ADSA
Module 3

Divide and Conquer AND Greedy Algorithms

ITDO 5014 ADSA SYLLABUS

Module 3 : Divide and Conquer AND Greedy Algorithms : (9 Hrs)

1. Introduction to Divide and Conquer
   Analysis of :
2. Binary search
3. Merge sort and Quick sort
4. Finding the minimum and maximum algorithm

3. Introduction to Greedy Algorithms
4. Knapsack problem
5. Job sequencing with deadlines
6. Optimal storage on tape
7. Optimal merge pattern
8. Analysis of All these algorithms and problem solving.

Self-learning Topics:
Implementation of minimum and maximum algorithm, Knapsack problem, Job sequencing using deadlines.

Contents of Module 3 : Divide and Conquer :

1. Introduction to Divide and Conquer

Analysis of :

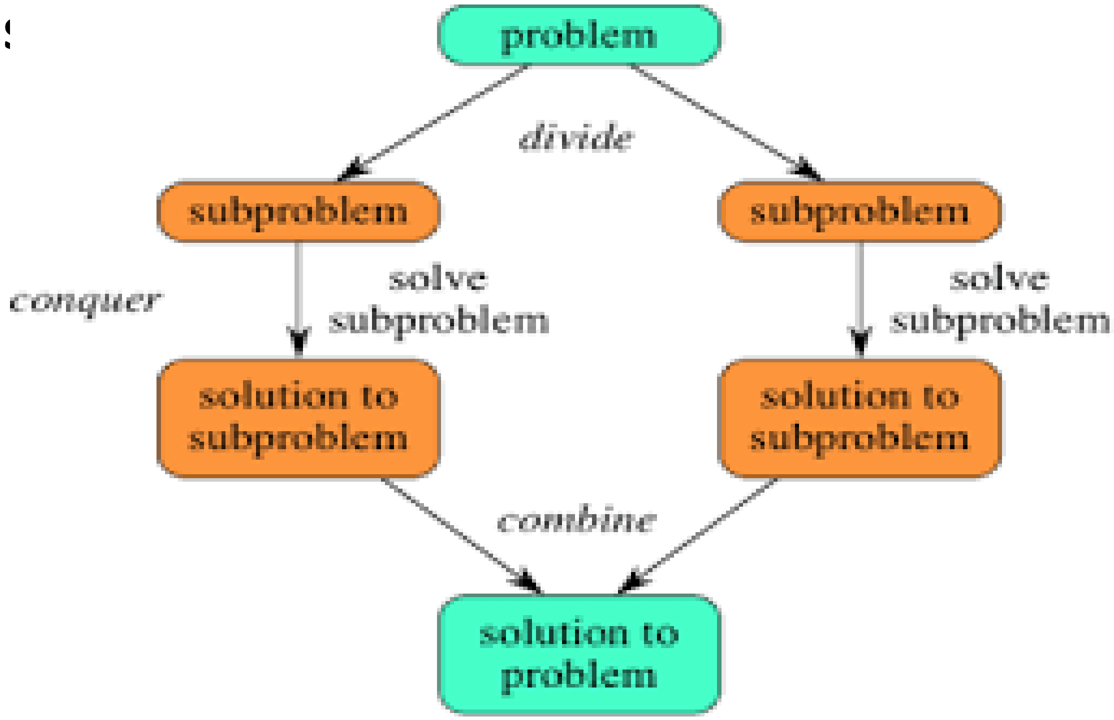
2. Binary search
3. Merge sort
4. Quick sort
5. Finding the minimum and maximum algorithm

Divide and Conquer :

Q. Explain Divide and Conquer strategy. List any four problems that can be solved using D&C. (10M)

Divide and Conquer is an algorithmic paradigm.
A typical Divide and Conquer algorithm solves a problem using following three steps.

1. Divide: **Break the given problem into smaller sub-problems of same or related type.**
2. Conquer: **Recursively solve these sub-problems**
3. Combine: **Combine the solution of the sub-problems**
problem.




```
Algo DC (P)
{        If P is too small then
                 return solution of P
         else
                 divide P into P1,P2,P3,........Pn
                 where n >=1
                 Apply DC to each sub-problem
                 Return Combine ( DC(P1), DC(P2),......DC(Pn) )
}
```

Following standard algorithms are Divide and Conquer algorithms .

1) Binary Search is a searching algorithm.

In each step, the algorithm compares the input element x with the value of the middle element in array. If the values match, return the index of middle. Otherwise, if x is less than the middle element, then the algorithm recurs for left side of middle element, else recurs for right side of middle element.

2) Quicksort is a sorting algorithm.

The algorithm picks a pivot element, rearranges the array elements in such a way that all elements smaller than the picked pivot element move to left side of pivot, and all greater elements move to right side. Finally, the algorithm recursively sorts the subarrays on left and right of pivot element.

3) Merge Sort is also a sorting algorithm.

The algorithm divides the array in two halves, recursively sorts them and finally merges the two sorted halves.

4) Strassen's Algorithm is an efficient algorithm to multiply two matrices.

A simple method to multiply two matrices need 3 nested loops and is O(n^3).

Strassen's algorithm multiplies two matrices in O(n^2.8974) time.

Advantage and disadvantage of Divide and Conquer Approach :

Advantage :

Divide and conquer approach supports parallelism as sub-problems are independent.

Hence, an algorithm, which is designed using this technique, can run on the multiprocessor system or in different machines simultaneously.

Disadvantage :

In this approach, most of the algorithms are designed using recursion, hence memory requirement is very high.

