UNIT-3: DIVIDE AND CONQUER ALGORITHM

Assistant Professor

IT-ICT Department, LJIET

prayag.patel@ljinstitutes.edu.in

Mr. Prayag Patel

Topics to be covered

- Introduction of Divide and Conquer Technique.
- Recurrence and Different methods to solve recurrence.
- Problem Solving using divide and conquer algorithm :
  1. Multiplying large Integers Problem
  2. Binary Search
  3. Merge Sort
  4. Quick Sort
  5. Max – Min Problem
  6. Matrix Multiplication
  7. Exponential

Introduction of Divide and Conquer Technique

- Many useful algorithms are **recursive** in structure.
- To solve a given problem, they call them selves recursively one or more times.
- These algorithms typically follow a **divide-and-conquer** approach.
- **Divide-and-conquer** is a top-down approach.

Divide and Conquer Technique

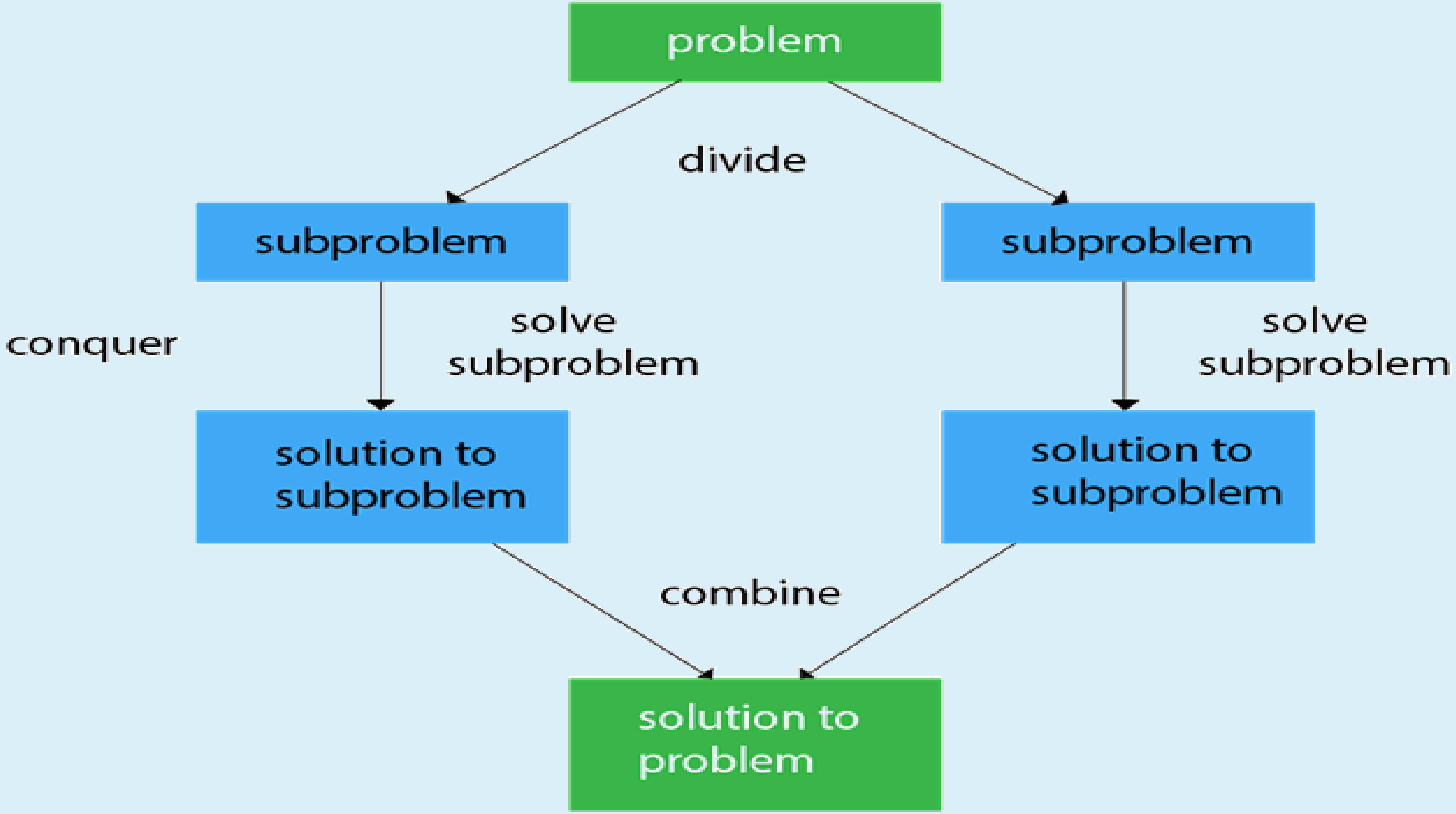
- The divide-and-conquer approach involves **three steps** at each level of the recursion:

  1. **Divide:** Break the problem into several sub problems that are similar to the original problem but smaller in size.

