SUBHAPREET PATRO
2211CS010547
GROUP-7A

DAA Assignment - 1

1) A) Write the characteristics of an algorithm.

Ans: An algorithm is a finite sequence of instructions, each of which has a clear meaning and can be performed with a finite amount of effort in a finite length of time to perform a particular task.

• Characteristics of algorithm :-

(i) Input :- Zero or more quantities are externally supplied.

(ii) Output :- Atleast one quantity is produced.

(iii) Definiteness :- Each instruction must be unambiguous.

(iv) Finiteness :- The algorithm will terminate after a finite number of steps.

(v) Effectiveness :- Every instruction must be sufficiently basic.

B) Describe the pseudo code conventions.

Ans. Pseudo code is an artificial and informal language that helps programmers to develop algorithms. It is a 'text-based' detail design tool.

Pseudo code consists of,

(i) single line comments start with //

(ii) Multi-line comments occur between /* and */.

(iii) Blocks are represented using brackets. Blocks can be used to represent compound statements or the procedures.

```
{
  statements
}
```

(iv) statements are delimited by semicolon.

(v) Assignment statements indicate that the result of evaluation of the expression will be stored in the variable.

< variable > := < expression >

(vi) The Boolean expression 'x > y' returns true if x is greater than y, else returns false.

(vii) if <condition> then <statement>

(viii) This condition is an enhancement of the above 'if' statement. It can also handle the case where the condition isn't satisfied.

if <condition> then <statement1> else <statement 2>

(ix) switch case (C or C++)

```
case {
:<condition 1>: <statement 1>
...
...
:<condition n>: <statement n>
: default: <statement n+1>
}
```

(x) while loop

```
while <condition> {
statements;
}
```

(xi) do-while loop

```
repeat
statements;
until <condition>
```

(xii) for loop

```
for variable := value 1 to value 2 {
statements;
}
```

(xiii) input instruction

```
Read
```

(xiv) output instruction

Print

(xv) The name of the algorithm <name> and the arguments are stored in the <parameter list>

Algorithm <name> (<parameter list>)

2) Illustrate space complexity and Time complexity with suitable examples.

Ans: • Time complexity:-

The time complexity of an algorithm is the total amount of ~~the~~ time required by an algorithm to complete its execution.

To calculate the time complexity of an algorithm;-

→ It requires 1 unit of time for Arithmetic and logical operations

→ It requires 1 unit of time for ~~Assign~~ Assignment and Return value.

→ It requires 1 unit of Time for Read and write operations.

Ex:-

```
int sum (int a, int b)
{
a = 5;
b = 6;
c = a + b;
}
```

→ It requires 1 unit of time for a=5, 1 unit of time for b = 6 and 1 unit of time to calculate c = a + b.

→ Totally it takes 3 units of time to complete its execution.

so, T(n) = 3

• Space Complexity:-

Total amount of computer memory required by an algorithm to complete its execution is called as space complexity of that algorithm.

→ Instruction space:- It is the amount of memory used to store compiled version of instructions

→ Environmental stack:- It is the amount of memory used to store information of partially executed functions at the time of function call.

→ Data space:- It is the amount of memory used to store all the variables and constants.

To calculate the space complexity, ~~of~~ we must know the memory required to store different data type values (according to the compiler).

For example, the C programming language compiler requires the following...

→ 2 bytes to store Integer value

→ 4 bytes to store Floating Point value.

→ 1 byte to store Character value.

→ 6 (or) 8 bytes to store double value.

Ex:-

```
int sum(int a, int b)
{
a=5;
b = 6;
c = a+b;
}
```

→ It requires 1 word space for a variable a, 1 word space for a variable b and 1 word space for variable c.

→ Totally it takes 3 word spaces to complete its execution

so, $S(n) = 3$

3) Summarize the Asymptotic Notations with examples.

Ans. → Asymptotic notations are the mathematical notations used to describe the running time of an algorithm when the input tends towards a particular value or a limiting value.