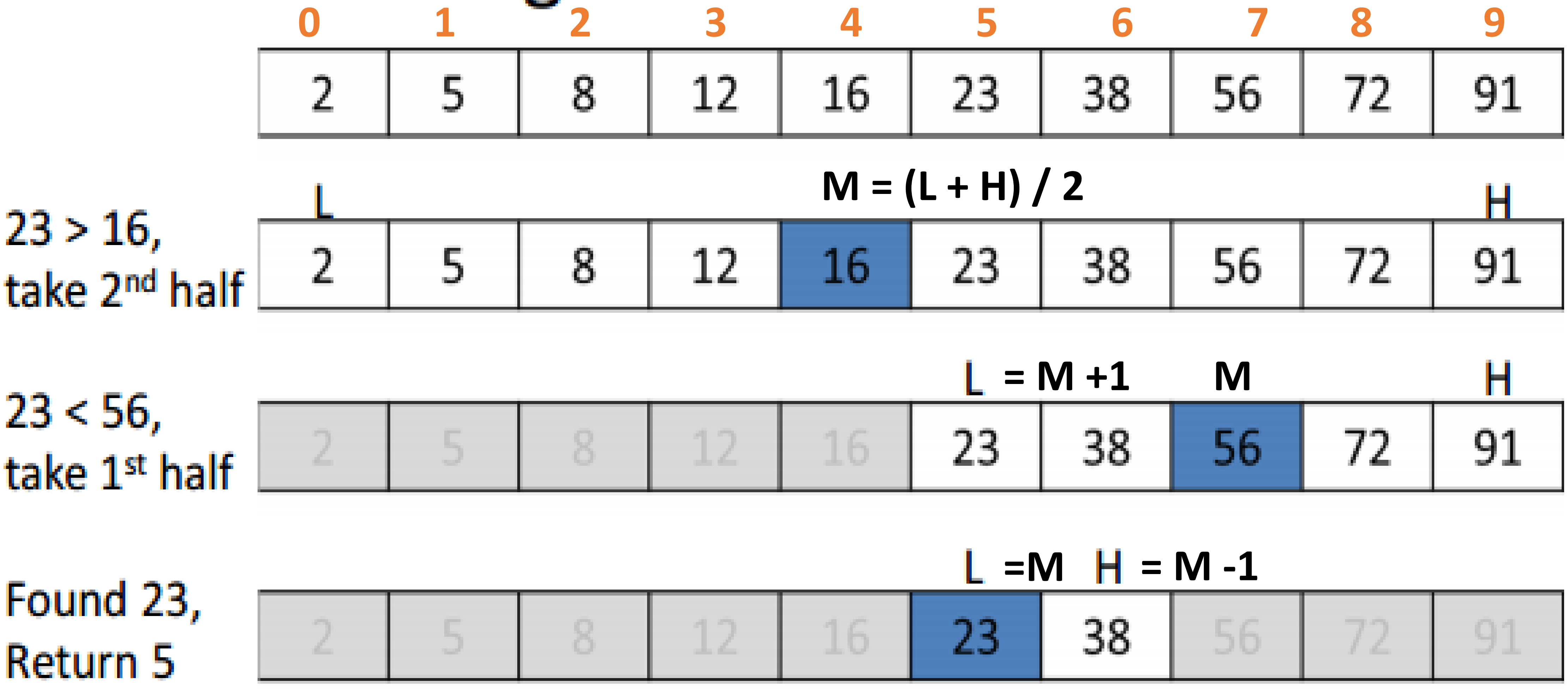
For recursive function stack is used, where function state needs to be stored.

Divide and Conquer : Recursive Binary Search :

Q. Describe binary search. Derive its time complexity (10M)

Q. Write the algorithm and derive the complexity of Binary search algorithm. (10M)

If searching for 23 in the 10-element array:

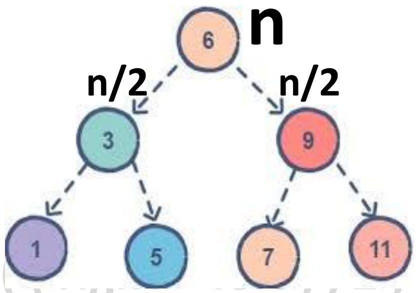


Divide and Conquer : recursive Binary Search :

If element x is present in given array arr[l..r] , then returns location of x in array arr[l..r] , otherwise -1.

l = 0 r = n-1

2	5	8	9	12	20	27	34




```
int   binarySearchRecursive  ( int arr[],   int l,  int r,  int x)
T(n)
{ int l = 0;    int  r  = n-1                       // r = array.length - 1
  if ( l < = r )
  {
      int mid = (l + r) / 2;                        // int mid = l + (r - l)/2;      // mid = left +
      if ( x  ==  arr[mid]  )                       // If the element is present at the middle
          return mid;
      else if ( x < arr[mid] )         // If element is smaller than mid, then it can only be present in
          return binarySearchRecursive (arr, l, mid-1, x); T(n/2)          //     LEFT SUB- ARRAY
      else if ( x > arr[mid] )          // Else the element can only be present in RIGHT SUB- ARRAY
          return binarySearchRecursive (arr, mid+1, r, x); T(n/2)
  }
   return -1;      // We reach here when element is not  present in array
}
 result = binarySearchRecursive (arr, 0, n-1, x);                  // Function call
```

Time Complexity:

The time complexity of Binary Search can be written as

T(n) = T(n/2) + c   // Search is reduced only to half of the array if (x < or > arr[mid]),
// so, not 2T(n/2)

The above recurrence can be solved either using Recurrence Tree method or Master method.

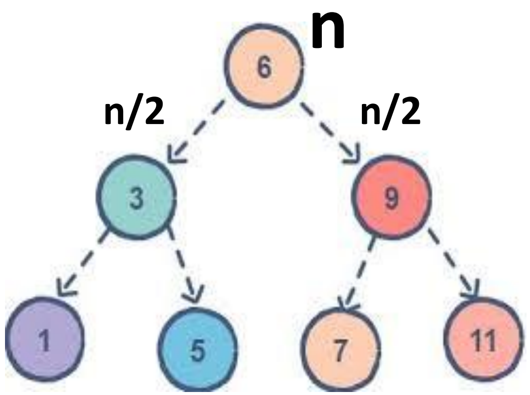
It falls in case II of Master Method and solution of the recurrence is Theta(Log n).

a = 1,   b = 2 ,   k = 0 ,   p = 0

1 = 2^0

1 = 1   case 2 p = 0 > -1 then $T(n) = \Theta\left(n^{\log_b a} \log^{p+1} n \right)$

$= \Theta(\log n)$



Auxiliary Space: O(1) in case of iterative implementation.
In case of recursive implementation, O(Log n) recursion call stack space.
Max no. of recursive calls = eq. to height of tree = h = O(h), h = log n, Space (n) = O(log n)

Finding the minimum and maximum from a list using D & C :
Example :

Q. Write an algorithm for finding maximum and minimum number from a given set.(10M)

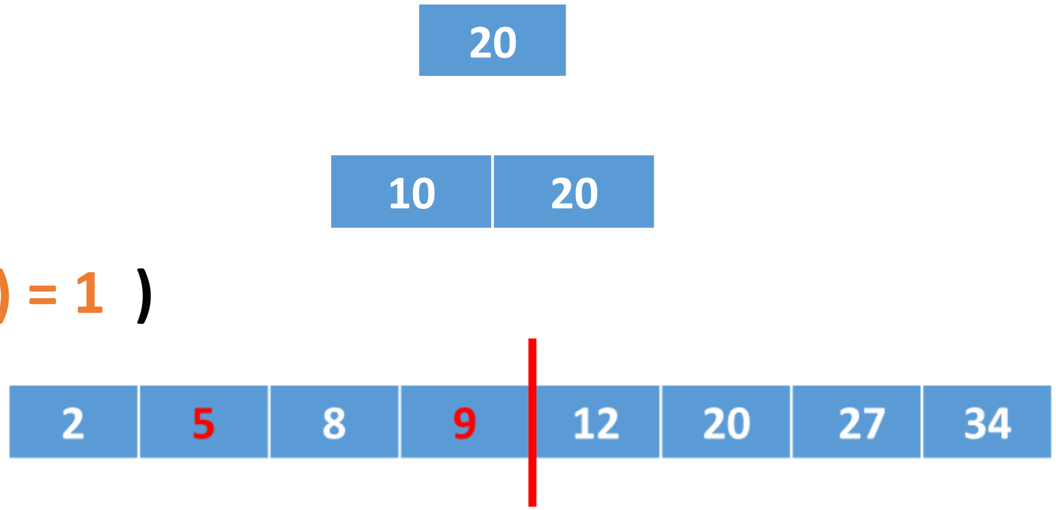
Three approaches to find the minimum and maximum from a list are :

1. **Sort the array.** **Min = 1st element , Max = last element**
2. **Assign min = 999999**
   max = 0
   Iteratively check min and max with each element of the array.

