Divide and Conquer Approach

Divide and conquer approach breaks the problem into several sub-problems, that are similar to original problems, but smaller in size. Then it solves the sub-problems recursively and then combines the solutions to obtain the solution of the original problem.

Steps:

Divide: Divide the problem into 'a' number of sub-problems of size (n/b).

Conquer: Solves the sub-problem recursively. If the size of a sub-problem is sufficiently small, solve it directly.

Combine: Combine the solutions of the sub problem to obtain solution of original problem.

Running time of algorithm, that is based on divide and conquer approach is expressed as $T(n) = aT(n/b)+ f(n)$,

where, a is the number of sub=problem

(n/b) is the size of each sub problem

f (n) = divide time + combine time

Problems:

Binary Search, Merge sort, Quick Sort, Heap sort, Closest pair problem, convex hull problem, Multiplication of large integer, Stassen's matrix multiplication

Binary Search:

Binary search is an efficient search technique that is applied on a sorted array to search an item.

Given a sorted array and a search key, binary search involved in following steps:

- Compare the search key with the middle element of the array.
- If key is found in middle, searching process stops.
- If key < middle element, then the search process is continued recursively in the left half of the array.
- If key > middle element, then the search process is continued recursively in the right half of the array.

1	2	3	4	5	6	7	8
10	20	30	40	50	60	70	80

Key = 50

lb = 1, ub =8, mid=4

Step -1: key > A [4], search in right half

1	2	3	4	5	6	7	8
10	20	30	40	50	60	70	80

Key = 50

lb = 5, ub =8, mid = 6

Step -2: key < A [6], search in left half

1	2	3	4	5	6	7	8
10	20	30	40	50	60	70	80

Key = 50

lb = 5, ub =5 mid = 5

Step -3: key = A [5], the key is present in mid position.

```
BinarySearch(A,p,r,key)
//Search the key in a sorted array A[p…..r]. Return
//index if the key is present and 0 if the key is
//not present in array.
1. if p>r
2.   return 0
3. mid = ⌊(p+r)/2⌋
4. if key = A[mid]
5.       return mid
6. if key < A[mid]
7.       return BinarySearch (A,p,mid-1,key)
8. else,return BinarySearch(A,mid+1,r,key)
```

Analysis:

Worst-case efficiency occurs, If the search key is present in last position or not present at all, it is worst case of the input instance.

The algorithm take *O(1)* to find the mid position and to compare the key . Algorithm makes ne recursive call out of two. At each recursive call, the size of the array is reduced to half of its size. The running time is described as

$$T(n) = T(n/2) + O(1) = O(\log n)$$

Best-case efficiency occurs if the key is present at middle position. Only one comparison is needed to find the key. So best case efficiency is O(1)

Merge Sort:

Merge sort follows divide and conquer approach. It works as follows.

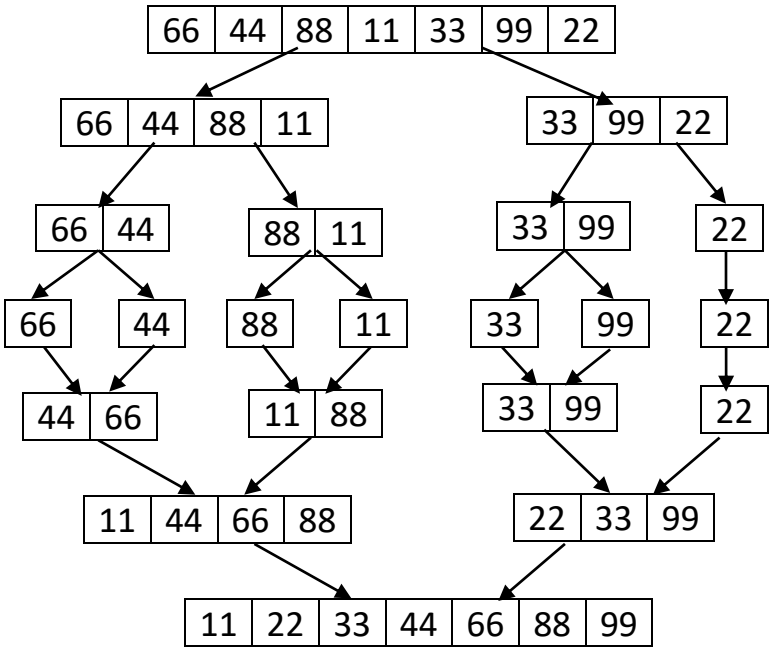
Given an n-element array A[p..r]

Divide : Divide the array into two sub array A [p….q] and A[q+1…r].Size of each problem is n/2.

