byteXL **Name of Faculty:** ***Suraj Mourya***

| | |
|---|---|
| FACULTY OF ENGG. & TECH. - **PIET** | Sem: **5th** |
| LESSON PLAN Academic Year: **2024** | Name of Department: **CSE** |

**Module 2**

**Divide & Conquer Algorithms:**
Structure of divide-and-conquer algorithms, examples: Binary search, quick sort, Merge sort, Strassen Multiplication; Max-Min problem

# Introduction to Divide and Conquer Algorithm:

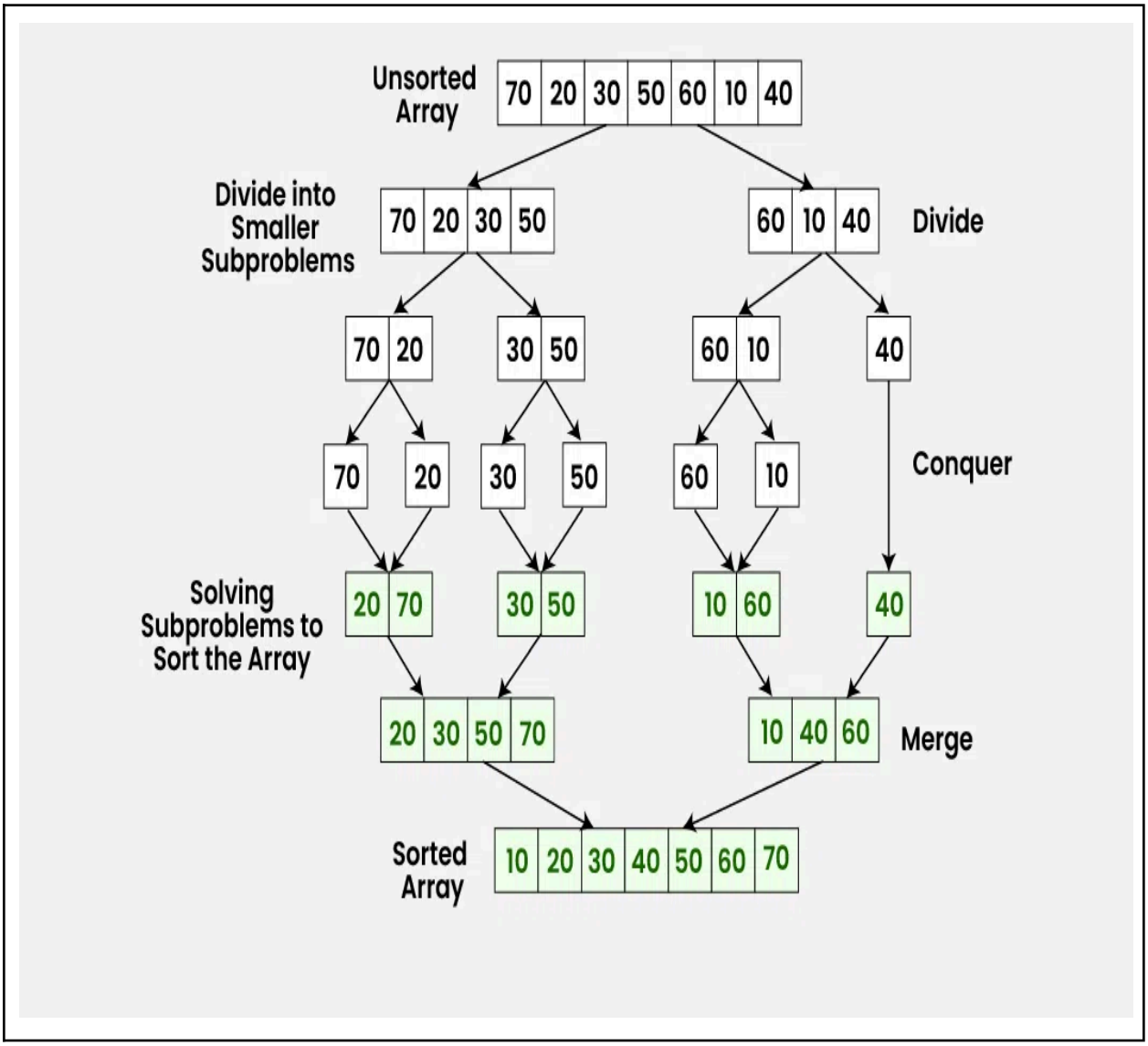
**Divide and Conquer Algorithm** is a problem-solving technique used to solve problems by dividing the main problem into subproblems, solving them individually and then merging them to find solution to the original problem.

## Divide and Conquer Algorithm Definition:

- Divide and Conquer Algorithm involves breaking a larger problem into smaller subproblems, solving them independently, and then combining their solutions to solve the original problem.
- The basic idea is to recursively divide the problem into smaller subproblems until they become simple enough to be solved directly.
- Once the solutions to the subproblems are obtained, they are then combined to produce the overall solution.

## Working of Divide and Conquer Algorithm:

Divide and Conquer Algorithm can be divided into three steps: **Divide**, **Conquer** and **Merge** .




**1. Divide:**

- Break down the original problem into smaller subproblems.
- Each subproblem should represent a part of the overall problem.
- The goal is to divide the problem until no further division is possible.

**2. Conquer:**

- Solve each of the smaller subproblems individually.
- If a subproblem is small enough (often referred to as the “base case”), we solve it directly without further recursion.
- The goal is to find solutions for these subproblems independently.

**3. Merge:**

- Combine the sub-problems to get the final solution of the whole problem.
- Once the smaller subproblems are solved, we recursively combine their solutions to get the solution of larger problem.
- The goal is to formulate a solution for the original problem by merging the results from the subproblems.

## Characteristics of Divide and Conquer Algorithm:

Divide and Conquer Algorithm involves breaking down a problem into smaller, more manageable parts, solving each part individually, and then combining the solutions to solve the original problem. The characteristics of Divide and Conquer Algorithm are:

- **Dividing the Problem**: The first step is to break the problem into smaller, more manageable subproblems.
- **Independence of Subproblems**: Each subproblem should be independent of the others, meaning that solving one subproblem

does not depend on the solution of another. This allows for parallel processing or concurrent execution of subproblems, which can lead to efficiency gains.

- **Conquering Each Subproblem**: Once divided, the subproblems are solved individually. This may involve applying the same divide and conquer approach recursively until the subproblems become simple enough to solve directly, or it may involve applying a different algorithm or technique.
- **Combining Solutions**: After solving the subproblems, their solutions are combined to obtain the solution to the original problem.

## Advantages of Divide and Conquer Algorithm:

- **Solving difficult problems:** Divide and conquer technique is a tool for solving difficult problems conceptually. e.g. Tower of Hanoi puzzle.
- **Algorithm efficiency:** The divide-and-conquer algorithm often helps in the discovery of efficient algorithms.
- **Parallelism:** Normally Divide and Conquer algorithms are used in multi-processor machines having shared-memory systems where the communication of data between processors does not need to be planned in advance, because distinct sub-problems can be executed

on different processors.

- **Memory access:** These algorithms naturally make an efficient use of memory caches. Since the subproblems are small enough to be solved in cache without using the main memory that is slower one. Any algorithm that uses cache efficiently is called cache oblivious.

## Disadvantages of Divide and Conquer Algorithm:

- **Overhead:** The process of dividing the problem into subproblems and then combining the solutions can require additional time and resources. This overhead can be significant for problems that are already relatively small or that have a simple solution.
- **Complexity:** Dividing a problem into smaller subproblems can increase the complexity of the overall solution. This is particularly true when the subproblems are interdependent and must be solved in a specific order.
- **Difficulty of implementation:** Some problems are difficult to divide into smaller subproblems or require a complex algorithm to do so. In these cases, it can be challenging to implement a divide and conquer solution.
- **Memory limitations:** When working with large data sets, the memory requirements for storing the intermediate results of the subproblems can become a limiting factor.

