Unit 2

Divide & Conquer Algorithms

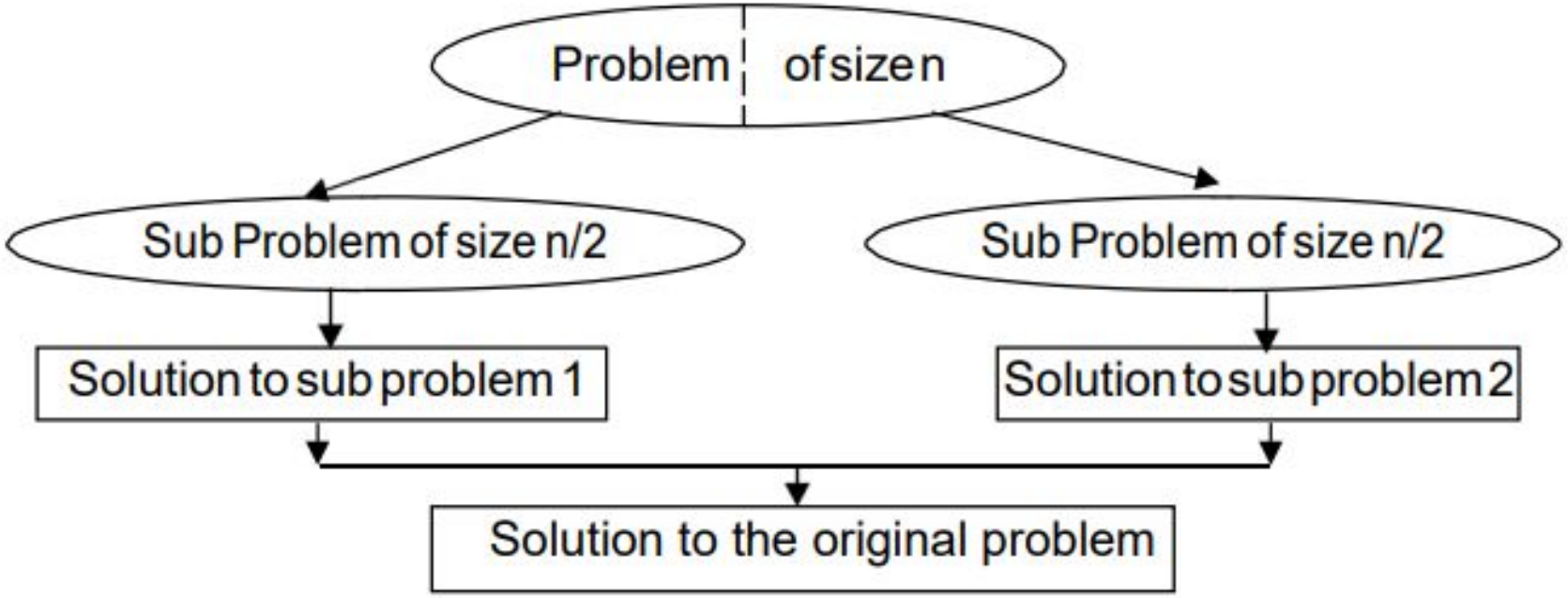
2. Divide & Conquer Algorithms:

Structure of divide-and-conquer algorithms,

Examples: Binary search, quick sort, Merge sort, Strassen Multiplication; Max-Min problem

Divide & Conquer Algorithms:

Divide & Conquer Algorithms



What is Divide & Conquer ?

- Divide and Conquer is an algorithm design paradigm in which a problem is divided into smaller subproblems, each subproblem is solved recursively, and the solutions are combined to obtain the final result.
- Instead of solving one large problem directly, the problem is broken into smaller independent parts.
- Divide and conquer is a design strategy which is well known to breaking down efficiency barriers. When the method applies, it often leads to a large improvement in time complexity. For example, from O (n^2) to O (n log n) to sort the elements.

Why Divide & Conquer Algorithms

Large problems are difficult to solve directly.

Breaking them into smaller problems

- reduces complexity,
- improves efficiency,
- allows recursive implementation,
- simplifies algorithm design.

General Structure of divide-and-conquer.

Every Divide and Conquer algorithm follows three steps.

- **Step 1 : Divide**

  Divide the problem into smaller subproblems.