Conquer : Sort the sub-arrays recursively

Combine : Merge two sorted sub-arrays to produce entire sorted array.

Example : Apply merge sort on [66,44,33,11,88,99]



Merging two sorted sub-array into one sorted array.

```
Algorithm : Merge(A,p,q,r)
//It mergers two sorted sub-array A [p…q] and A [q+1….r] into array A [p…r]
1.  n1 = q - p +1
2.  n2 = r – q
3.  Create two arrays Left[1…n1+1] and Right[1……n2+1]
4.  for i = 1 to n1
5.       Left[i] = A[p + i -1]
6.   for j = 1 to n2
7.       Right[i] = A[q+j]
8.  Left[n1+1] =Right[n2+1] = ∞
9.  i = j =1
10. for k =p to r
11.      if Left[i] ≤ Right[j]
12.           A[k] = Left[i]
12.           i = i + 1
13.      else,    A[k] = Right[j]
14.           j = j + 1
```

Analysis of Merge () procedure

Loop in line number 4 runs O(n1) time . As n1≈ (n/2), it is O(n/2). So, as loop in line number 6.

Loop in line number 10 runs for k = p to r. Let the size of array in n, the loop checks n items of the array from index p to r. So it run O(n)

Total running time = O(n)/2 + O(n/2) +O(n) = O(n)

```
MergeSort(A,p,r)
// This algorithm sorts the array A[p…r].
1. if p < r
2.        q = ⌊(p + r)/2⌋
3.        MergeSort(A,p,q)
4.        MergeSort(A,q+1,r)
5.        Merge(A,p,q,r)
```

Analysis of MergeSort() algorithm

If p ≥ r , the array contains 0 or 1 element, which is sorted already. So T(n) =O(1)

If p< r, It takes divide time of O(1) , combine time of (n) by calling Merge() procedure. Two recursive call to MergeSort() with input size of n/2 can be represented as T(n/2). Total running time of MergeSort algorithm is—

$$T(n) = 2T(n/2) + O(n) = O(n \log n)$$

Quick Sort

This sorting is based on divide and conquer approach. It sorts all the elements in place. It works as follows.

Given an array A[p….r],

Divide: Partition the array A[p…r] into two sub arrays A[p…q-1] and A[q+1…r]. such that element in A[p…q-1] and less than or equal to A[q] and element in A[q+1…r] are greater than or equal to A[q]. Thus, A[q] get its place.

Conquer: Sorts two sub arrays A[p…q-1] and A[q+1…r] recursively.

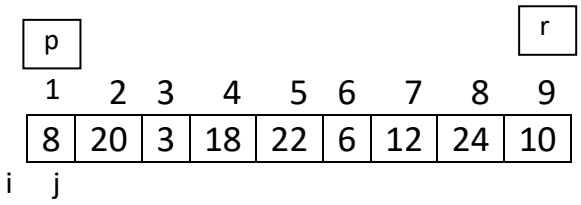
Combine: No work needed to combine as the sub arrays are sorted in place.

Algorithm for partitioning array

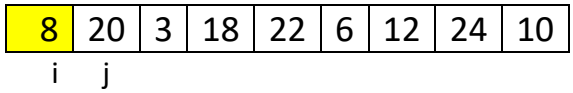
```
Partition (A,p,r)
// This algorithm partition the array A and return position of pivot element A[q] such that A[p…q-1] ≤A[q] ≤A[q+1…r]

1.  x = A[r]
2.  i = p-1
3.  for j = p to r-1
4.  if A[j] ≤ x
5.       i =i+1
6.       exchange (A[i],A[j])
7.  exchange (A[i+1],A[r])
8.  return i+1
```

Example: Apply Quick sort on [8, 20, 3, 18, 22, 6, 12, 24, 10]