## Frequently Asked Questions (FAQs):

**1. What is the Divide and Conquer algorithm?**

*Divide and Conquer is a problem-solving technique where a problem is divided into smaller, more manageable subproblems. These subproblems are solved recursively, and then their solutions are combined to solve the original problem.*

**2. What are the key steps involved in the Divide and Conquer algorithm?**

*The main steps are:*

***Divide**: Break the problem into smaller subproblems.*

***Conquer**: Solve the subproblems recursively.*

***Combine**: Merge or combine the solutions of the subproblems to obtain the solution to the original problem.*

**3. What are some examples of problems solved using Divide and Conquer?**

*Divide and Conquer Algorithm is used in sorting algorithms like Merge Sort and Quick Sort, finding closest pair of points, Strassen's Algorithm, etc.*

**4. How does Merge Sort use the Divide and Conquer approach?**

*Merge Sort divides the array into two halves, recursively sorts each half, and then merges the sorted halves to produce the final sorted array.*

**5. What is the time complexity of Divide and Conquer algorithms?**

*The time complexity varies depending on the specific problem and how it's implemented. Generally, many Divide and Conquer algorithms have a time complexity of O(n log n) or better.*

**6. Can Divide and Conquer algorithms be parallelized?**

*Yes, Divide and Conquer algorithms are often naturally parallelizable because independent subproblems can be solved concurrently. This makes them suitable for parallel computing environments.*

**7. What are some strategies for choosing the base case in Divide and Conquer algorithms?**

*The base case should be simple enough to solve directly, without further*

*division. It's often chosen based on the smallest input size where the problem can be solved trivially.*

# Divide and Conquer :-

```
DAC (P)
{
  if (small (P))
  {
    S(P);
  }
  else
  {
    divide P into P1, P2, P3, ... Pk
    Apply DAC(P1), DAC(P2), ...
    Combine (DAC(P1), DAC(P2), ..
  }
```

Note: Problem P and subproblems ($P_1, P_2, P_3, \ldots$) should be of same type. means like If P is problem of sorting then all $P_1, P_2, P_3, \ldots$ all must sorting problem only.

## 1. Binary Search :-

arr:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 6 | 8 | 12 | 14 | 17 | 25 | 29 | 31 | 36 | 42 | 47 | 53 | 55 | 62 |

key = 42

| l | h | mid = $\lfloor \frac{l+h}{2} \rfloor$ |
|---|---|---|
| 1 | 15 | $\frac{1+15}{2} = 8$ |
| 9 | 15 | $\frac{9+15}{2} = 12$ |
| 9 | 11 | $\frac{9+11}{2} = 10$ |
| 11 | 11 | $\frac{11+11}{2} = 11$ |

key = 30

| l | h | mid = $\lfloor \frac{l+h}{2} \rfloor$ |
|---|---|---|
| 1 | 15 | $\frac{1+15}{2} = 8$ |
| 9 | 15 | $\frac{9+15}{2} = 12$ |
| 9 | 11 | $\frac{20}{2} = 10$ |
| 9 | 9 | $\frac{9+9}{2} = 9$ |
| 9 | 8 | ⊗ (l > h) |

key = 12

| l | h | mid = $\lfloor \frac{l+h}{2} \rfloor$ |
|---|---|---|
| 1 | 15 | $\frac{1+15}{2} = 8$ |
| 1 | 7 | $\frac{1+7}{2} = 4$ |

# Flow of code :-

## Binary Search Algorithm:

Below is the step-by-step algorithm for Binary Search:

- Divide the search space into two halves by finding the middle index "mid".
- Compare the middle element of the search space with the **key**.
- If the **key** is found at middle element, the process is terminated.
- If the **key** is not found at middle element, choose which half will be used as the next search space.
  - If the **key** is smaller than the middle element, then the **left** side is used for next search.
  - If the **key** is larger than the middle element, then the **right** side is used for next search.
- This process is continued until the **key** is found or the total search space is exhausted.

```
def search(self, nums: List[int], target: int) -> int:
    low = 0
    high = len(nums) - 1

    while low <= high:
        mid = (low + high) // 2
        if nums[mid] == target:
            return mid
        elif nums[mid] > target:
            high = mid - 1
        else:
            low = mid + 1

    return -1
```

min → 1 unit ⇒ $O(1)$

max → $\log n$ ⇒ $O(\log_2 n)$

Recurrence Relation :

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

$T(n) = T(n/2) + 1$ — (i)

$T(n/2) = T(n/4) + 1$ — (ii)

$T(n/4) = T(n/8) + 1$ — (iii)

$\Rightarrow T(n) = [T(n/4) + 1] + 1$

$= T(n/4) + 2$

$= [T(n/8) + 1] + 2$

$= T(n/8) + 3$

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

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

$\Rightarrow k = \log_2 n$

$\Rightarrow T(1) + \log_2 n$

$\Rightarrow 1 + \log_2 n \Rightarrow O(\log_2 n)$

## Complexity Analysis of Binary Search Algorithm:

- **Time Complexity:**
  - Best Case: O(1)
  - Average Case: O(log N)
  - Worst Case: O(log N)

## Advantages of Binary Search:

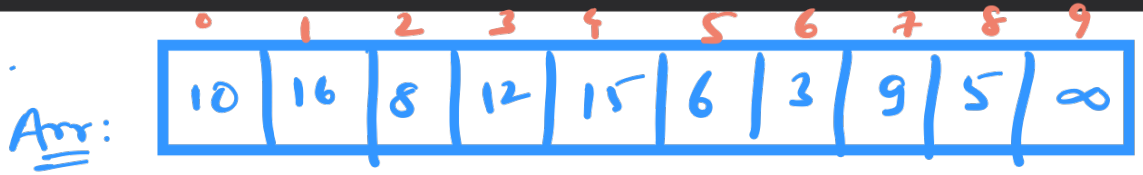
- Binary search is faster than linear search, especially for large arrays.
- More efficient than other searching algorithms with a similar time complexity, such as interpolation search or exponential search.
- Binary search is well-suited for searching large datasets that are stored in external memory, such as on a hard drive or in the cloud.

## Disadvantages of Binary Search:

- The array should be sorted.
- Binary search requires that the data structure being searched be stored in contiguous memory locations.
- Binary search requires that the elements of the array be comparable, meaning that they must be able to be ordered.

# 2. Quick Sort :-

**QuickSort** *is a sorting algorithm based on the Divide and Conquer algorithm that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array.*



