# UNIT II
# DIVIDE AND CONQUER

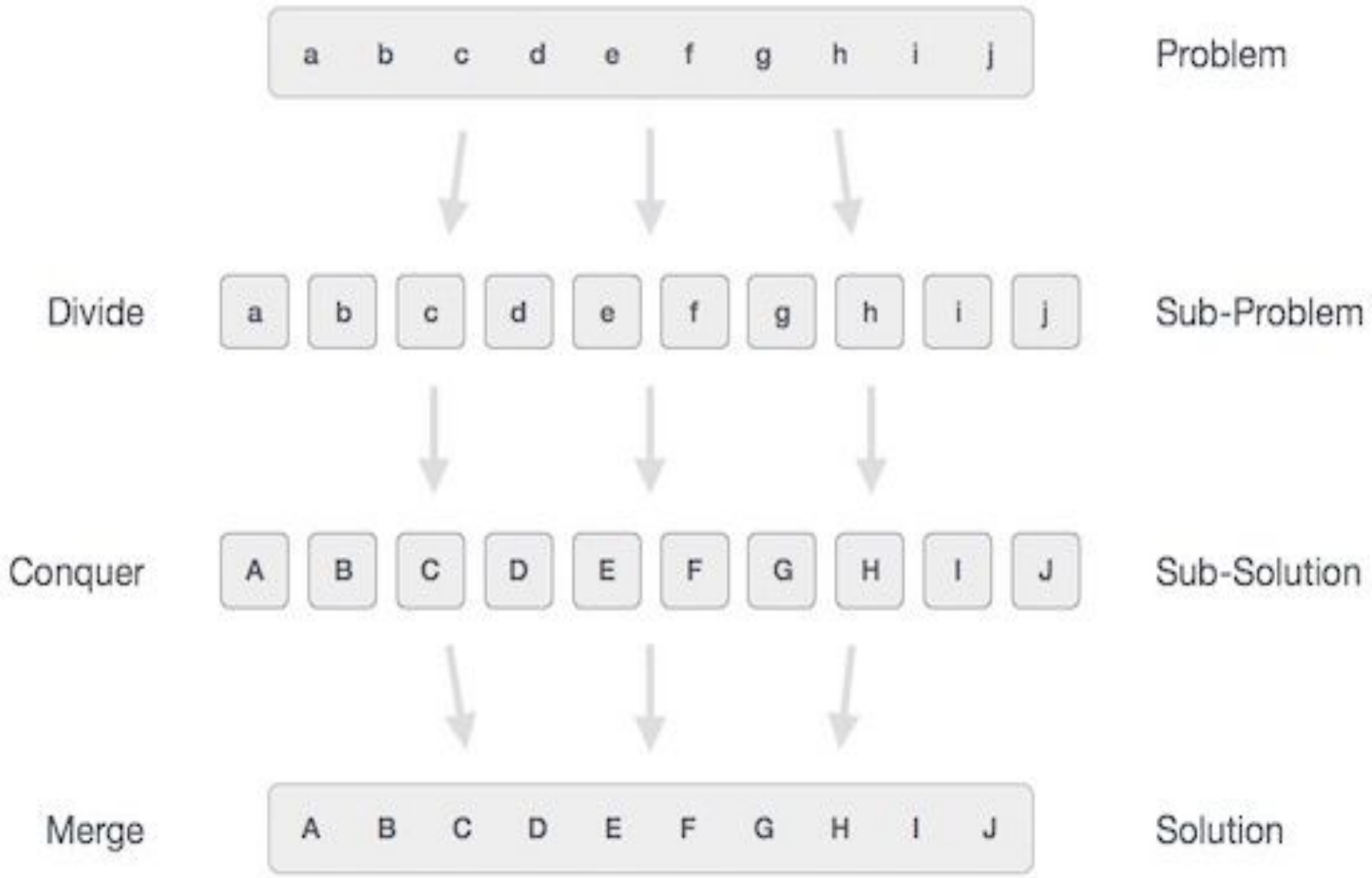
## Session – 9

# Syllabus

- **Introduction, Binary Search** - Merge sort and its algorithm analysis - Quick sort and its algorithm analysis - Strassen's Matrix multiplication - Finding Maximum and minimum - Algorithm for finding closest pair - Convex Hull Problem

# Introduction

- Divide / Break
- Conquer / Solve
- Merge / Combine


a b c d e f g h i j
Problem
Divide
a b c d e f g h i j
Sub-Problem
Conquer
A B C D E F G H I J
Sub-Solution
Merge
A B C D E F G H I J
Solution

# BINARY SEARCH ALGORITHM

```
Binarysearch(a[n], key)
start=0;  end=n-1;
While(start<=end)
{
      mid=(start+end)/2;
            return mid
      if(a[mid]==key)
            return mid;
      else if(a[mid]<key)
            end=mid-1;
            else
            start=mid+1;
}
return -1;
```

# ANALYSIS

**Worst case analysis:** The key is not in the array

Let T(n) be the number of comparisons done in the worst case for an array of size n. For the purposes of analysis, assume n is a power of 2, ie n = $2^k$.

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

$= 2 + 2 + T(\frac{n}{2^2})$ // 2nd iteration

$= 2 + 2 + 2 + T(n/2^3)$ // 3rd iteration

...

$= i * 2 + T(n/2^i)$ // ith iteration

... $= k * 2 + T(1)$

Note that k = logn, and that T(1) = 2.

So T(n) = 2logn + 2 = O(logn)

# Using Master's theorem

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

**T(n) = aT(n/b) + f(n) where a >= 1 and b > 1**

a = 1 b = 2 $f(n) = 1 = n^0$

1. If $f(n) < O(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$.
3. If $f(n) > \Omega(n^{\log_b a})$, and f(n) satisfies the regularity condition, then $T(n) = \Theta(f(n))$.

Calculate $n^{\log_b a} = n^{\log_2 1} = n^0$

Compare with f(n). Since $f(n) < n^{\log_b a}$

i.e. $n^0 = n^0$

Case 3 is satisfied hence complexity is given as **$T(n) = \Theta(f(n)) = \Theta(n^0 \log n) = \Theta(\log n)$**

# Worksheet No. 9

# UNIT II
# DIVIDE AND CONQUER

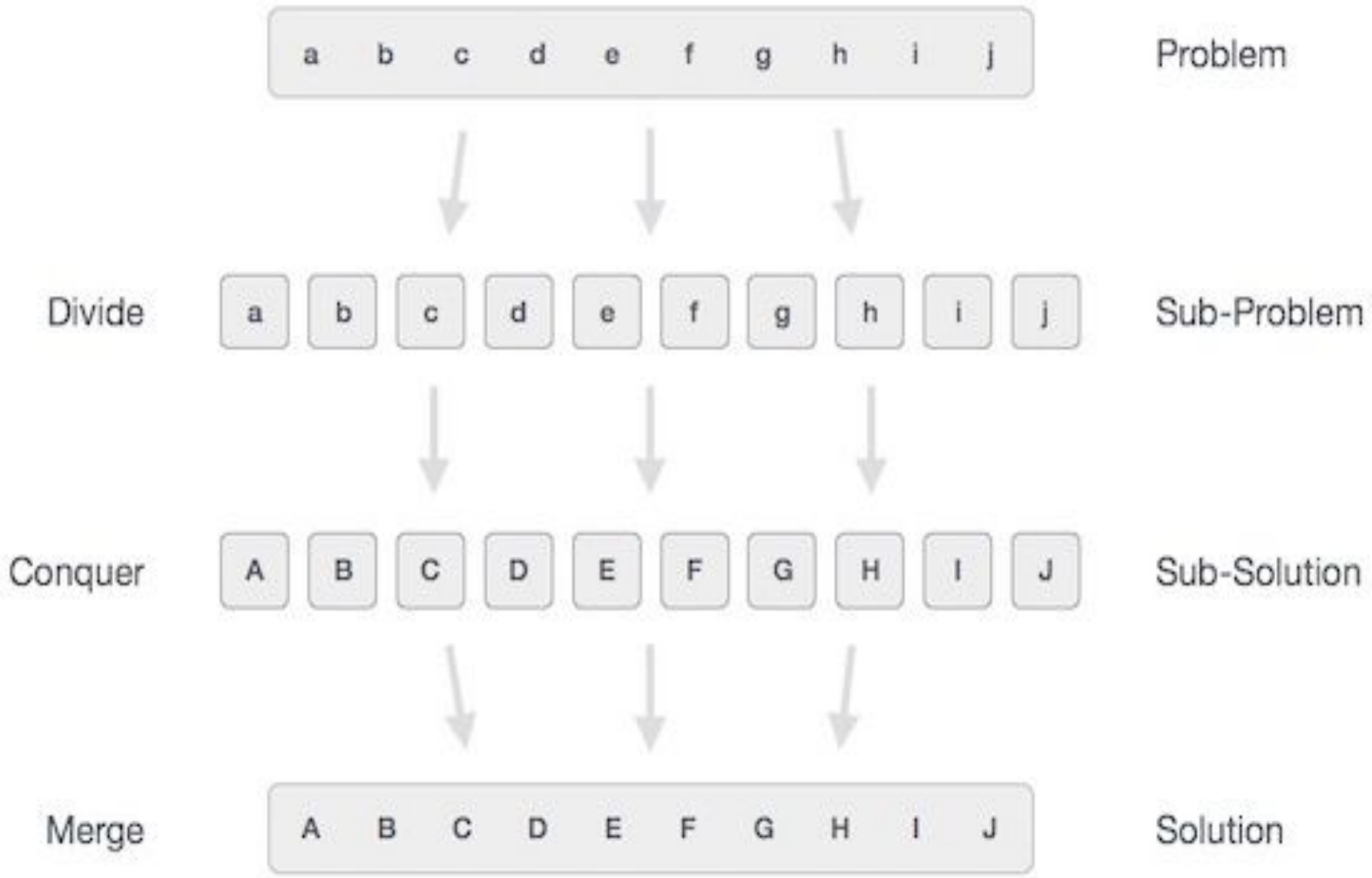
## Session – 10

# Syllabus

- Introduction, Binary Search - **Merge sort and its algorithm analysis** - Quick sort and its algorithm analysis - Strassen's Matrix multiplication - Finding Maximum and minimum - Algorithm for finding closest pair - Convex Hull Problem

# Introduction

- Divide / Break
- Conquer / Solve
- Merge / Combine


a b c d e f g h i j
Problem
Divide
a b c d e f g h i j
Sub-Problem
Conquer
A B C D E F G H I J
Sub-Solution
Merge
A B C D E F G H I J
Solution

# Merge Sort algorithm

- Merge sort is based on Divide and conquer method.
- It takes the list to be sorted and divide it in half to create two unsorted lists.
- The two unsorted lists are then sorted and merged to get a sorted list.
- The two unsorted lists are sorted by continually calling the merge-sort algorithm; we eventually get a list of size 1 which is already sorted. The two lists of size 1 are then merged.

**Steps using Divide and Conquer strategy**

- **Step 1** − if it is only one element in the list it is already sorted, return.
- **Step 2** − divide the list recursively into two halves until it can no more be divided.
- **Step 3** − merge the smaller lists into new list in sorted order.

