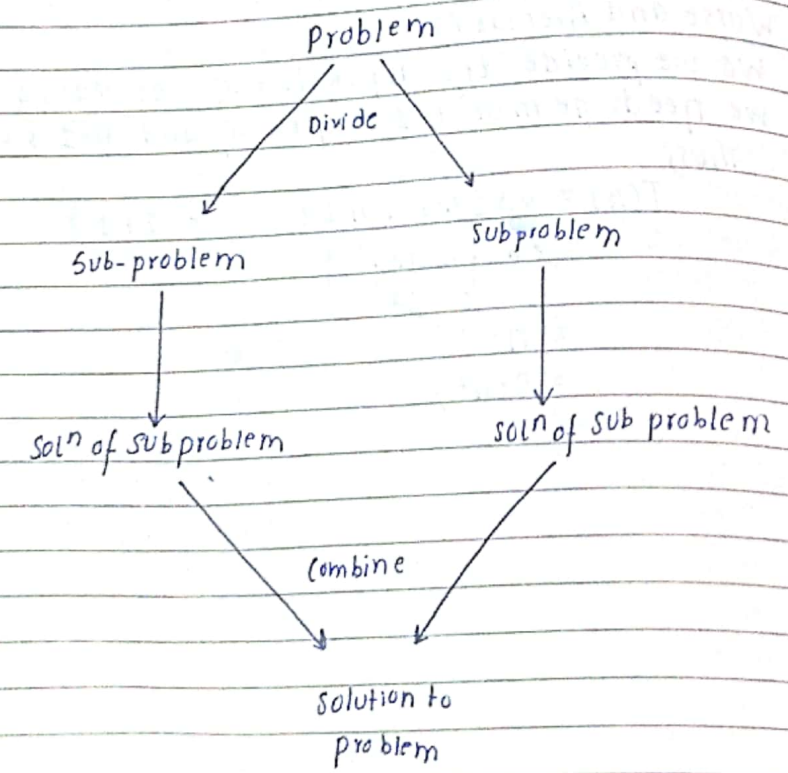
Divide and Conquer

Chapter-3

Divide and Conquer

→ Divide and Conquer is a algorithmic method, where large input are broken into minor pices or sub problem, we find the solution of that sub problem and merge those piecewise solution into a global solution.



Searching Algorithm

→ Searching is an operation or technique that helps to find the place of the given element or value in the list.

Some of the standard searching technique that are being followed are

a) Sequential Search
b) Binary Search

Binary Search using Recursion

Algorithm :-

```
BinarySearch (arr, l, r, key)
  {
  if (l < r)
    {
    mid = (l+r)/2;
    if ( arr[mid] == key)
      {
      return 1;
      }
    else if (arr[mid] > key)
      {
      return BinarySearch (arr, l, mid-1, key);
      }
    else return BinarySearch (arr, mid+1, r, key);
    }
  return 0;
  }
```

Binary Search using divide and Conquer

Algorithm :-

```
BinarySearch (arr, size, key)
  {
  left = 0;
  right = size-1;
  while ( left < right )
    {
    mid = (left+right) /2;
    if ([mid] == key)
      {
      return 1;
      }
    else if ( arr[mid] > key)
      {
      Right = mid-1;
      }
    else {
      left = mid+1;
      }
    }
  return 0;
  }
```

Time Complexity:-

The running time of the above algorithm is,

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

On solving the recurrence relation, we get,

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

Best Case :- The best case ~~oupt~~ output is obtained when the key is at middle.

Thus,

$$T(n) = O(1)$$

Average Case :- $O(\log n)$

Finding minimum and maximum element in the List of item (Min-Max)

i) Iterative method :-

```
MinMax(arr)
  {
  min = arr[0];
  max = arr[0];

  for(i = 1 to n)
    {
    if (arr[i] > max)
      {
        max = arr[i])
      }
    if (arr[i] < min)
      {
        min = arr[i]
      }
    }
  return (max, min)
  }
```

Time Complexity.

Since loop run for 'n' time,

time complexity = O(n)

(iii) Divide and Conquer Min-Max

```
MinMax (left, right)
{
If (left == right)      //only 1 element left
  {
  min = max = arr[left];
  }
else if (left = right-1)      //only 2 element
  {
  if (arr[left] < arr[right])
    {
    max = arr[right];
    min = arr[left];
    }
  else
    {
    max = arr[left]
    min = arr[right];
    }
  }
else                    //more than 2 element using Divide
  {
  mid = (left + right)/2
{max, min} = MinMax(left, mid);
{max1, min1} = MinMax(mid+1, right);
If (max1 > max)
  {
  max = max1;
  }
If (min > min1)
  {
  min = min1;
  }
  }
}
```