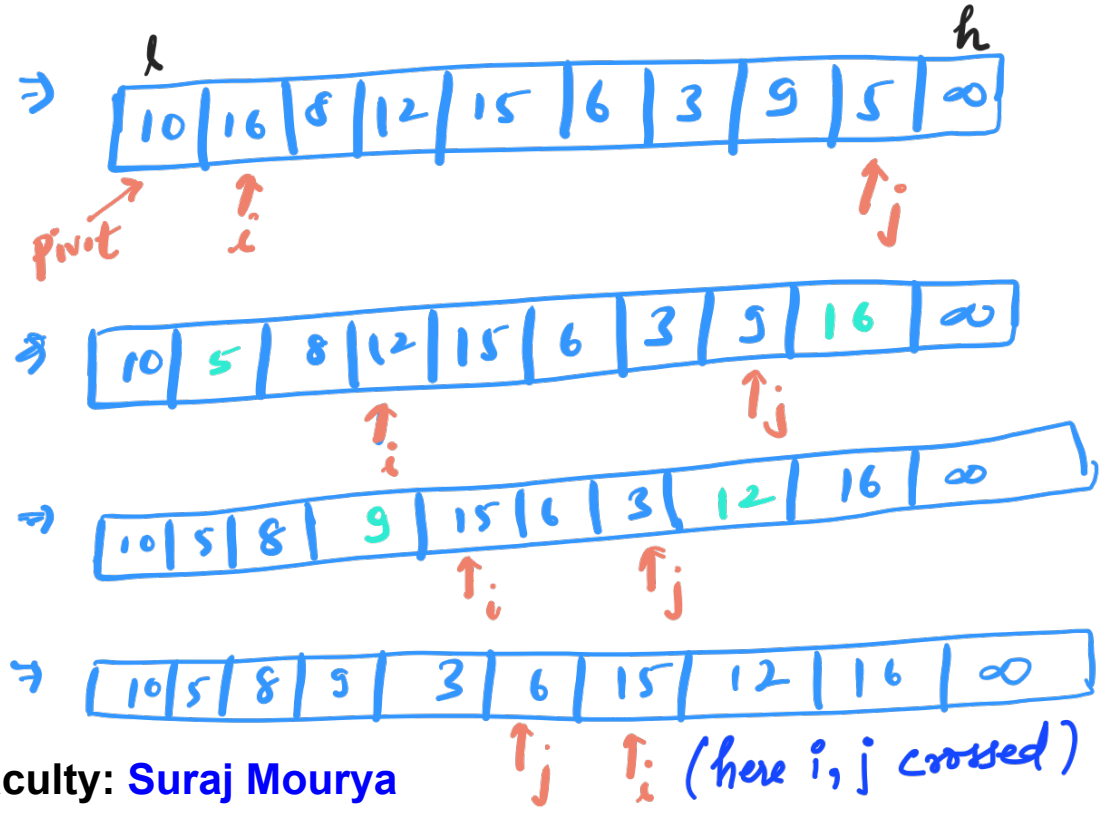

## Points to Remember :-

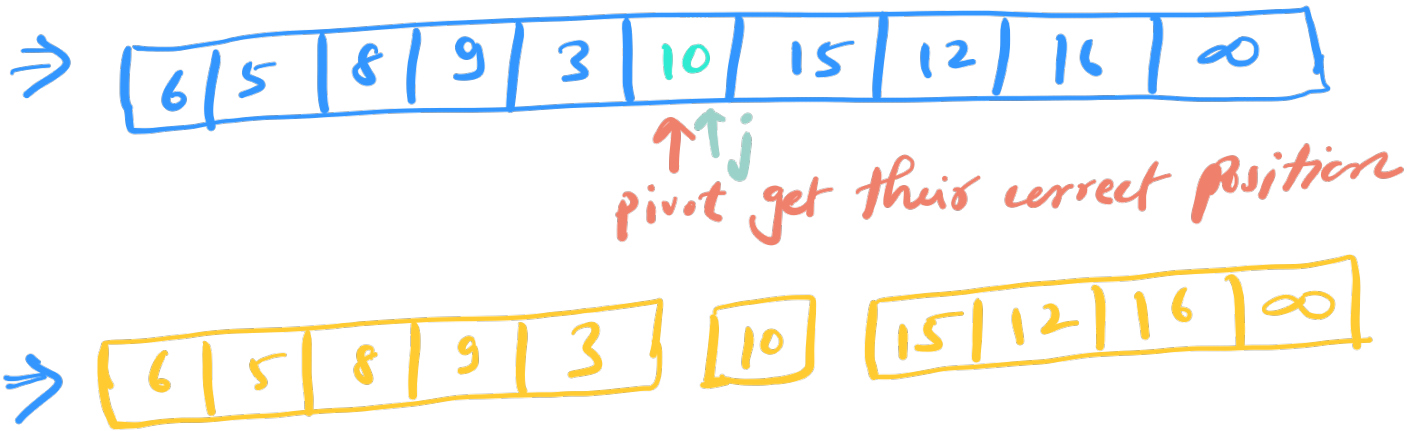
i ⇒ increment i untill you find greater than pivot

j ⇒ decrement j until u find lesser than pivot

if i and j not crossed ⇒ swap(i, j)

if i and j crossed ⇒ swap(pivot, j)






```
Partition(l, h)
{
  pivot = A[l]
  i = l; j = h;
  while (i < j)
  {
    do
    {
      i++;
    }while (A[i] ≤ pivot);
    do
    {
      j--;
    }while (A[j] > pivot);
    if (i < j)
      swap(A[i], A[j]);
  }
  swap(A[l], A[j]);
  return j;
}
```

```
QuickSort(l, h)
{
  if (l < h)
  {
    j = partition(l, h);
    QuickSort(l, j);
    QuickSort(j+1, h);
  }
}
```

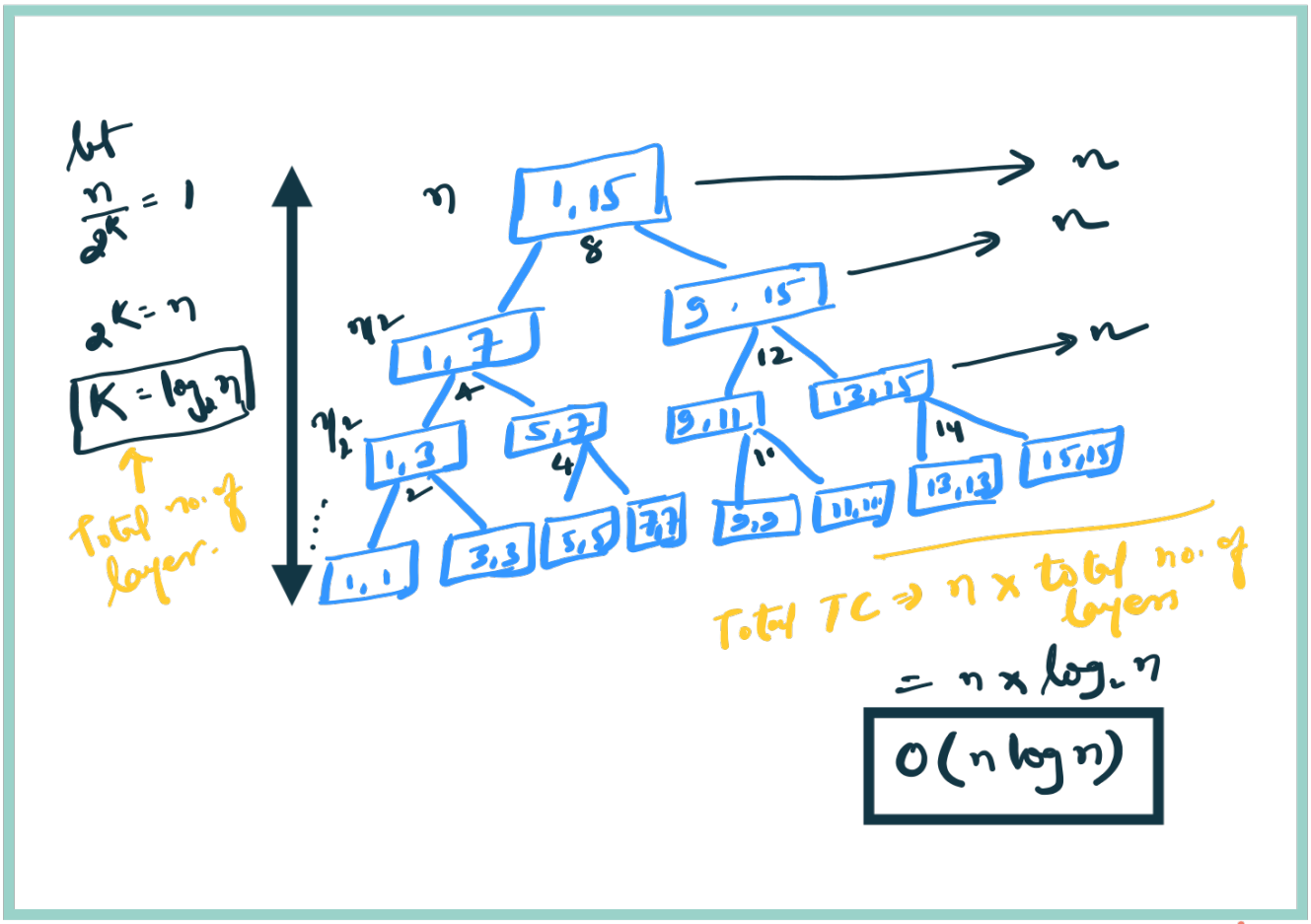
**Faculty:** Suraj Mourya

# Quick Sort Analysis :-

**Time Complexity:**

- **Best Case** : Ω (N log (N))
  The best-case scenario for quicksort occur when the pivot chosen at the each step divides the array into roughly equal halves.
  In this case, the algorithm will make balanced partitions, leading to efficient Sorting.
- **Average Case: θ ( N log (N))**
  Quicksort's average-case performance is usually very good in practice, making it one of the fastest sorting Algorithm.