  2. **Conquer:** Solve the sub problems recursively. If the sub problem sizes are small enough, just solve the sub problems in a straight forward manner.

  3. **Combine:** Combine these solutions to create a solution to the original problem.

Divide and Conquer Technique



BINARY SEARCH

Assistant Professor

IT/ICT Department, LJIET

prayag.patel@ljinstitutes.edu.in

Mr. Prayag Patel

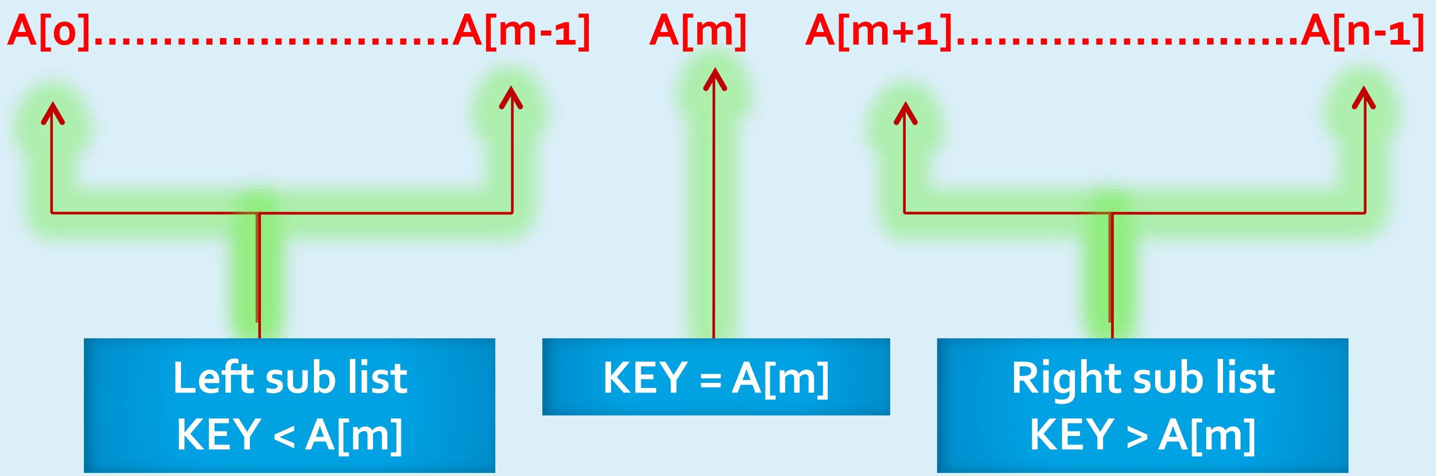
Binary Search

- Use Divide and conquer Technique.
- Data collection should be in the sorted form.
- An element which is to be searched is called KEY element.
- Find the middle element A[m].

Binary Search

- There are three conditions that needs to be tested.

1) If KEY = A[m] then desired element is present in the list at location m.
2) Otherwise if KEY < A[m] then search the **left sub list**.
3) Otherwise if KEY > A[m] then search the **right sub list.**



Binary Search - Example

Input: sorted array of integer values. **Key = 7**

[0]	[1]	[2]	[3]	[4]	[5]	[6]
3	6	7	11	32	33	53

low = 0

high = 6

Find middle point m

m = (low + high)/2
m = (0+6)/2
m = 3

Binary Search - Example

Key = 7

[0]	[1]	[2]	[3]	[4]	[5]	[6]
3	6	7	11	32	33	53

↑

m=3

[0]	[1]	[2]	[3]	[4]	[5]	[6]
3	6	7	11	32	33	53

↑

Check KEY = midpoint? 7 = A[3] ? 7 = 11 ? No

Binary Search - Example

Key = 7

[0]	[1]	[2]	[3]	[4]	[5]	[6]
3	6	7	11	32	33	53

Is Key < midpoint? 7 < 11 ? Yes.

[0]	[1]	[2]	[3]	[4]	[5]	[6]
3	6	7	11	32	33	53

Search the left sub list.

