UNIT-3

Divide and Conquer Algorithms: [Imp]

A divide-and-conquer algorithm recursively breaks down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem. It consists of following steps:

Divide → The original problem is divided into a set of sub-problems.

Conquer → Solve every problem individually, recursively.

Combine → Put together solutions of the sub-problems to get the solution to the whole problem.

Fundamentals of Divide and Conquer Strategy:

i) Relational Formula: It is a formula that we generate from the given technique. After generation of formula we apply divide and conquer strategy.

ii) Stopping Condition: We should know that for how much time, we need to apply divide and conquer. So, the condition where we need to stop our recursion steps is called a stopping condition.

DAC → divide-and-conquer

⊛. Searching Algorithms:

1) Binary Search: [Imp]

Algorithm:

1. Start
2. Read the search element from the user.
3. Find the middle element in the sorted list.
4. Compare the search element with middle element.
5. If both are matching, then display "Given element found" and terminate the function.
6. If both are not matching, then check whether the search element is smaller or larger than the middle element.
7. If search element is smaller than middle element, then go to step 3, for the left sub list of the middle element.

8. If the search element is larger than middle element, then go to step 3, for the right sub list of the middle element.
9. Repeat the same process until we find the search element in the list or until sub-list contains only one element.
10. If last element also doesn't match with the search element, then display "Element not found in the list" and terminate the function.
11. Stop.

Pseudo Code:

```
BinarySearch (a,l,r,key)
{
  int m;
  int flag=0;
  if(l<=r)
  {
     m=(l+r)/2;
     if (key==a[m])
        flag =m;
     else if (key<a[m])
        return BinarySearch (a,l,m-1,key);
     else
        return BinarySearch (a,m+1,r,key);
  }
  else
  return flag;
}
```

Analysis:

From the above algorithm we can say that the running time of the algorithm is $T(n)=T(n/2)+O(1)$.

By any one of the recurrence relation solving technique we get,

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

In the best case output is obtained at one run i.e, $O(1)$ if the key is at middle.

In the worst case the output is at the end of the array so running time is $O(\log n)$ time. In the average case also running time is $O(\log n)$.

2) MIN-MAX Finding:

To find the maximum and minimum numbers in a given array of elements of size n is called min-max finding algorithm. There are two methods for finding minimum and maximum element from given array of elements: Iterative approach and divide and conquer approach.

i) Iterative Method: This is a basic method to solve any problem. In this method, the maximum and minimum number can be found seperately.

Algorithm:

```
Max-Min-Element(numbers[])
{
    Max = numbers[0]
    Min = numbers[0]
    for i=1 to n do
        if numbers[i] > Max then
            Max = numbers[i]
        if numbers[i] < Min then
            Min = numbers[i]
    Return (Max, Min)
}
```

Analysis:

The number of comparision in this method is $2n-2$.

$= O(1) \times O(n) - O(1)$

$= O(n) - O(1)$

$= O(n)$

The number of comparisions can be reduced by using divide and conquer approach.

ii) Divide and Conquer Approach for min-max finding:

In this approach, the array is divided into two halves. Then using recursive approach maximum and minimum numbers in each halves are found. Later, return the maximum of two maxima of each half and the minimum of two minima of each half. The main idea behind the algorithm is if the number of elements is 1 or 2 then min and max are obtained trivially.

Otherwise split problem into approximately equal part and solve recursively.

Algorithm सोदो code लाई Youtube बाट बुझेर यहाँ दुरुस्तसँग theory मा लेख्नो।

Pseudocode:

```
MinMax(l,r)
{ if(l==r)
    max=min=A[l];
  else if(l=r-1)
  {
      if (A[l]<A[r])
       { max=A[r];
         min = A[l];
       }
      else
       { max= A[l];
         min = A[r];
       }
  }
  else
  { //Divide the problems
      mid=(l+r)/2;
    //solve the sub-problems.
     {min, max} = MinMax (l, mid);
     {min1, max1} = MinMax (mid+1, r);
    //Combine the solutions.
     if (max1>max)
          max=max1;
     if (min1<min)
          min = min1;
  }
}
```

Analysis:

Since there are two sub-problems each of size n/2. So, size of sub-problems = n/2.

Dividing and merging sub-problems = constant time = O(1).