3. Divide and Conquer :

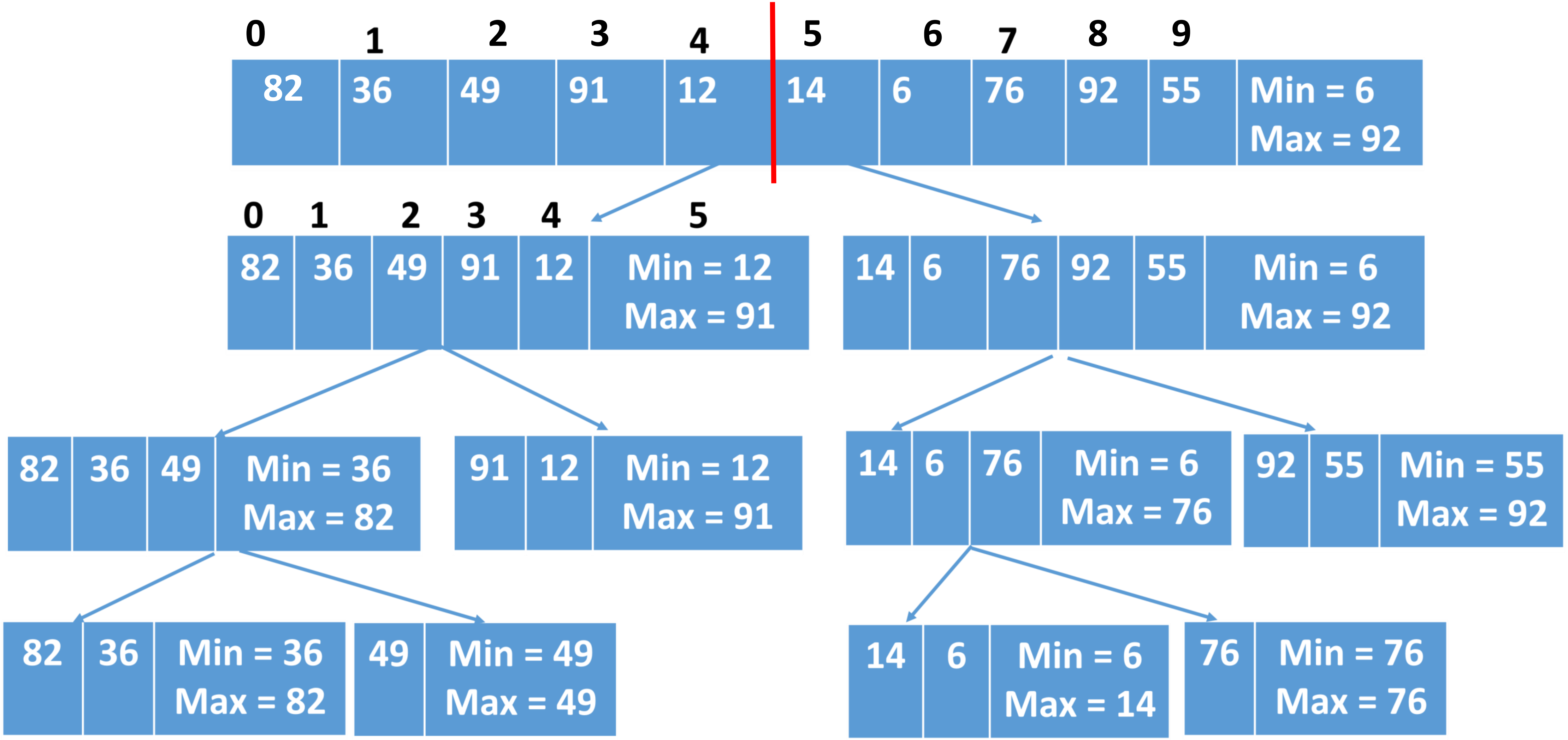
Gives better complexity compared to 1 & 2.

Divide the array into two parts and compare the maximums and minimums of the two parts to get the maximum and the minimum of the whole array.




```
Pair MaxMin(array, array_size) T(n)
  if array_size = 1
    return element as both max and min       ( T(1) = 0 )
  else if array_size = 2
    one comparison to determine max and min    ( T(2) = 1 )
    return that pair
  else    /* array_size  > 2 */
    recur for max and min of left half         ( T(n/2) )
    recur for max and min of right half        ( T(n/2) )
    one comparison determines true max of the two candidates   ( C = 1)
    one comparison determines true min of the two candidates   ( C = 1)
    return the pair of max and min
```

Finding the minimum and maximum from a list using D & C :



// structure pair is used to return two values : MIN and MAX from minMax()

```
struct pair
{
  int min;
  int max;
};
struct pair getMinMax(int arr[], int low, int high)
{
  struct pair minmax, mml, mmr;       int mid;

 /* Base case 1 : If there is only one element */
  if (low == high)
  {
    minmax.max = arr[low];
    minmax.min = arr[low];
    return minmax;
  }
```

L						L	H
2	5	8	9	12	20	27	34

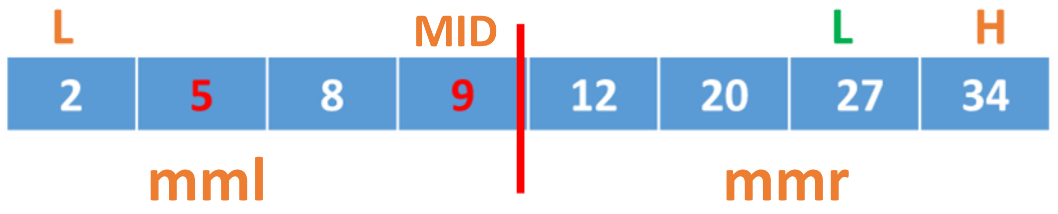
```
// Base case 2 : If there are two elements

if (high == low + 1)
{
  if (arr[low] > arr[high])
  {
    minmax.max = arr[low];
    minmax.min = arr[high];
  }
  else
  {
    minmax.max = arr[high];
    minmax.min = arr[low];
  }
  return minmax;
}
```

```
/* If there are more than 2 elements */
else
{ mid = (low + high)/2;
  mml = getMinMax(arr, low, mid);
  mmr = getMinMax(arr, mid+1, high);

  /* compare minimums of two parts*/
  if (mml.min < mmr.min)
    minmax.min = mml.min;
  else
    minmax.min = mmr.min;

  /* compare maximums of two parts*/
  if (mml.max > mmr.max)
    minmax.max = mml.max;
  else
    minmax.max = mmr.max;
    return minmax;
}
```

```
struct pair
{
  int min;
  int max;
};

 struct pair minmax, mml, mmr;
```