Binary Search - Example

Key = 7

[0]	[1]	[2]	[3]	[4]	[5]	[6]
3	6	7	11	32	33	53

low = 0

high = 2

Find middle point m

m = (low + high)/2

m = (0+2)/2

m = 1

Binary Search - Example

Key = 7

[0]	[1]	[2]	[3]	[4]	[5]	[6]
3	6	7	11	32	33	53

m = 1

[0]	[1]	[2]	[3]	[4]	[5]	[6]
3	6	7	11	32	33	53

Check KEY = midpoint?   7 = A[1] ? 7 = 6 ? No

Binary Search - Example

Key = 7

[0]	[1]	[2]	[3]	[4]	[5]	[6]
3	6	7	11	32	33	53

Check KEY < midpoint? 7 < A[1] ? 7 < 6 ? No

[0]	[1]	[2]	[3]	[4]	[5]	[6]
3	6	7	11	32	33	53

Check KEY > midpoint? 7 > A[1] ? 7 > 6 ? Yes

Binary Search - Example

Key = 7

[0]	[1]	[2]	[3]	[4]	[5]	[6]
3	6	7	11	32	33	53

Search the right sub list

[0]	[1]	[2]	[3]	[4]	[5]	[6]
3	6	7	11	32	33	53

low = 2
High = 2

Binary Search - Example

Key = 7

[0]	[1]	[2]	[3]	[4]	[5]	[6]
3	6	7	11	32	33	53

low = 2
High = 2

Find middle point m

m = (low + high)/2
m = (2+2)/2
m = 2

Binary Search - Example

Key = 7

[0]	[1]	[2]	[3]	[4]	[5]	[6]
3	6	7	11	32	33	53

m = 2

[0]	[1]	[2]	[3]	[4]	[5]	[6]
3	6	7	11	32	33	53

Check KEY = midpoint? 7 = A[2] ? 7 = 7 ? Yes

So KEY is present at A[2] location

Binary Search Algorithm

- **Algorithm: Binary search(A[0,…,n-1],Key)**

```
low ← 0
high ← n-1
While(low <= high) do
    {
        m ← (low + high)/2
        if (KEY = A[m]) then
                return m
        else if (KEY < A[m]) then
                high ← m-1
        else
                low ← m+1
    }
```

Binary Search Analysis

Worst case:

Recurrence Relation for Binary search

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

Time required to compare left or right sub list

One compare is made with middle element

$$T(1) = 1$$

Binary Search Analysis

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

$T(1) = 1$

$$T(n) = aT(n/b) + f(n)$$

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

$T(n)= \Theta(n^d \log n)$ if $a = b^d$

$a=1, \quad b=2, \quad d=0$

$1 = 2^0$ means $a = b^d$

so consider Case-2

Worst case and Average case

Thus, $T(n)= \Theta(n^d \log n)$

$= \Theta(n^0 \log n)$

So Time complexity is $\Theta(\log n)$

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

Best Case Time complexity is $\Theta(1)$

MERGE SEARCH

Assistant Professor
IT/ICT Department, LJIET
prayag.patel@ljinstitutes.edu.in
Mr. Prayag Patel

Merge Sort

- The merge sort is sorting algorithm that uses the divide and conquer strategy.
- Merge sort on an input array with n elements consists of three steps.

1) Divide : Partition array into two sub list S1 and S2 with n/2 elements each.
2) Conquer: Then sort Sub list S1 and Sub list S2
3) Combine: Merge Sub list S1 and Sub list S2 into unique sorted group.