Time Complexity :-

Here, the problem is divided into two-sub problem each of size $n/2$.
and, Every time problems are divided into 2 sub-problem and, we do not need to merge those sub-problem.
The recurrence relation from above problem is given by

$$T(n) = 2T\left(\frac{n}{2}\right) + 1, \text{ if } n > 2$$

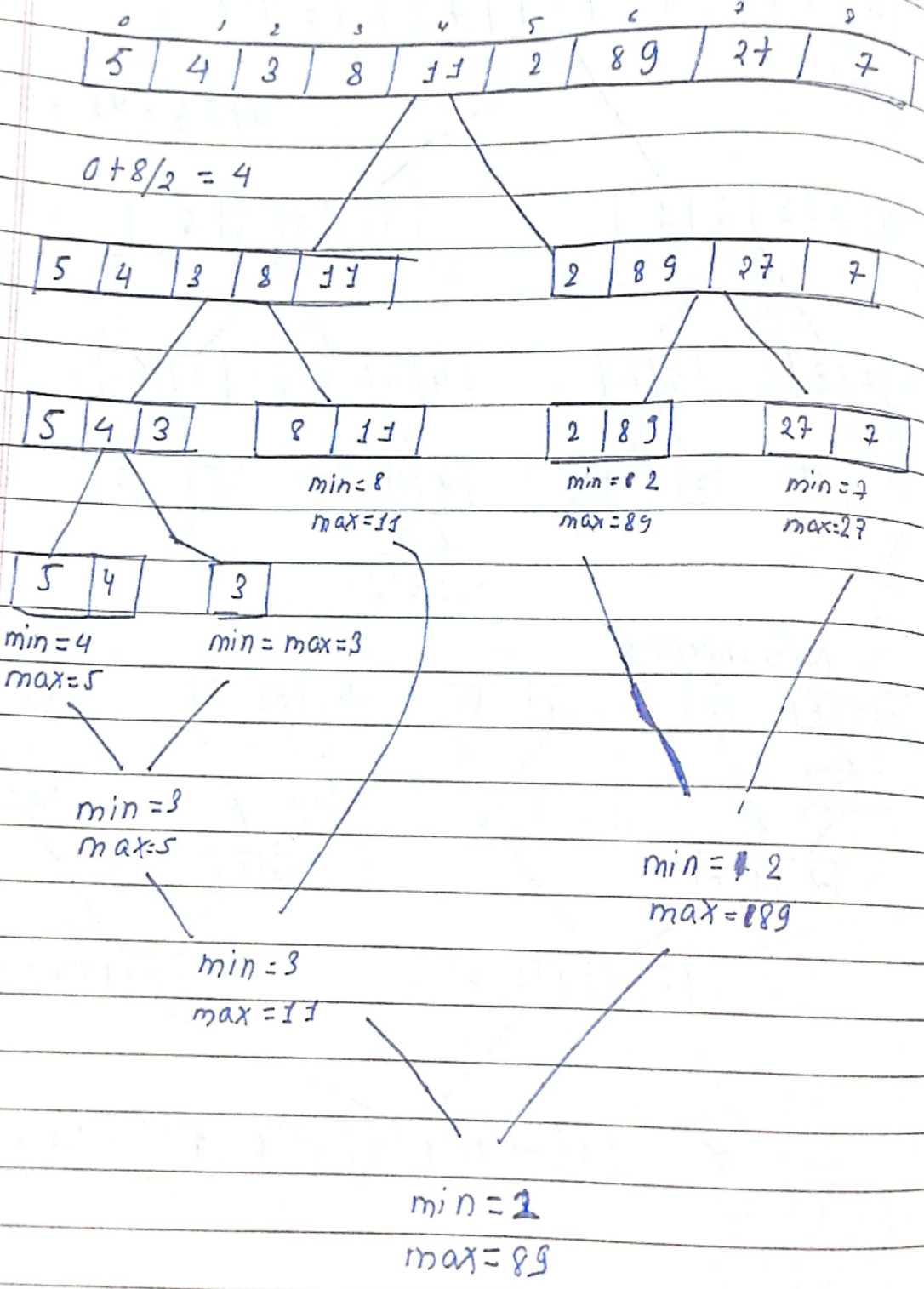
$$T(n) = 1 \quad \text{if } n \leq 2$$

on solving above recurrence relation we get

$$T(n) = O(n)$$

Q2) A = {5, 4, 3, 8, 11, 2, 89, 27, 7}

Trance Min-Max

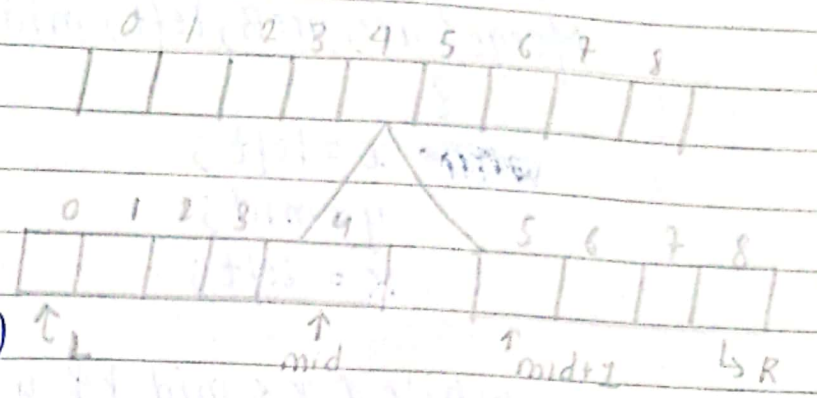


Merge Sort :-

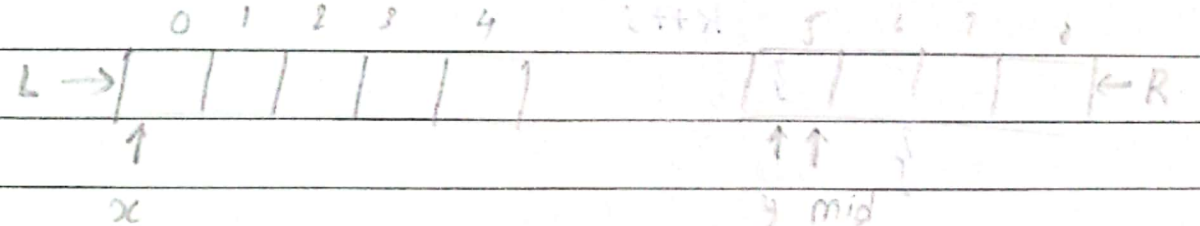
Algorithm :-

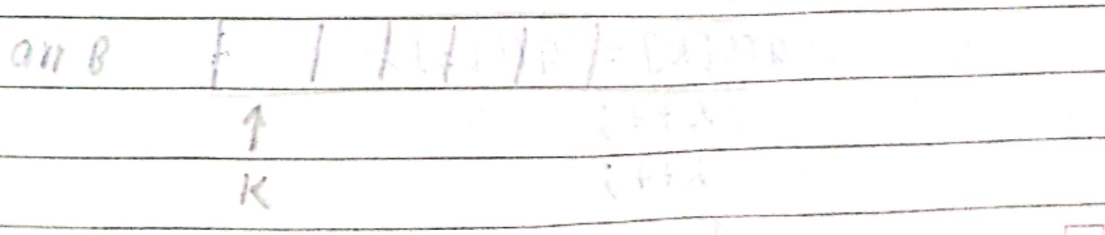
```
MergeSort (arr, left, right)
    {
    If (left < right)
        {
        mid = (left + right) / 2 ;
        MergeSort (arr, left, mid) ;
        MergeSort (arr, mid+1, right);
        let arrB;
        Merge (arr, arrB, left, mid+1, right)
        }
    }
```

In MergeSort



In Merge,





```
Merge(arr, arrB, left, mid, right)
{
    x = left;
    y = mid;
    k = left;

while ( x < mid && y < right )
    {
If ( arr[x] < arr[y] )
        {
        arrB[k] = arr[x];
          x++;
          k++;
        }
else
        {
        arrB[k] = arr[y];
          y++;
          k++;
        }
}

while ( x < mid)
    {
arr[k] = arr[x];
  x++;
  k++;
}
```

```
while ( y < right )
  {
  arr[k] = arr[y];
    y++;
    k++;
  }

for(i=0; i<n; i++)        // copying data from arrB to arr
  {
  arr[i] = arrB[i];
  }

}
```

Time Complexity :-

The recurrence relation for Merge Sort is

$$T(n) = 1 \quad \text{when } n = 1$$

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n) \quad \text{when } n > 1$$