Best case :- when we assuming each time pivot dividing the dataset into 2 equal halves. = $O(n \log n)$



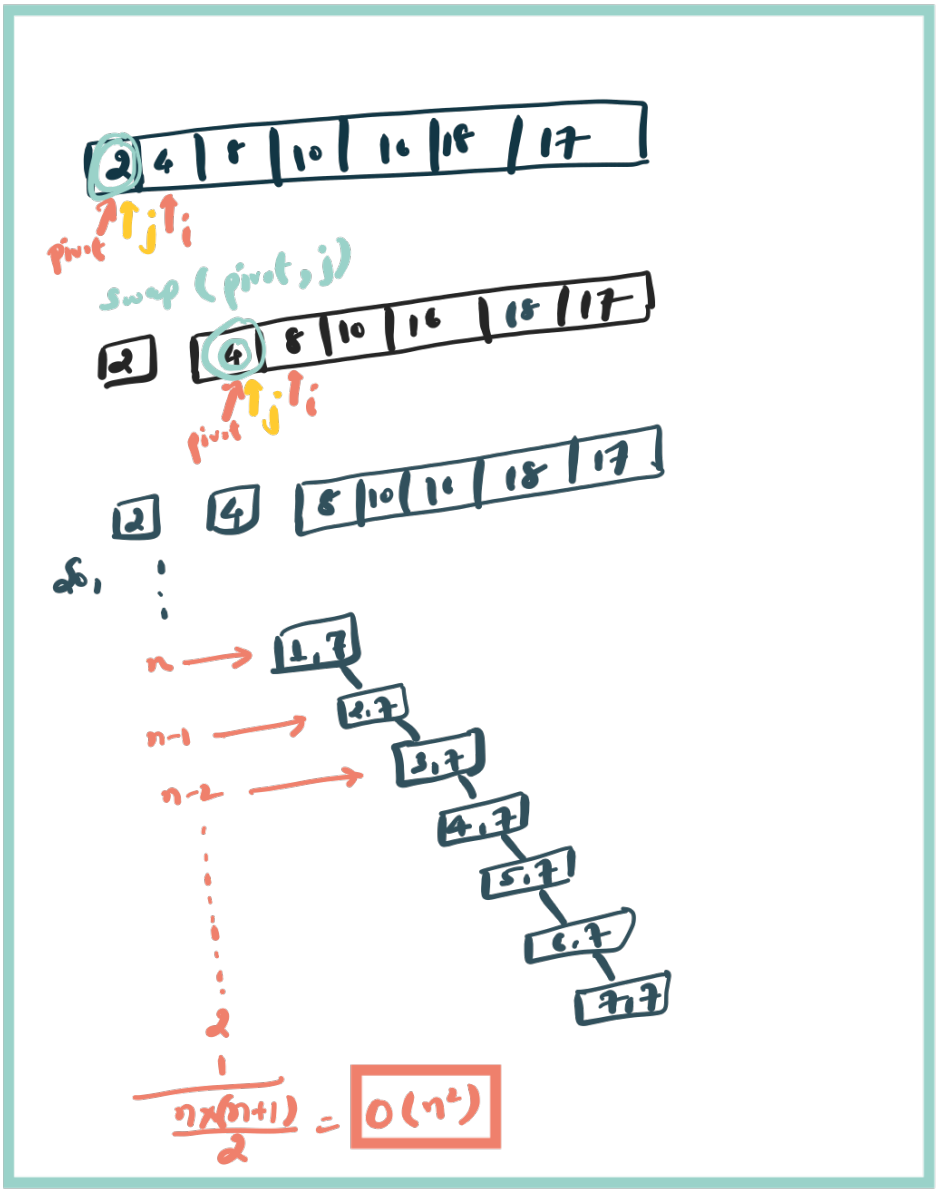

→ But getting Best case is rare because how will we ensure the selected pivot value is the median of dataset when the given data is unsorted.

**Faculty:** Suraj Mourya

## Worst case:-

- **Worst Case: O(N2)**
  The worst-case Scenario for Quicksort occur when the pivot at each step consistently results in highly unbalanced partitions. When the array is already sorted and the pivot is always chosen as the smallest or largest element. To mitigate the worst-case Scenario, various techniques are used such as choosing a good pivot (e.g., median of three) and using Randomized algorithm (Randomized Quicksort ) to shuffle the element before sorting.

→ when the data is given either ascending or descending order and we are selecting Ist or last element as pivot respectively then we will get worst case. ⇒ $O(n^2)$




**Faculty:** Suraj Mourya

Suggestion to avoid worst case :-

1. Select middle element as pivot.
2. Select Random element as pivot.

## Space Complexity :-

→ Since Quick sort using recursion, By the way Quick sort not using any extra space but Quick sort uses recursion hence, for recursion stack is used.

→ what could be the maximum size of stack ?

* In best case, height is optimised ⇒ $\log n$
* In worst case, skewed Tree ⇒ $n$

So, space complexity will be ⇒ $\log n$ to $n$

- **Auxiliary Space:** O(1), if we don't consider the recursive stack space. If we consider the recursive stack space then, in the worst case quicksort could make $O(N)$.

**Faculty:** Suraj Mourya

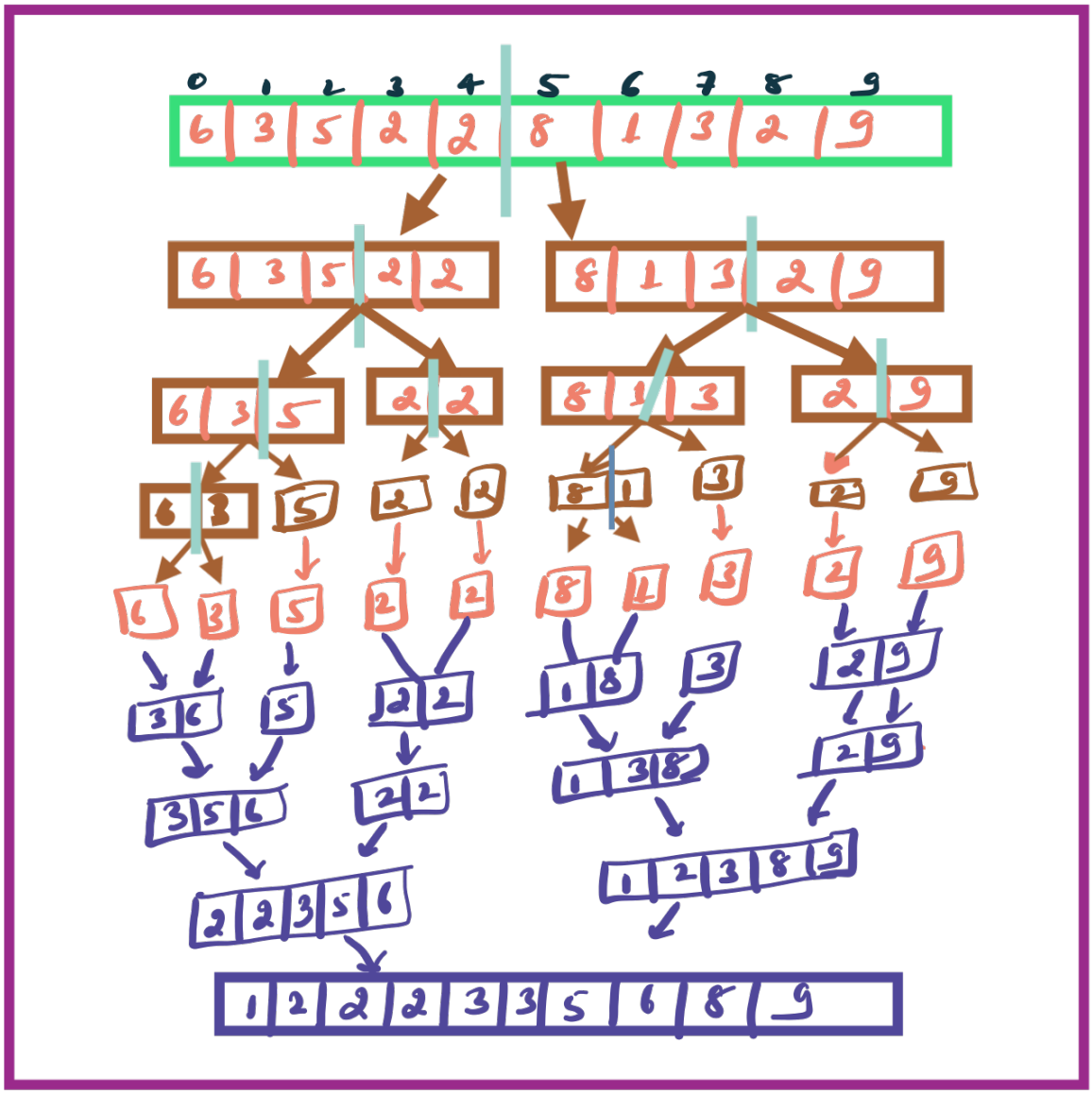
## Advantages of Quick Sort:

- It is a divide-and-conquer algorithm that makes it easier to solve problems.
- It is efficient on large data sets.
- It has a low overhead, as it only requires a small amount of memory to function.

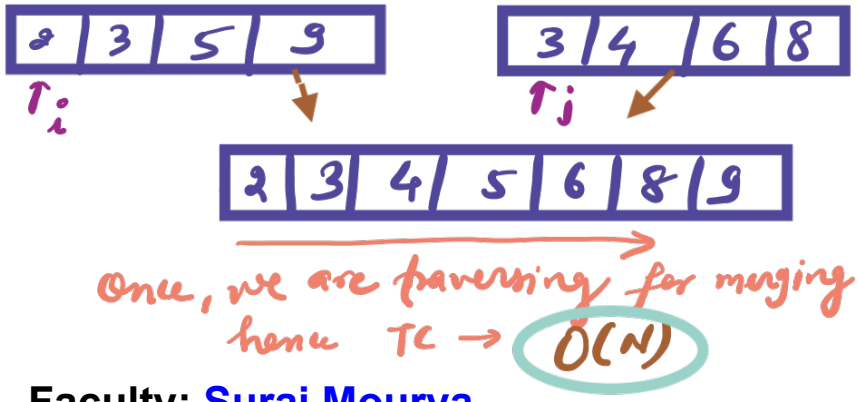
## Disadvantages of Quick Sort:

- It has a worst-case time complexity of $O(N^2)$, which occurs when the pivot is chosen poorly.
- It is not a good choice for small data sets.
- It is not a stable sort, meaning that if two elements have the same key, their relative order will not be preserved in the sorted output in case of quick sort, because here we are swapping elements according to the pivot's position (without considering their original positions).

# Merge Sort:




But Question is, How Merging is happening ?




```
if (i<j):
    add arr[i] in result
    i++
else:
    add arr[j] in result
    j++
```

**Faculty:** Suraj Mourya

```
void mergeSort (arr, start, end)
{
   if (start == end)
        return;
    mid = (start+end)/2
   mergeSort (arr, start, mid)
   mergeSort (arr, mid+1, end)
    merge (arr, start, mid, end)
}
```

```
merge (arr, start, mid, end)
{
   temp [end+1]  => array for storing result
    left = start, right = mid+1, index = 0
    while (left <= mid && right <= end)
    {
       if arr [left] <= arr [right]
        {
            temp [index] = arr [left]
            index++, left++;
        }
       else
       {
           temp [index] = arr [right]
           index++, right++;
       }
   } // left array remaining
      while (left <= mid)
      {
         temp [index] = arr [left];
         index++, left++;
      } // Right array remaining
        while (right <= end)
        {  temp [index] = arr [right]
           index++, right++;
        }
}
```

**Faculty:** Suraj Mourya

```
merge_sort (arr, start, end)
{
  if (start < end)
    mid = ⌊(start + end) / 2⌋
    merge_sort(arr, start, mid)
    merge_sort(arr, mid+1, end)
    merge (arr, start, mid, end)
}
```

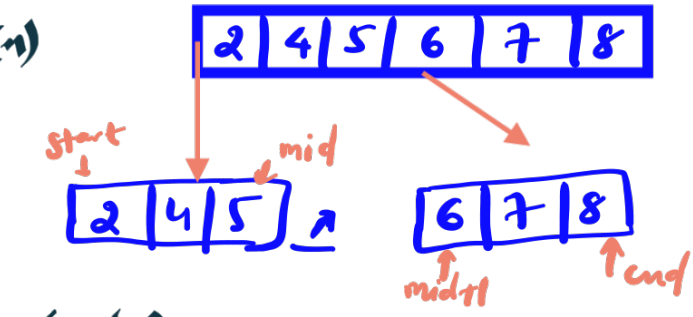
$T(n)$

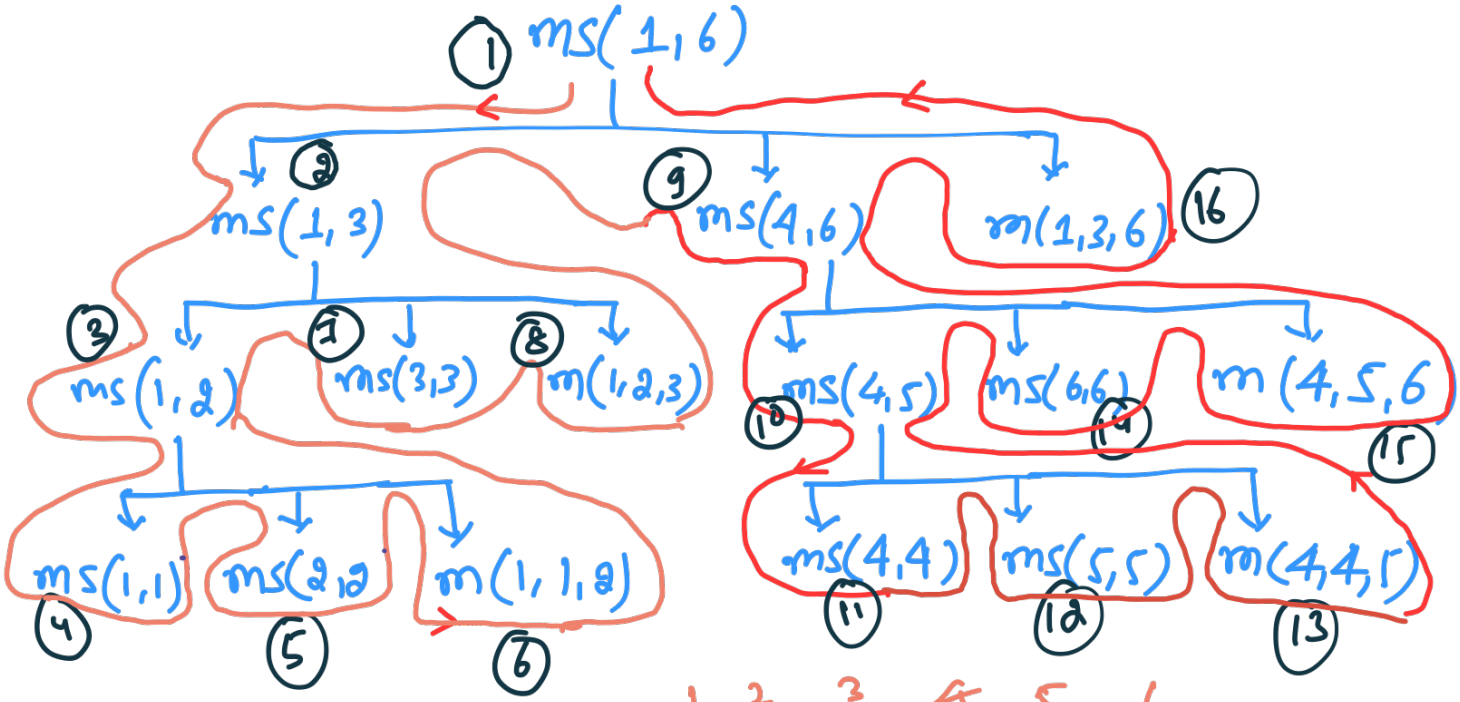
$merge\_sort(arr, start, mid) \rightarrow T(n/2)$

