(n \log (\log n))}$$

$$\boxed{S(n) \neq \Omega(n \log n)}$$