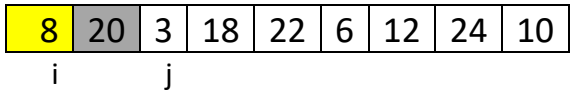
p=1,r=9, x=10, i=0 , j=1



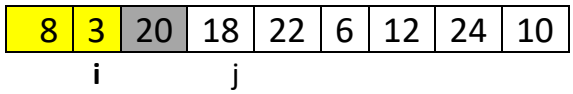
A[1] < x, So i=1, Swap (8,8), j=j+1 = 2



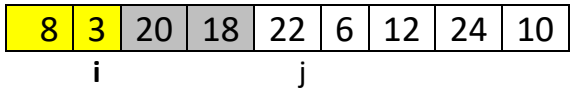
A[2] > x, So j=j+1 = 3



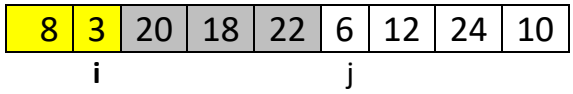
A[3] > x, So i=2, Swap (20,3), j=j+1=4



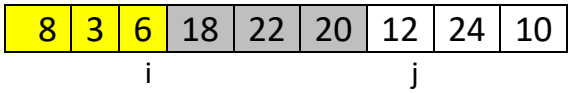
A[4] > x, j=j+1=5



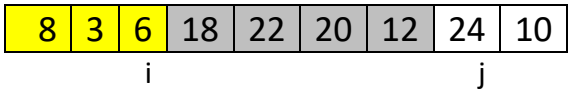
A[5] > x, j=j+1=6



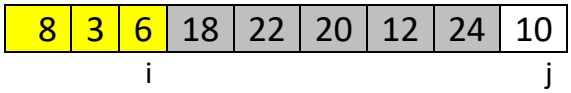
A[6] < x, i=3, Swap (20,6), j=j+1=7



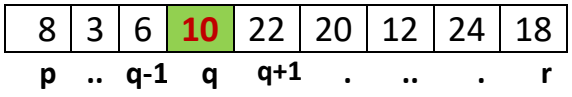
A[7] > x, j=j+1=8



A[8] > x, j=j+1=9



Loop terminates, A[i+1]↔A[r]



Quick sort Algorithm:

```
QuickSort(A,p,r)
// Sorts all the elements of the array A[p…r]
1. if p<r
2.   q = Partition(A,p,r)
3.   QuickSort(A,p,q-1)
4.   QuickSort(A,q+1,r)
```

Running time of partition algorithm

Loop in line number 3 checks each element of array A[p..r] exactly once and move j one-step forward. So the loop runs O(n) time, where n is the number of element in the array.

Performance of Quick Sort algorithm

Performance of Quick sort depends on whether the partition is balanced or not.

Partitioning is balanced, if the partition algorithm produces one sub-array with size n/2 and other sub-array of size nearly equal to n/2.

Partitioning is unbalanced, if the partition algorithm produces one sub-array with size (n-1) and another with size 0.

Worst case Partitioning

It occurs if each recursive call to the QuickSort algorithm, there is unbalanced partitioning. Size of one sub-array is (n-1) and other is 0. Therefore, the running time is –

$$T(n) = T(n-1) + T(0) + O(n)$$

$$= T(n-1) + O(n) = O(n^2)$$

Best case Partitioning

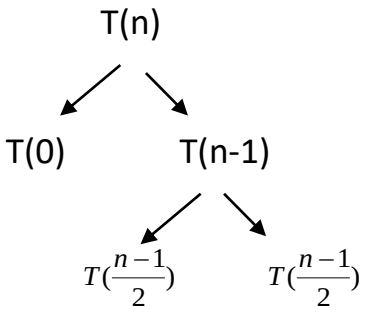
It occurs if each recursive call to the QuickSort algorithm there is balanced partitioning. Size of each sub-array is (n/2). Therefore, the running time is –