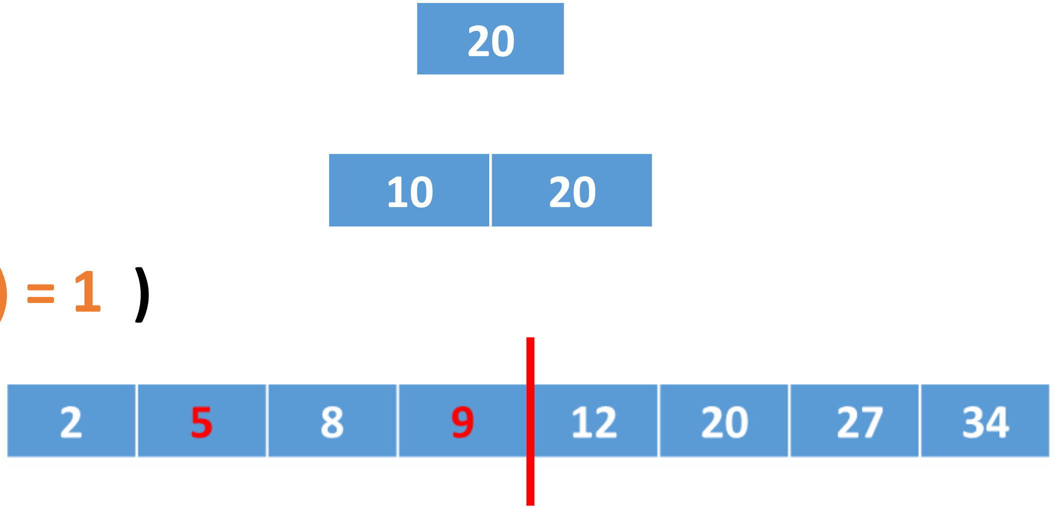
CALL :

```
struct pair minmax = getMinMax(arr, 0, arr_size-1);
```



Divide the array into two parts and compare the maximums and minimums of the two parts to get the maximum and the minimum of the whole array.

```
Pair MaxMin(array, array_size) T(n)
  if array_size = 1
    return element as both max and min        ( T(1) = 0 )
   else if array_size = 2
    one comparison to determine max and min   ( T(2) = 1 )
    return that pair
  else    /* array_size > 2 */
    recur for max and min of left half        ( T(n/2) )
    recur for max and min of right half       ( T(n/2) )
    one comparison determines true max of the two candidates   ( C = 1)
    one comparison determines true min of the two candidates   ( C = 1)
    return the pair of max and min
```



Time Complexity of Divide and Conquer MIN-MAX Algo : O(n)

Total number of comparisons: let number of comparisons be T(n).

T(n) can be written as follows:

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

$T(2) = 1$

$T(1) = 0$

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

After solving above recursion using master's theorem, we get

$a = 2$, $b = 2$, $k = 0$, $p = 0$

a b^k

2 > 2^0 Condition 1 => $T(n) = \Theta (n^{\log_b a})$

$= \Theta (n^{\log_2 2})$

$= \Theta (n)$

Merge Sort :

Q1. Sort the list of elements in ascending order using merge-sort technique. Give output for each pass.

i. **90,20,80,89,70,65,85,74**

ii. **100,20,38,14,48,07,17,57,93,35** **(Any one) (10M)**

Merge Sort :

Like QuickSort, Merge Sort is a Divide and Conquer algorithm.

The merge sort algo is composed of two independent parts.

Merge : merge(arr, l, m, r) and

mergeSort(arr, l, r)

MergeSort : -divides input array in two halves,
calls itself for the two halves and
-then calls merge function
to merge the two sorted halves.
-MergeSort uses merge
to sort a list (array) of elements.

Merge : function is used for merging the two halves.
The merge() function merges two sorted arrays to give another sorted array.

The merge(arr, l, m, r) process assumes that arr[l..m] and arr[m+1..r] (eventually single element arrays) are sorted and merges the two sorted sub-arrays into another sorted array.

```
void mergeSort(int arr[], int l, int r) T(n)
{
  if (l < r)
  {
    int m = ( l + r ) / 2;

    // Sort first and second halves
    mergeSort(arr, l, m);          T(n/2)
    mergeSort(arr, m+1, r);        T(n/2)

    merge(arr, l, m, r);           Theta(n)
  }
}
```

Merge Sort Algorithm : The array is recursively divided in two halves till the size becomes 1. Once the size becomes 1, the merge processes comes into action and starts merging arrays back till the complete array is merged.

MergeSort(arr[], l, r)

If l < r // divide the array till u get single element arrays i.e.
// when l = r = single ele arr=> stop split (recursion)

1. Find the middle point to divide array into two halves:
   middle m = (l+r)/2
2. Call mergeSort for first half:
   Call mergeSort(arr, l, m)
3. Call mergeSort for second half:
   Call mergeSort(arr, m+1, r)
4. Merge the two halves sorted in step 2 and 3:
   Call merge(arr, l, m, r)

```
void mergeSort(int arr[], int l, int r) T(n)
{
    if (l < r)
    {
        int m = ( l + r ) / 2;

        // Sort first and second halves
        mergeSort(arr, l, m);           T(n/2)
        mergeSort(arr, m+1, r);         T(n/2)

        merge(arr, l, m, r);            Theta(n)
    }
}
```

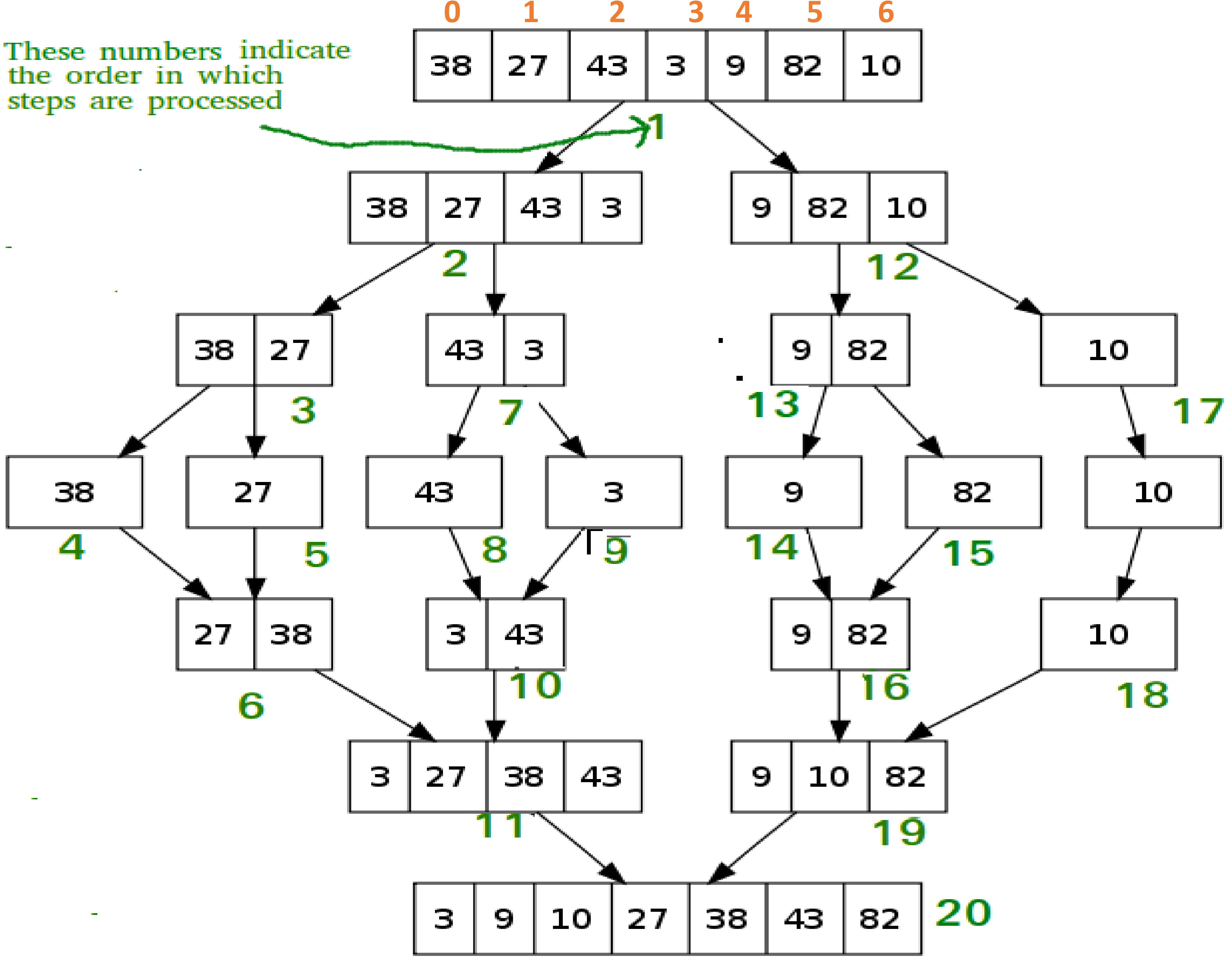
The array is recursively Divided in two halves till the size becomes 1.