**MergeSort(arr[], l, r)**

If r > l

1. Find the middle point to divide the array into two halves:

   middle m = (l+r)/2

2. Call mergeSort for first half:

   Call mergeSort(arr, l, m)

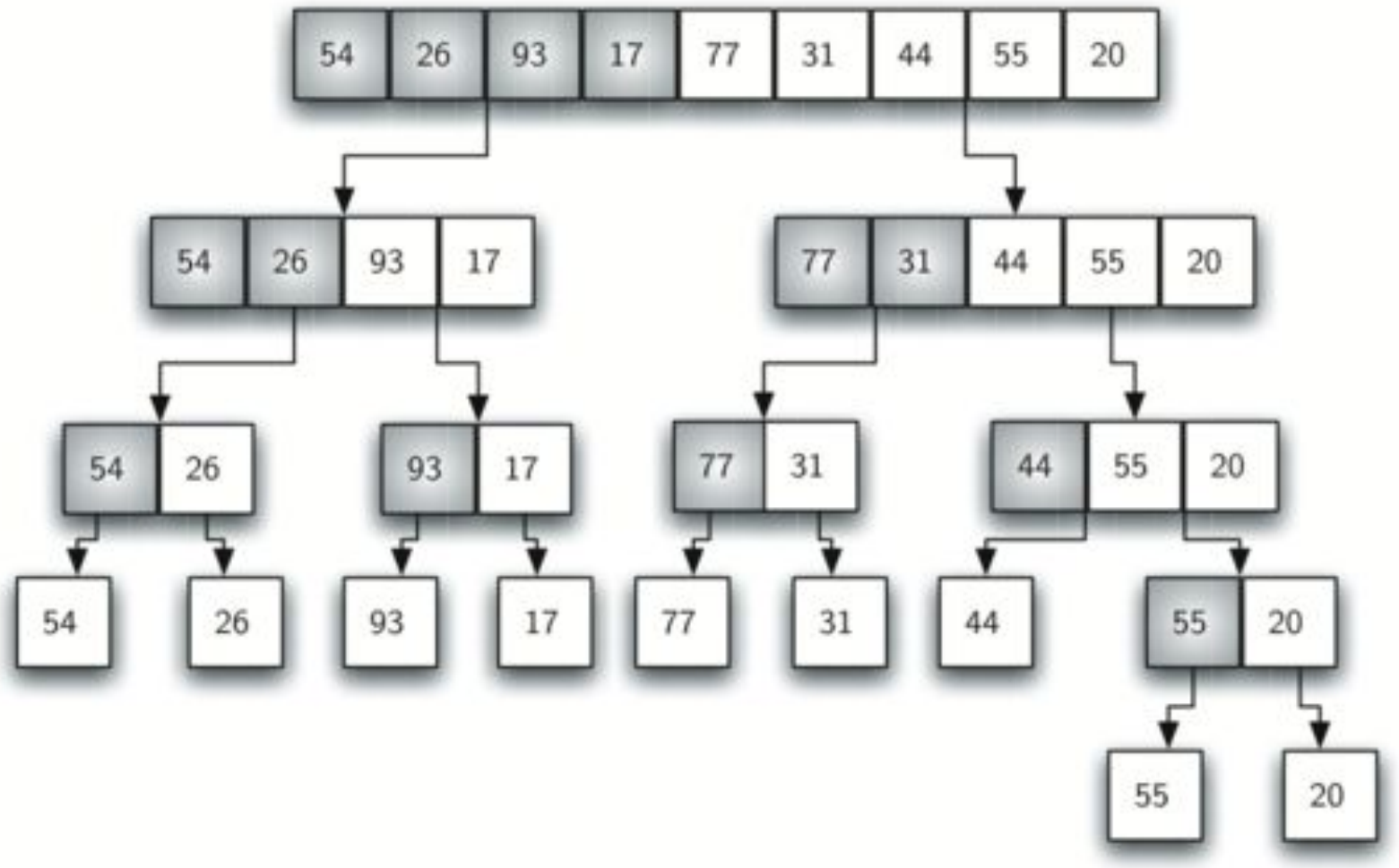
3. Call mergeSort for second half:

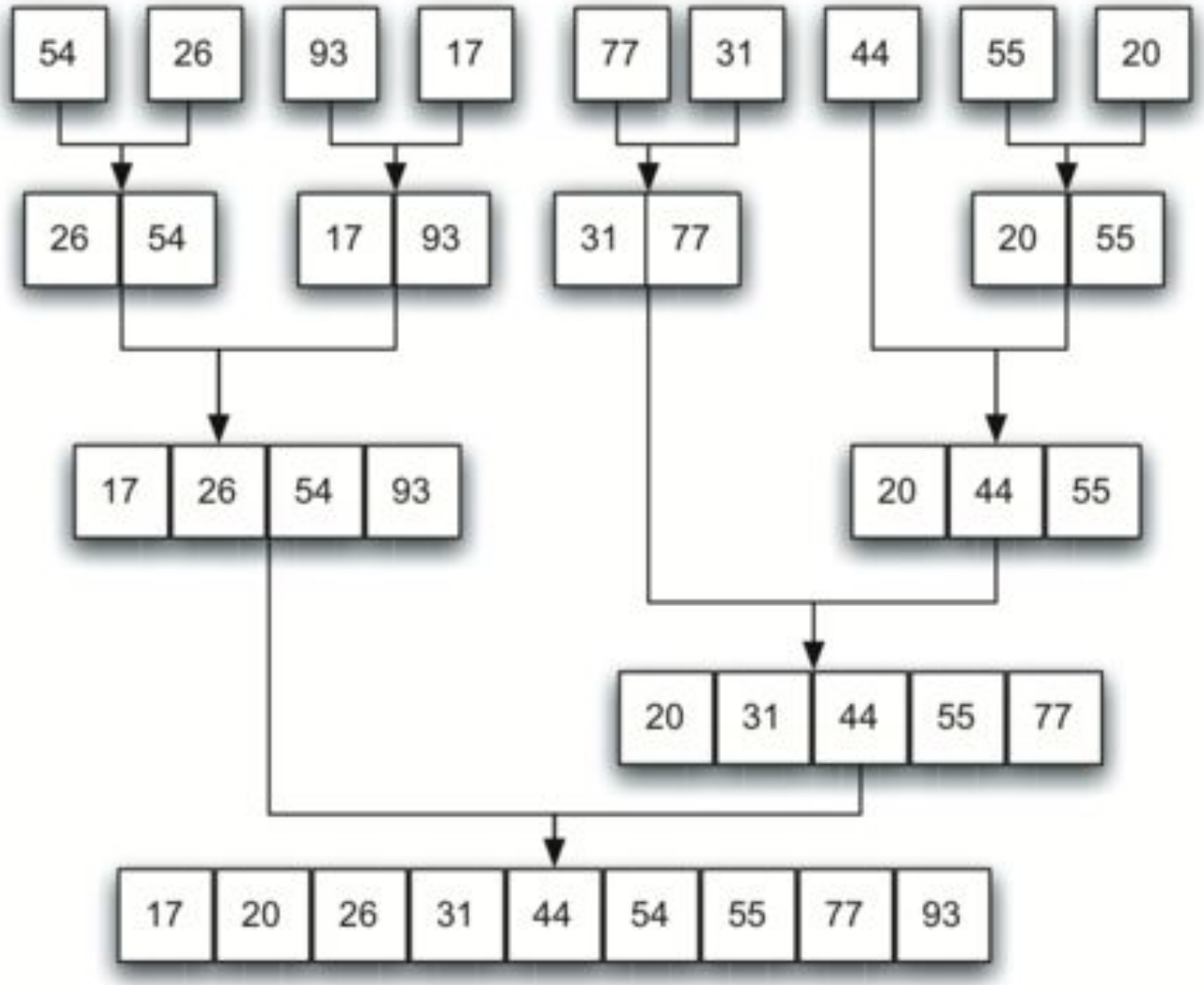
   Call mergeSort(arr, m+1, r)

4. Merge the two halves sorted in step 2 and 3:

   Call merge(arr, l, m, r)

# Mergesort(A,p,r)




54
26
93
17
77
31
44
55
20
26
54
17
93
31
77
20
55
17
26
54
93
20
44
55
20
31
44
55
77
17
20
26
31
44
54
55
77
93

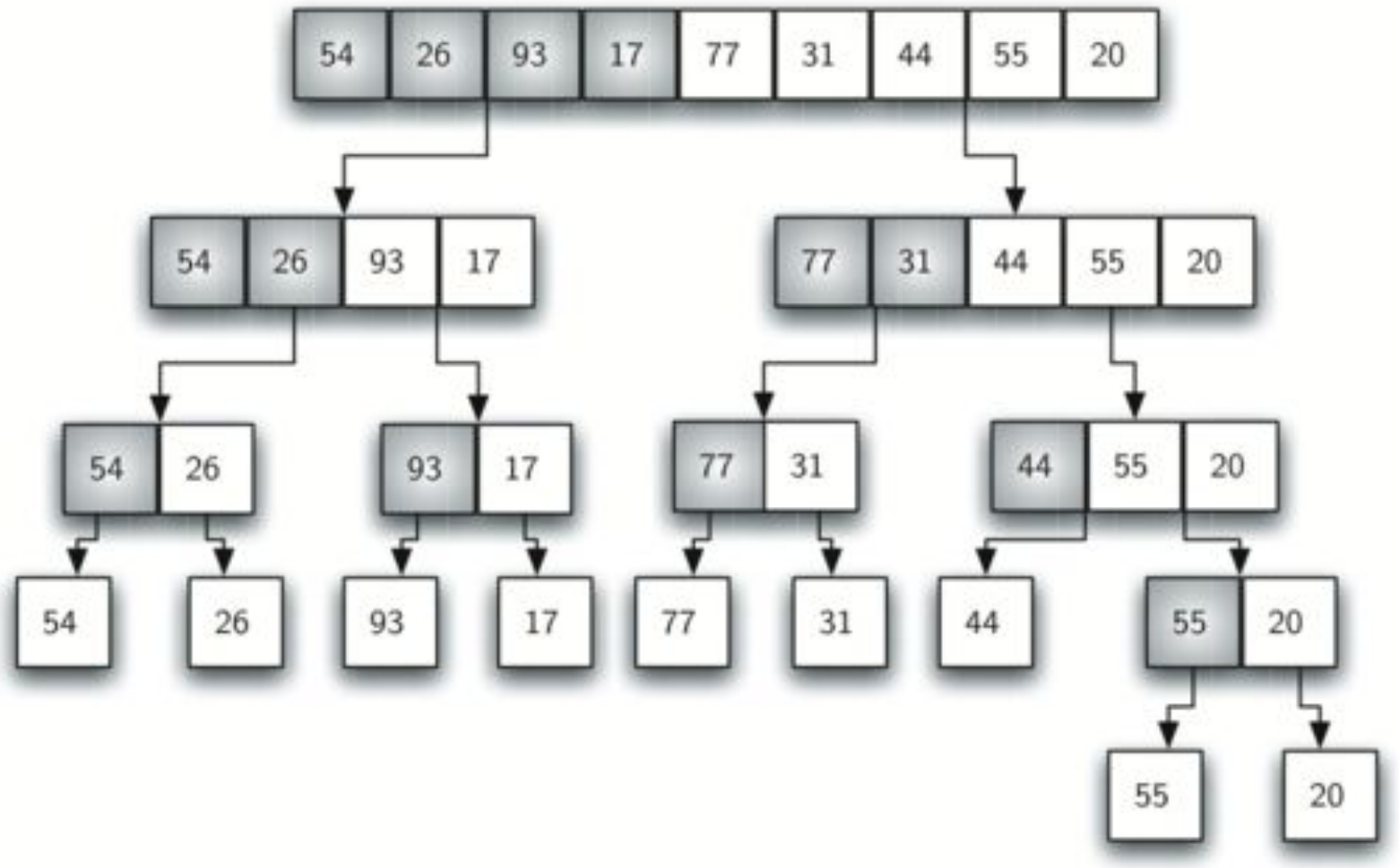
# Mergesort() function

**MERGE-SORT (*A, p, r)***

1. WHILE *p < r // Check for base case*
2. q *= FLOOR[(p + r)/2] // Divide step*
3. MERGE-SORT (A, *p, q)*
4. MERGE-SORT (A, *q + 1, r)*
5. MERGE (A, *p, q+1, r)*

- **MergeSort() function / Split step**
- We find the middle index using start and end index of the input array and again invoke the same function with two parts one starting from start to mid and other being from mid+1 to end.
- Once base condition is hit, we start winding up and call merge function.