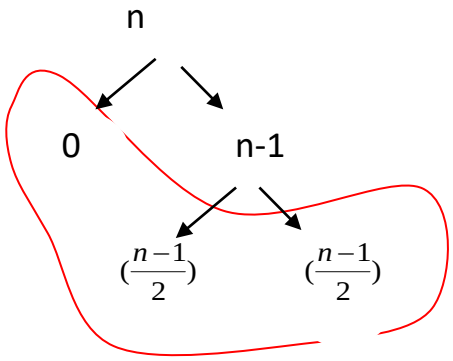
$$T(n) = T(n/2) + T(n/2) + O(n)$$

$$= 2T(n/2) + O(n) = O(\log n) = O(n \log n)$$

Average case Partitioning

Average case running time of quick sort is much closer to the best case. This case occurs, when balanced and unbalanced partition is distributed randomly throughout each recursive steps. Assuming balance and unbalanced partition occurs in alternative steps.





$$T(n) = T(0) + T(\frac{n-1}{2}) + T(\frac{n-1}{2}) + O(n)$$

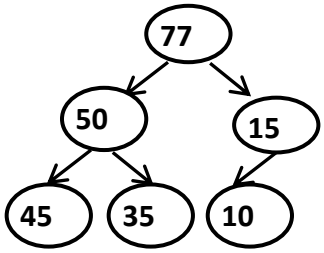
$$= 2T(\frac{n-1}{2}) + O(n)$$

$$\leq 2T(\frac{n}{2}) + O(n) = O(n \log n)$$

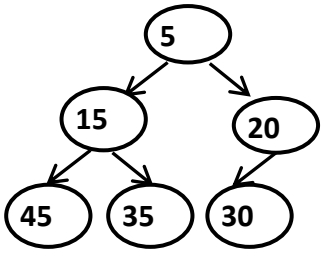
Heap Sort

Heap or a binary heap data structure is an array of objects, which can be viewed as a complete binary tree. Each node in the tree represents an element of the array.

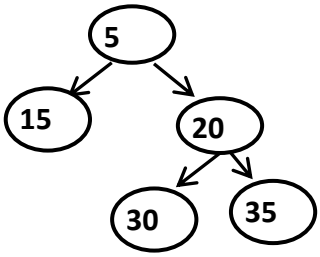
A complete binary tree is said to be a heap or max-heap if the key value of any node is greater than or equal to the value of its children



[Max Heap]



[Min Heap]

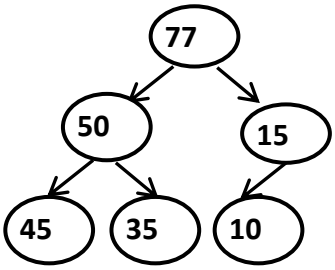


[Not a heap]

Representation of Heap

Each element of the array represents a node in a complete binary tree level wise. Root node contains 1st array element , two children of root contains next two elements of array and so on.

Tree representation of array A [77,50,15,45,35,10] is



For a heap array A,

length(A) : is the number of element present in Array A

Heapsize(A) : is the number of element of array constitute a heap

for any nod at index 'k'

index of parent (k) = Floor (k/2)

index of Left Child (k) = 2×k

index of Right Child (k) = 2×k +1

Sorting an array using heap requires two steps

- *Building heap / maintaining heap property*
- *Sorting using heap*

Maintaining Heap property

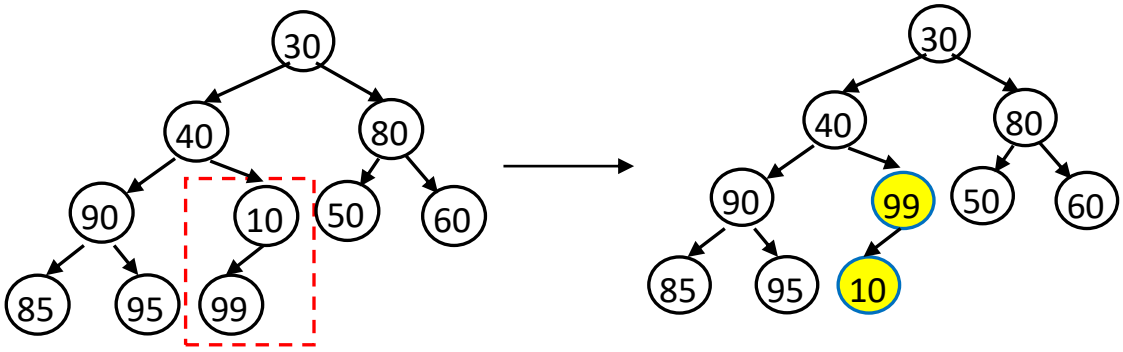
When an n- element array is represented in a complete binary tree, the elements in A[n/2+1 ….n] are in leaf nodes. The elements in A[1 . ..n/2] are in non-leaf nodes. All the sub-trees rooted at leaf nodes already maintain the heap property.