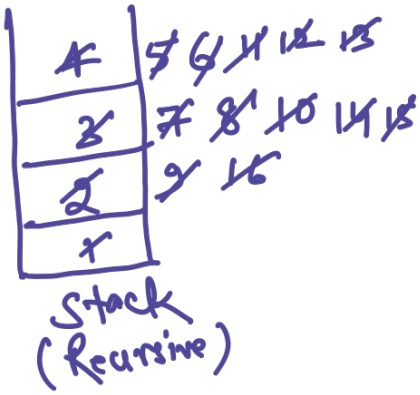
$merge\_sort(arr, mid+1, end) \rightarrow T(n/2)$

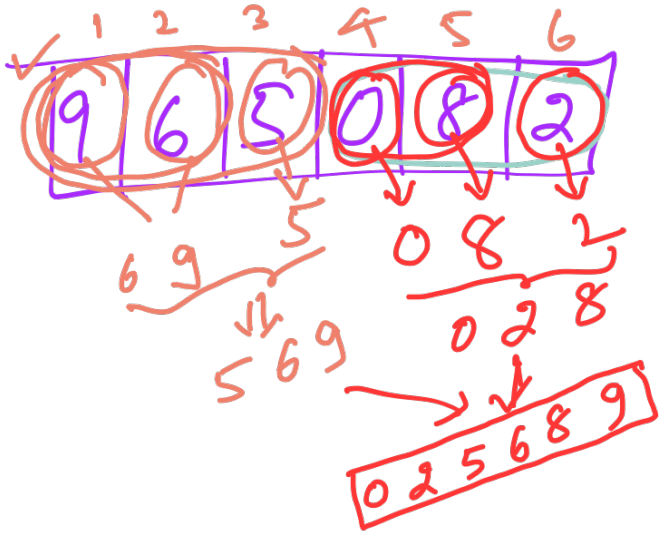
$merge(arr, start, mid, end) \rightarrow O(n)$

$\Rightarrow T(n) = 2 * T(n/2) + O(n)$




**Faculty:** Suraj Mourya

→ Size of stack = $(\log n) \Rightarrow O(\log n)$

→ merge → $O(n)$ [Since we have created global array for merging of length n]

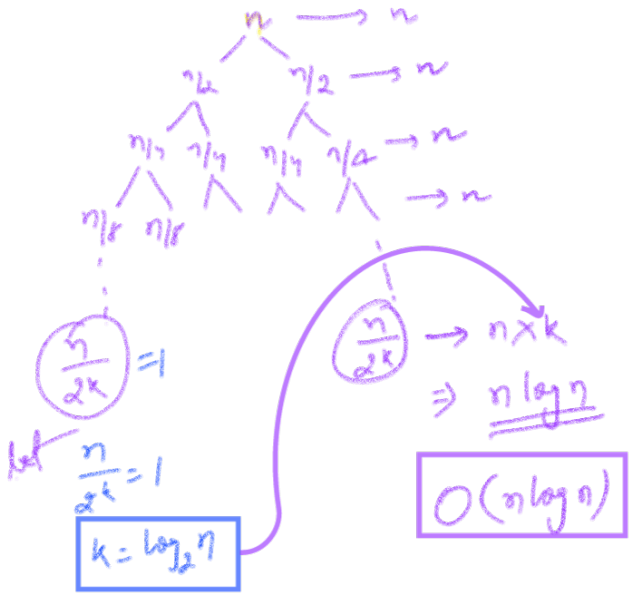
Stack → $O(\log n)$

$O(n + \log n)$ ⇐ Total space is required.

$\Rightarrow \boxed{O(n)} \Rightarrow$ Space complexity.

**Time Complexity :-**

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




Que: Given "logn" sorted lists each of size "n/logn", what is the total time required to merge them into one single list. [GATE 2017]

(a) $n \log n$

(b) $\log n$

(c) $\log \log n$

(d) $n \log \log n$

## Complexity Analysis of Merge Sort:

**Time Complexity:**

- **Best Case:** O(n log n), When the array is already sorted or nearly sorted.
- **Average Case:** O(n log n), When the array is randomly ordered.
- **Worst Case:** O(n log n), When the array is sorted in reverse order.

**Space Complexity:** O(n), Additional space is required for the temporary array used during merging.

## Applications of Merge Sort:

- Sorting large datasets
- External sorting (when the dataset is too large to fit in memory)
- Inversion counting (counting the number of inversions in an array)
- Finding the median of an array

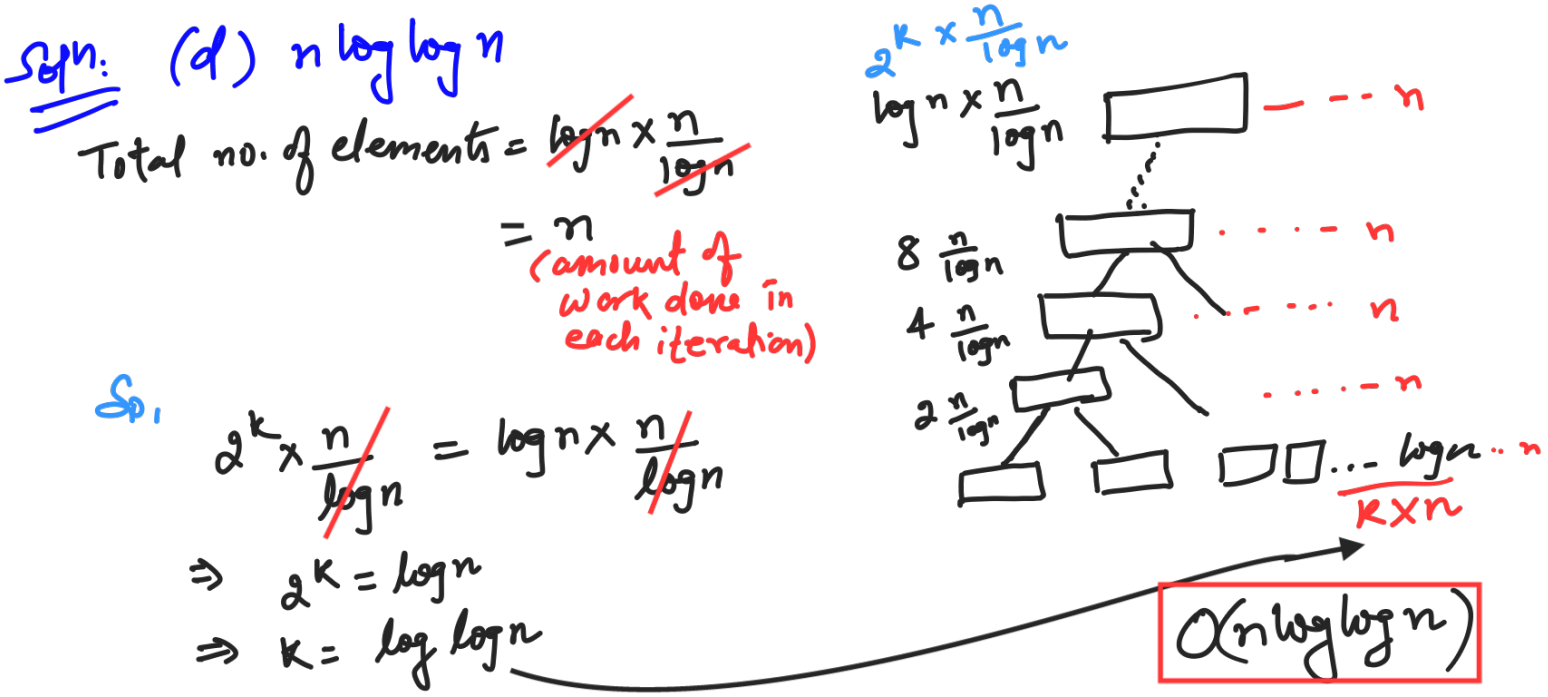
## Advantages of Merge Sort:

- **Stability** : Merge sort is a stable sorting algorithm, which means it maintains the relative order of equal elements in the input array.
- **Guaranteed worst-case performance:** Merge sort has a worst-case time complexity of **O(N logN)** , which means it performs well even on large datasets.
- **Simple to implement:** The divide-and-conquer approach is straightforward.