→ There are mainly three asymptotic notations :-

1. Big - O notation
2. Omega notation
3. Theta notation.

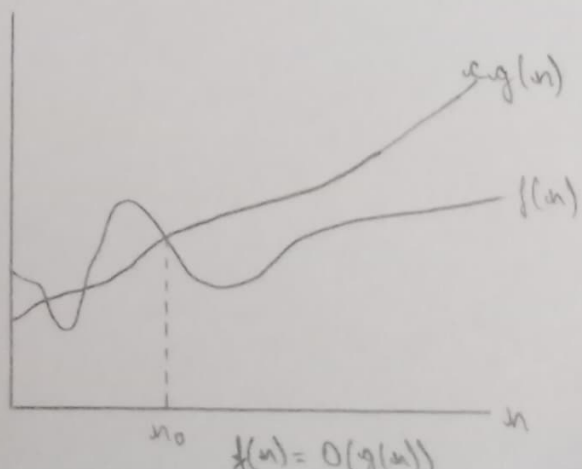
(1) Big - O notation (O - notation) :-

→ Big - O notation represents the upper bound of the running time of an algorithm. Thus, it gives the worst - case complexity of an algorithm.

→ For any value of n, the maximum time required by the algorithm is given by Big - O $O(g(n))$.

→ A function $f(n)$ belongs to ~~to~~ the set $O(g(n))$ if there exists a positive constant c and n_0 such that

$$f(n) \leq c^* g(n) \text{ for all } n \geq n_0$$



Ex:- $f(n) = 3n+2$ $\quad f(n) = O(g(n))$

$$f(n) \leq c \cdot g(n)$$

$$\Rightarrow 3n+2 \leq c \cdot g(n)$$

$$\Rightarrow 3n+2 \leq 4 \cdot n$$

$n=1 \Rightarrow 5 \leq 4$ (False)

$n=2 \Rightarrow 8 \leq 8$ (True)

$n=3 \Rightarrow 11 \leq 12$ (True)

$n=4 \Rightarrow 14 \leq 16$ (True)

$\boxed{n \geq 2} \Rightarrow \boxed{n \geq n_0}$

$\Rightarrow \boxed{n_0 = 2}$

$\Rightarrow \boxed{c = 4}$

$\Rightarrow \boxed{g(n) = n}$

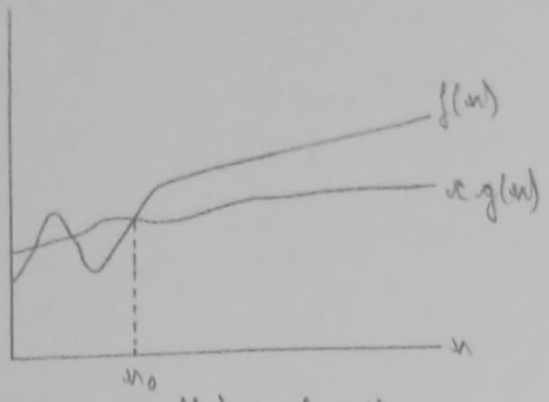
(ii) Omega Notation (Ω - notation):-

→ Omega notation represents the lower bound of the running time of an algorithm.

→ For any value of n, the minimum time required by an algorithm is given by Omega $\Omega(g(n))$.

→ A function $f(n)$ belongs to the set $\Omega(g(n))$ if there exists a positive constant c and n_0 such that,

$$\boxed{f(n) \geq c * g(n) \text{ for all } n,\ n \geq n_0}$$



$f(n) = \Omega(g(n))$

Ex:- $f(n) = 3n+2$ $\qquad f(n) = \Omega(g(n))$

$f(n) \geq c.g(n)$

$3n+2 \geq 2.n$

$n=1 \Rightarrow 5 \geq 2$ (True)

$n=2 \Rightarrow 8 \geq 4$ (True)

$n=3 \Rightarrow 11 \geq 6$ (True)