Once the size becomes 1,

the merge processes comes into action and starts merging arrays back

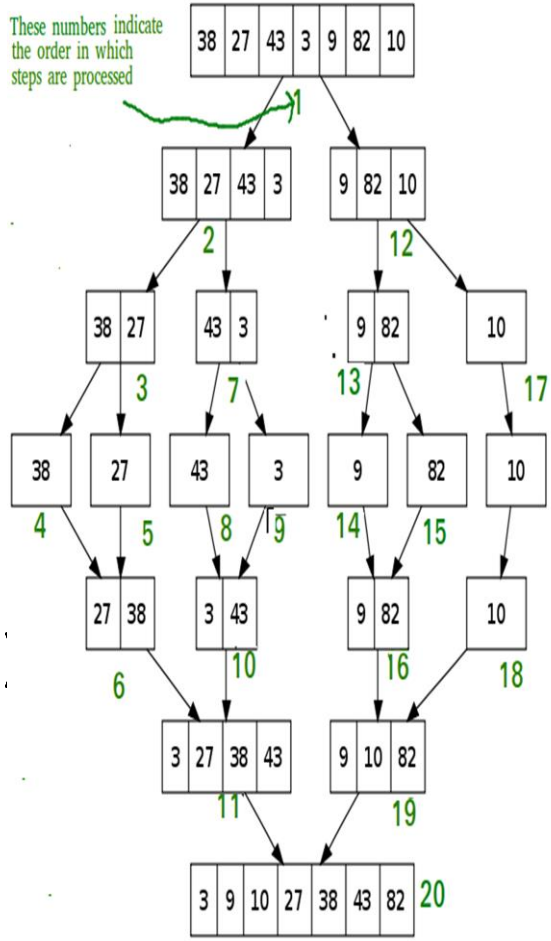
till the complete array is merged.

```
void mergeSort(int arr[], int l, int r)
{
   if (l < r)
   {
      int m = ( l + r ) / 2;

      // Sort first and second halves
      mergeSort(arr, l, m);
      mergeSort(arr, m+1, r);          T(n/2)

      merge(arr, l, m, r);
   }
}
```

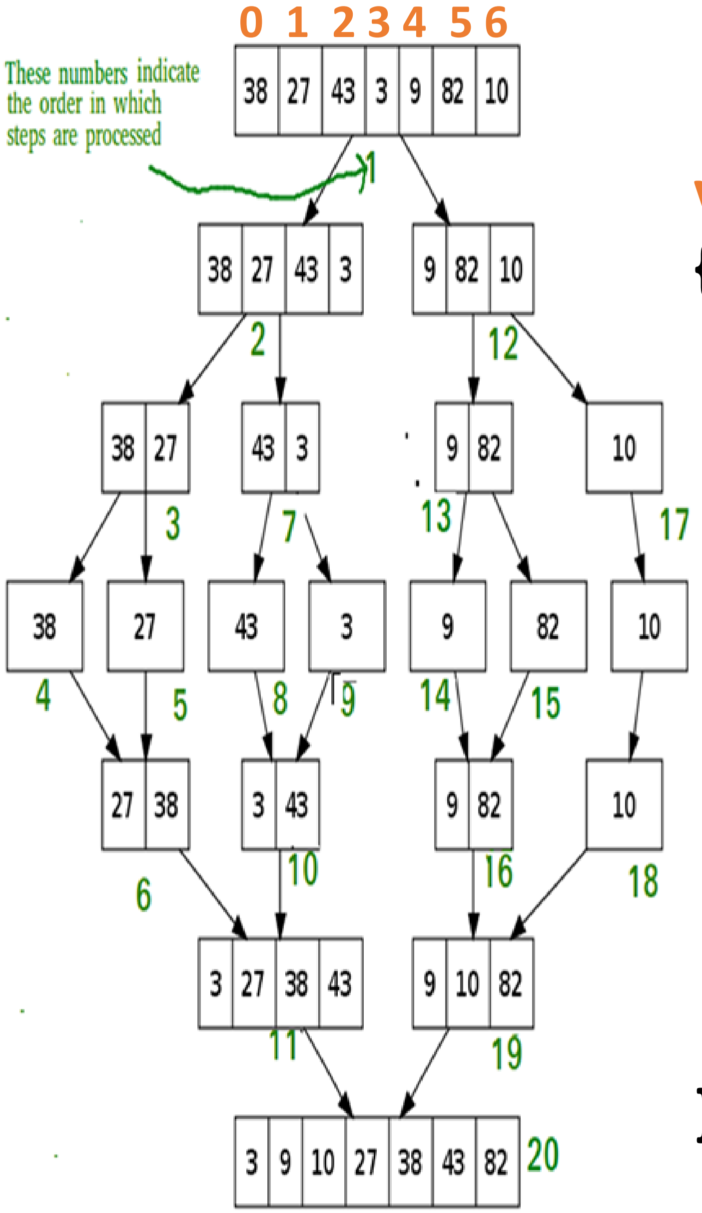


```
// Merges two subarrays of arr[].
// First subarray is arr[l..m]
// Second subarray is arr[m+1..r]
void merge(int arr[], int l, int m, int r)
{
    int i, j, k;
    int n1 = m - l + 1;
    int n2 =  r - m;

    int L[n1], R[n2];  /* create temp arrays */

    /* Copy data to temp arrays L[] and R[] */
    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1+ j];

/* Merge the temp arrays back into arr[l..r]*/
    i = 0;      // Initial index of first subarray
    j = 0;      // Initial index of second subarray
    k = l;      // Initial index of merged subarray
    while ( i < n1  &&  j < n2 )
    {   if (L[i] < R[j])
        {           arr[k] = L[i];
                    i++;
        }
        else  if ( R[j] < L[i] )
        {           arr[k] = R[j];
                    j++;
        }
        else   //    if ( R[j] == L[i] )
        {           arr[k] = L[i];
                    i++;   j++;
        }
        k++;
    }
```