## Disadvantages of Merge Sort:

- **Space complexity:** Merge sort requires additional memory to store the merged sub-arrays during the sorting process.
- **Not in-place:** Merge sort is not an in-place sorting algorithm, which means it requires additional memory to store the sorted data. This can be a disadvantage in applications where memory usage is a concern.

**Faculty: Suraj Mourya**




# Strassen's Matrix Multiplication :-

Why Strassen's matrix algorithm is better than normal matrix multiplication and How to multiply two matrices using Strassen's matrix multiplication algorithm?

So the main idea is to use the divide and conquer technique in this algorithm – divide matrix A & matrix B into 8 submatrices and then recursively compute the submatrices of C.

**Faculty:** Suraj Mourya

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \times B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = C = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}$$

$2 \times 2$ $\quad$ $2 \times 2$ $\quad$ $2 \times 2$

Brute force

```
for (i=0; i<n; i++){
    for( j=0; j<n; j++){
        c[i,j]=0;
        for( k=0; k<n; k++){
            c[i,j] += A[i,k] * B[k,j];
        }
    }
}
```

$\Rightarrow O(N^3)$

$$c_{11} = a_{11} * b_{11} + a_{12} * b_{21}$$
$$c_{12} = a_{11} * b_{12} + a_{12} * b_{22}$$
$$c_{21} = a_{21} * b_{11} + a_{22} * b_{21}$$
$$c_{22} = a_{21} * b_{12} + a_{22} * b_{22}$$

If we use formula it will take constant time.

→ If we assume [2×2] matrix is the smallest matrix and treat as base case then: we can solve larger matrix also of order $2^n$

**Faculty:** Suraj Mourya

Example:-

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}$$

$A_{11}$ $A_{12}$ $A_{21}$ $A_{22}$

$\frac{4}{2} \times \frac{4}{2}$

$$B = \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{bmatrix}$$

$B_{11}$ $B_{12}$ $B_{21}$ $B_{22}$

$\frac{4}{2} \times \frac{4}{2}$

Algorithm

```
MM(A, B, n)
{
    if (n <= 2)
    {
        c11 = a11 * b11 + a12 * b21
        c12 = a11 * b12 + a12 * b22
        c21 = a21 * b11 + a22 * b21
        c22 = a21 * b12 + a22 * b22
    }
    else
    {
        MM(A11, B11, n/2) + MM(A12, B21, n/2)
        MM(A11, B12, n/2) + MM(A12, B22, n/2)
        MM(A21, B11, n/2) + MM(A22, B21, n/2)
        MM(A21, B12, n/2) + MM(A22, B22, n/2)
    }
}
```

## Recurrence Relation:-

$$T(n) = \begin{cases} 1 & n \le 2 \\ 8T(n/2) + n^2 & n > 2 \end{cases}$$

By using Master method:

$a = 8$
$b = 2$
$f(n) = n^2$
$k = 2$

$\log_a b = \log_2 8 = 3$

$\log_a b > k$

case-I

$\Theta(n^k \log n) \Rightarrow \boxed{\Theta(n^3)}$

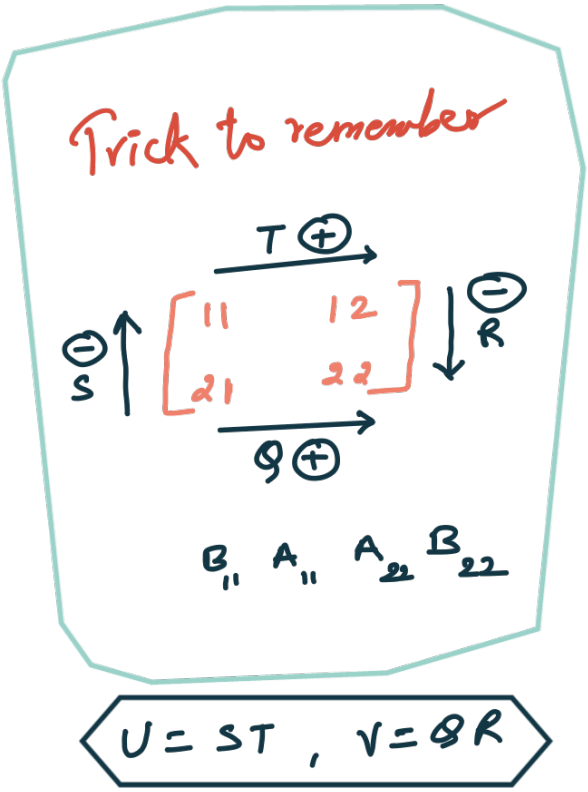
→ By using Strassen's Algorithm, STRASSEN proposed formulas:

### Formula:-

$$P = (A_{11} + A_{22})(B_{11} + B_{22})$$
$$Q = B_{11}(A_{21} + A_{22})$$
$$R = A_{11}(B_{12} - B_{22})$$
$$S = A_{22}(B_{21} - B_{11})$$
$$T = B_{22}(A_{11} + A_{12})$$
$$U = (A_{21} - A_{11})(B_{11} + B_{12})$$
$$V = (B_{21} + B_{22})(A_{12} - A_{22})$$

Then after:

$$C_{11} = P + S - T + V$$
$$C_{12} = R + T$$
$$C_{21} = Q + S$$
$$C_{22} = P + R - Q + U$$




$U = ST$ , $V = QR$

Recurrence Relation :-

$$T(n) = \begin{cases} 1 & n \leq 2 \\ 7T(n/2) + n^2 & n > 2 \end{cases}$$

$\log_2 7 = 2.81 > K = 2$ (case I)

$$\boxed{\theta(n^{2.81})}$$

By this way Strassen's reduced TC somewhat.

Que: Multiply two matrices using Strassen's Algorithm

$$A = \begin{bmatrix} 5 & 6 \\ -4 & 3 \end{bmatrix} ; \quad B = \begin{bmatrix} -7 & 6 \\ 5 & 9 \end{bmatrix}$$