# Mergesort(A,p,r)



# Merge() function

- MERGE (A, p, q, r ) //A –array, p – start index, q – middle,

r – last index

//Merges two subarrays of arr[].First subarray is arr[p..q] Second subarray is arr[q+1..r]

1. n1 ⟵ q − p + 1

2. n2 ⟵ r − q

// Create temperary arrays L[1 .. n1+1] and R[1 ..n2+1]

3. **FOR i ⟵ 1 to n1**

4. **do L[i] ⟵ A[p + i − 1]**

5. **FOR j ⟵ 1 TO n2**

6. **DO R[j] ⟵ A[q + j ]**

7. L[n1 + 1] ⟵ ∞

8. R[n2 + 1] ⟵ ∞

9. i ← 1

10. j ← 1

11. for **k ← p to r**

12. Do if **L[i ] ≤ R[ j]**

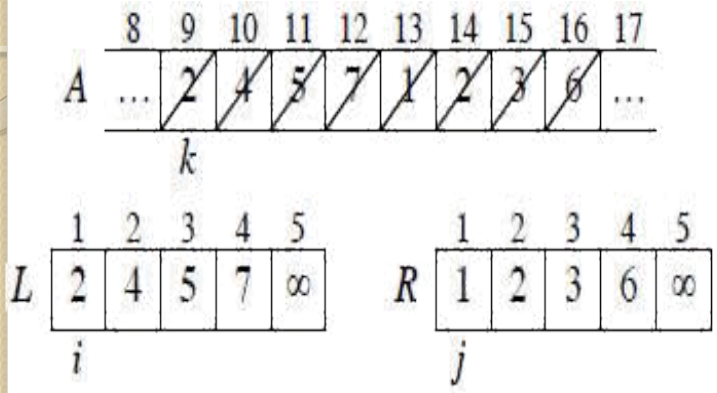
13. then **A[k] ← L[i]**

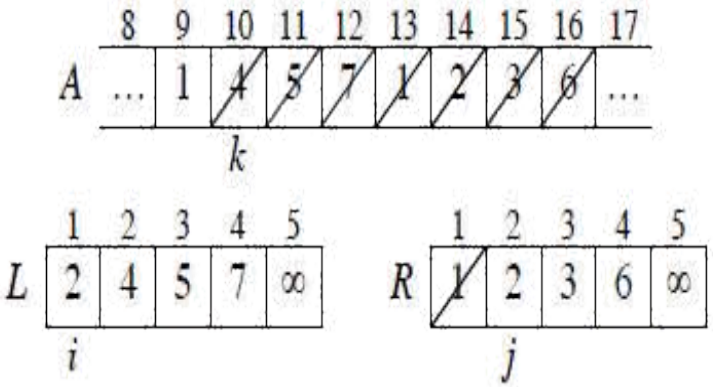
14. i ← i + 1

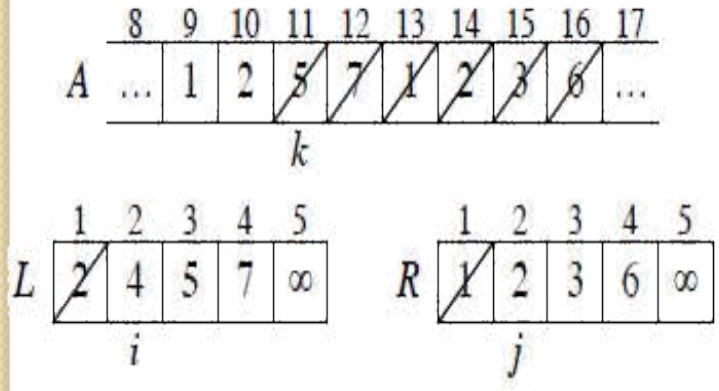
15. else **A[k] ← R[j]**

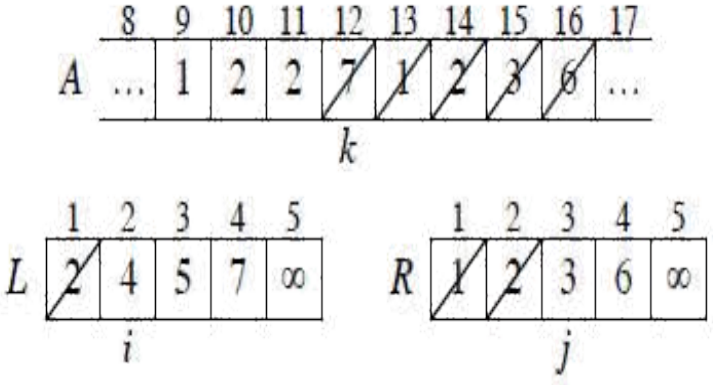
16. j ← j + 1

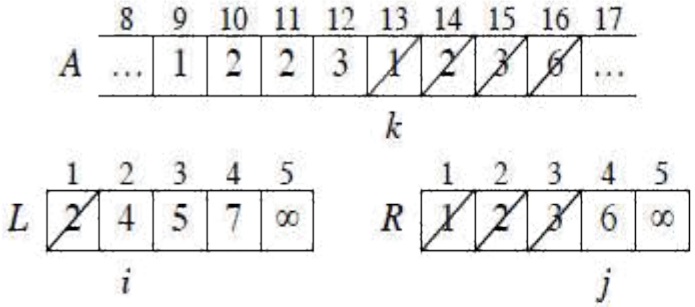
- **Merge() function / Merge step**
- Merge function merges two sub arrays (one from start to mid and other from mid+1 to end) into a single array from start to end. This array is then returned to upper calling function which then again sort two parts of array.

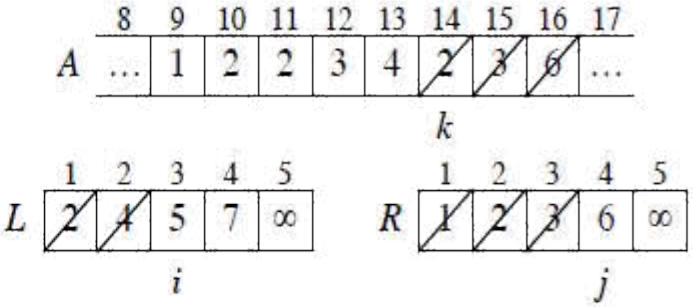

8 9 10 11 12 13 14 15 16 17
A … 2 4 5 7 1 2 3 6 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j

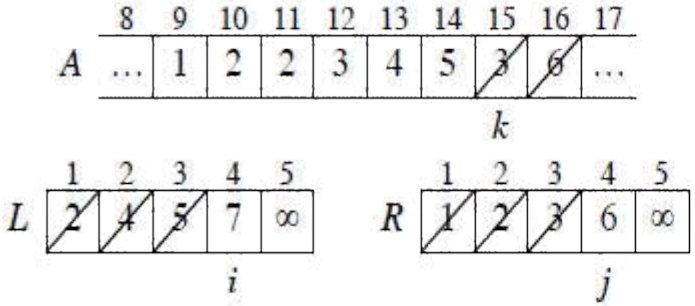

8 9 10 11 12 13 14 15 16 17
A … 1 4 5 7 1 2 3 6 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j

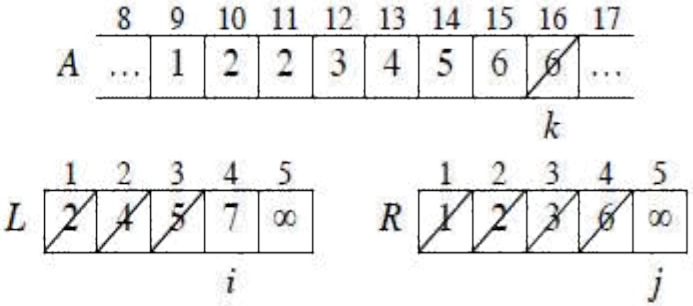

8 9 10 11 12 13 14 15 16 17
A … 1 2 5 7 1 2 3 6 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j

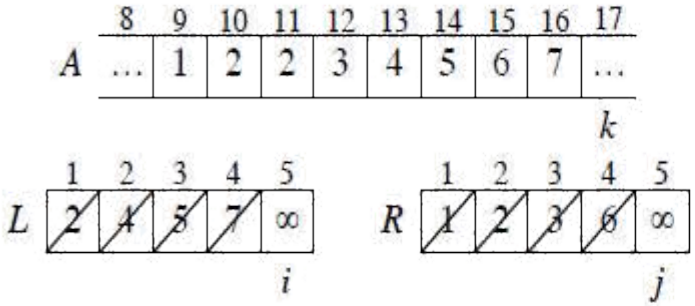

8 9 10 11 12 13 14 15 16 17
A … 1 2 2 7 1 2 3 6 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j


8 9 10 11 12 13 14 15 16 17
A … 1 2 2 3 1 2 3 6 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j


8 9 10 11 12 13 14 15 16 17
A … 1 2 2 3 4 2 3 6 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j


8 9 10 11 12 13 14 15 16 17
A … 1 2 2 3 4 5 3 6 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j


8 9 10 11 12 13 14 15 16 17
A … 1 2 2 3 4 5 6 6 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j


8 9 10 11 12 13 14 15 16 17
A … 1 2 2 3 4 5 6 7 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j

# Merge sort analysis

- For simplicity, assume that $n$ is a power of 2 so that each divide step yields two subproblems, both of size exactly $n/2$.
- The base case occurs when $n = 1$.
- When $n \geq 2$, time for merge sort steps:
- **Divide**: Just compute $q$ as the average of $p$ and $r$, which takes constant time i.e. $\Theta(1)$.
- **Conquer**: Recursively solve 2 subproblems, each of size $n/2$, which is $2T(n/2)$.
- **Combine**: MERGE on an $n$-element subarray takes $\Theta(n)$ time.
- Summed together they give a function that is linear in $n$, which is $\Theta(n)$. Therefore, the recurrence for merge sort running time is

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 2T(n/2) + \Theta(n) & \text{if } n > 1. \end{cases}$$

# Using Master's theorem

**T(n) = 2T(n/2) +** $\Theta(n)$

**T(n) = aT(n/b) + f(n) where a >= 1 and b > 1**

a = 2 b = 2 f(n) = n

1. If $f(n) < O(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a}\log^k n)$ with $k \geq 0$, then $T(n) = \Theta(n^{\log_b a}\log^{k+1} n)$.
3. If $f(n) > \Omega(n^{\log_b a})$, and f(n) satisfies the regularity condition, then $T(n) = \Theta(f(n))$.