0 1 2 3 4 5 6



```
/* Copy the remaining elements of L[], if there
   are any */
while (i < n1)
{
    arr[k] = L[i];
    i++;
    k++;
}

/* Copy the remaining
   elements of R[],
   if there are any */
while (j < n2)
{
    arr[k] = R[j];
    j++;
    k++;
}
}
```




```
/* l is for left index and r is right index of the
   sub-array of arr to be sorted */

void mergeSort(int arr[], int l, int r)      T(n)
{
    if (l < r)
    {
        int m = ( l + r ) / 2;

        // Sort first and second halves
        mergeSort(arr, l, m);                T(n/2)
        mergeSort(arr, m+1, r);              T(n/2)

        merge(arr, l, m, r);                 Theta(n)
    }
}
```

```
/* UTILITY FUNCTIONS */
/* Function to print an array */

void printArray(int A[], int size)
{
   int i;
   for (i=0; i < size; i++)
      printf("%d ", A[i]);
   printf("\n");
}
```

```
/* Driver program to test above functions */
int main()
{
   int arr[] = {12, 11, 13, 5, 6, 7};
   int arr_size = sizeof(arr)/sizeof(arr[0]);

   printf("Given array is \n");
   printArray(arr, arr_size);

   mergeSort(arr, 0, arr_size - 1);

   printf("\nSorted array is \n");
   printArray(arr, arr_size);
   return 0;
}
```

Time Complexity:

Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.

$T(n) = 2T(n/2) + \text{Theta}(n)$: The above recurrence can be solved either using Recurrence Tree method or Master method.

$= 2T(n/2) + n$ a = 2 , b = 2 , k = 1, p = 0 ; a : b^ k ; 2 = 2 ^ 1 => 2 ; p = 0 > -1 => 2a

2. If $a = b^k$ then a. If $p > -1$ then $T(n) = \Theta (n^{\log_b a} \log^{p+1} n)$

i.e It falls in case II of Master Method.

$= \Theta$ (n Logn).

Time complexity of Merge Sort is = Θ(n Log n) in all 3 cases (worst, average and best) as :

merge sort always divides the array in two halves and take linear time to merge two halves irrespective of whether the i/p array is already sorted or unsorted .

Space complexity of Merge Sort :

Auxiliary Space:

Space for merge function = n + C

(n = n1 +n2) and (c = 5 for l, j, k, n1, n2)

= O(n)

Stack space for recursion of mergeSort function= O(log n)

Stack space for recursion = height of recursion tree

= O (log n) , n = no of elements in array.

So, total auxiliary space = n + log n = O(n)

Parameter space : for merge function = n + 3 (arr[] = n l,r,m = 3)

for mergeSort function = n + 2 (arr[] = n l,r = 2)

So, total Parameter space = n+3 + n+2

= 2n + 5 = O(n)

Total space = Parameter space + Auxiliary Space

= O(n) + O(n)

= O(n)

```
void mergeSort(int arr[], int l, int r) T(n)
{
    if (l < r)
    {
        int m = ( l + r ) / 2;
        // Sort first and second halves
        mergeSort(arr, l, m);          T(n/2)
        mergeSort(arr, m+1, r);   T(n/2)

        merge(arr, l, m, r);      Theta(n)
    }
}
void merge(int arr[], int l, int m, int r)
{
    int i, j, k;
    int n1 = m - l + 1;
    int n2 =  r - m;
 // create temp arrays
    int L[n1], R[n2];
// Copy data to temp arrays L[] and R[]
    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
            ------
            ------
```

Applications of Merge Sort :

1. **Merge Sort is useful for sorting linked lists in O(n Log n) time.**

2. **Used in External Sorting**

3. **Inversion Count Problem :**

   Inversion Count for an array indicates – how far (or close) the array is from being sorted.

   If array is already sorted then inversion count is 0.
   If array is sorted in reverse order that inversion count is the maximum.

   Two elements a[i] and a[j] form an inversion if $a[i] > a[j]$ and $i < j$.

   Example:
   The sequence 2, 4, 1, 3, 5 has three inversions (2, 1), (4, 1), (4, 3).

Merge sort = : space required = height of tree = h = log n

SC (n) = Theta (log n)

Merge Sort :

auxiliary space = n + log n = O(n)

Total space = Parameter space + Auxiliary Space

= O(n) + O(n)

= O(n)

Complexity of merge sort is same as that of the average case complexity of quick sort. However, the memory requirement of this merge sort algorithm is too large and hence considered generally as less efficient

Quick sort : [Cormen]

Q1. Sort the following using quick sort also derive its time complexity show all passes of execution. (10M)

i. 50,31,71,38,77,81,12,33 ii. 65,70,75,80,85,60,55,50,45.

Q2. Explain Quick sort algorithm. Dive its time complexity with suitable example.

Quick sort : [Cormen]

Like Merge Sort, QuickSort is a Divide and Conquer algorithm.

It picks an element as pivot and partitions the given array around the picked pivot.

There are many different versions of quickSort that pick pivot in different ways.

- Always pick first element as pivot.
- Always pick last element as pivot (Cormen : implemented below)
- Pick a random element as pivot.
- Pick median as pivot.

The key process in quickSort is partition().

Function of partition : Given an array and an element x of array as pivot,

- put x at its correct position in sorted array and
- put all elements smaller than x, before x, and
- put all elements greater than x, after x. All this should be done in linear time.

Pseudo Code for recursive QuickSort function :

```
/* low  --> Starting index,
   high  --> Ending index */

quickSort(arr[], low, high)
{
   if (low < high)
   {
// pi is partitioning index, arr[pi] is now at right place

      pi = partition(arr, low, high);

      quickSort(arr, low, pi - 1);      // Before pi
      quickSort(arr, pi + 1, high);     // After pi
   }
}
```

Low							High
9	6	5	0	8	2	4	7

6	5	0	2	4	9	8	7

Low							High
6	5	0	2	4	7	8	9

Pi

```
partition (arr[], low, high)
{// pivot Element to be placed at its right position
   pivot = arr[high];
   i = (low - 1)  // Index of smaller element
   for (j = low; j <= high- 1; j++)
   {
      if (arr[j] <= pivot)
      {   i++;       swap arr[i] and arr[j] ;  }
   }
   swap arr[i + 1] and arr[high])
   return (i + 1)
}
```

i	J	A[i]	A[j]	<	pivot	Action	1 = low	2	3	4	5	6	7	8= high = pivot
0	1	--	9	≮	7	J++	9	6	5	0	8	2	4	7
							9	6	5	0	8	2	4	7
0	2	--	6	<	7	i++ => i=1 , swap a[1] & a[2]	9	6	5	0	8	2	4	7
							6	9	5	0	8	2	4	7
1	3	6	5	<	7	i++ => i=2 , swap a[2] & a[3]	6	9	5	0	8	2	4	7
2							6	5	9	0	8	2	4	7
2	4	9	0	<	7	i++ => i=3 , swap a[3] & a[4]	6	5	9	0	8	2	4	7
3							6	5	0	9	8	2	4	7
3	5	0	8	≮	7	J++	6	5	0	9	8	2	4	7

```
partition (arr[], low, high)
{// pivot Element to be placed at its right position
   pivot = arr[high];
   i = (low - 1)  // Index of smaller element
   for (j = low; j <= high- 1; j++)
   {
      if (arr[j] <= pivot)
      {
         i++;
         swap arr[i] and arr[j] ;
      }
   }
   swap arr[i + 1] and arr[high])
   return (i + 1)
}
```