Merge Sort Example

Unsorted Array

724	521	2	98	529	31	189	451
1	2	3	4	5	6	7	8

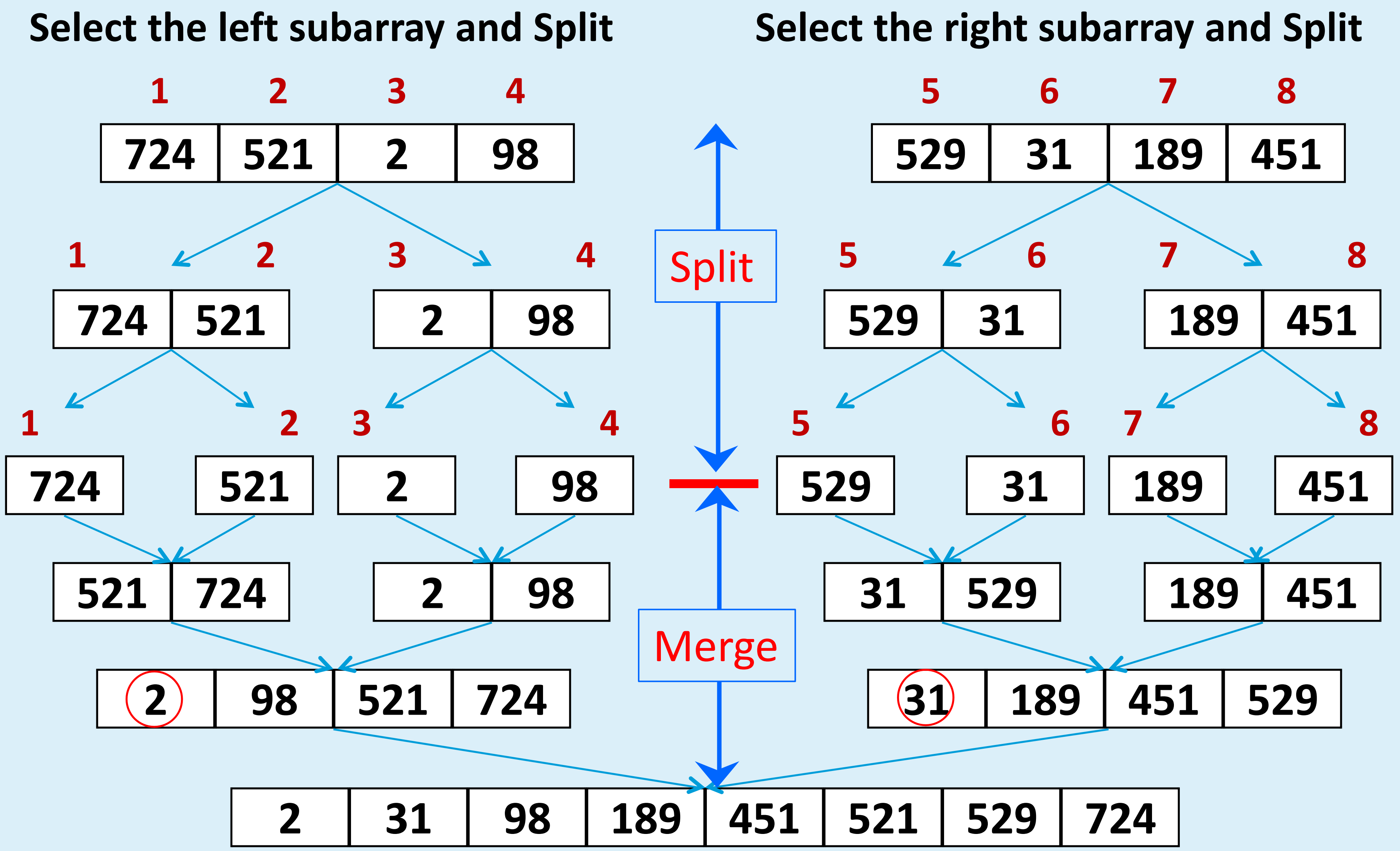
Step 1: Split the selected array

1	2	3	4	5	6	7	8
724	**521**	**2**	**98**	**529**	**31**	**189**	**451**

724	**521**	**2**	**98**
1	2	3	4

529	**31**	**189**	**451**
5	6	7	8

Merge Sort Example



Merge Sort Algorithm

```
Algorithm Merge_Sort(int A[0...n-1], low, high)
    if (low < high) then
     {
       mid ← (low + high) / 2
       Merge_Sort(A, low, mid)
       Merge_Sort(A, mid + 1, high)
       Combine(A, low, mid, high)
      }
```

```
Combine(A[0…..n-1], low, mid, high)
 {
    k ← low
    i ← low
    j ← mid + 1
  while (i <= mid and j <= high) do
    {
       If (A[i] <= A[j]) then
        {
           temp[k]  ← A[i]
            i  ← i+1
            k  ← k+1
        }
       else
        {
           temp[k]  ← A[j]
            j  ← j+1
            k  ← k+1
        }
while (i <= mid) do
       {
           temp[k]  ← A[i]
            i  ← i+1
            k  ← k+1
       }
   while (j <= high) do
       {
           temp[k]  ← A[j]
            j  ← j+1
            k  ← k+1
       }
}
```

```
Algorithm Merge_Sort(int A[0…n-1], low, high)
    if (low < high) then
     {
       mid ← (low + high) / 2
       Merge_Sort(A, low, mid)
       Merge_Sort(A, mid + 1, high)
       Combine(A, low, mid, high)
      }
```