Then we can define their recurrance relation as; $T(n)=2T(n/2)+1$, if $n>2$

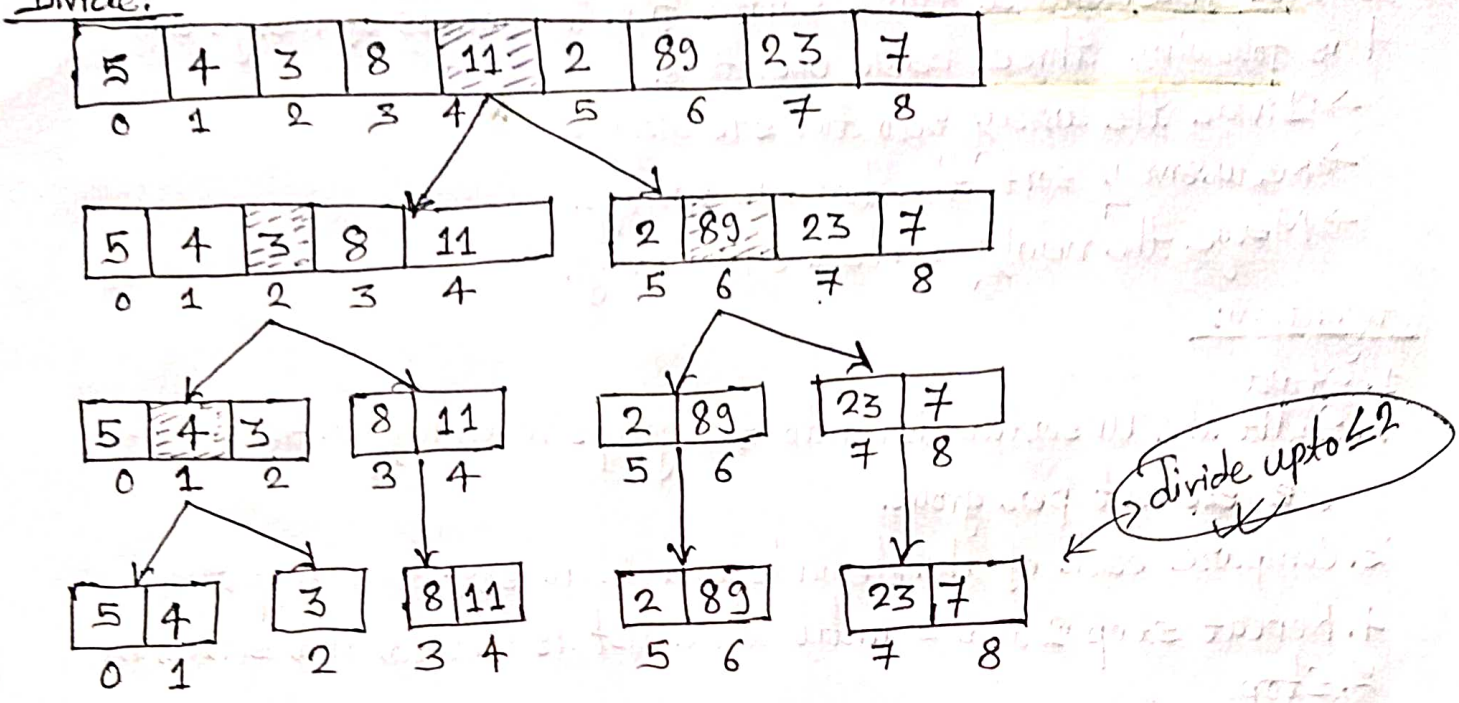
$T(n)=1$, if $n\leq 2$.

Solving the recurrence relation we get $T(n)=O(n)$.

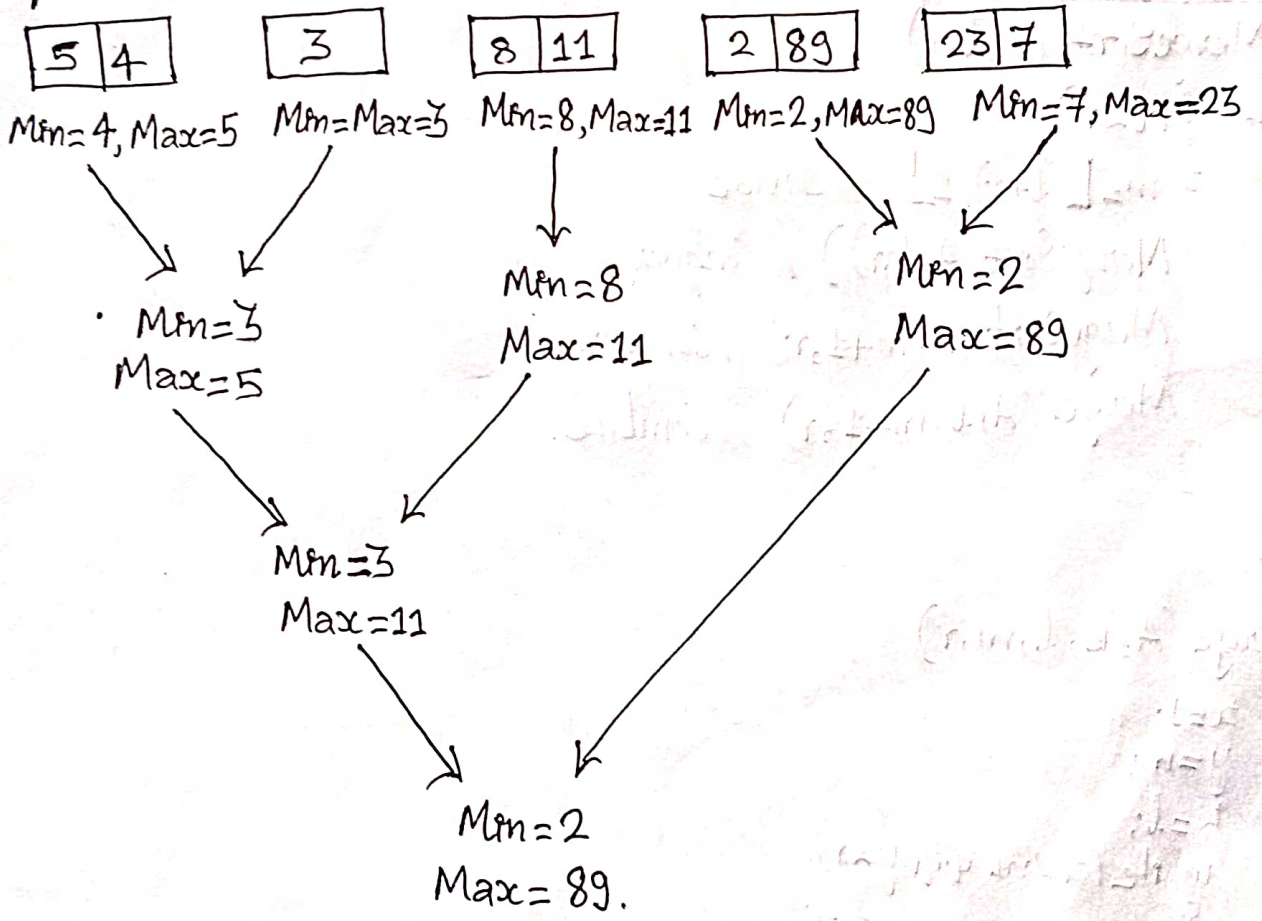
Tracing: Trace the algorithm for following array of elements by using recursive approach of min-max algorithm.