i	J	A[i]	A[j]	<	pivot	Action	1 = low	2	3	4	5	6	7	8= high = pivot
3	5	0	8	<	7	J++	6	5	0	9	8	2	4	7
3	6	0	2	<	7	i++ => i=4 , swap a[4] & a[6]	6	5	0	9	8	2	4	7
4							6	5	0	2	8	9	4	7
4	7	2	4	<	7	i++ => i=5 , swap a[5] & a[7]	6	5	0	2	8	9	4	7
							6	5	0	2	4	9	8	7
5	8					Swap a[6] & a[high = 8]	6	5	0	2	4	9	8	7
							6	5	0	2	4	7	8	9
							6= low	5	0	2	4= high	7	8 = low	9= high

partition (arr[], low, high)

```
{// pivot Element to be placed at its right position
   pivot = arr[high];
   i = (low - 1)  // Index of smaller element
   for (j = low; j <= high- 1; j++)
   {
      if (arr[j] <= pivot)
      {
         i++;
         swap arr[i] and arr[j] ;
      }
   }
   swap arr[i + 1] and arr[high])
   return (i + 1)
}
```

Pass1

Pi =i+1

= 6

Partition Algorithm :

The logic is simple, we start from the

leftmost element and keep track of index of smaller (or equal to) elements as i.

This function takes last element as pivot,

places the pivot element at its correct position in sorted array, and

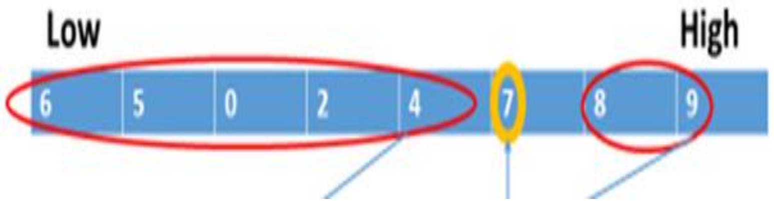
places all elements smaller than pivot to left of pivot and

all elements greater than pivot element to right of pivot .

```
partition (arr[], low, high)
{
   // pivot Element to be placed at its right position

   pivot = arr[high];
   i = (low - 1)                 // Index of smaller element

   for (j = low; j <= high- 1; j++)
   {
      // If current element is smaller than or
      // equal to pivot
      if (arr[j] <= pivot)
      {
         i++;   // increment index of smaller element
         swap arr[i] and arr[j]
      }
   }
// Place pivot ele : arr[high] in its right place arr[i+1]
   swap arr[i + 1] and arr[high])
   return (i + 1)
}
```

Psudo Code for D & C i.e. Recursive Quicksort :

```
/* low  --> Starting index,
   high  --> Ending index */

quickSort(arr[], low, high)                    T(n)
{
   if (low < high)
   {
      //  pi is partitioning index,
      //  arr[pi] is now at right place
      pi = partition(arr, low, high);          Θ(n)

      quickSort(arr, low, pi - 1);  // Before pi   T(k)
      quickSort(arr, pi + 1, high);  // After pi   T(n-k-1)
   }
}
```




```
partition (arr[], low, high)
{   // pivot Element to be placed at its right position

   pivot = arr[high];
   i = (low - 1)                  // Index of smaller element

   for (j = low; j <= high- 1; j++)
   {
      // If current element is smaller than or
      // equal to pivot
      if (arr[j] <= pivot)
      {
         i++;    // increment index of smaller element
         swap arr[i] and arr[j]
      }
   }
// Place pivot ele : arr[high] in its right place arr[i+1]
   swap arr[i + 1] and arr[high])
   return (i + 1)
}
```

An Example:

p
pivot=11
r
17 9 22 31 7 12 10 21 13 29 18 20 11
9 17 22 31 7 12 10 21 13 29 18 20 11
9 7 22 31 17 12 10 21 13 29 18 20 11
9 7 10 31 17 12 22 21 13 29 18 20 11
9 7 10 11 17 12 22 21 13 29 18 20 31
q

Analysis of QuickSort :

Time taken by QuickSort in general can be written as following.

$T(n) = T(k) + T(n-k-1) + \Theta(n)$ // $T(n-k-1)$: -1 because 1 element is pivot

The first two terms are for two recursive calls, the last term: $\Theta(n)$ is for the partition process.

k is the number of elements which are smaller than pivot.

The time taken by Quick Sort depends upon the input array and partition strategy.

Worst Case: The worst case occurs when the partition process always picks greatest or smallest element as pivot.

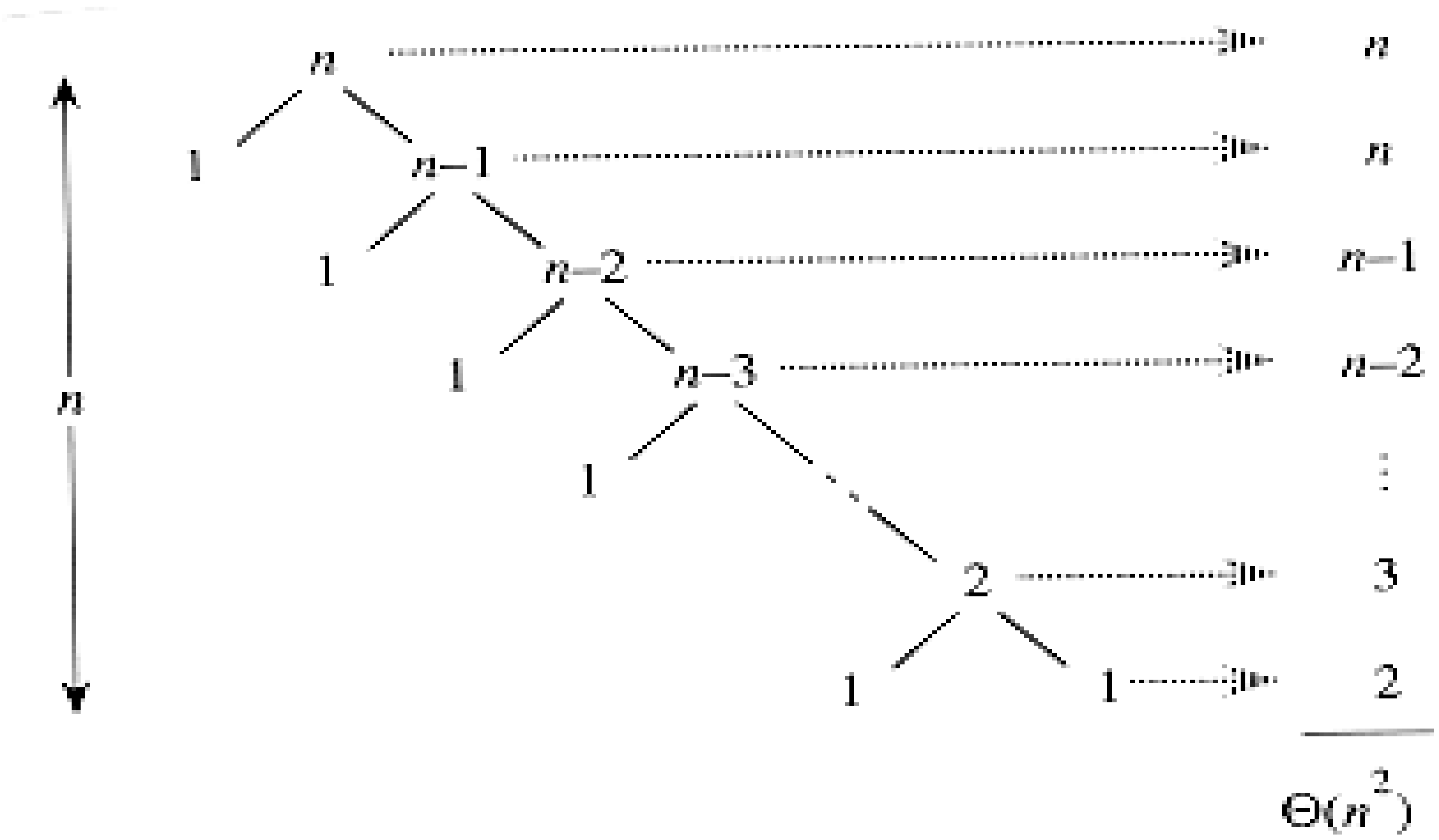
If we consider above partition strategy where last element is always picked as pivot, the worst case would occur when the array is already sorted in increasing or decreasing order.