Combine(A[0…..n-1], low, mid, high)

```
 {
    k ← low
    i ← low
    j ← mid + 1
  while (i <= mid and j <= high) do
     {
       If (A[i] <= A[j]) then
        {
          temp[k]  ← A[i]
           i  ← i+1
           k  ← k+1
        }
       else
        {
          temp[k]  ← A[j]
           j  ← j+1
           k  ← k+1
        }
while (i <= mid) do
     {
        temp[k]  ← A[i]
         i  ← i+1
         k  ← k+1
     }
  while (j <= high) do
     {
        temp[k]  ← A[j]
         j  ← j+1
         k  ← k+1
     }
}
```

Merge Sort Analysis

Recurrence relation:

$T(n) = T(n/2) + T(n/2) + c\,n \quad \text{if } n>1$

$T(1) = 0$

Time required by left sub list to get sorted

Time required by right sub list to get sorted

Time taken for combining two sub lists

- Each recursive call focuses on n/2 elements of the list.
- After two recursive calls one call is made to combine two sub list.

Merge Sort Analysis

Recurrence relation:

T(n) = 2T(n/2) + c n   if n>1

T(1) = 0

$$T(n) = aT(n/b) + f(n)$$

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

$$T(n) \in \begin{cases} \Theta(n^d), & a < b^d \\ \Theta(n^d \log n), & a = b^d \\ \Theta(n^{\log_b a}), & a > b^d \end{cases}$$

Case - 1

Case - 2

Case - 3

a=2,   b=2,   d=1

$2 = 2^1$ means $a = b^d$

so consider Case-2

Thus, $T(n)=\Theta(n^d \log n)$

$= \Theta(n^1 \log n)$

$= \Theta(n \log n)$

Consider f(n) is $\Theta(n^d)$

So Time complexity is $\Theta(n \log n)$

QUICK SORT

Assistant Professor

IT/ICT Department, LJIET

prayag.patel@ljinstitutes.edu.in

Mr. Prayag Patel

Quick Sort

- The Quick sort is sorting algorithm that uses the divide and conquer strategy.
- Quick sort on an input array with n elements consists of three steps.

1) Divide : Split the array into two sub arrays that each element in the left sub array is less than or equal the middle element and each element in the right sub array is greater than the middle element.
2) Conquer: Recursively solve the two sub arrays.
3) Combine: Combine all the sorted elements in a group to form a list of sorted elements.

Quick Sort

- Quick sort chooses the first element as a pivot element, a **lower Bound (low) is the first index** and an **upper bound (high) is the last index**.
- The array is then **partitioned** on either side of the pivot.
- Elements are moved so that, those **greater** than the pivot are shifted to its **right** whereas the others are shifted to its **left.**
- Each Partition is **internally sorted recursively.**

Pivot Element

0	1	2	3	4	5	6	7	6	7
42	**23**	**74**	**11**	**65**	**58**	**94**	**36**	**99**	**87**

LB **UB**

Quick Sort Example

0	1	2	3	4	5	6	7	8	9
42	23	74	11	65	58	94	36	99	87
i									j

low=i= 0,

High=j= 9

Pivot =A[low]= 42

Swap

42	23	36	11	65	58	94	74	99	87
	i								j

Swap

11	23	36	42	65	58	94	74	99	87
		i					j		

A[i]<= Pivot
i = i + 1

Step-1

A[j] > Pivot
j = j - 1

Step-2

Check if(i<=j)

Step-3

YES

swap(A[i],A[j])

NO

swap(A[low],A[j])