So we have to start heapifying from the node indexed at $\left\lfloor \frac{n}{2} \right\rfloor$ down to index 1. It is to be noted that heapifying a node requires its left sub-tree and right sub-tree to be heapified before.

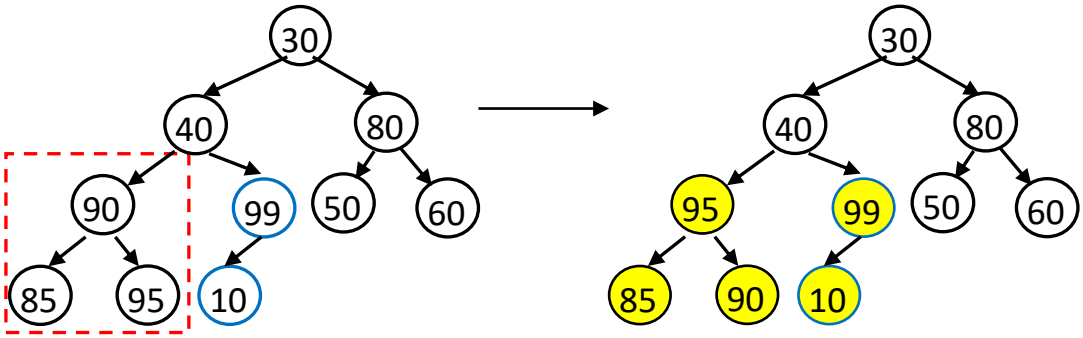
To heapify a sub-tree rooted at index i, find the index of largest value among A[i], A[left[i]] and A[right [i]]. Then exchange A[i] and A[largest]. Then heapify the sub-tree rooted at A[largest] if it is not a leaf node.

Example: Heapify the array A=[30,40,80,90,10,50,60,85,95,99] of length n=10

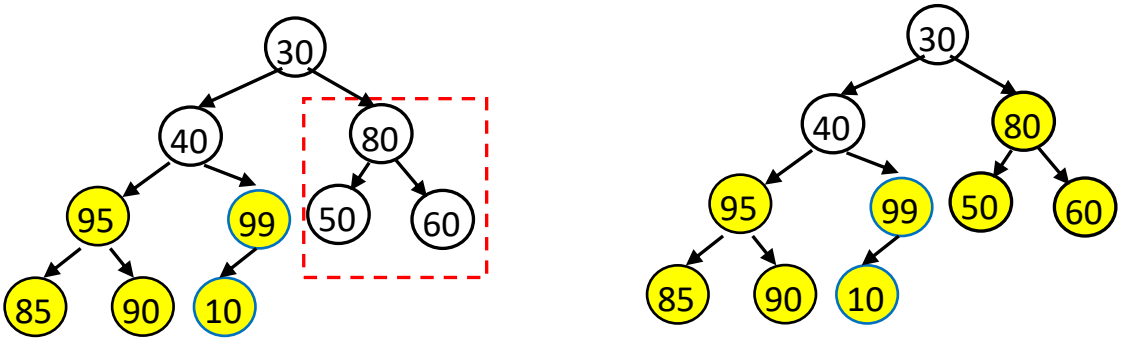
Step 1: Heapify the Sub-tree rooted at node 10



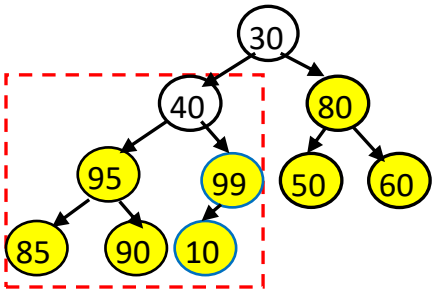
Step 2: Heapify the Sub-tree rooted at node 90