Following is recurrence for worst case.

$T(n) = T(0) + T(n-1) + \Theta(n)$ // $T(n-1)$ because 1 ele is pivot

$T(n) = T(n-1) + \Theta(n)$ The solution of above recurrence using substitution is = $\Theta(n^2)$.

Worst case :



Best case => pivot is middle element =>

It partitions array in two almost equal halves of size (n/2) each.

So, $T(n) = 2T(n/2) + n$ as in case of Merge Sort ………..
The above recurrence can be solved either using Recurrence Tree method or Master method.

$= 2T(n/2) + n$ a = 2 , b = 2 , k = 1, p = 0 ; a : b^ k ; 2 = 2 ^ 1 => 2 ; p = 0 > -1 => 2a

2. If $a = b^k$ then a. If $p > -1$ then $T(n) = \Theta (n^{\log_b a} \log^{p+1} n)$

i.e It falls in case II of Master Method.

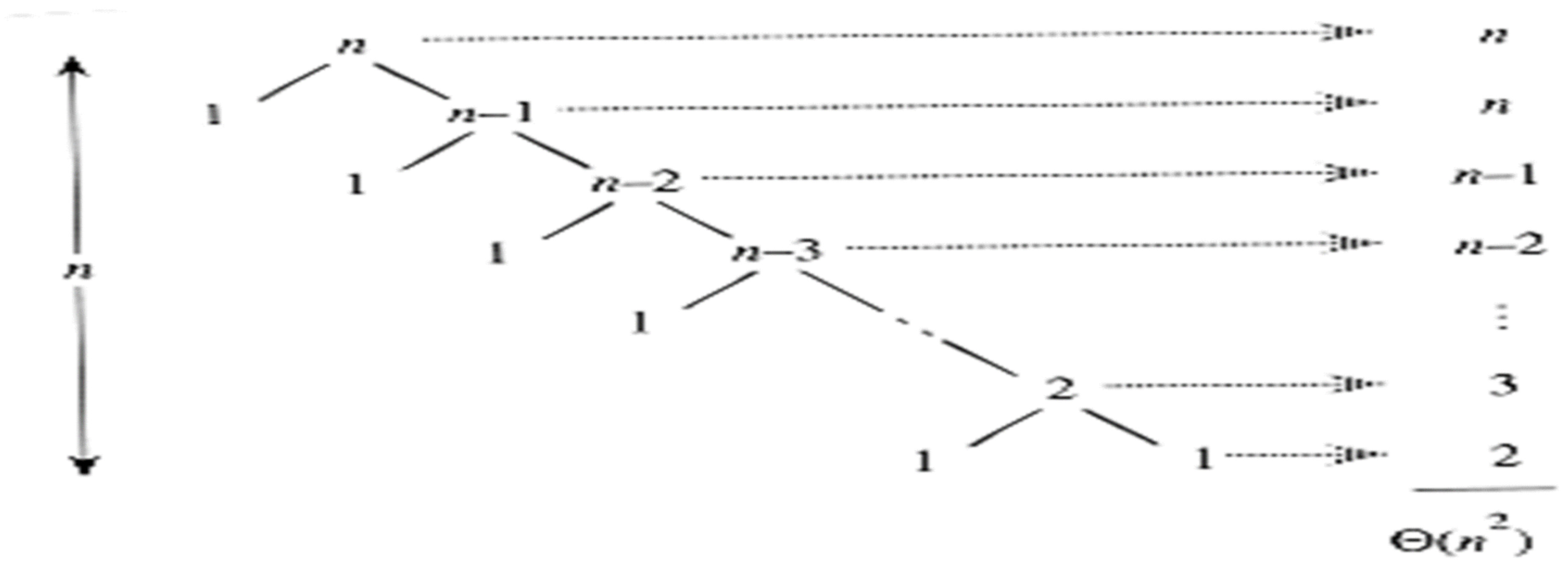
$= \Omega(n \log n)$.

Best Case Time complexity of Quick Sort is $= \Omega(n \log n)$
Worst Case Time complexity of Quick Sort is $= \Theta(n^2)$.

Space Complexity of Quick sort :

In best case , as it partitions array in two equal halves of size (n/2) each. => balanced tree of height = log n = > Space = O(log n) in best case.

In worst case , every time it may partition the array in two partitions of size 1 and (n-2) elements resp . => height of tree (skewed on one side) = n = > Space = O(n) in worst case.



BASIS FOR COMPARISON	QUICK SORT	MERGE SORT
Partitioning of the elements in the array	The splitting of a list of elements is not necessarily divided into half.	Array is always divided into half (n/2).
Worst case complexity	$O(n^2)$	O(n log n) (worst, average and best)
Works well on	Smaller array	Operates fine in any type of array.
Speed	Faster than other sorting algorithms for small data set.	Consistent speed in all type of data sets.
Additional storage space requirement	Less	More
Efficiency	Inefficient for larger arrays.	More efficient.
Sorting method	Internal	External

Divide and Conquer Algorithm	Time Complexity	Space Complexity
Binary Search	Θ (log n)	O (log n)
MIN-MAX Algorithm	Θ (n)	
Merge Sort	worst, average and best : Θ(n Log n)	O(n)
Quick Sort	Best Case : Ω(n Log n) Worst Case : O(n^2)	Best Case : Ω (log n) Worst Case : O(n)
Matrix Multiplication (Naïve)	O(N^3)	
Matrix Multiplication D & C	O(N^3)	
Strassen's Matrix Multiplication D & C	Θ($n^{2.8074}$)	