Quick Sort Example

Low High

0	1	2	3	4	5	6	7	8	9
11	23	36	42	65	58	94	74	99	87

11	23	36
i		j

Low High

11	23	36	42	65	58	94	74	99	87

23	36
i	j

11	23	36	42	65	58	94	74	99	87

Quick Sort Example

				Low					High
				4	5	6	7	8	9
11	23	36	42	65	58	94	74	99	87

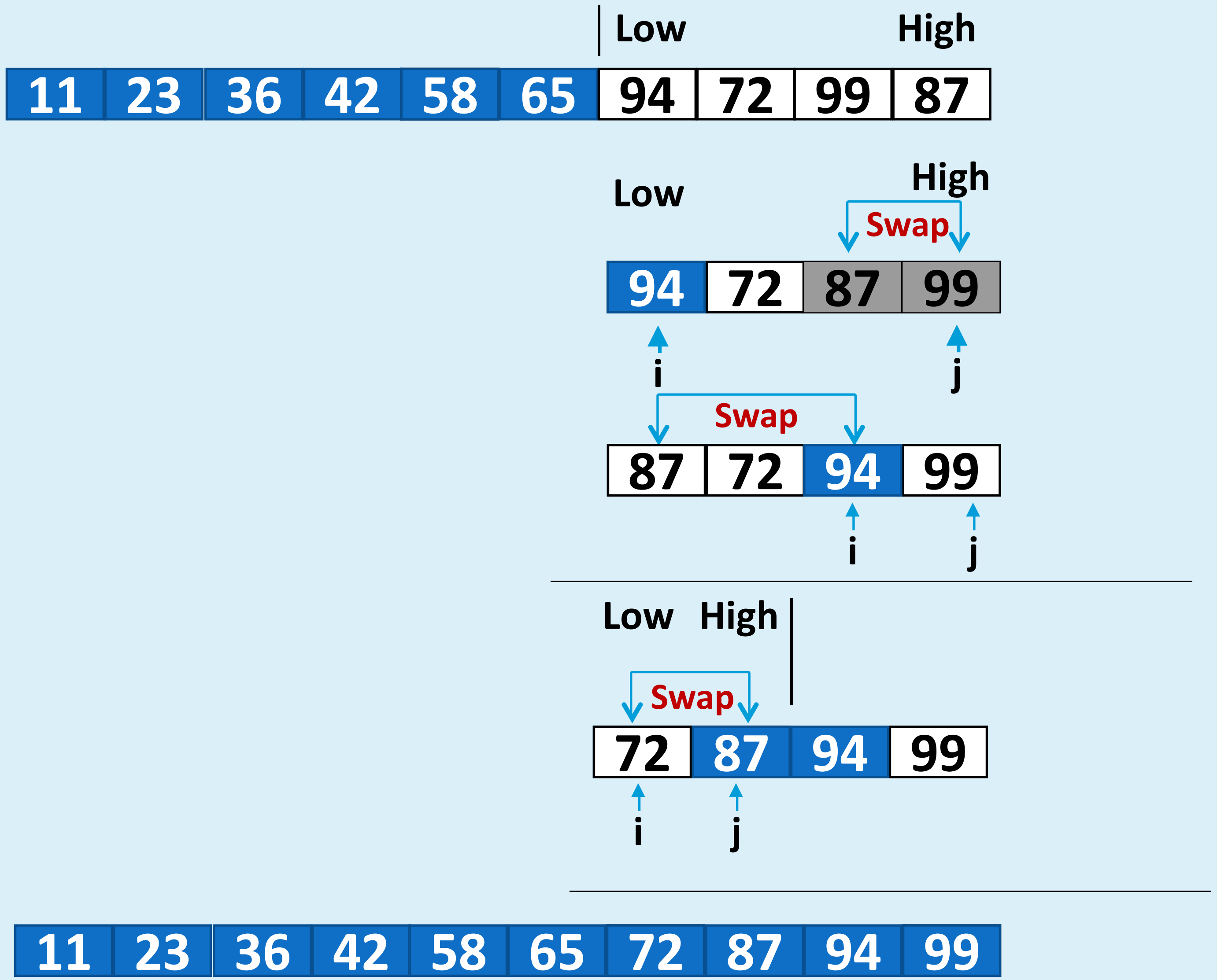
Swap

58	65	94	72	99	87
i					j

58	65	94	72	99	87

						Low			High
11	23	36	42	58	65	94	72	99	87

Quick Sort Example



Quick Sort Algorithm

```
Algorithm Quick(A[0…n-1], low, high)
{
    if (low < high) then
      {
         m = Partition(A[low…..high])
         Quick(A[low….m-1], low, m-1)
         Quick(A[m+1….high], m+1, high)
      }
}
```