Calculate $n^{\log_b a} = n^{\log_2 2} = n$

Compare with f(n). Since $f(n) = n^{\log_b a}$

i.e. n = n

Case 2 is satisfied hence complexity is given as

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

# Video for Merge sort

Merge sort

# Worksheet No. 10

# UNIT II
# DIVIDE AND CONQUER

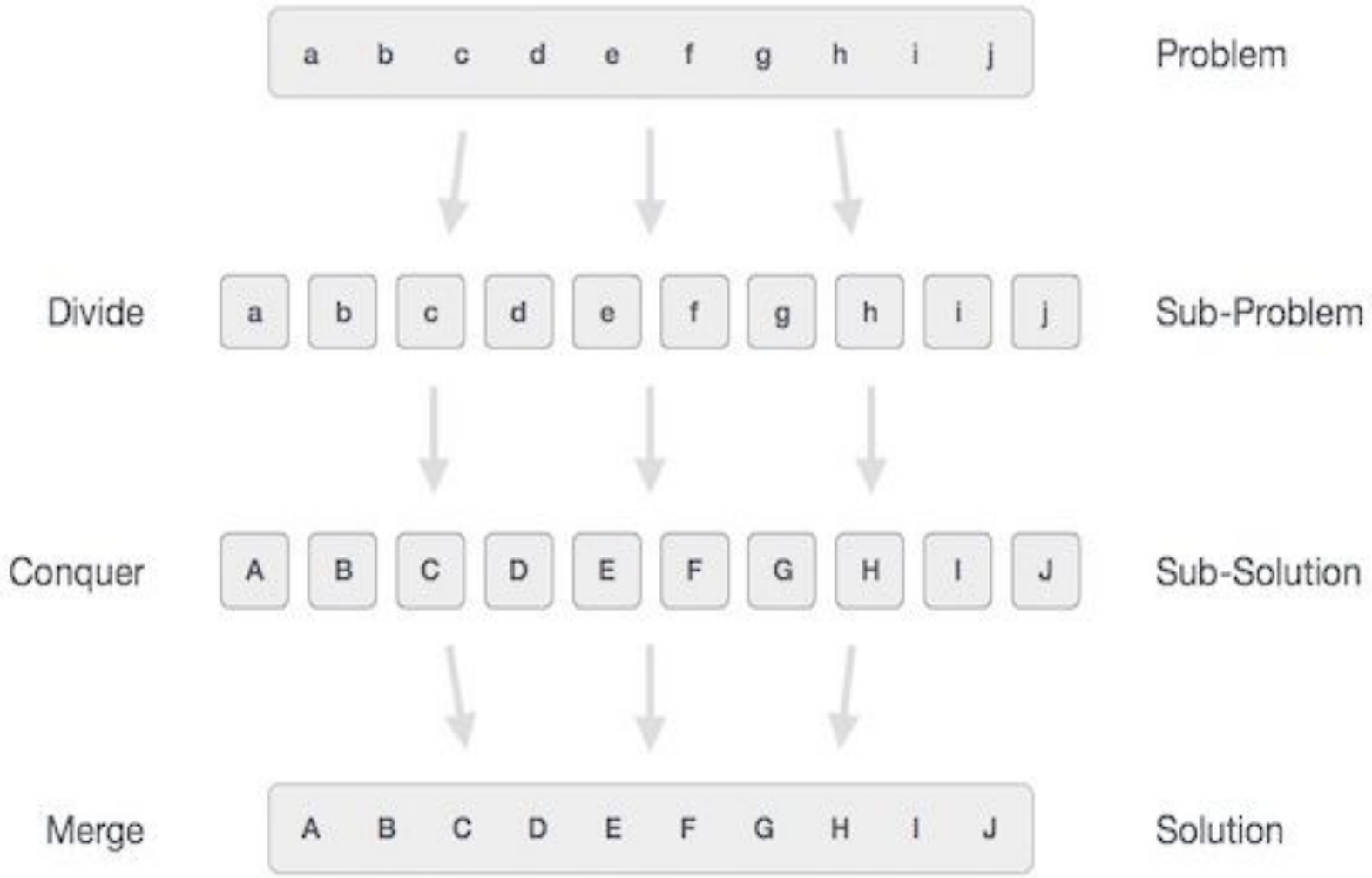
## Session – 1 1

# Syllabus

- Introduction, Binary Search - Merge sort and its algorithm analysis - **Quick sort and its algorithm analysis** - Strassen's Matrix multiplication - Finding Maximum and minimum - Algorithm for finding closest pair - Convex Hull Problem

# Introduction

- Divide / Break
- Conquer / Solve
- Merge / Combine


a b c d e f g h i j
Problem
Divide
a b c d e f g h i j
Sub-Problem
Conquer
A B C D E F G H I J
Sub-Solution
Merge
A B C D E F G H I J
Solution

# Quick Sort algorithm

- Quick sort works by partitioning a given array A[p . . r] into two non-empty sub array A[p . . q] and A[q+1 . .r] such that every key in A[p . . q] is less than or equal to every key in A[q+1 . . r].
- Then the two sub arrays are sorted by recursive calls to Quick sort. The exact position of the partition depends on the given array and index q is computed as a part of the partitioning procedure

# Quicksort()

**QuickSort ()**

1. If p < r then
2. Partition (A, p, r)
3. Recursive call to Quick Sort (A, p, q)
4. Recursive call to Quick Sort (A, q + 1, r)

- As a first step, Quick Sort chooses as pivot one of the items in the array to be sorted. Then array is then partitioned on either side of the pivot.
- Elements that are less than or equal to pivot will move toward the left and elements that are greater than or equal to pivot will move toward the right.