By solving the recurrence relation we get

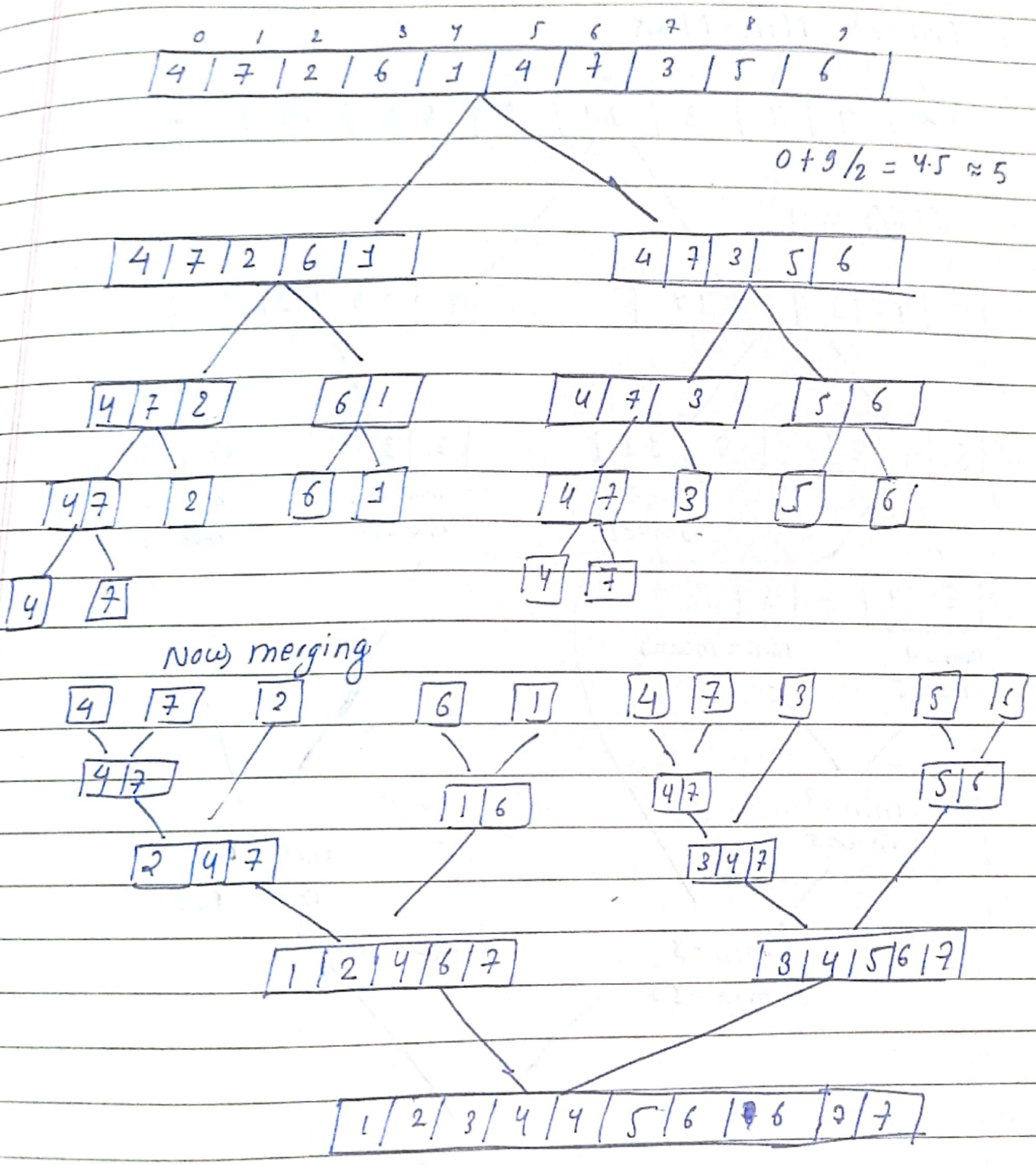
$$T(n) = O(n \log n)$$

Here,

Best Case = Worst Case = Avg Case = $O(n \log n)$.

DATE

Q. A = {4, 7, 2, 6, 1, 4, 7, 3, 5, 6} using Merge Sort



PAGE

QuickSort :-

Algorithm :-

```
QuickSort (arr, left, right)
  {
  if (left < right)
    {
    pi = Partation (arr, left, right);
    QuickSort (arr, left, pi-1);
    QuickSort (arr, pi+1, right);
    }
  }

Partation (arr, left, right)
  {
  x = left;
  y = right;
  pivot = arr[left];
  while (x < y)
    {
  while (arr[x] <= pivot)
      {
      x++;
      }
```

```
  while (arr[y] > pivot)
      {
      y--;
      }
  if (x < y)
    {
    swap (arr[x], arr[y])
    }
  }
  swap (arr[left], arr[y])
    return y;
  }
```

Imp

Time Complexity :-

Best Case :- QuickSort gives best time complexity when the array is divided into two equal size.

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

on solving,

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

DATE

Worst Case :-

Quick Sort give worse case complexity when the elements are already sorted.
Therefore, its recurrence relation is given by

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

After solving,

$$T(n) = O(n^2)$$

Average Case :-

On average case, we have balanced and unbalance split.