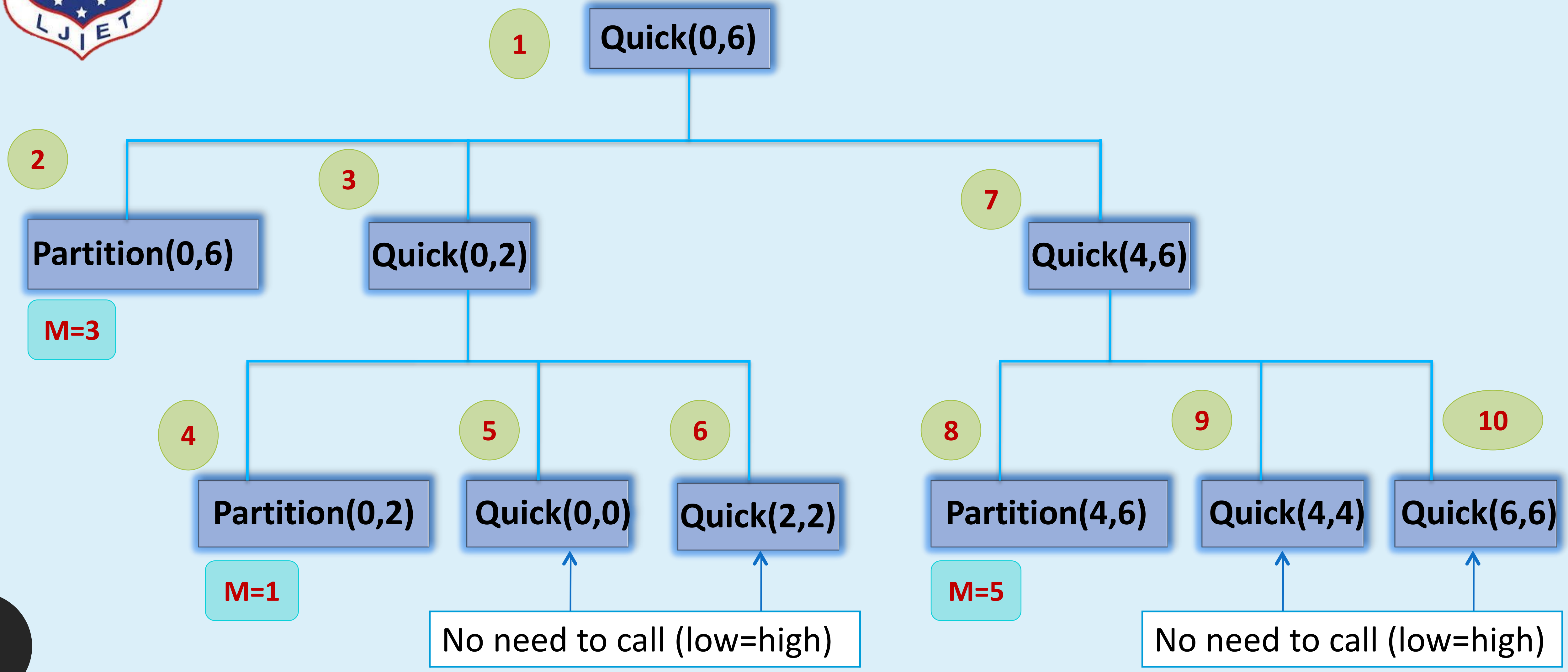
```
Partition(A[low…..high])
{
    pivot ← A[low]
     i ← low
     j ← high
    while(i <= j) do
     {
        while (A[i] <= pivot) do
              i ← i + 1
        while (A[j] > pivot) do
              j ← j - 1
        if ( i <= j) then
              swap(A[i], A[j])
     }
 swap(A[low], A[j])
return j
}
```

Pivot Element

0	1	2	3	4	5	6	7	6	7
42	23	74	11	65	58	94	36	99	87

Low | High

Quick Sort - Recursive calls



Quick Sort Best case Analysis

- If the array is always partitioned at the middle then it brings the best case.

Recurrence relation:

$T(n) = T(n/2) + T(n/2) + n \quad \text{if } n>1$

$T(1) = 0$

Time required by left sub list to get sorted

Time required by right sub list to get sorted

Time required for partitioning the sub array

Quick Sort Best case Analysis

Recurrence relation:

T(n) = 2T(n/2) + n if n>1

T(1) = 0

$$T(n) = aT(n/b) + f(n)$$

$$T(n) \in \begin{cases} \Theta(n^d), & a < b^d \\ \Theta(n^d \log n), & a = b^d \\ \Theta(n^{\log_b a}), & a > b^d \end{cases}$$

Case - 1

Case - 2

Case - 3

Consider f(n) is $\Theta(n^d)$

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

a=2, b=2, d= 1

$2 = 2^1$ means $a = b^d$

so consider Case-2

Thus, $T(n) = \Theta(n^d \log n)$

$= \Theta(n^1 \log n)$

$= \Theta(n \log n)$

So Time complexity is $\Theta(n \log n)$

Quick Sort Best case Analysis

- We can get an idea of average case by considering the case when partition puts O(n/9) elements in one set and O(9n/10) elements in other set.

Following is recurrence for this case.