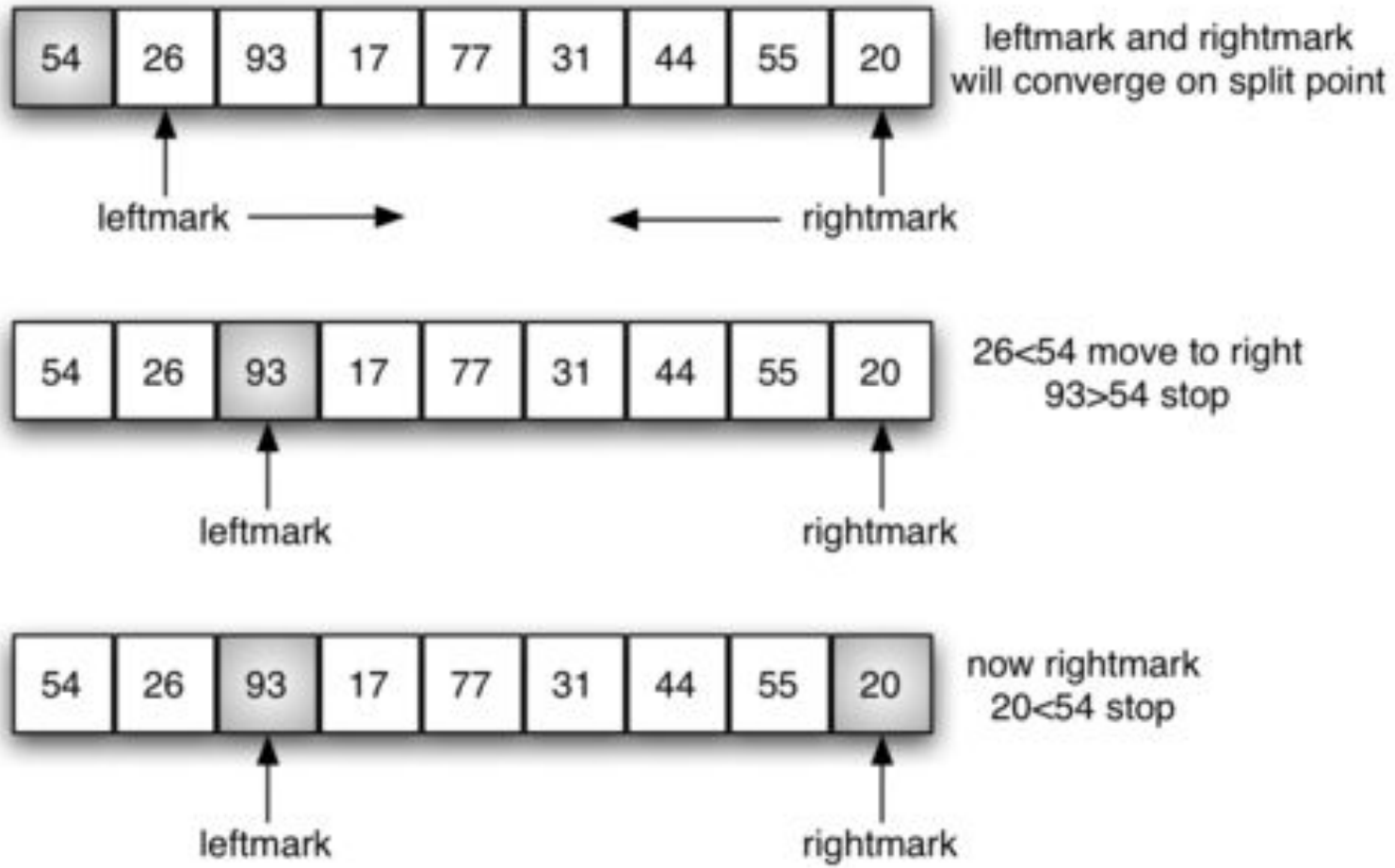
# Partitioning procedure

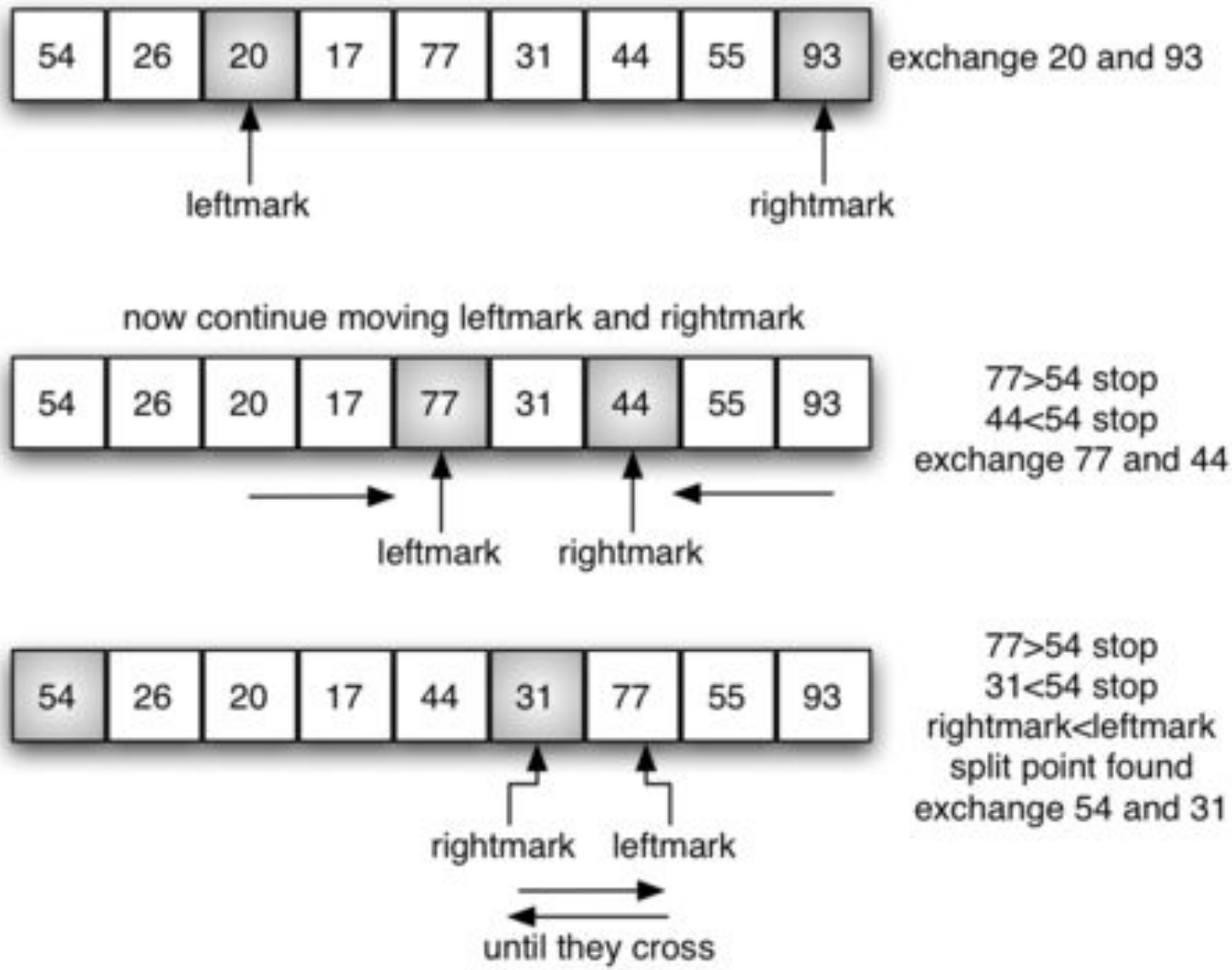
- **PARTITION (*A, p, r)***

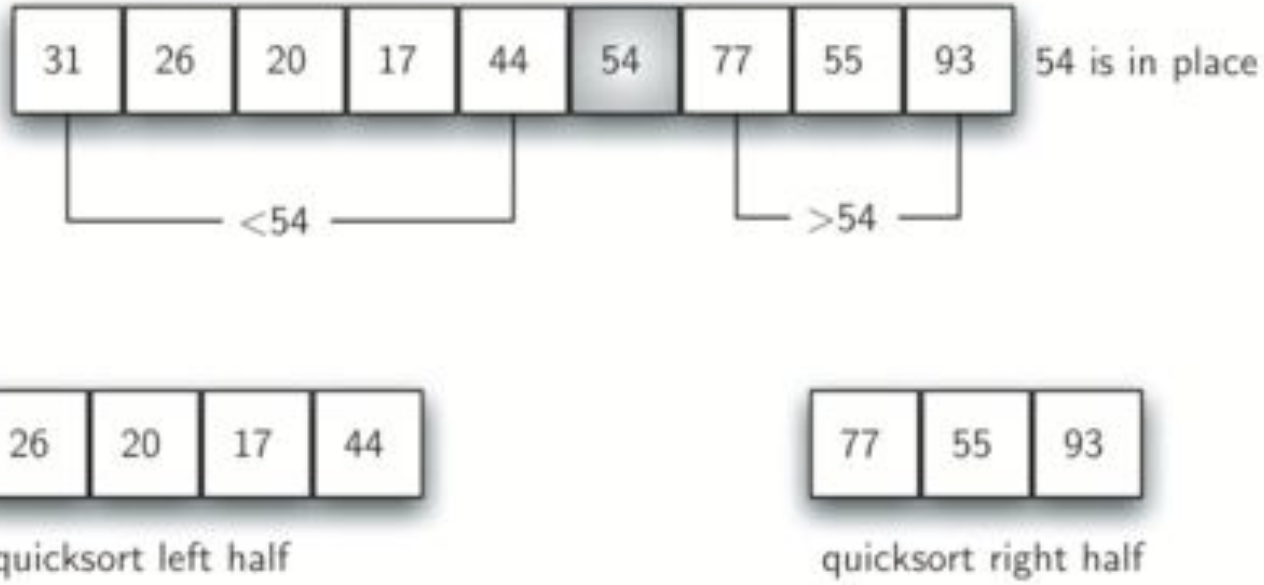
```
1. x ← A[p]
2. i ← p+1
3. j ← r
4. while TRUE do
5.  Repeat j ← j-1
6.  until A[j] ≤ x
7.  Repeat i ← i+1
8.  until A[i] ≥ x
9.  if i < j
10.          then exchange A[i] ↔ A[j]
11.     else return j
```

# Example

- Partitioning begins by locating two position markers—let's call them leftmark and rightmark—at the beginning and end of the remaining items in the list (positions 1 and 8). The goal of the partition process is to move items that are on the wrong side with respect to the pivot value while also converging on the split point. This process as we locate the position of 54.


54 26 93 17 77 31 44 55 20
leftmark and rightmark will converge on split point
leftmark
rightmark
54 26 93 17 77 31 44 55 20
26<54 move to right
93>54 stop
leftmark
rightmark
54 26 93 17 77 31 44 55 20
now rightmark
20<54 stop
leftmark
rightmark


54 26 20 17 77 31 44 55 93
exchange 20 and 93
leftmark
rightmark
now continue moving leftmark and rightmark
54 26 20 17 77 31 44 55 93
77>54 stop
44<54 stop
exchange 77 and 44
leftmark
rightmark
54 26 20 17 44 31 77 55 93
77>54 stop
31<54 stop
rightmark<leftmark
split point found
exchange 54 and 31
rightmark
leftmark
until they cross


31
26
20
17
44
54
77
55
93
54 is in place
<54
>54
31
26
20
17
44
quicksort left half
77
55
93
quicksort right half

# Analysis of Quick Sort

## Best Case

The best thing that could happen in Quick sort would be that each partitioning stage divides the array exactly in half. In other words, the best to be a median of the keys in *A[p . . r] every time procedure 'Partition'* is called. The procedure 'Partition' always split the array to be sorted into two equal sized arrays.

- If the procedure 'Partition' produces two regions of size *n/2. the recurrence relation is then*
- T(n) = T(*n/2) + T(n/2) + O(n)*
- = 2T(*n/2) + O(n)*
- And from case 2 of Master theorem

T(n) = (*n lg n)*

- **Worst-case**

When quicksort always has the most unbalanced partitions possible, then the original call takes cn, time for some constant c, the recursive call on n-1 elements takes c(n-1) time, the recursive call on n-2elements takes $c(n-2)$ and so on.

- When we total up the partitioning times for each level, we get
- $c(n) + c(n-1) + c(n-2) + c(n-3) + .. 2c = c(n+(n-1)+(n-2)+...(2))$

  $= c(1+2+3+...+n) - c(1)$

  $= c(n(n+1)/2) - c$

  $= \Theta(n^2)$

# Using Master's theorem

**Best Case**

**$T(n) = 2T(n/2) + \Theta(n)$**

**T(n) = aT(n/b) + f(n) where a >= 1 and b > 1**

a = 2 b = 2 f(n) = n

1. If $f(n) < O(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a}\log^k n)$ with $k \geq 0$, then $T(n) = \Theta(n^{\log_b a}\log^{k+1} n)$.
3. If $f(n) > \Omega(n^{\log_b a})$, and f(n) satisfies the regularity condition, then $T(n) = \Theta(f(n))$.

Calculate $n^{\log_b a} = n^{\log_2 2} = n$

Compare with f(n). Since $f(n) = n^{\log_b a}$

i.e. n = n

Case 2 is satisfied hence complexity is given as

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

# Worksheet No. 11

# UNIT II
# DIVIDE AND CONQUER

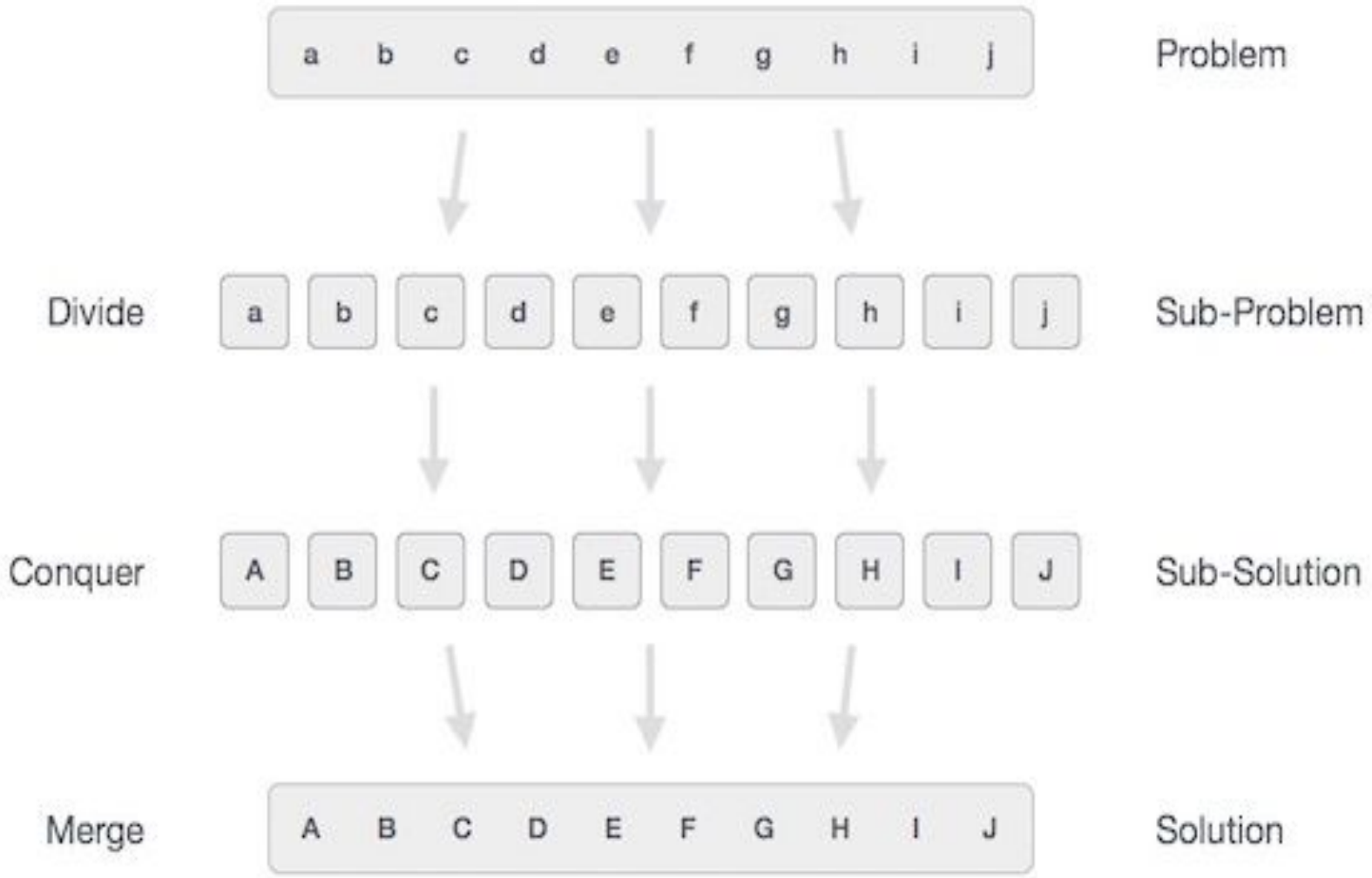
## Session – 12

# Syllabus

- Introduction, Binary Search - Merge sort and its algorithm analysis - Quick sort and its algorithm analysis - **Strassen's Matrix multiplication** - Finding Maximum and minimum - Algorithm for finding closest pair - Convex Hull Problem

# Introduction

- Divide / Break
- Conquer / Solve
- Merge / Combine


a b c d e f g h i j
Problem
Divide
a b c d e f g h i j
Sub-Problem
Conquer
A B C D E F G H I J
Sub-Solution
Merge
A B C D E F G H I J
Solution

# Strassen's Multiplication – Naive Method

- Given two square matrices A and B of size **n x n** each, find their multiplication matrix.

***Naive Method***

```
void multiply(int A[][N], int B[][N], int C[][N])
{
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
        {
            C[i][j] = 0;
            for (int k = 0; k < N; k++)
            {
                C[i][j] += A[i][k]*B[k][j];
            }
        }
    }
}   // Time Complexity of above method is O(N^3).
```

# Strassen's Multiplication – Divide and Conquer

- Following is simple Divide and Conquer method to multiply two square matrices.

**1) Divide matrices A and B in 4 sub-matrices of size N/2 x N/2 as shown in the below diagram.**

**2) Calculate following values recursively. ae+bg, af+bh, ce+dg and cf+dh**.

- In the above method, we do 8 multiplications for matrices of size N/2 x N/2 and 4 additions. Addition of two matrices takes $O(N^2)$ time. So the time complexity can be written as $T(N) = 8T(N/2) + O(N^2)$
- From Master's Theorem, time complexity of above method is $O(N^3)$ which is unfortunately same as the above naive method.