Recurrence relation for balance and unbalance are,

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n) \quad -(i) \quad \text{(balance)}$$

$$T(n) = T(n-1) + O(n) \quad -(ii) \quad \text{(unbalance)}$$

On solving eq (i) n eq (ii), we get

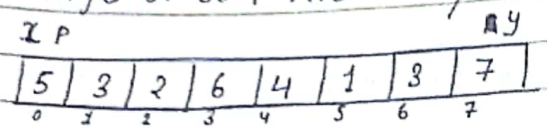
$$T(n) = O(n \log n)$$

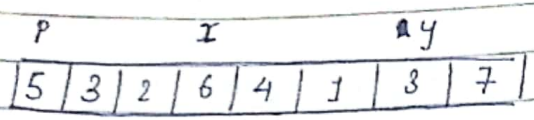
classmate

Q10

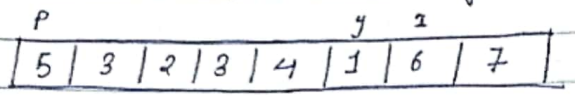
A[] = {5,3,2,6,4,1,3,7}

Arrange or Sort the Array using QuickSort

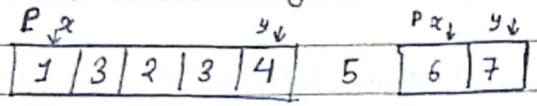


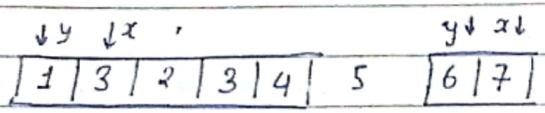


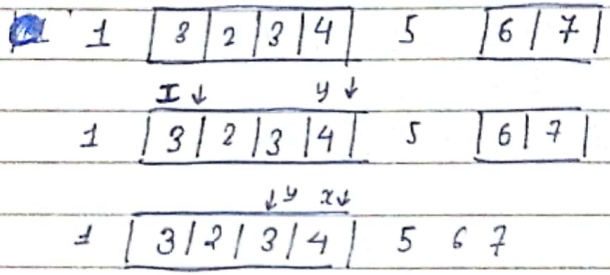
Swap [x] and [y] since x<y

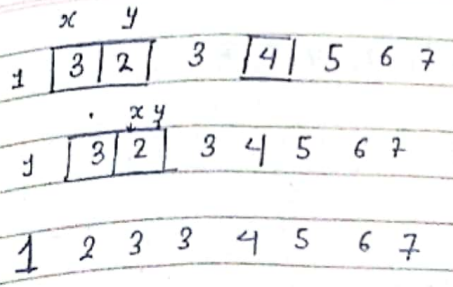


Swap pivot and y index element, since x>y









1 2 3 3 4 5 6 7

Randomize Quick Sort

Algorithm:-

```
RanQuickSort (arr, left, right)
    {
    if (left < right)
      {
      pi = RandPartation (arr, left, right)
      RandQuickSort (arr, left, pi-1);
      RandQuickSort (arr, pi+1, right);
      }
    }

RandPartation ( arr, left, right).
      {
    k = random (left, right)
    swap (arr[left], arr[k])
     return Partation (arr, left, right);
      }

Partation (arr, left, right)
      {
       // same as QuickSort (if 10 marks then includethis)

      }
```

Time Complexity:-

Worst Case :-

Let 'k' be the partition element then there are two sub-problem k and n-k.

where,

k is some partition point generated by random number generator.

Since there are 'n' elements, so we need at most O(n) time for dividing.

Thus,

Their recurrence relation can be defined as

$$T(n) = T(k) + T(n-k) + O(n)$$

on solving,

$$T(n) = O(n^2)$$

Average Case and Best Case :-

Now, consider the probability of choosing the pivot from n element is equally like ie $\frac{1}{n}$.

Now, we give recurrence relation for the algorithm as

$$T(n) = \frac{1}{n} \sum_{k=1}^{n-1} \left(T(k) + T(n-k)\right) + O(n)$$

on solving recurrence relation,

we get,

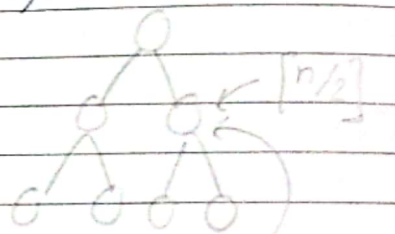
$$T(n) = O(n \log n)$$

Heap Sort

Algorithm:-

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

BuildHeap (arr, n)
  {
  for (i= n/2; i>0; i--)
    {
    Heapify (arr, n, i)
    }
  }
```

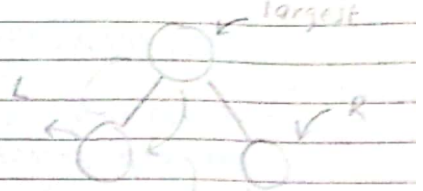


for here we start to Heapify

since, no need for ... node to ...
leaf