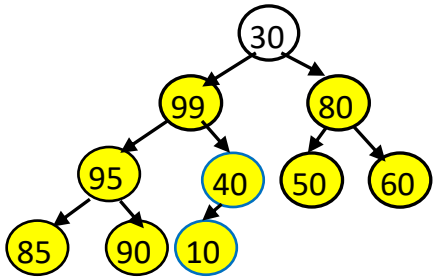


Step 3: Heapify the Sub-tree rooted at node 80

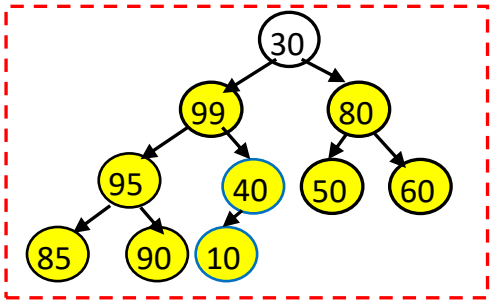


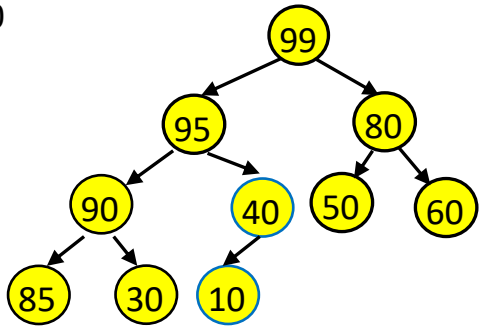
Step 4: Heapify the Sub-tree rooted at node 40





Step 5: Heapify the Sub-tree rooted at node 30





Now the heap array is [99 , 95, 80, 90, 40, 50, 60, 85, 30, 10]

```
Algorithm : Heapify(A,i)
// This algorithm heapify only one sub-tree rooted at
element at index i
1. left = 2* i
2. right =2* i + 1
3. if left ≤ heapsize and A[left] > A[i]
4.       largest = left
5. else, largest = i
6. if right ≤ heapsize and A[right] > A[i]
7.       largest = right
8. if largest ≠ i
9.       Exchange(A[i],A[largest])
10. Heapify(A, largest)
```

Running time of Heapify algorithm:

This algorithm requires O(1) time to find the largest value among A[i],A[Left[i]] ad A[right[i]].

Time to run heapify() on a sub-tree rooted at one of children of node (x) is calculated as below.

size of the sub-tree of a tree having node is at most (2n/3) in worst case. Worst case occurs if the last row of the tree is exactly half-full. So, the running time is –

$$T(n) = T(\frac{2n}{3}) + O(1) = O(\log n).$$

```
Algorithm: BuildHeap(A)
// This algorithm builds a heap from the original array
by repeatedly calling HEAPIFY() to heapify the sub-
trees rooted at all non leaf nodes.
1. heapsize = length(A)
2. for i = ⌊length(A)/2⌋ downto 1
3.       Heapify(A,i)
```

Running time of Build Heap algorithm:

Time required to call heapify algorithm is O(logn). As there are n/2 calls to heapify algorithm, total tie taken by Build Heap algorithm is n/2*logn = O(nlogn).

However, this is a loose bound. More tighter analysis to BuildHeap() algorithm can reduce the running time to **O(n).**

Heap Sort

Given an array of n items, first build a heap by calling BUILDHEAP(). As the new array is a max heap, the largest element is stored at A[1] (i,e at root). It can be placed into its proper position by exchanging it with A[heapsize]. Then discard it from the heap by decreasing the heapsize. Then call to HEAPIFY(A,1) to maintain max heap property at root. Above process is repeated until heapsize = 1.

```
Algorithm: HEAPSORT(A)
// This algorithm sorts an array A using heap.
1. BuildHeap(A)                          .. .. .. O(n)
2. for i = length(A) to  down to 2       .. .. ..
3.       Exchange(A[1],A[i])
4.       heapsize = heapsize – 1
5.       Heapify(A,1)
```

Running time of Build Heap algorithm:

Line number 1 takes O(n) to build a heap. Line number 5 takes O(logn) to call heapify algorithm. As there are n-1 call to this algorithm, altogether it takes O(nlogn) time. So the running time of Heapsort algorithm is-