$\Rightarrow \boxed{n \geq 1}$

$\Rightarrow \boxed{n_0 = 1}$

$\Rightarrow \boxed{c = 2}$

$\Rightarrow \boxed{g(n) = n}$

(iii) Theta Notation (θ- notation):-

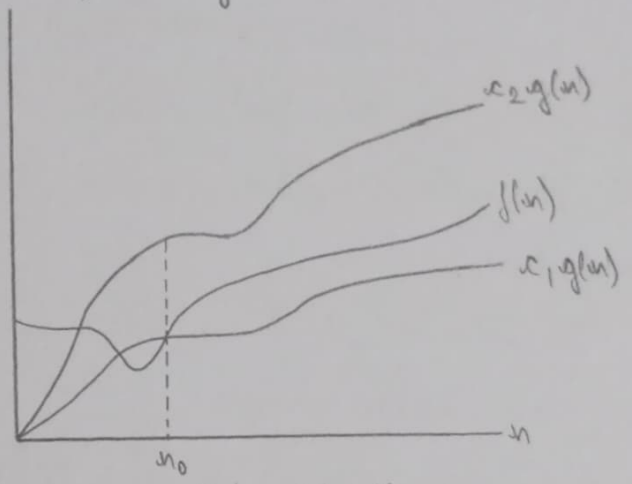
→ Theta notation encloses the function from above and below. Since it represents the upper and the lower bound of the running time of an algorithm.

→ For any given value of n, the average time required by the algorithm is given by Theta $\theta(g(n))$

→ A function $f(n)$ belongs to the set $\theta(g(n))$ if there exists a positive constant c_1, c_2 and n_0 such that,

$$c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0$$

→ If a function $f(n)$ lies anywhere in between $c_1 g(n)$ and $c_2 g(n)$ for all $n \geq n_0$, then $f(n)$ is said to be tight bound.



$f(n) = \theta(g(n))$

Ex:- $f(n) = 4n+2$ $\qquad$ $f(n) = \theta(g(n))$

$$c_1 g(n) \leq f(n) \leq c_2 g(n)$$

$$3n \leq 4n+2 \leq 5n$$

$n=1 \Rightarrow 3 \leq 6 \leq 5$ (False)

$n=2 \Rightarrow 6 \leq 10 \leq 10$ (True)

$n=3 \Rightarrow 9 \leq 14 \leq 15$ (True)

$n=4 \Rightarrow 12 \leq 18 \leq 20$ (True)

$\therefore \boxed{n \geq 2}$

$\boxed{n_0 = 2}$

$\boxed{c_1 = 3}$

$\boxed{c_2 = 5}$

$\boxed{g(n) = n}$

4) Interpret the Recursive and Iterative Binary Search algorithm with suitable examples.

Ans. Binary search is a searching algorithm used in a sorted array by repeatedly dividing the search interval in half.

The basic steps to perform Binary Search are :-

① Sort the array in ascending order

② Set the low index of the first element of the array and high index to the last element.

③ Set the middle index to the average of the low and high indices.

④ If the element at the middle index is the target element, return the middle index.

⑤ If the target element is less than the element at the middle index, set the high index to the middle index − 1.

⑥ If the target element is greater than the element at the middle index, set the low index to the midde index+1.

⑦ Repeat steps 3-6 until the element is found or it is clear that the element is not present in the array.

Binary search can be implemented in the following two ways:-

→ Iterative method

→ Recursive method

- Iterative method :-

```
binarySearch (arr, key, low, high)
    repeat till low <= high
        mid = (low + high) / 2
            if (key == arr[mid])
                return mid
            else if (key > arr[mid])    // key is on the right side
                low = mid + 1
            else                        // key is on the left side
                high = mid - 1
```

- Recursive method :-

```
binarySearch (arr, key, low, high)
    if low > high
        return False
    else
        mid = (low + high) / 2
        if key == arr[mid]
            return mid
        else if key > arr[mid]      // key is on the right side
            return binarySearch (arr, key, mid + 1, high)
        else                        // key is on the left side
            return binarySearch (arr, key, low, mid - 1)
```

2	5	9	16	18	36	52

Total number of elements in array = 7

Maximum comparison depends on the height of the tree.