```
Heapify (arr, n, i)
  {
  largest = i;
  L = left(i);
  R = Right(i);

  if (L ≤ n && arr[L] > arr[largest])
    {
    largest = L;
    }
  if (R ≤ n && arr[R] > arr[largest])
    {
    largest = R;
    }
  if (largest != i)
    {
    swap (arr[left], arr[largest]);
    Heapify (arr, n, largest)
    }
  }
```



swap with ... which is bigger than ... largest

Time Complexity :-

Build heap takes $O(n)$ time
for loop executes at most $O(n)$ time
and,
within for loop, heapify operation takes
at most $O(\log n)$ time,

Thus, total time complexity,

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

Here,
Best Time Complexity = $O(n \log n)$
Avg Time Complexity = $O(n \log n)$
Worst Time Complexity = $O(n \log n)$

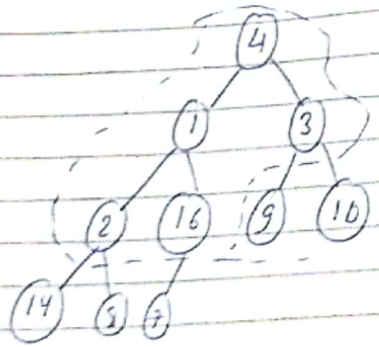
Q.D

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

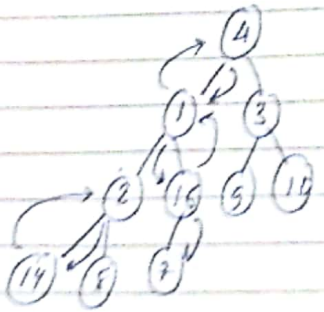
Sort using heapsort

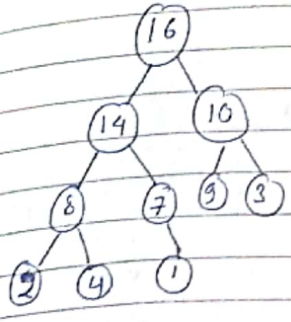
Soln,

Constructing Binary tree

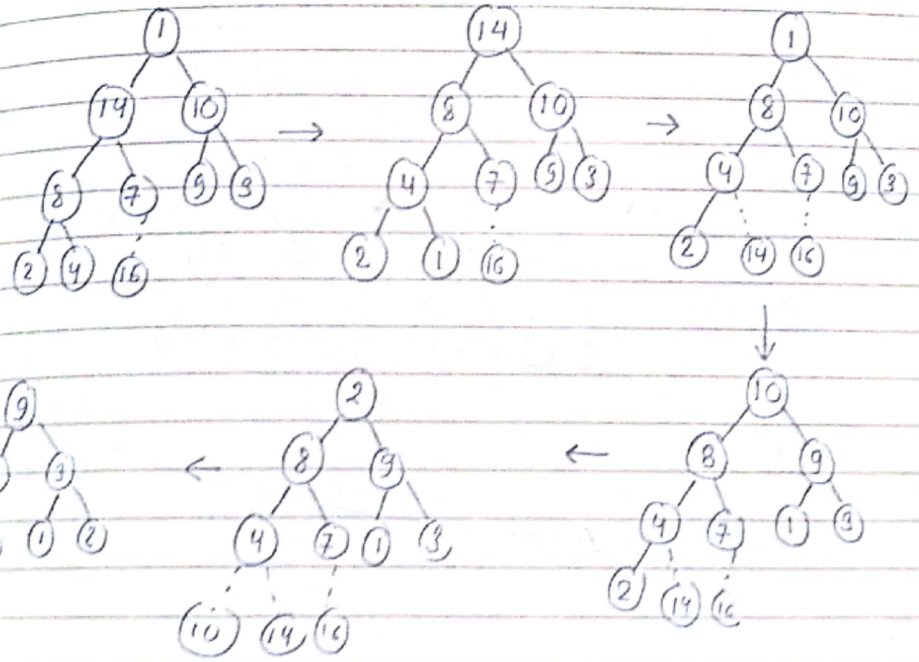


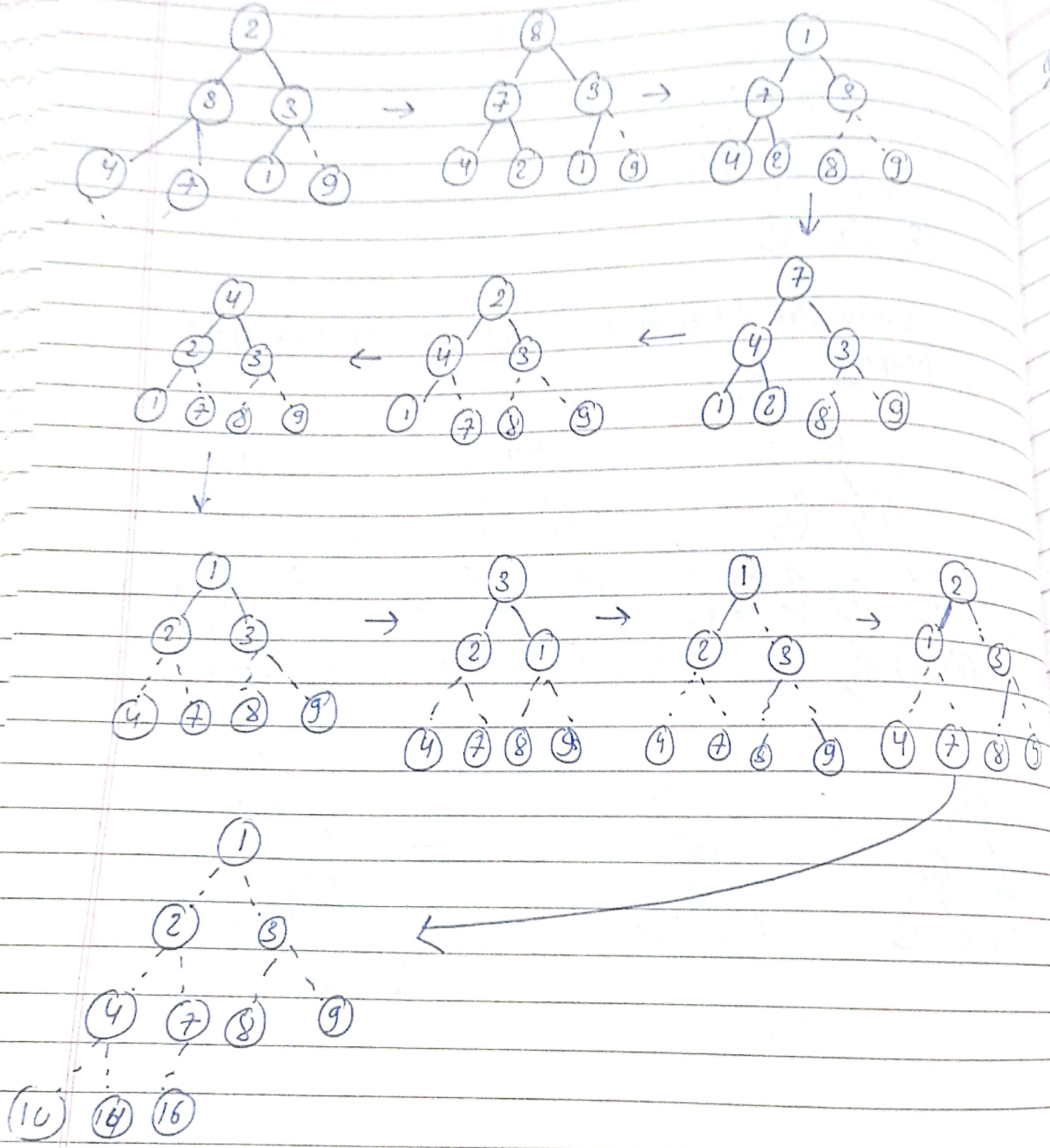
Now, Constructing heap of given tree





Swap the 1st element with last element and again heapify





Order Statistics

→ Order statistics refers to the value of element in a sample when elements are arranged in ascending or descending order.