# *Simple Divide and Conquer also leads to $O(N^3)$, can there be a better way?*

- In the above divide and conquer method, the main component for high time complexity is 8 recursive calls.
- The idea of Strassen's method is to reduce the number of recursive calls to 7. Strassen's method is similar to above simple divide and conquer method in the sense that this method also divides matrices to submatrices of size N/2 x N/2 as shown in the above diagram, but in Strassen's method, the four sub-matrices of result are calculated using following formulae.

$$p1 = a(f - h) \qquad p2 = (a + b)h$$
$$p3 = (c + d)e \qquad p4 = d(g - e)$$
$$p5 = (a + d)(e + h) \qquad p6 = (b - d)(g + h)$$
$$p7 = (a - c)(e + f)$$

The A x B can be calculated using above seven multiplications.
Following are values of four sub-matrices of result C

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} p5 + p4 - p2 + p6 & p1 + p2 \\ p3 + p4 & p1 + p5 - p3 - p7 \end{bmatrix}$$

A B C

A, B and C are square metrices of size N x N
a, b, c and d are submatrices of A, of size N/2 x N/2
e, f, g and h are submatrices of B, of size N/2 x N/2
p1, p2, p3, p4, p5, p6 and p7 are submatrices of size N/2 x N/2

**Generally, Strassen's Method is not preferred for practical applications for following reasons.**

1. The constants used in Strassen's method are high and for a typical application Naive method works better.
2. For Sparse matrices, there are better methods especially designed for them.
3. The sub matrices in recursion take extra space.
4. Because of the limited precision of computer arithmetic on non integer values, larger errors accumulate in Strassen's algorithm than in Naive Method

# Using Master's theorem

**Time Complexity**

**$T(n) = 7T(n/2) + \Theta(n^2)$**

**T(n) = aT(n/b) + f(n) where a >= 1 and b > 1**

**a = 7 b = 2 $f(n) = n^2$**

1. If $f(n) < O(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a} \log^k n)$ with $k \geq 0$, then $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$.
3. If $f(n) > \Omega(n^{\log_b a})$, and f(n) satisfies the regularity condition, then $T(n) = \Theta(f(n))$.

Calculate $n^{\log_b a} = n^{\log_2 7} = n^{\mathbf{2.8074}}$

Compare with f(n). Since $f(n) < n^{\log_b a}$

i.e. $n^2 = n^{\mathbf{2.8074}}$

Case 1 is satisfied hence complexity is given as

**$T(n) = \Theta(n^{\log_b a}) = \Theta(n^{2.8074})$**

# Worksheet No. 12

# UNIT II
# DIVIDE AND CONQUER

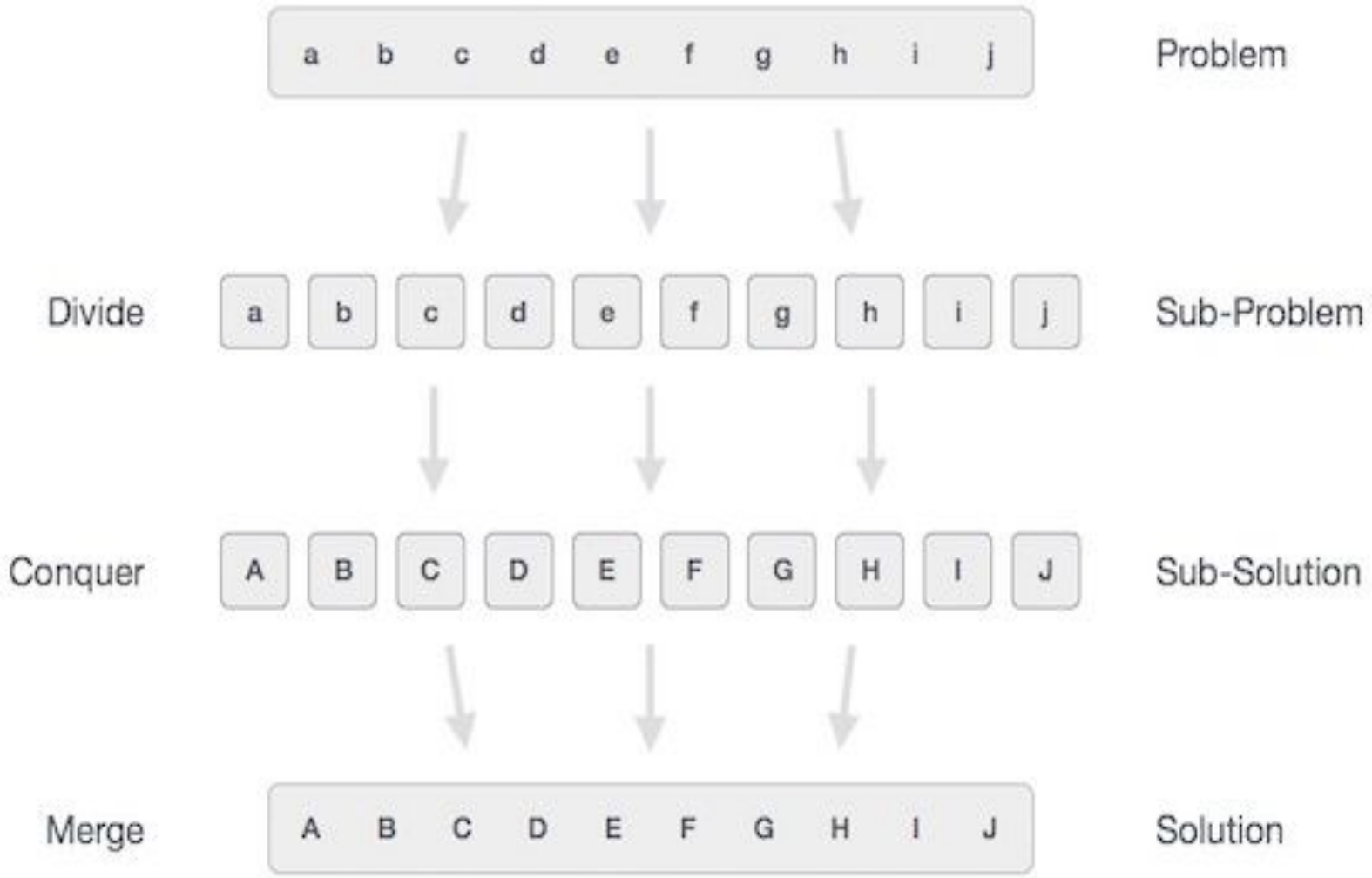
## Session – 13

# Syllabus

- Introduction, Binary Search - Merge sort and its algorithm analysis - Quick sort and its algorithm analysis - Strassen's Matrix multiplication - **Finding Maximum and minimum** - Algorithm for finding closest pair - Convex Hull Problem

# Introduction

- Divide / Break
- Conquer / Solve
- Merge / Combine


a b c d e f g h i j
Problem
Divide
a b c d e f g h i j
Sub-Problem
Conquer
A B C D E F G H I J
Sub-Solution
Merge
A B C D E F G H I J
Solution

# Finding Maximum and Minimum

- **Algorithm straightforward**

  Finding the maximum and minimum elements in a set of (n) elements

**Algorithm straightforward (a, n, max, min)**

```
Input: array (a) with (n) elements
Output: max: max value, min: min value
max = min = a(1)
for i = 2 to n do
    begin
     if (a(i)>max) then max=a(i)
     if (a(i)<min) then min=a(i)
    end
End
```

**Complexity**

**best= average =worst= 2(n-1) comparisons**

# Modify it for betterment

- If we change the body of the loop as follows:

```
max=min=a(1)
for i=2 to n do
    begin
     if (a(i)>max) then max=a(i)
     else if (a(i)<min) then min=a(i)
    end
```

- **Complexity**

best case: elements in increasing order No. of comparisons = (n-1)

Worst case: elements in decreasing order No. of comparisons = 2(n-1)

Average case: a(i) is greater than max half the time

No. of comparisons = 3n/2 – 3/2

=(1/2)((n-1)+(2n-2))

# Divide and Conquer approach

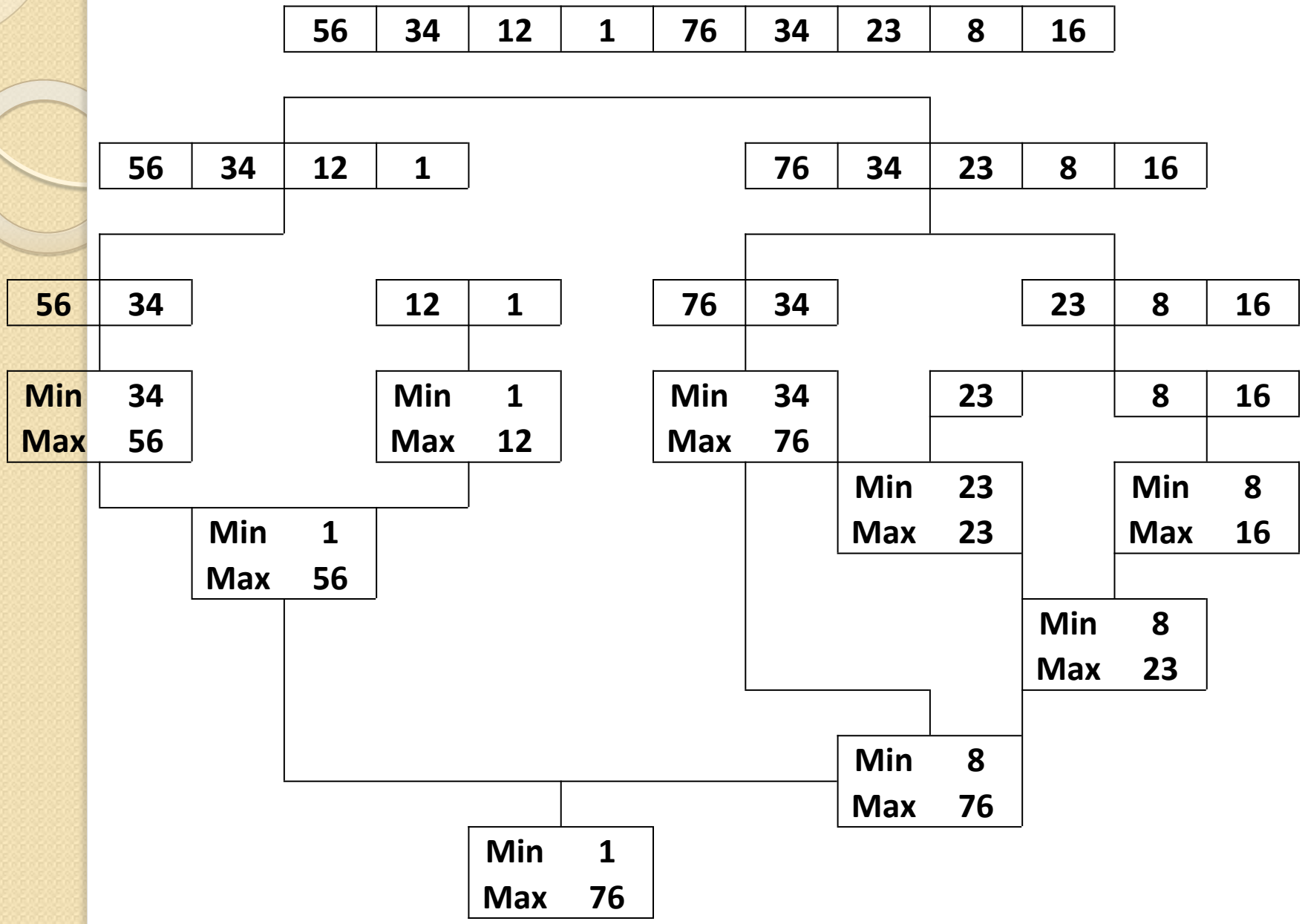
Problem: is to find the maximum and minimum items in a set of (n) elements.