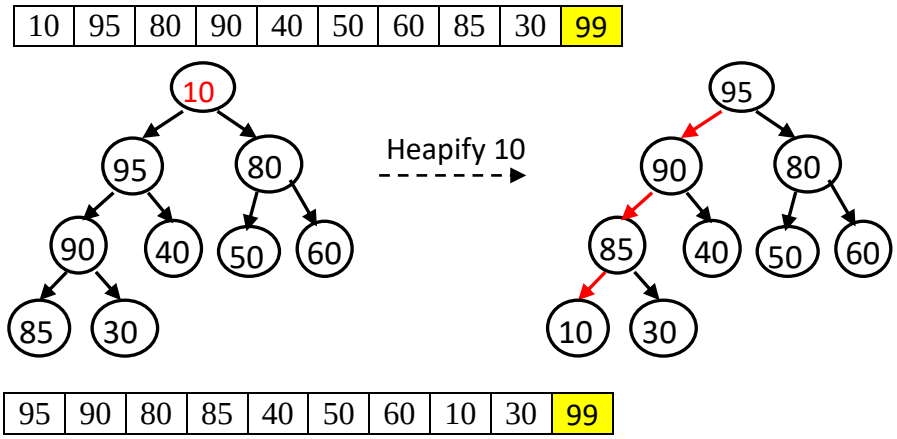
$$T(n) = O(n) + O(n\log n) = O(n\log n)$$

Example: Apply Heap sort on the array A=[30,40,80,90,10,50,60,85,95,99] .

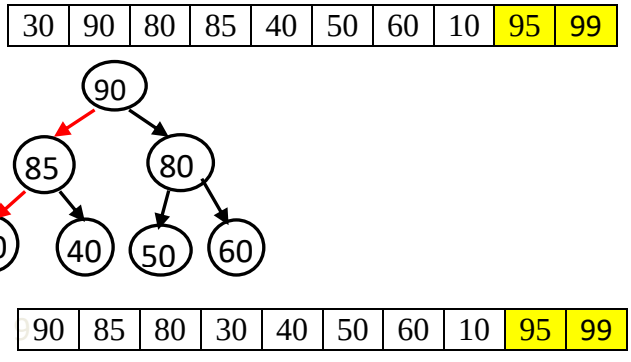
Solution: Line number 1 builds the heap array as below (see previous example)

Heap Array - [99 , 95, 80, 90, 40, 50, 60, 85, 30, 10] ,heapsize=length=10

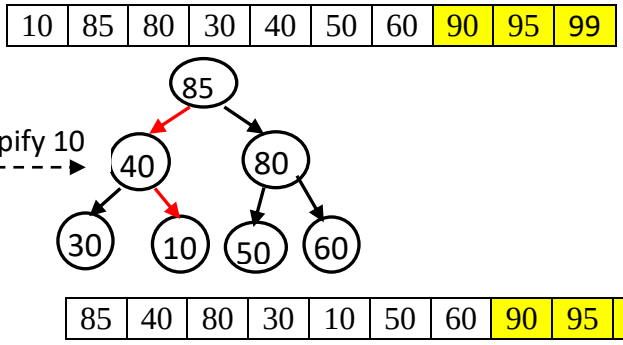
Step 1: i= 10, exchange A[1] and A[10], Decrease heapsize, HEAPIFY(A,1)



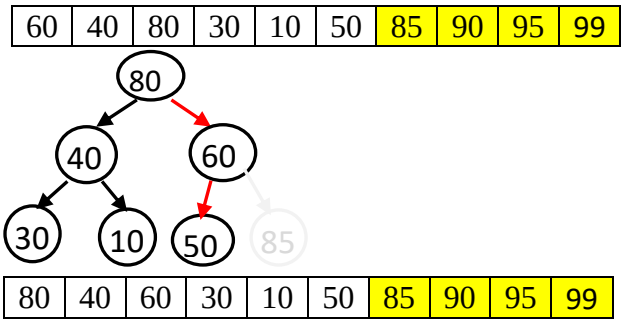
Step 2: i= 9, Repeat above process.



Step 3: i= 8, Repeat three steps.