for example:-

→ you have elements {4, 1, 7, 3, 9}

If you arrange these element is ascending order then we get

{1, 3, 4, 7, 9}

Here,
The second order statistic (k=2) in this case is 3.
This mean '3' is the second smallest value in the list of elements.

So, ith order of set of element gives ith smallest/largest element.

→ Order Statistic is denoted by X(K), to represent kth order of statisic in a sample.

→ A median is given by ith order statistic, where

$$i = \frac{n+1}{2} \quad \text{for odd } n$$

and,

$$i = \frac{n}{2} \text{ and } \frac{n}{2+1} \quad \text{for even } n.$$

Selection in Worst Case Linear Time

→ Select (i, n)

1) Divide the n element into ⌊ groups of 5 Then find the median of each 5 element group.

2) Recursively select the median x of the $\left(\frac{n}{5}\right)$ groups to be pivot.

3) Partation the array taking pivot = x;

4) if i = k then return x

else if (i < K)
then recursively find ith smallest element in first partation

else recursively find (i-k)th smallest element in second partation

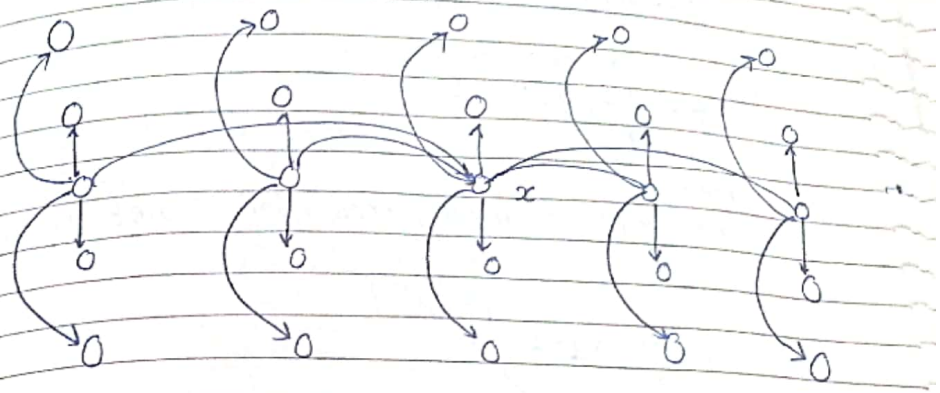


fig: Selection in Worst case

The Time Complexity of this algorithm is O(n).

Imp

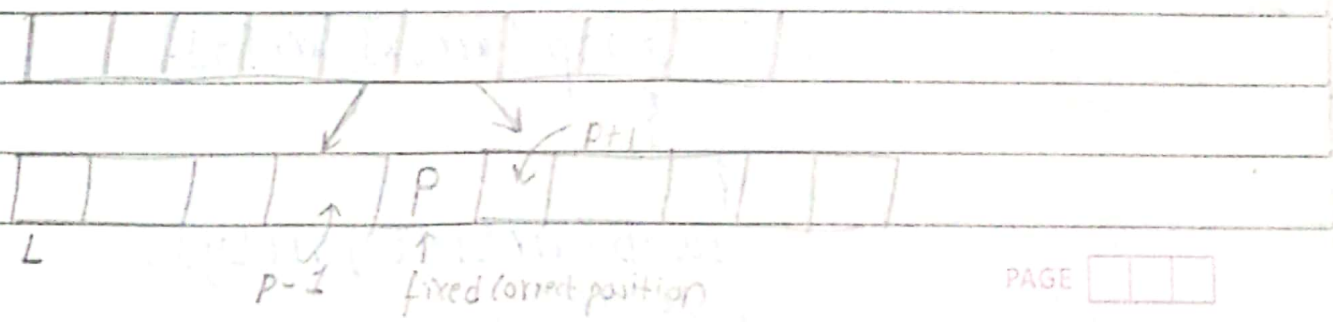
DATE

Selection in Expected Linear Time

Algorithm :-