A[] = {5, 4, 3, 8, 11, 2, 89, 23, 7}

Divide:



Conquer:



⊛. Sorting Algorithms:

1) Merge Sort: Merging is the process of combining two or more sorted files into a third sorted file. The process of merge sort has following three basic operations: [Imp]

→ Divide the array into two sub arrays.
→ Recursively sort the two sub arrays.
→ Merge the newly sorted sub arrays.

Algorithm:

1. Start
2. Split the unsorted list into two groups recursively until there is one element per group.
3. Compare each of the elements and then sort and group them.
4. Repeat step 2 and 3 until whole list is merged and sorted.
5. Stop.

Pseudo Code:

```
MergeSort (A,l,r)
{ if (l<r)
   { m=⌊(l+r)/2⌋  //Divide
     MergeSort (A,l,r ) //Conquer
     MergeSort (A,m+1,r) //Conquer
     Merge (A,l,m+1,r) //Combine.
   }
}

Merge (A,B,l,m,r)
{ x=1;
  y=m;
  k=l;
  while (x<m && y<r)
  { if (A[x]<A[y])
    { B[k]=A[x];
      k++;
      x++;
    }
```

```
        else
        { B[k] = A[y];
          k++;
          y++;
        }
    }
    while (x<m)
    { A[k] = A[x];
      k++;
      x++;
    }
    while (y<r)
    { A[k] = A[y];
      k++;
      y++;
    }
    for (q=1; q<=r; q++)
        A[q] = B[q]
}
```

Time Complexity:

No. of sub-problems = 2.
Size of each sub-problem = n/2.
Dividing cost = constant
Merging cost = n.

Thus recurrence relation for Merge sort is;

$T(n) = 1$ if $n = 1$

$T(n) = 2T(n/2) + O(n)$ if $n > 1$.

By solving recurrence relation we get,

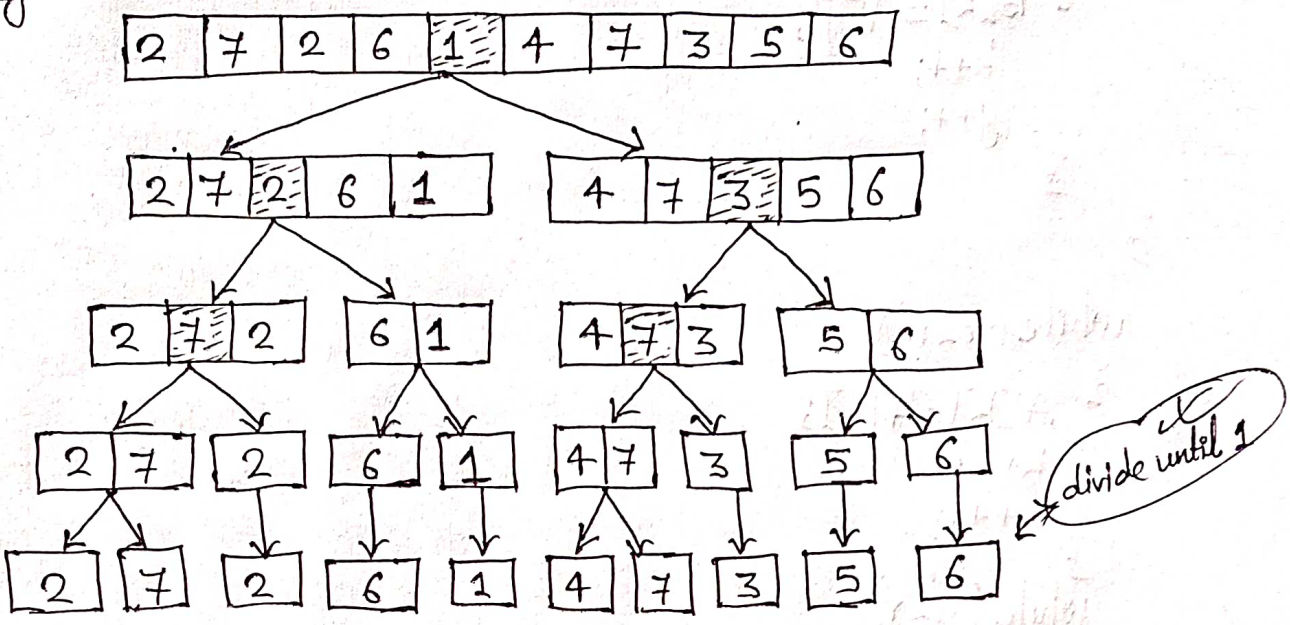
Time complexity $T(n) = O(n \log n)$.

Space Complexity:

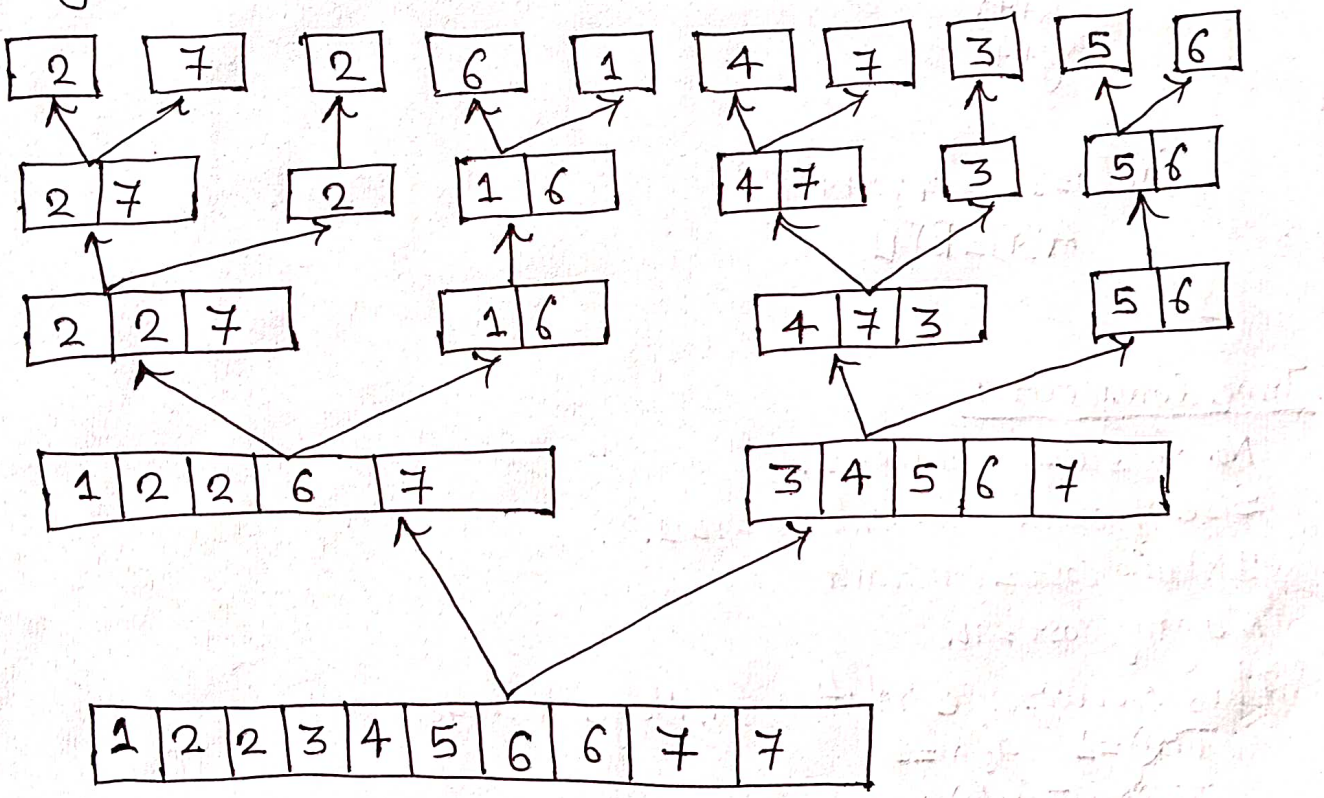
Merge sort uses additional memory for left and right sub arrays. Hence, space complexity $= O(n)$.

Example/Tracing: A[] = {2, 7, 2, 6, 1, 4, 7, 3, 5, 6}

Dividing:



Merging:



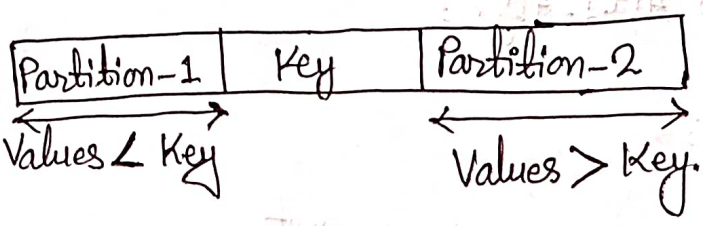
2) Quick Sort: [Imp]

As its name implies, quick sort is a fast divide-and-conquer algorithm. It has two phases: The partition phase and the sort phase.

Divide: Partition the array A[l...r] into two sub-arrays A[l...pivot] and A[pivot+1...r], such that each element of A[l...pivot] is smaller than or equal to each element in A[pivot+1...r].

Conquer: Recursively sort A[l...pivot] and A[pivot+1...r] using Quick sort.

Combine: The arrays are sorted in place. No additional work is required to combine them.



Algorithm:

1. Start
2. Choose a pivot.
3. Select a left pointer and right pointer.
4. Compare the left pointer element (lelement) with the pivot and the right pointer element (relement) with the pivot.
5. Check if lelement < pivot and relement > pivot:
   → if yes, increament the left pointer and decreament the right pointer.
   → If not, swap the lelement and relement.
6. When left >= right, swap the pivot with either left or right pointer.
7. Repeat steps 1 to 5 on the left half and right half of the list till the entire list is sorted.
8. Stop.

Pseudo Code:

```
QuickSort (A, l, r)
{ if (l<r)
  { p=partition(A,l,r);
    QuickSort (A,1,p-1);
    QuickSort (A,p+1,r);
```

```
    }
}
Partition(A,l,r)
{
    x=l;
    y=r;
    p=A[l];
    while(x<y)
    { while(A[x]<=p)
        x++;
      while(A[y]>=p)
        y--;
      if(x<y)
        swap(A[x],A[y]);
    }
    A[l]=A[y];
    A[y]=p;
    return y;  //return position of pivot
}
```

⊛ Time Complexity:

In Best Case, time complexity $T(n)=O(n \log n)$.

In Worst Case, time complexity $T(n)=O(n^2)$

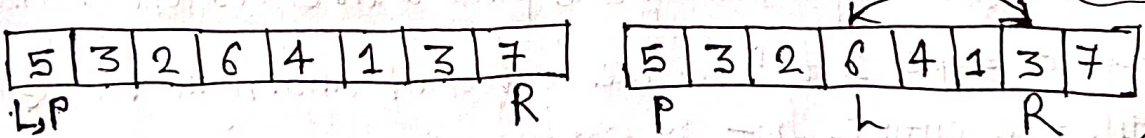
In Average Case, time complexity $T(n)=O(n \log n)$.

Example: Sort the following data items by using Quick sort.