- **Algorithm MaxMin(i, j, max, min)**

***input:*** *array of N elements, i lower bound, j upper bound*

***output:*** *max: largest value;* min: smallest value.

```
if (i=j) then max=min=a(i)
else if (i=j-1) then
      if (a(i)<a(j)) then max= a(j) and  min= a(i)
      else                max= a(i) and  min= a(j)
       else
      mid=(i+j)/2
      maxmin(i, mid, max, min)
      maxmin(mid+1, j, max1, min1)
      if (max<max1) then max = max1
      if (min>min1) then min = min1
end
```


56 34 12 1 76 34 23 8 16
56 34 12 1
76 34 23 8 16
56 34
12 1
76 34
23 8 16
Min 34
Max 56
Min 1
Max 12
Min 34
Max 76
23
8 16
Min 1
Max 56
Min 23
Max 23
Min 8
Max 16
Min 8
Max 23
Min 8
Max 76
Min 1
Max 76

# Video

- Click here

# Complexity Analysis

1.if size= 1 return current element as both max and min //base condition

2.else if size= 2 one comparison to determine max and min //base condition

3.else /* if size > 2 find mid and call recursive function */

recur for max and min of left half recur for max and min of right half.

one comparison determines to max of the two subarray, update max.

one comparison determines min of the two subarray, update min.

4. finally return or print the min/max in whole array.

**Recurrence relation can be written as T(n)=2(T/2)+2**

solving which give you T(n) =(3n/2)-2 which is the exact no. of comparisons but still the worst time complexity will be **T(n)=O(n)** and best case time complexity will be **O(1)** when you have only one element in array, which will be candidate for both max and min.

$$C_n = \begin{cases} C\left(\lceil \frac{n}{2} \rceil\right) + C\left(\frac{n}{2} \rfloor\right) + 2 & for\ n > 2 \\ 1 & n = 2 \\ 0 & n = 1 \end{cases}$$

When n is a power of 2, $n = 2^k$

$$\begin{aligned} C_n &= 2C_{\frac{n}{2}} + 2 \\ &= 2\left(2C_{\frac{n}{4}} + 2\right) + 2 \\ &= 4C_{\frac{n}{4}} + 4 + 2 \\ &\vdots \\ &= 2^{k-1}C_2 + \sum_{i=1}^{k-1} 2^i \\ &= 2^{k-1} + 2^k - 2 \\ &= \frac{3n}{2} - 2 \end{aligned}$$

# Worksheet No. 13

# UNIT II
# DIVIDE AND CONQUER

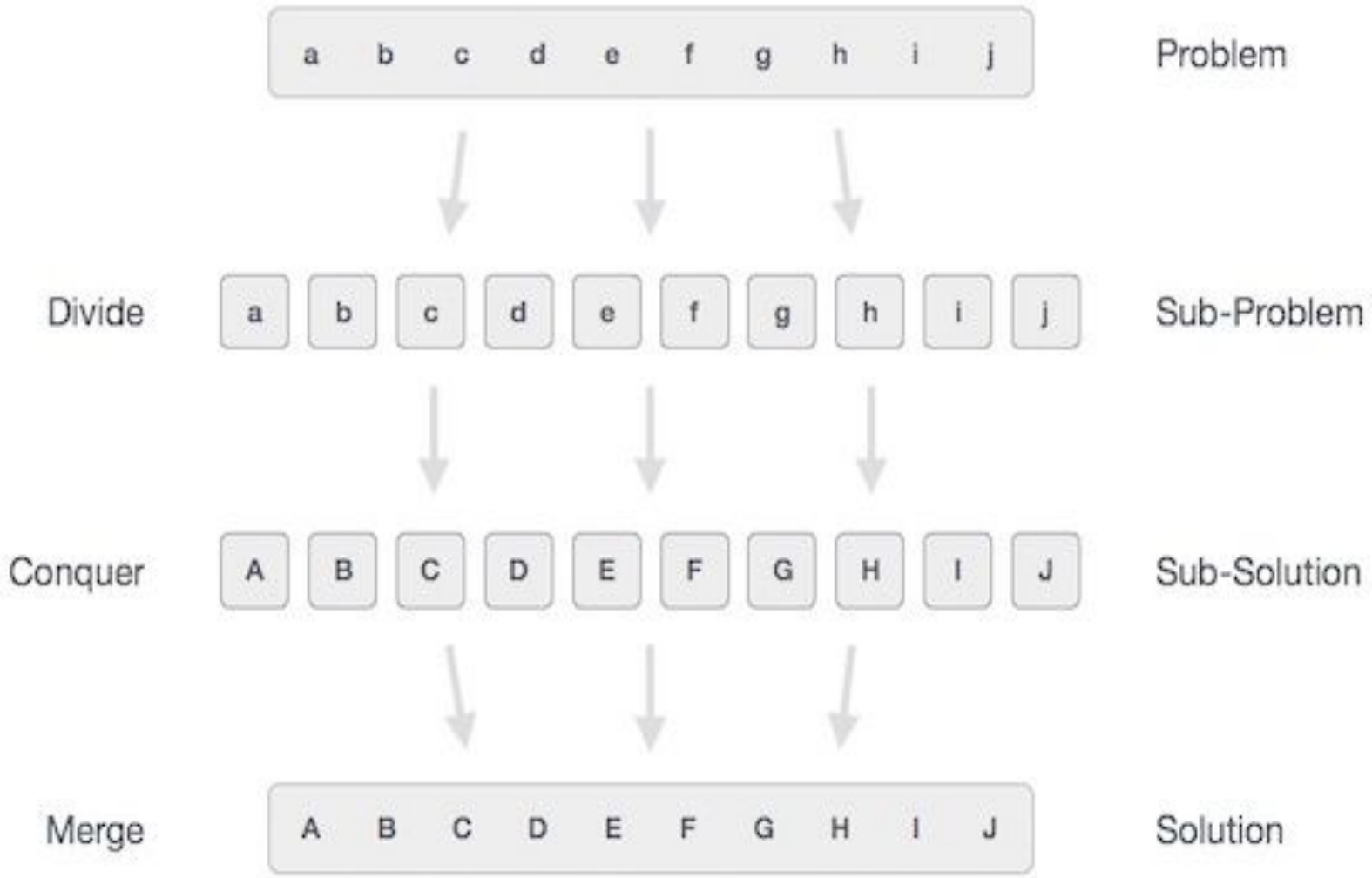
## Session – 14

# Syllabus

- Introduction, Binary Search - Merge sort and its algorithm analysis - Quick sort and its algorithm analysis - Strassen's Matrix multiplication - Finding Maximum and minimum - **Algorithm for finding closest pair** - Convex Hull Problem

# Introduction

- Divide / Break
- Conquer / Solve
- Merge / Combine


a b c d e f g h i j
Problem
Divide
a b c d e f g h i j
Sub-Problem
Conquer
A B C D E F G H I J
Sub-Solution
Merge
A B C D E F G H I J
Solution

# Algorithm for finding closest pair

- **We are given an array of n points in the plane, and the problem is to find out the closest pair of points in the array.**
- For example, in air-traffic control, you may want to monitor planes that come too close together, since this may indicate a possible collision.
- Recall the following formula for distance between two points p and q.

$$\|pq\| = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}.$$

- The Brute force solution is $O(n^2)$, compute the distance between each pair and return the smallest.
- We can calculate the smallest distance in O(nLogn) time using Divide and Conquer strategy.

# Algorithm

Input: An array of n points P[]

Output: The smallest distance between two points in the given array.

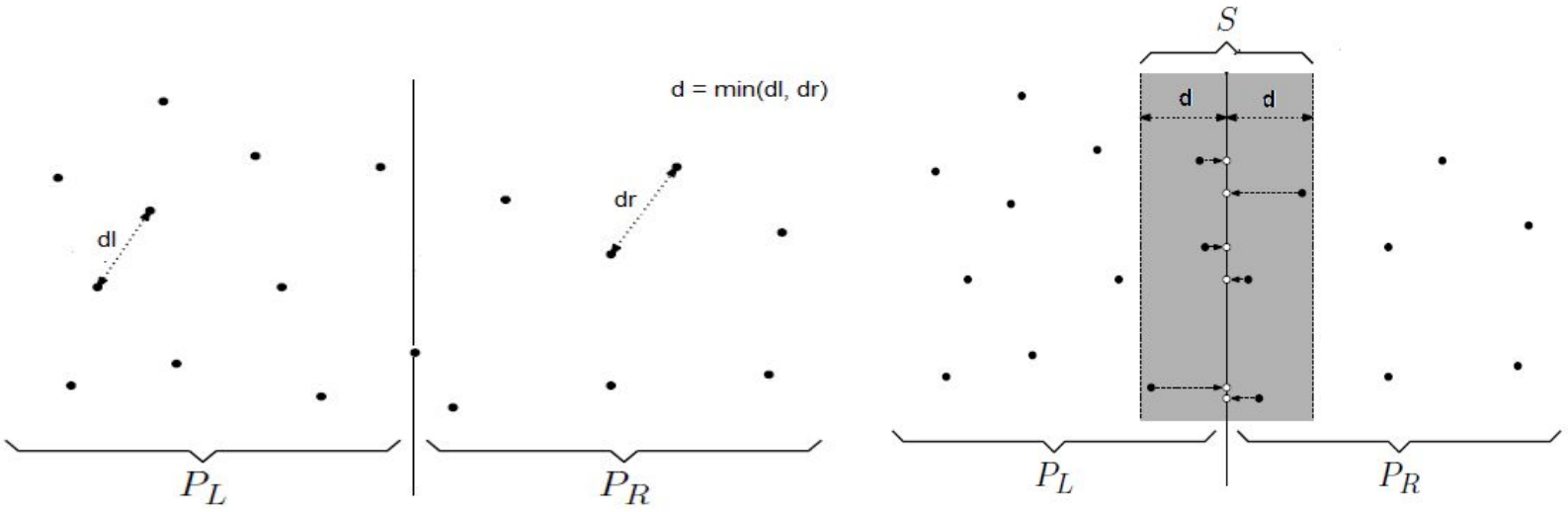
As a pre-processing step, input array is sorted with x coordinates.

1) **Find the middle point** in the sorted array, we can take P[n/2] as middle point.

2) **Divide the given array in two halves.** The first subarray contains points from P[0] to P[n/2]. The second subarray contains points from P[n/2+1] to P[n-1].

3) **Recursively find the smallest distances** in both subarrays. Let the distances be *dl* and *dr*. Find the minimum of *dl* and *dr*. Let the minimum be *d*.

**4) From above 3 steps, we have an upper bound d of minimum distance. Now we need to consider the pairs** such that one point in pair is from left half and other is from right half. Consider the vertical line passing through passing through P[n/2] and find all points whose x coordinate is closer than d to the middle vertical line. Build an array strip[] of all such points.




**5) Sort the array strip[] according to y coordinates. This step is O(nLogn). It can be optimized to O(n) by** recursively sorting and merging.

**6) Find the smallest distance in strip[]. This is tricky. From first look, it seems to be a $O(n^2)$ step, but it is** actually O(n). It can be proved geometrically that for every point in strip, we only need to check at most 7 points after it (note that strip is sorted according to Y coordinate).

**7)** Finally return the minimum of d and distance calculated in above step 6

# Time Complexity

**Let Time complexity of above algorithm be T(n).**

- **Let us assume that we use a O(nLogn)** sorting algorithm. The above algorithm divides all points in two sets and recursively calls for two sets.
- After dividing, it finds the strip in **O(n)** time, sorts the strip in **O(nLogn)** time and finally finds the closest points in strip in **O(n)** time.

So T(n) can expressed as follows:

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

**$T(n) = 2T(n/2) + O(nLogn)$**

# Using Master's theorem

**$T(n) = 2T(n/2) + O(nLogn)$**

**$T(n) = aT(n/b) + f(n)$ where $a >= 1$ and $b > 1$**

**$a = 2$ $b = 2$ $f(n) = nlogn$**

1. If $f(n) < O(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a})$.
2. **If $f(n) = \Theta(n^{\log_b a}\log^k n)$ with $k \geq 0$, then $T(n) = \Theta(n^{\log_b a}\log^{k+1} n)$.**
3. If $f(n) > \Omega(n^{\log_b a})$, and $f(n)$ satisfies the condition, then $T(n) = \Theta(f(n))$.