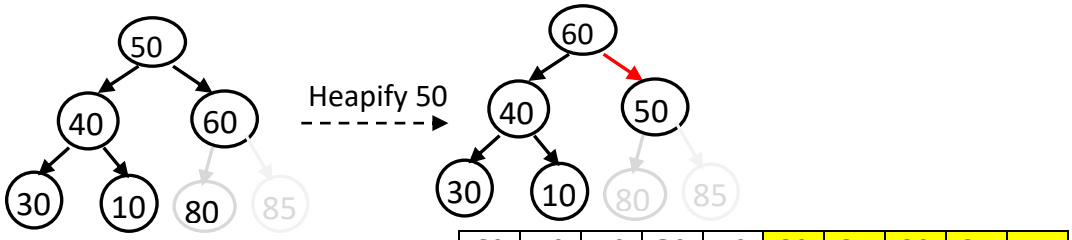
Step 4: i= 7, Repeat three steps



SSG Mishra

Step 5: i= 6, Repeat three steps

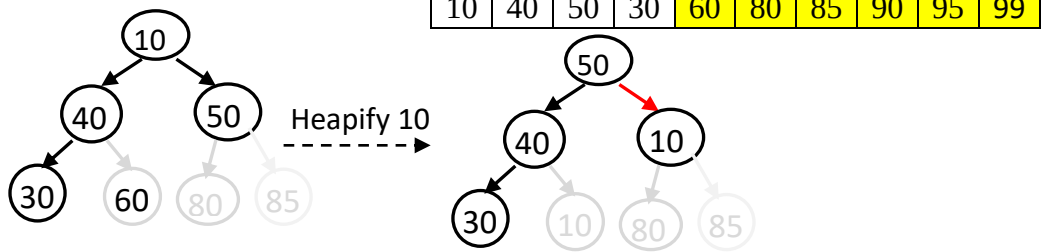
50	40	60	30	10	80	85	90	95	99



60	40	50	30	10	80	85	90	95	99

Step 6: i= 5, Repeat three steps

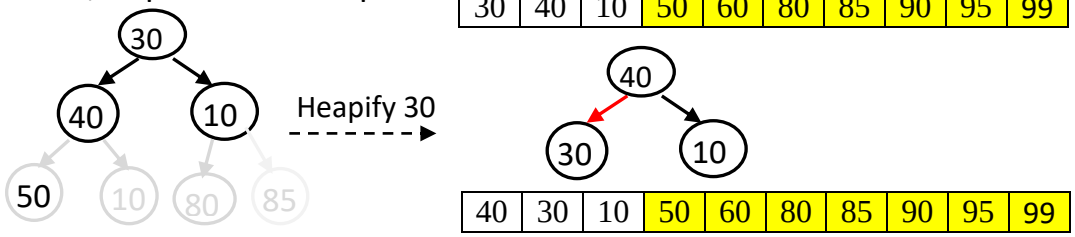
10	40	50	30	60	80	85	90	95	99



50	40	10	30	60	80	85	90	95	99

Step 7: i= 4, Repeat three steps

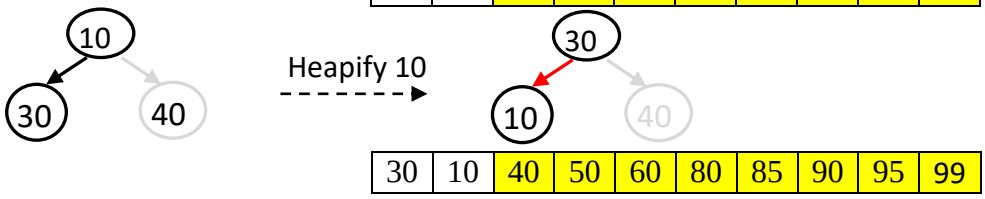
30	40	10	50	60	80	85	90	95	99



40	30	10	50	60	80	85	90	95	99

Step 8: i= 3, Repeat three steps.

10	30	40	50	60	80	85	90	95	99



30	10	40	50	60	80	85	90	95	99

Step 9: i= 2, Repeat the steps

10	30	40	50	60	80	85	90	95	99

10	30	40	50	60	80	85	90	95	99

Sorted Array