$A[]=\{5,3,2,6,4,1,3,7\}$

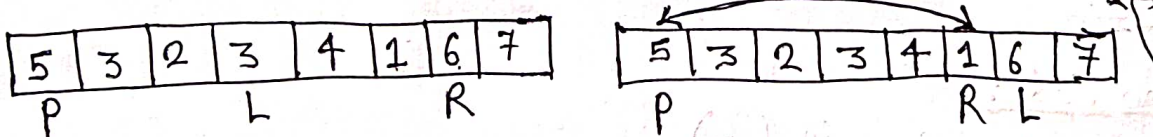
Solution:

Pass1:



① Take 1st as pivot. L will move right until element greater than pivot is found.

② R will decrement until ≤ P is found

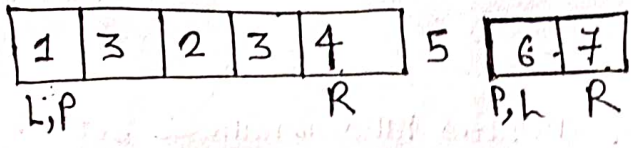


③ Then we swap elements if L>R and continue same process.

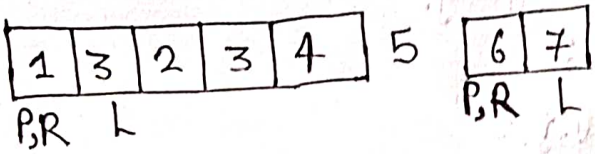
④ When pivot goes to right in between then problem is divided into 2 parts and solved seperately

If L and R cross eachother then swap with pivot

Pass2:

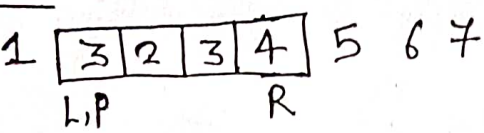


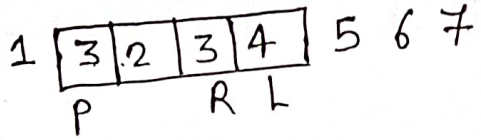
Left र right मा Pass1 जस्तै same process repeat गर्ने seperately. L र R ले cross गर्‍यो but L>P Ee भने त्यसैले रहन्छ।



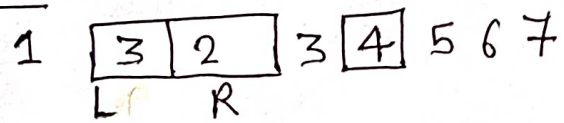
कुनै change भएन sequence मा भने pivot change गर्ने र past pivot लाई as sorted बाहिर निकाल्ने as in step3 or Pass3.
कुनै part मा एउटा मात्र रहन्छ भने त्यो पनि बाहिर आउछ, for eg. 7 here.

Pass3:

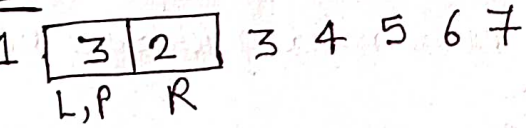


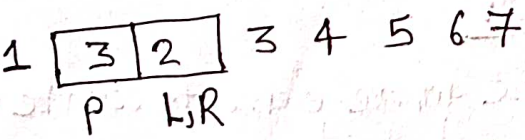


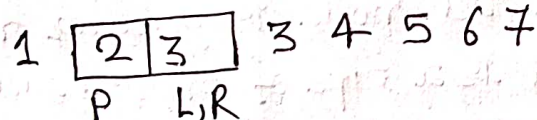
Pass4:



Pass5:







1 2 3 4 5 6 7

3) Randomized Quick Sort: We use random approach for selecting pivot. The algorithm is called randomized if its behaviour depends on input as well as random value generated by random number generator. The beauty of the randomized algorithm is that no particular input can produce worst-case behaviour of an algorithm. The worst case only depends on random number generated.

Algorithm:

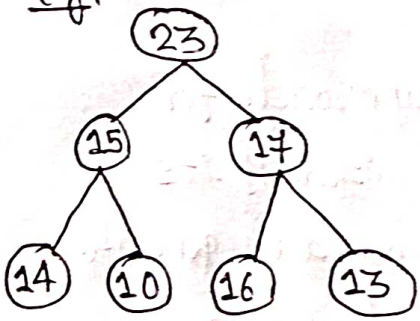
```
RandPartition (A,l,r) {
        k=random(l,r); //generates random number between l and r.
        swap(A[l], A[k]);
        return Partition(A,l,r);
}

RandQuickSort(A,l,r)
{   if (l<r)
    {m= RandPartition(A,l,r);
      RandQuickSort (A,l,m-1);
      RandQuickSort (A,m+1,r);
    }
}
```

Note: Time complexity analysis for best, worst and average case is same as quick sort, for randomized quick sort also.

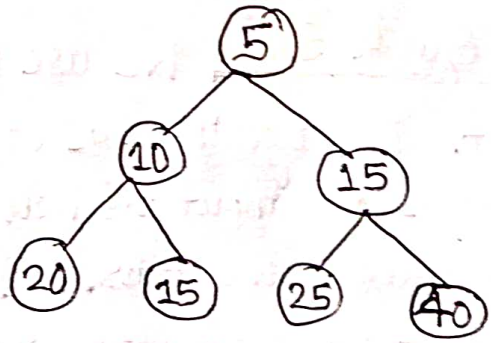
⊛ Concept of Heap Data Structures:

Heap is a data structure where elements can be viewed in a binary tree and in which each node is greater or smaller than it's childrens. It is a type of priority queue. A heap in which each node is greater than children node is called max heap otherwise it is called min-heap.