Calculate $n^{\log_b a} = n^{\log_2 2} = n^1$

Compare with $f(n)$. Since $f(n) < n^{\log_b a}$ i.e. $nlogn = n^1\log^1 n$

Case 2 is satisfied hence complexity is given as

**$T(n) = \Theta(n^{\log_b a}\log^{k+1} n) = \Theta(n\log^2 n)$**

# Worksheet No. 14

# UNIT II
# DIVIDE AND CONQUER

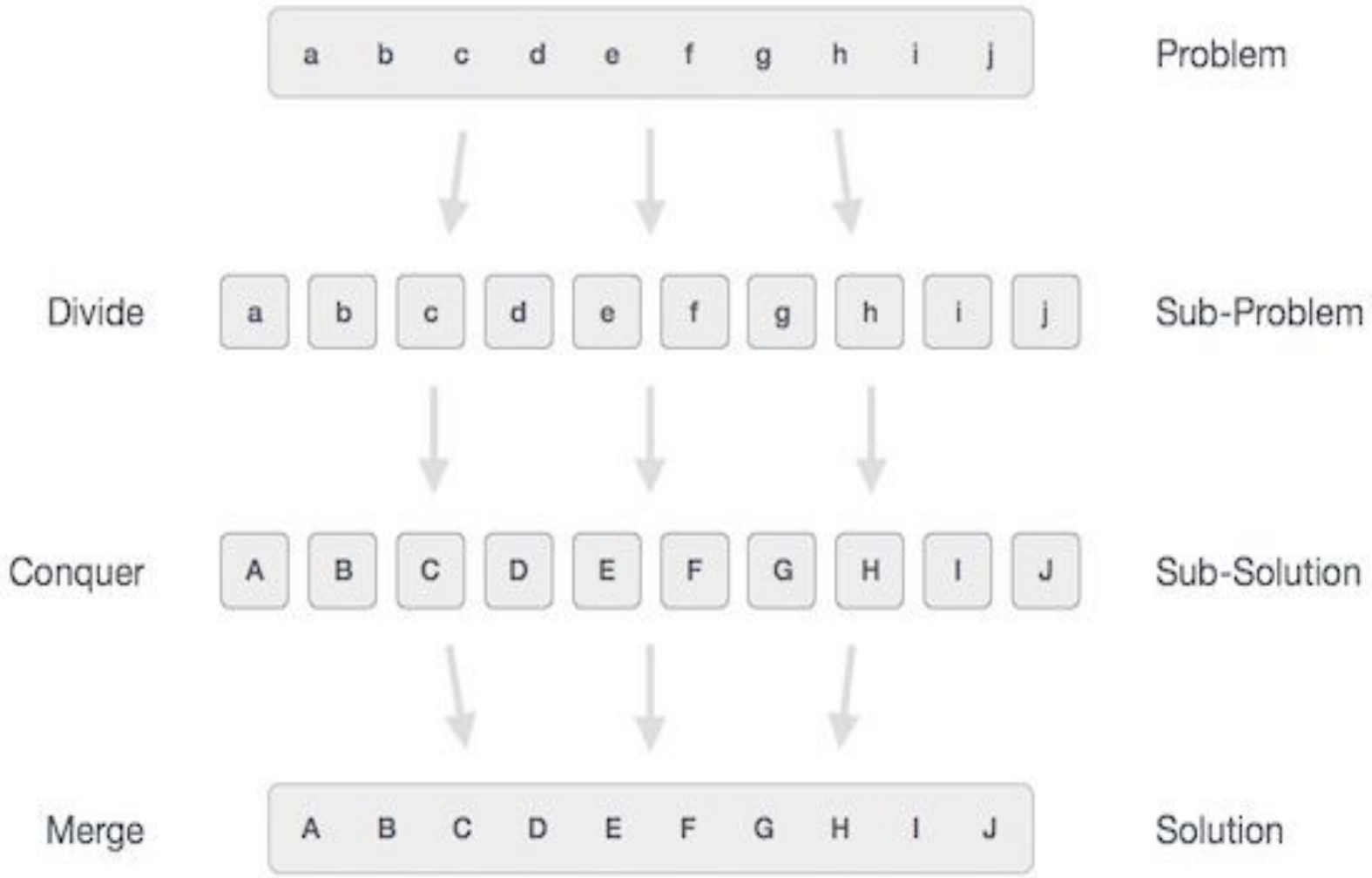
## Session – 15

# Syllabus

- Introduction, Binary Search - Merge sort and its algorithm analysis - Quick sort and its algorithm analysis - Strassen's Matrix multiplication - Finding Maximum and minimum - Algorithm for finding closest pair - **Convex Hull Problem**

# Introduction

- Divide / Break
- Conquer / Solve
- Merge / Combine


a b c d e f g h i j
Problem
Divide
a b c d e f g h i j
Sub-Problem
Conquer
A B C D E F G H I J
Sub-Solution
Merge
A B C D E F G H I J
Solution

# CONVEX HULL PROBLEM

- A polygon is **convex if any line segment joining two points on the boundary stays within the polygon.**
- The **convex hull of a set of points in the plane is the smallest convex polygon for which each point is either** on the boundary or in the interior of the polygon.
- A **vertex is a corner of a polygon. For example, the highest, lowest, leftmost and rightmost points are** all vertices of the convex hull.

# Different algorithms

- Graham Scan,
- Jarvis March
- Divide & Conquer

We present the algorithms under the **assumption that no 3 points are collinear (on a straight line)**

# Graham Scan

- The idea is to identify one vertex of the convex hull and sort the other points as viewed from that vertex.
- Then the points are scanned in order.
- Let **x0 be the leftmost point (which is guaranteed to be in the convex hull) and number the remaining points** by angle from **x0 going counter clockwise:**

  **x1; x2; : : : ; xn-1.**

  **Let xn = x0, the chosen point. Assume that no two** points have the same angle from **x0.**

# Graham Scan Algorithm

- The algorithm is simple to state with a single stack:

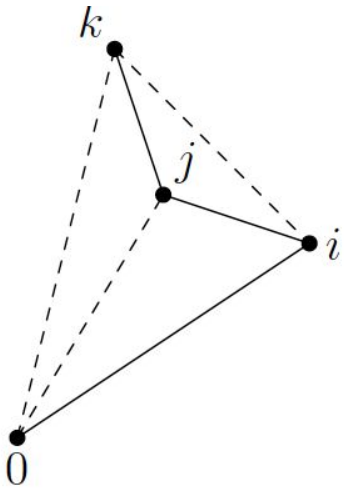
1. Sort points by angle from **x0**
2. Push **x0 and x1. Set i=2**
3. While **i <=n do:**

   If **xi makes left turn w.r.t. top 2 items on stack, then** { push **xi; i++ }**

   else { pop and discard }

- To prove that the algorithm works, it success to argue that:
- A discarded point is not in the convex hull.
- If $x_j$ is discarded, then for some i < j < k the points xi --> xj --> xk form a right turn. So, $x_j$ is inside the triangle x0, xi, xk and hence is not on the convex hull.

- What remains is convex. This is immediate as every turn is a left turn.

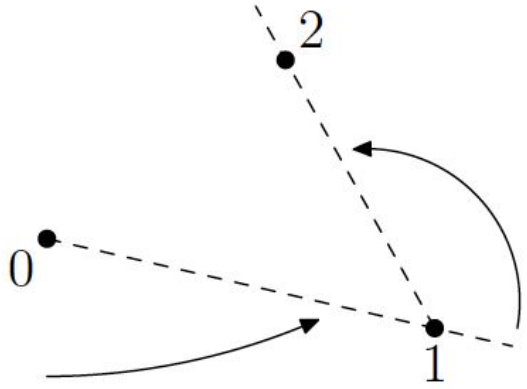


# Running time of Grahams scan

- **Each time the while loop is executed, a point is either stacked or discarded. Since a** point is looked at only once, the loop is executed at most **2n times.**
- **There is a constant-time subroutine for** checking, given three points in order, whether the angle is a left or a right turn.
- This gives an **O(n) time** algorithm, apart from the initial sort which takes time **O(n log n).**

# Jarvis March

- This is also called the **wrapping algorithm.**
- **This algorithm finds the points on the convex hull in the order** in which they appear.
- It is quick if there are only a few points on the convex hull, but slow if there are many.
- Let **x0 be the leftmost point. Let x1 be the first point counter clockwise when viewed fromx0. Then x2 is the** first point counter clockwise when viewed from **x1, and so on.**



# Jarvis March Algorithm

**i = 0**

while not done do

**xi+1 = first point counter clockwise from xi**

Finding xi+1 takes linear time.

The while loop is executed at most n times. More specifically, the while loop is executed h times where h is the number of vertices on the convex hull. So Jarvis March takes time **O(nh)**.

The best case is h = 3.

The worst case is h = n, when the points are, for example, arranged on the circumference of a circle.

# Divide and Conquer method

1. Divide the n points into two halves.

2. Find convex hull of each subset.

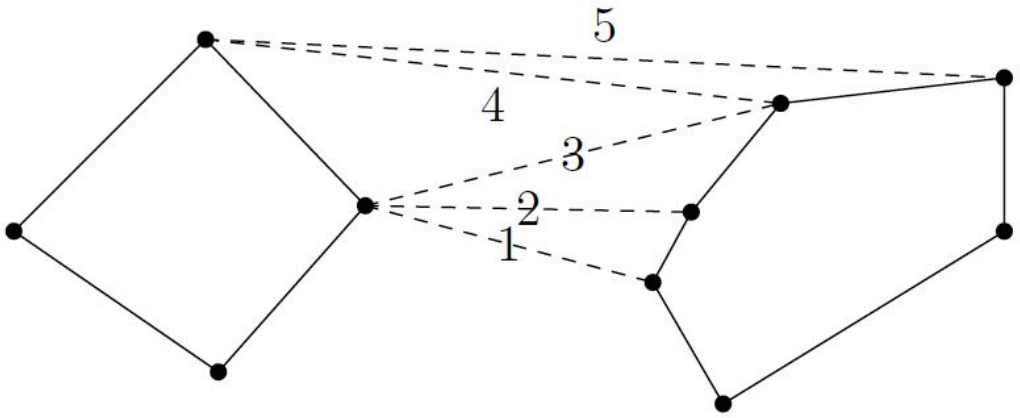
3. Combine the two hulls into overall convex hull.

**Combining two hulls**

1. It helps to work with convex hulls that do not overlap. To ensure this, all the points are **presorted from left** to right. So we have a left and right half and hence a left and right convex hull.
2. Define a **bridge as any line segment joining a vertex on the left and a vertex on the right that does not cross** the side of either polygon. What we need are the **upper and lower bridges.**

**The following produces the upper bridge.**

1. Start with any bridge. For example, a bridge is guaranteed if you join the rightmost vertex on the left to the leftmost vertex on the right.
2. Keeping the left end of the bridge fixed, see if the right end can be raised. That is, look at the next vertex on the right polygon going clockwise, and see whether that would be a (better) bridge. Otherwise, see if the left end can be raised while the right end remains fixed.
3. If made no progress in (2) (cannot raise either side), then stop else repeat (2).



# Running time

- **The key is to perform step (2) in constant time. For this it is sufficient that** each vertex has a pointer to the next vertex going clockwise and going counter clockwise.
- Hence the choice of data structure: we store each hull using a **doubly linked circular linked list.**
- It follows that the total work done in a merge is proportional to the number of vertices. This means that the overall algorithm takes time **O(nlog n).**

# Worksheet No. 15