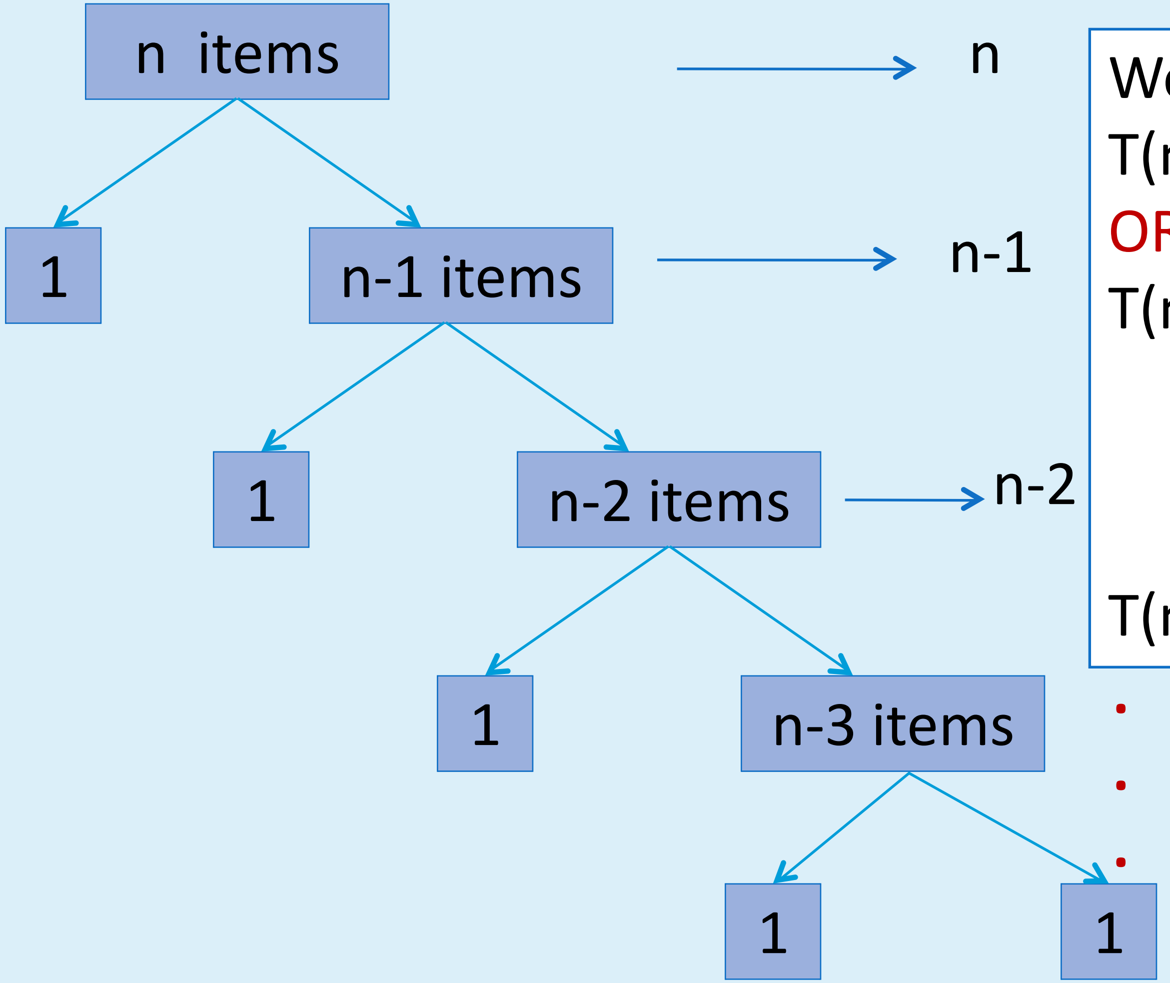
$T(n) = T(n/9) + T(9n/10) + (n)$

Solution of above recurrence is also O(n log n)

So Average case Time complexity is Θ(n log n)

Quick Sort Algorithm

- The worst case of quick sort occurs when the array is partitioned into one sub array with n-1 elements and other with 0 element.



We can write as

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

OR

$T(n)= n +(n-1)+(n-2)+\ldots+2+1$

$= n(n+1)/2$

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

$= n^2/2 + n/2$

$T(n) = \Theta(n^2)$

So worst case Time complexity is $\Theta(n^2)$

THANK YOU FOR WATCHING!

Mr. Prayag Patel

prayag.patel@ljinstitutes.edu.in