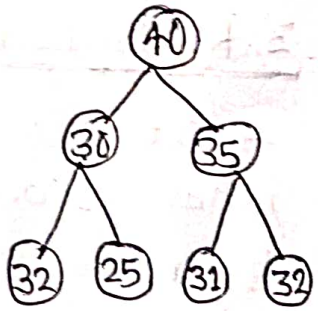
e.g:



Max heap



Min heap



Not a heap

4) Heap Sort: [Imp]

Algorithm:

1. Build a max-heap from the array.
2. Swap the root (maximum element) with the last element in the array.
3. "Discard" this last node by decreasing the heap size.
4. Perform Max-Heapify operation on the new root node.
5. Repeat this process until only one node remains.

Pseudo Code:

```
HeapSort (A)
{ BuildHeap(A);
  n = length [A];
  for (i=n; i>=2; i--)
  {
     swap (A[1], A[n]);
     n=n-1;
     Heapify (A,1);
  }
}
```

Analysis:

Build heap takes O(n) time.
For loop executes at most O(n) time.
Within for loop heapify operation takes at most O(log n) times.
Thus total time complexity $T(n) = O(n) + O(n) + O(\log n)$.

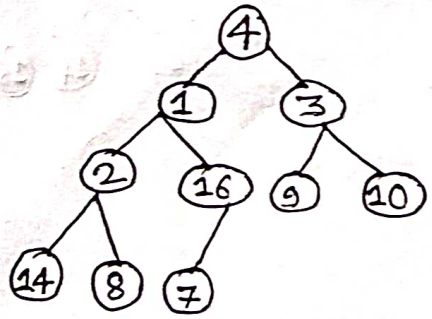
$$\Rightarrow T(n) = O(n \log n).$$

Example: Sort following elements by using heap sort.

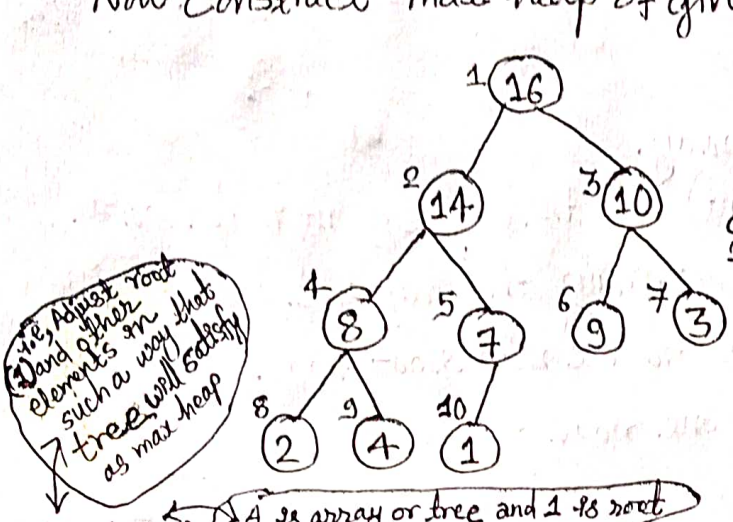
A[ ] = {4, 1, 3, 2, 16, 9, 10, 14, 8, 7}

Solution:

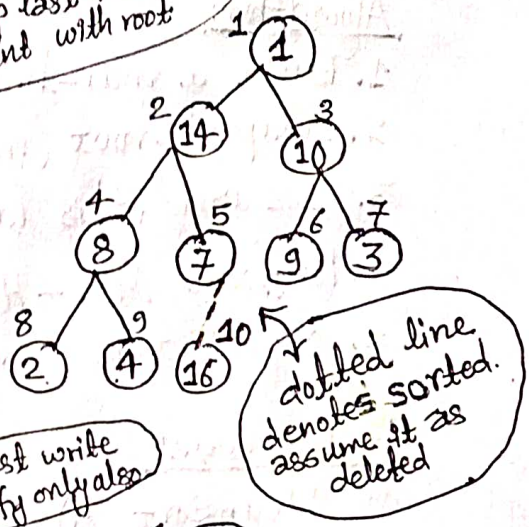
At first we construct a binary tree of given array.



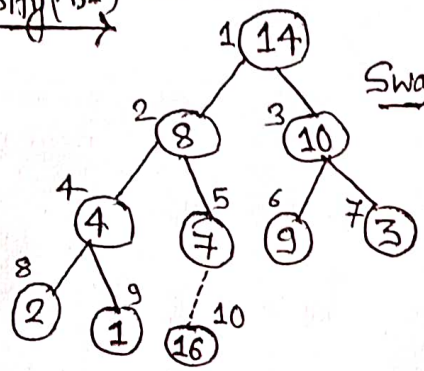
Now construct max heap of given tree and sort as;



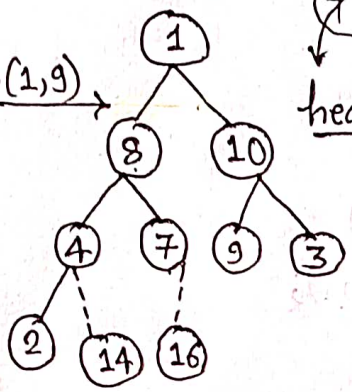
heapify(A,1) — A is array or tree and 1 is root



heapify(A,1)

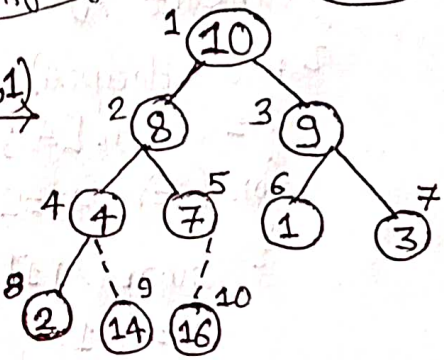


Swap(1,9)