Height of the tree = $\log n$

Recurrence relation, $\boxed{T(n) = T(n/2) + 1}$

$$T(n) = T\left(\frac{n}{2}\right) + 1 \quad ; n > 0$$

$$= 1 \quad ; n = 0$$

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

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

$$T\left(\frac{n}{2^2}\right) = T\left(\frac{n}{2^3}\right) + 1 \longrightarrow (3)$$

$$T\left(\frac{n}{2^3}\right) = T\left(\frac{n}{2^4}\right) + 1 \longrightarrow (4)$$

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

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

$$\Rightarrow T(n) = T\left(\frac{n}{2^3}\right) + 1 + 2 = T\left(\frac{n}{2^3}\right) + 3$$

$$\Rightarrow T(n) = T\left(\frac{n}{2^4}\right) + 1 + 3 = T\left(\frac{n}{2^4}\right) + 4$$

$$\vdots$$

$$\Rightarrow T(n) = T\left(\frac{n}{2^k}\right) + k$$

$$\Rightarrow \frac{n}{2^k} = 1$$

$$\Rightarrow n = 2^k$$

Apply $\log_2$ on both sides

$\Rightarrow \log_2 n = \log_2 2^k$

$\Rightarrow \log_2 n = k \log_2 2$

$\Rightarrow \log_2 n = k$

$\Rightarrow \boxed{T(n) = O(\log_2 n)}$

• Time Complexity :-

Minimum time is 1 and maximum Time is $\log n$

Best Case = $O(1)$, Element to be searched is at Root

Average Case = $O(\log n)$, All possible cases of search

Worst Case = $O(\log n)$, Element to be searched is not in the list.

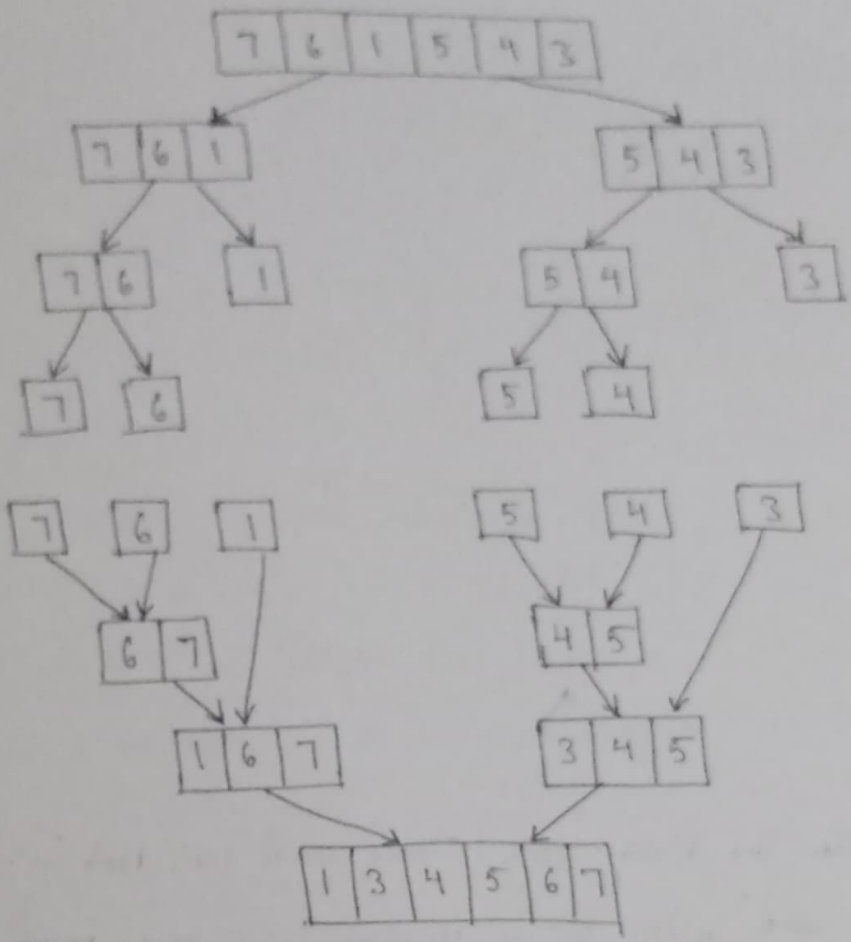
5) Demonstrate the Merge sort (Stable sort (OR) Not in-place) algorithm and solve for $n = 10$ elements: 13, 48, 26, 19, 37, 28, 42, 33, 22 and 17.

Ans. → Merge sort algorithm is a classic example of divide and conquer.

→ Given a sequence of 'n' elements $a[1], \ldots, a[n]$ the general idea is to split into 2 sets $a[1], \ldots, a[n/2]$ and $a[\lfloor n/2 \rfloor + 1], \ldots, a[n]$.