Soln: find,

P, Q, R, S, T, U, V by using formula:

and then,

$C_{11} = -5$

$C_{12} = 84$

$C_{21} = 43$

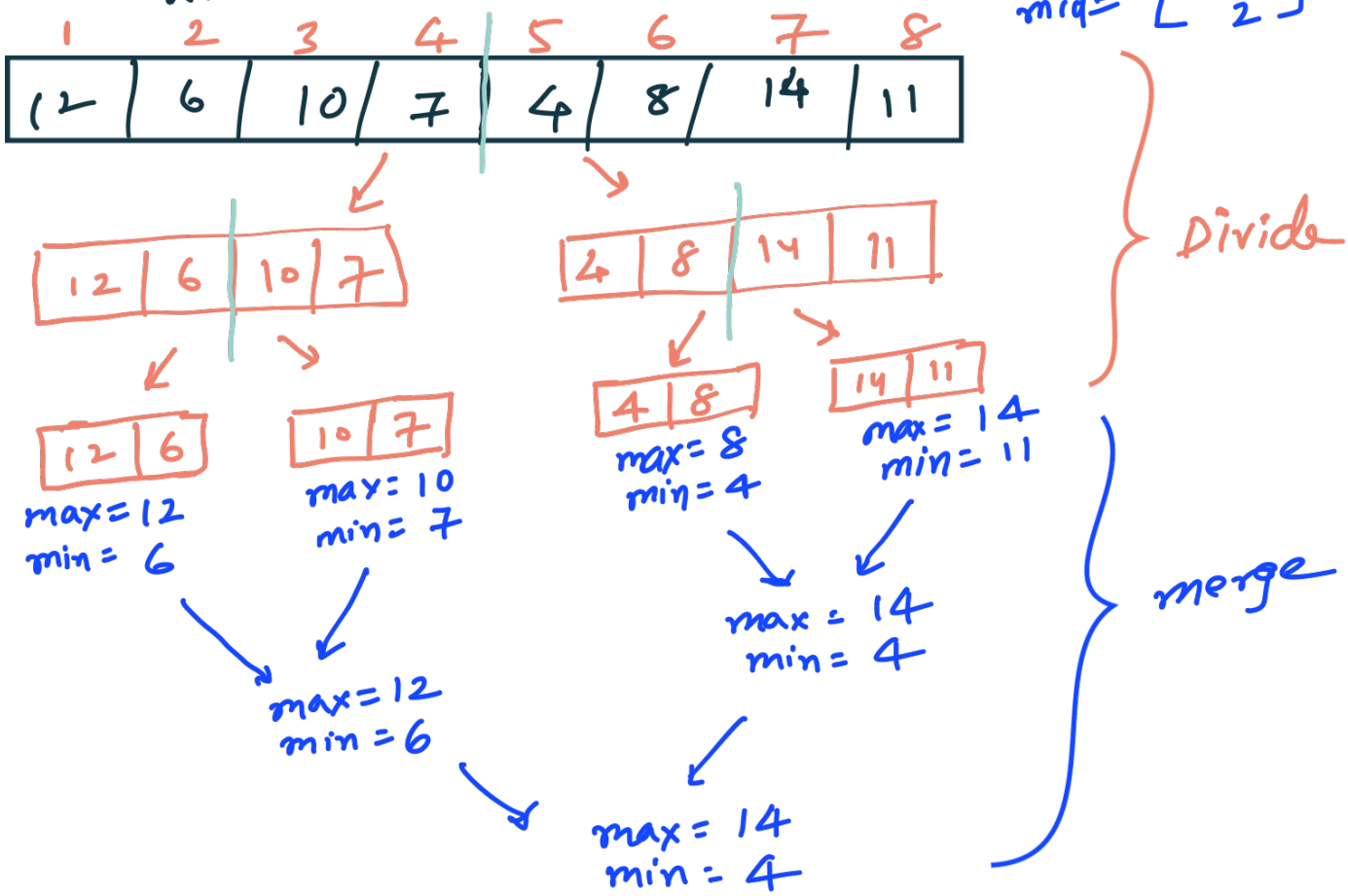
$C_{22} = 3$

$$\boxed{\begin{bmatrix} -5 & 84 \\ 43 & 3 \end{bmatrix}}$$

Ans

# MAX - MIN PROBLEM

→ find the maximum and minimum element from an arr.




Recurrence Relation :-

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

- dividing into two equal parts
- cost of comparing & merging in each iteration

**Faculty:** Suraj Mourya

```
DAC Max_Min(A, i, j, max, min)
{
    int mid;
    if (i==j-1)
    {
        if (A[i] < A[j])
            max = A[j]; min = A[i];
        else
            max = A[i]; min = A[j];
    }
    else
    {
        mid = (i+j)/2
        DAC Max_Min(A, i, mid, max, min)
        DAC Max_Min(A, mid+1, j, max, min)
    }
}
```

$$T(n) = \begin{cases} 1 & n = 2 \\ 2T(n/2) + 2 & n > 2 \end{cases}$$

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

$T(n/2) = 2T(n/4) + 2$

$\Rightarrow T(n) = 2[2T(n/4)+2] + 2 = 2^2 T(n/4) + 2^2 + 2$

$= 2^2[2T(n/8)+2] + 2^2 + 2$

$= 2^3 T(n/8) + 2^3 + 2^2 + 2$

$\vdots$

$= 2^k T(n/2^k) + [2 + 2^2 + 2^3 + 2^4 + \dots + 2^k]$

$= 2^k \times 1 + [2(2^k - 1)]$ $\quad \because \frac{a(r^n-1)}{r-1}$

$= 2^k + 2 \times 2^k - 2$

$= \frac{n}{2} + 2 \times \frac{n}{2} - 2$

$\because \frac{n}{2^k} = 2$

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

$$= \frac{n}{2} + n - 2 = \boxed{\frac{3n}{2} - 2} = \boxed{O(n)}$$

## By iterative method:-

```
FIND MAX MIN (A, n)
{
  max = min = A[0];
for (i=1, i<n; i++)
 {  if (max < A[i])
        max = A[i];
    else if (min > A[i])
        min = A[i];
  }
  return (max, min);
}
```

→ (n-1) times iterates

} → (n-1) times comparison

} → (n-1) times comparison

Note: Since both statement if and else if is executing (n-1) times in each iteration.

So,

$$TC \Rightarrow 2(n-1) \Rightarrow 2n - 2 = \boxed{O(N)}$$

Note: So, precisely if you observe then you will find that iterative method takes "2n" TC whereas Divide and conquer is taking "1.5n" TC.

# Quick Sort questions:

**What is the average-case time complexity of Quick Sort?**

- a) O(n)
- b) O(n log n)
- c) O(n^2)
- d) O(log n)

**In the Quick Sort algorithm, what is the main role of the pivot element?**

- a) To merge sorted arrays
- b) To divide the array into two sub-arrays
- c) To find the median
- d) To count the elements in the array

**Which of the following scenarios would result in the worst-case time complexity of Quick Sort?**

- a) Choosing the first element as the pivot
- b) Choosing a random element as the pivot
- c) Choosing the median of three elements as the pivot
- d) Choosing the smallest or largest element as the pivot

**In a sorted array of 100 elements, what is the maximum number of comparisons required to find an element using Binary Search?**

- a) 8
- b) 20
- c) 50
- d) 100

**What is the main idea behind the Divide and Conquer approach?**

- a) Solving the problem directly
- b) Dividing the problem into smaller subproblems, solving them independently, and combining their solutions
- c) Using a single loop to solve the problem
- d) Storing solutions to subproblems to avoid redundant work

GATE 2014

Let P be a QuickSort Program to sort numbers in ascending order using the first element as pivot. Let t1 and t2 be the number of comparisons made by P for the inputs {1, 2, 3, 4, 5} and {4, 1, 5, 3, 2} respectively. Which one of the following holds?

**(A)** t1 = 5
**(B)** t1 < t2
**(C)** t1 > t2
**(D)** t1 = t2

GATE 1994

Algorithm design technique used in quicksort algorithm is?

**(A)** Dynamic programming
**(B)** Backtracking
**(C)** Divide and conquer
**(D)** Greedy method

GATE 1992

Assume that the last element of the set is used as partition element in Quicksort. If n distinct elements from the set [1.....n] are to be sorted, give an input for which Quicksort takes maximum time.

# MERGE SORT

**Assume that a mergesort algorithm in the worst case takes 30 seconds for an input of size 64. Which of the following most closely approximates the maximum input size of a problem that can be solved in 6 minutes?**

**a) 256**
**b) 512**
**c) 1024**
**d) 2048**

**GATE 2015 (Shift-3)**

If one uses straight two-way merge sort algorithm to sort the following elements in ascending order:

20, 47, 15, 8, 9, 4, 40, 30, 12, 17

then the order of these elements after second pass of the algorithm is:

(A) 8, 9, 15, 20, 47, 4, 12, 17, 30, 40

(B) 8, 15, 20, 47, 4, 9, 30, 40, 12, 17

(C) 15, 20, 47, 4, 8, 9, 12, 30, 40, 17

(D) 4, 8, 9, 15, 20, 47, 12, 17, 30, 40

**ISRO Test Series**