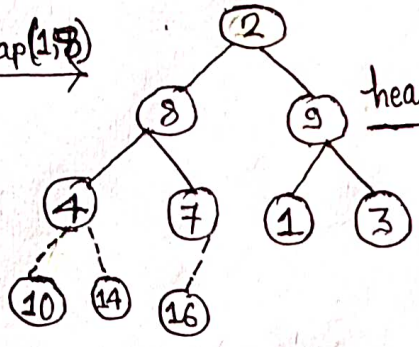


we can just write heapify only also

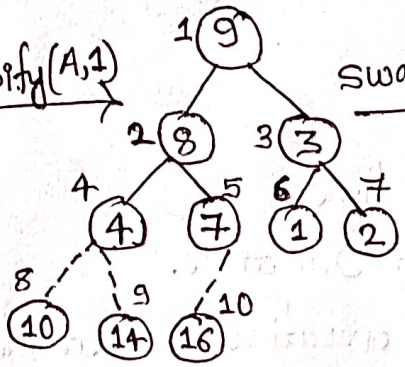
heapify(A,1)



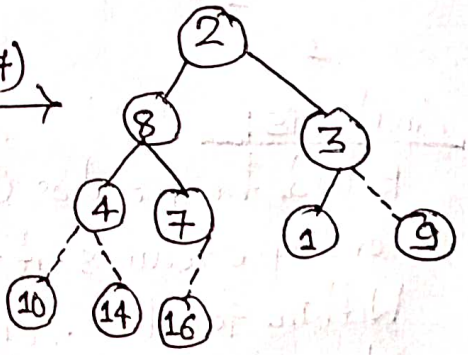
Swap(1,8)



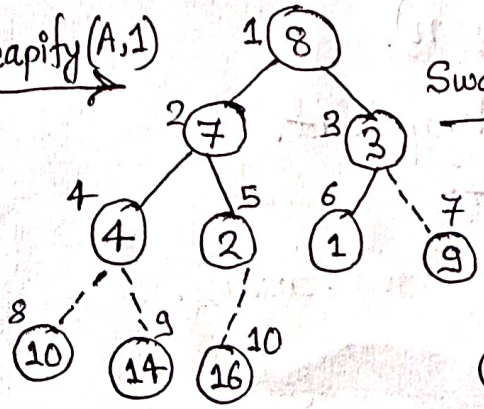
heapify(A,1)



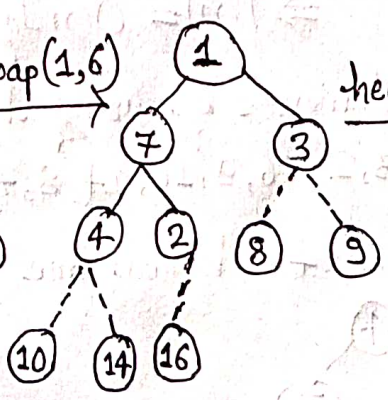
swap(1,7)



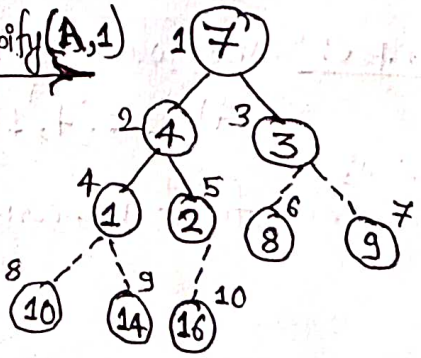
heapify(A,1)