→ Each set is individually sorted and the resulting sorted sequences are merged to produce a single sorted sequence of 'n' elements.

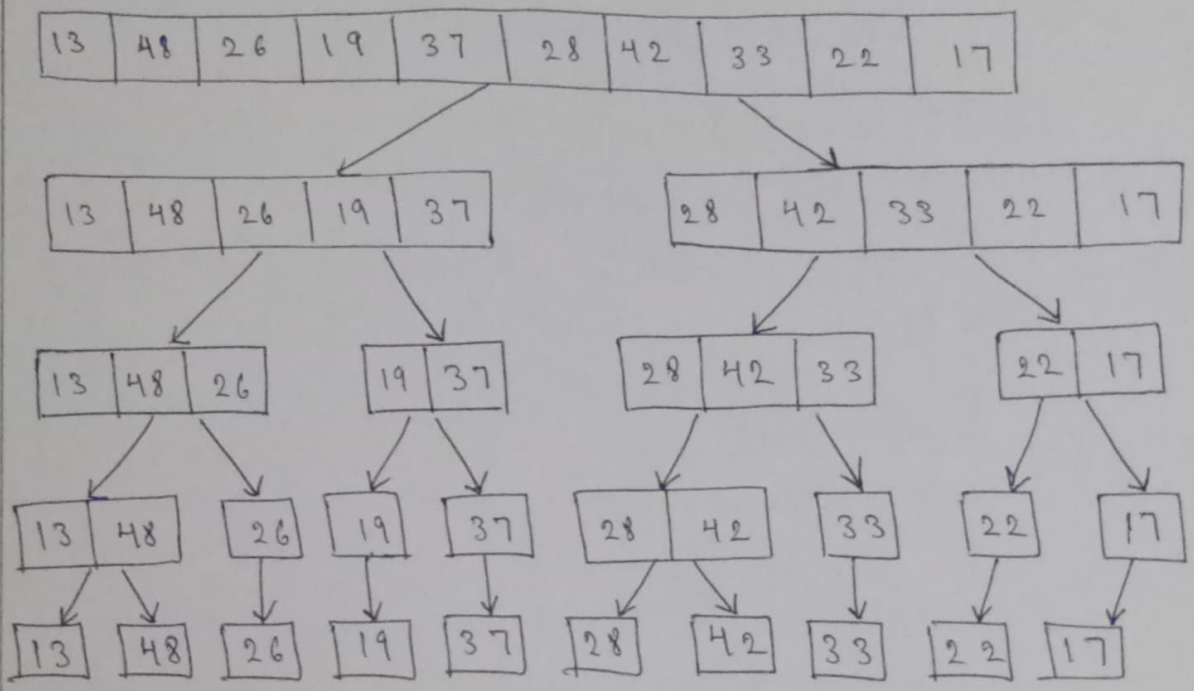
Ex:-

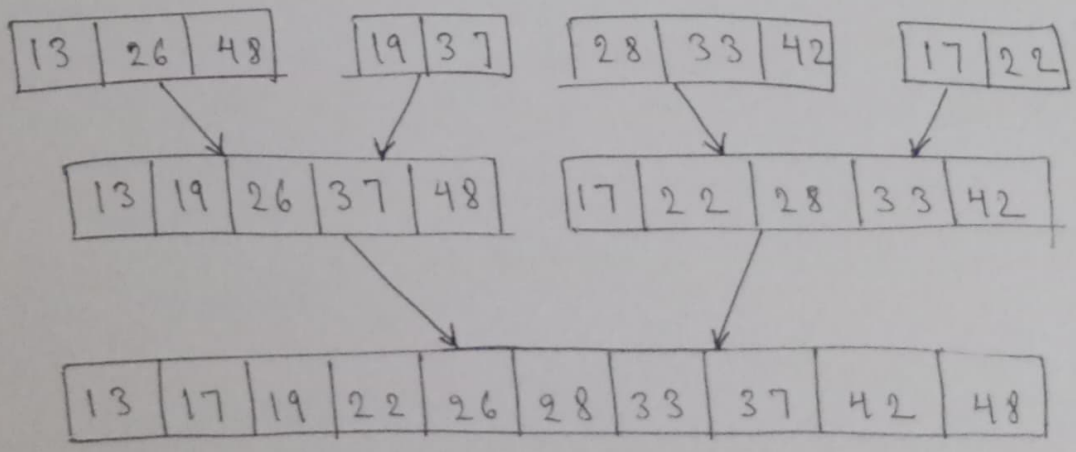


- Algorithm for merge sort:-

```
Algorithm MergeSort (array[], low, high)
{
if (low < high) then
{
mid = [(low + high)/2];
// divide the subproblems
mergesort (Array[], low, mid);
mergesort (Array[], mid+1, high);
// combine the solutions
merge (Array[], low, mid, high);
}}
```

```
Algorithm Merge (Array[], low, mid, high)
{
  int a[] = Array [mid - low];
  int b[] = Array [high - mid + 1];
  i = 0, j = 0, k = 0;
  while (i <= m && j <= n)
  {
    if (a[i] < b[j])
        c[k++] := a[i++];
    else
        c[k++] := b[j++];
  }
  for (int i; i < m; i++)
        c[k++] := a[i];
  for (int j; j < n; j++)
        c[k++] := b[j];
}
```


13 48 26 19 37 28 42 33 22 17
13 48 26 19 37
28 42 33 22 17
13 48 26
19 37
28 42 33
22 17
13 48
26
19
37
28 42
33
22
17
13
48
26
19
37
28
42
33
22
17


13 26 48
19 37
28 33 42
17 22
13 19 26 37 48
17 22 28 33 42
13 17 19 22 26 28 33 37 42 48

6) Generalize the steps of partition exchange sort (Quick sort) (OR) in-place sort (OR) unstable sort) algorithm and trace this algorithm for $n = 10$ elements: 13, 48, 26, 19, 37, 28, 42, 33, 22 and 17.

Ans. → In quick sort algorithm, partitioning of the list is performed using following steps:-

Step 1:- Consider the first element of the list as pivot (i.e, Element at first position in the list).

Step 2:- Define two variables i and j. Set i and j to first and last elements of the list respectively.

Step 3:- Increment i until list[i] > pivot then stop.

Step 4:- Decrement j until list[j] < pivot then stop.

Step 5:- If i < j then exchange list[i] and list[j]

Step 6:- Repeat steps 3, 4 and 5 until i > j

Step 7:- ~~End~~ Exchange the pivot element with list[j] element

- Quick sort Algorithm:-