```
RandomSelect ( arr, left, right, Index )
    {
    If ( left == right )
        {
        return arr [left] ;
        }
    p = RandPartation (arr, left, right )
    k = p - left + 1 ;

    If (index ≤ k)
            {
            return RandSelect (arr, left, p-1, index);
            }
    else
        {
        return RandSelect (arr, p+1, right, index - k);
        }
    }
```



classmate

PAGE

```
RandPartation (arr, left, right)
{
randomIndex = Rand(left, right);
swap (arr[left], arr[randomIndex]);
Partation (arr, left, right);
}

Partation (arr, left, right)
{
pivot = arr[left];
x = left;
y = right;
while (x < y)
{
while (arr[x] ≤ pivot)
{
x++;
}
while (arr[y] > pivot)
{
y--;
}
if (x < y)
{
swap (arr[x], arr[y])
}
}
swap (arr[left], arr[y])
return y;
}
```

Time Complexity :-

The Worst case running time of the algorithm is $O(n^2)$.
This happens if every time, when selected pivot is always the largest one.

The Expected time complexity of the algorithm is $O(n)$

Here,
The probability of selecting pivot is equal to all element i.e $\frac{1}{n}$ then we have recurrence relation

$$T(n) = \frac{1}{n}\sum_{j=1}^{n-1} T(\max(j, n-j))) + O(n)$$

Here, $\max(j, n-j) = j$ if $j > \lceil n/2 \rceil$
else
$\max(j, n-j) = n-j$

So, we can write

Here, $T(j)$ or $T(n-j)$ will repeate twice, one from 1 to $n/2$ and $n/2$ to $n-1$, so we can write

$$T(n) = \frac{2}{n}\sum_{j=n/2}^{n-1} T(j) + O(n)$$

So, solving we get

$$T(n) = O(n)$$

Selection Sort in non-linear :

```
Selection (arr, n, k)
  {
  for (i=0; i<k; i++)
    {
    minValue = arr[i];
    minIndex = i;

  for (j= i+1 ; j<n; j++)
    {
    ~~if (A[j] < Arr[~~
      if (arr [j] < minvalue)
        {
          minIndex = j ;
          minValue = arr [j] ;
        }
      swap (arr[i], arr[minIndex]);
    }
  }
  return A[r[k];
}
```

Time Complexity

when, i=0, inner loop runs for n-1 time
when, i=1, inner loop runs for n-2 time

When, i=k-1, inner loop runs for n-(k-1+1) times

In Worst case if k=n,
then,

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