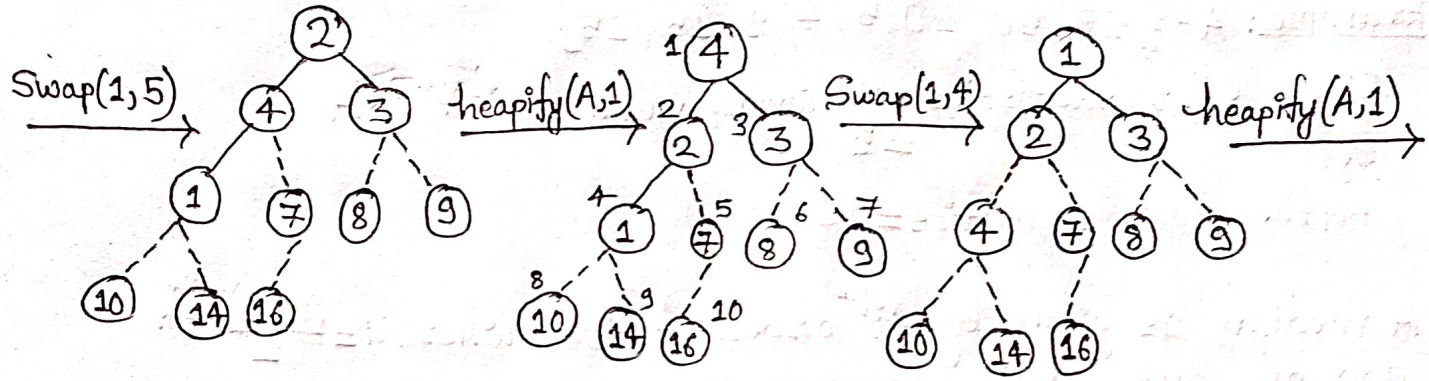


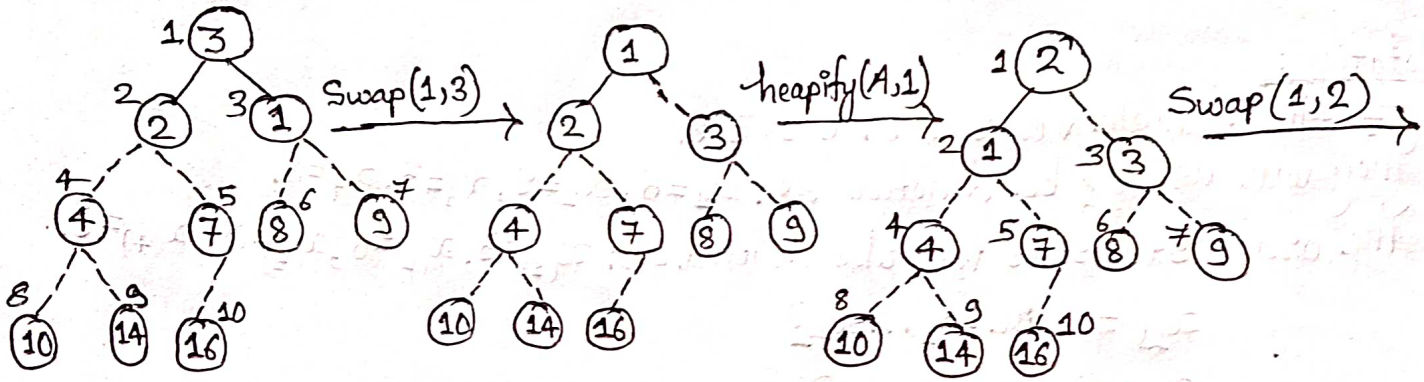
Swap(1,6)

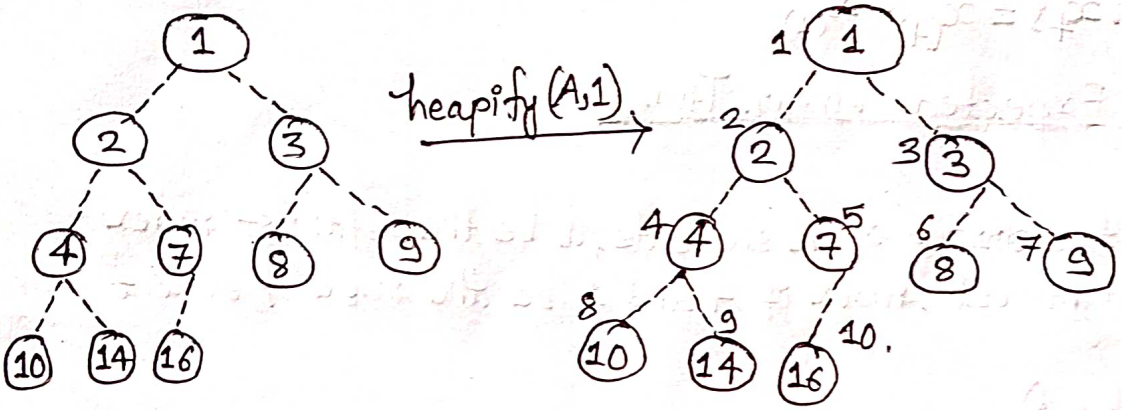


heapify(A,1)









Hence sorted arry is A[] = {1,2,3,4,7,8,9,10,14,16}

⊛ Order Statistics: (Selection Problem)

Order Statistics are smaple values placed in ascending order. The study of order statistics deals with the applications of these ordered values and their functions.

Definition: Given a set A of n elements, i^{th} order statistics gives i^{th} smallest (or largest) element in the set.

Problem:

Input: An array A of n distinct elements.

Output: An element $e \in A$ such that e is larger than exactly i-1 elemens.

Example: A={ 28, 50, 10, 5, 7, 9, 15, 25}

First order statistics = minimum value element of A
= 5

Fifth order statistics = 15

A median is given by i^{th} order statistic where $i = \frac{(n+1)}{2}$ for odd n and $i = \frac{n}{2}$ for even n.

Notation:

If the sample values are: 6, 9, 3, 8.
They will usually be denoted as: $x_1=6, x_2=9, x_3=3, x_4=8$.
The order statistics would be denoted as: $x_{(1)}=3, x_{(2)}=6, x_{(3)}=8, x_{(4)}=9$.

$$x_{(1)} = \min\{x_1 \ldots x_4\}$$
$$x_{(4)} = \max\{x_1 \ldots x_4\}$$
$$\text{Range}\{x_1 \ldots x_4\} = x_{(4)} - x_{(1)}$$

⊛ Selection in Expected Linear Time:

Algorithm:

Let A be the array of n elements, l be the leftmost index of A, r be the rightmost index of A and i be the index of element to be select.

```
RandSelect (A,l,r,i)
{ if(l==r)
     return A[l];
  p = RandPartition (A,l,r);
  k = (p-l+1);
  if (i≤k)
        return RandSelect (A,l,p-1,i);
  else
        return RandSelect (A,p+1,r,i-k);
}
RandPartition (A,l,r)
{ k = random(l,r);
  swap(A[l], A[k]);
  return Partition (A,l,r);
}
```

```
Partition (A,l,r)
{ x=l; y=r; p=A[l];
  while (x<y)
  { do{
      x++;
    }
    while(A[x]<=p);
    do{ y--;
    }
    while (A[y]>=p);
    if(x<y)
    swap(A[x], A[y]);
  }
  A[l]= A[y];
  A[y] = p;
  return y; //return position of pivot
}
```

do while loops inside while loop

Analysis:-

The worst case running time of this algorithm is $O(n^2)$. This happens if every time unfortunately the pivot chosen is always the largest one.

⊛Selection in Worst Case Linear Time:

Algorithm:

1. Divide the n elements of the input array into $\lfloor n/5 \rfloor$ groups of 5 elements each and at most one group made up of the remaining n mod 5 elements.
2. Find the median of each of the $\lceil n/5 \rceil$ groups by insertion sorting the elements of each group and taking its middle element.
3. Select recursively to find the median x of the $\lceil n/5 \rceil$ medians found in Step 2.

4. Partition the input array around the median-of-medians x using a modified version of Partition. Let k be the number of elements on the low side of the partition, so that $n-k$ is the number of elements on the high side.

5. Select recursively to find the ith smallest element on the low side if $i \leq k$, or the $(i-k)^{th}$ smallest element on the high side if $i > k$.

Analysis:

The running time complexity of this algorithm is $T(n) = O(n)$.