```
QuickSort (alist, start, end)
{
if end - start > 1:
    p = partition (alist, start, end)
    quicksort (alist, start, p)
    quicksort (alist, p+1, end)
}
```

```
Partition (alist, istart, end)
{
    pivot = alist [start];
    i = start + 1;
    j = end
    while (i < j):
    {
        while (alist [i] <= pivot):
            i = i++;
        while (alist [j] >= pivot):
            j = j--;
        if i <= j:
            alist [i], alist [j] = alist [j], alist [i];
    }
    alist [start], alist [j] = alist [j], alist [start];
    return j;
}
```

13	48	26	19	37	28	42	33	22	17

① Select a Pivot (P) :- In the first step, we select a pivot element. Let's choose the first element, which is 13, as the pivot.

Pivot → 13 | 48, 26, 19, 37, 28, 42, 33, 22, 17

② Partitioning :- Rearrange the elements so that all the elements less than the pivot (13) are on the left, and all elements greater than the pivot are on the right.

|13| 26, 19, 37, 28, 42, 33, 22, 17, 48

③ Recursively sort :- Now, we have two partitions, We recursively apply Quick Sort to each partition.

Left Partition (elements less than 13) :-

|13| 19, 28, 22, 17, 26

Right Partition (elements greater than 13) :-

37, 42, 33, 48

④ Repeat for Left partition :-

- Select a pivot (19) :-

|13|19| 28, 22, 17, 26

- Partition the left partition :-

|13, 17|19| 28, 22, 26

⑤ Repeat for Left partition (subpartition):-

- Select a pivot (17):-

  |13,17| 19 |22,26,28

- Partition the left partition (no change):-

  |13,17| 19 |22,26,28

⑥ Repeat for Right partition:-

- Select a pivot (42):-

  33, 37 |42| 48

- Partition the left partition (elements less than 42):-

  33,37|42|48

⑦ Repeat for Right partition (subpartition):-

- Select a pivot (37):-

  33 |37| 42

- Partition the left partition (no change):-

  33|37|42

⑧ Merge the sorted subarrays:- Merge all the sorted subarrays together to obtain the final ~~#~~ sorted array.

13	17	19	22	26	28	33	37	42	48

7) Discuss about Analysis of Quick Sort algorithm.

Ans.

- Time Complexity :-

→ Best Case: $O(n \log n)$

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

Worst Case: $O(n^2)$

→ To sort a sorted list with 'n' number of elements, we need to make $((n-1)+(n-2)+(n-3)+\ldots+1) = (n(n-1))/2$ number of comparisons in the worst case.

- Space Complexity :-

→ Space complexity of Quick Sort is $O(n)$

- Best and Average case time complexity :-

$$T(n) = 2T(n/2) + n \quad ; n > 1$$
$$= 1 \quad ; n = 1$$

$$T(n) = 2T(n/2) + n \longrightarrow (1)$$

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

$$T\left(\frac{n}{2^2}\right) = 2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2} \longrightarrow (3)$$

$$T\left(\frac{n}{2^3}\right) = 2T\left(\frac{n}{2^4}\right) + \frac{n}{2^3} \longrightarrow (4)$$

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

Sub (2) in above equation.

$$\Rightarrow T(n) = 2\left[2T\left(\frac{n}{2^2}\right) + \frac{n}{2}\right] + n$$

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

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

Sub ③ in above equation

$$\Rightarrow T(n) = 2^2\left[2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2}\right] + 2n$$

$$= 2^3 T\left(\frac{n}{2^3}\right) + 2^2\frac{n}{2^2} + 2n$$

$$= 2^3 T\left(\frac{n}{2^3}\right) + 3n$$

Sub ④ in above equation

$$\Rightarrow T(n) = 2^3\left[2T\left(\frac{n}{2^4}\right) + \frac{n}{2^3}\right] + 3n$$

$$= 2^4 T\left(\frac{n}{2^4}\right) + 2^3\frac{n}{2^3} + 3n$$

$$= 2^4 T\left(\frac{n}{2^4}\right) + 4n$$

$$\vdots$$

$$\Rightarrow T(n) = 2^k T\left(\frac{n}{2^k}\right) + kn$$

Now, $\frac{n}{2^k} = 1$

$\Rightarrow n = 2^k$

Apply $\log_2$ on both sides

$\Rightarrow \log_2 n = \log_2 2^k$

$\Rightarrow \log_2 n = k \log_2 2$

$\Rightarrow \log_2 n = k$

$\Rightarrow T(n) = 2^k T(1) + kn$

$= n \cdot 1 + kn$

$= n + n \log_2 n$

$\Rightarrow \boxed{T(n) = O(n \log_2 n)}$

• Worst case time complexity:-

$$T(n) = T(n-1) + n \quad ; n > 0$$
$$= 1 \quad ; n = 0$$

$$T(n-1) = T(n-2) + n-1 \longrightarrow (1)$$
$$T(n-2) = T(n-3) + n-2 \longrightarrow (2)$$
$$T(n-3) = T(n-4) + n-3 \longrightarrow (3)$$

$$T(n) = T(n-1) + n$$
$$\Rightarrow T(n) = T(n-2) + (n-1) + n$$
$$\Rightarrow T(n) = T(n-3) + (n-2) + (n-1) + n$$
$$\Rightarrow T(n) = T(n-4) + (n-3) + (n-2) + (n-1) + n$$
$$\vdots$$
$$\Rightarrow T(n) = T(n-k) + (n-(k-1)) + (n-(k-2)) + (n-(k-3)) + \ldots + n$$

$$n - k = 0 \Rightarrow \boxed{n = k}$$

$$\Rightarrow T(n) = T(0) + (1+2+3+4+\ldots+n)$$
$$\Rightarrow T(n) = 1 + \frac{n(n+1)}{2}$$
$$\Rightarrow T(n) = 1 + \frac{n^2+n}{2}$$
$$\Rightarrow \boxed{T(n) = O(n^2)}$$