- **Step 2 : Conquer**

  Solve each subproblem recursively.

  If the problem becomes very small,

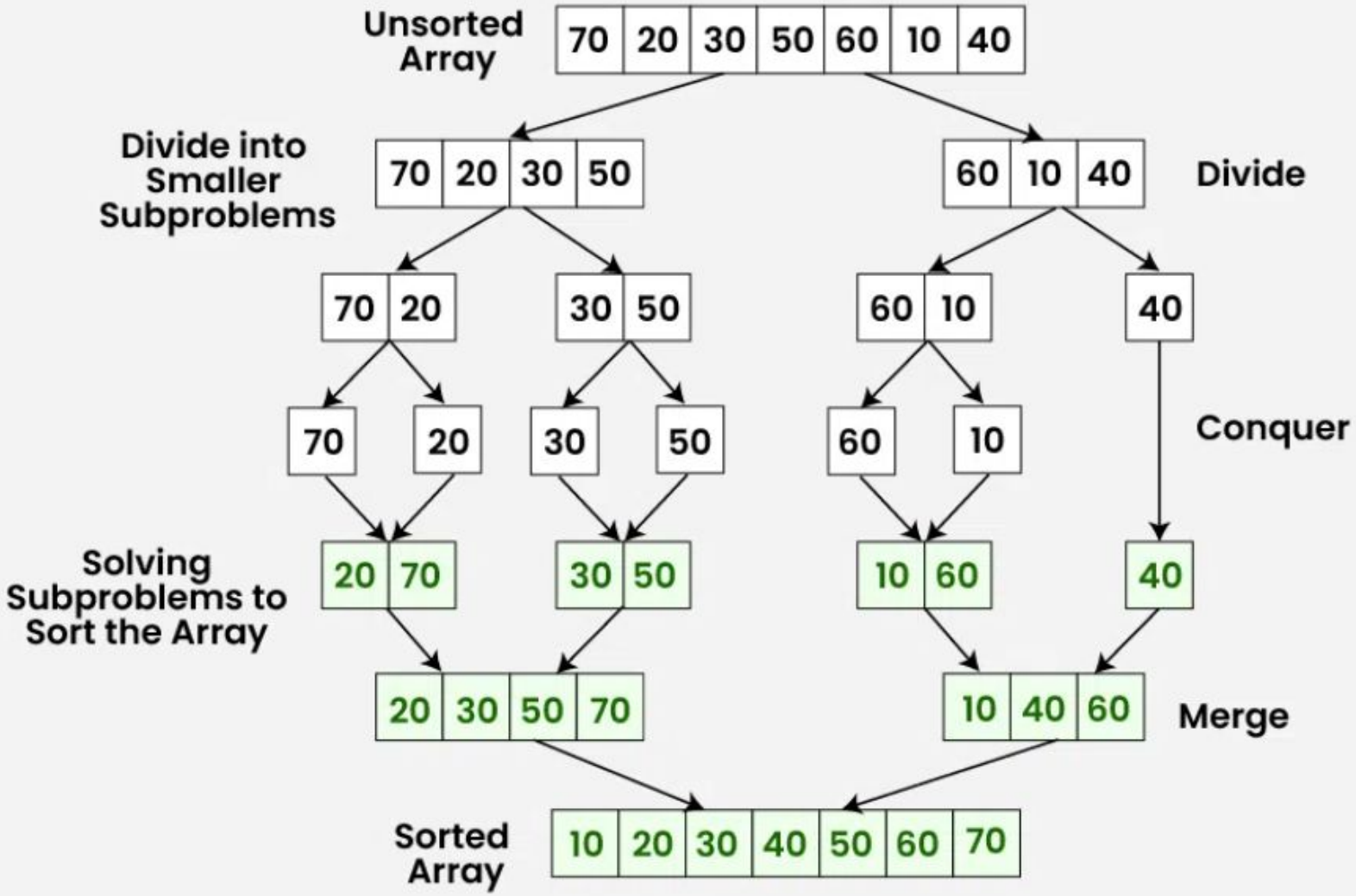
  solve it directly.

  (Base Case)

- **Step 3 : Combine**

  Merge the solutions of all subproblems to obtain the final answer.

Example


Unsorted Array
70 20 30 50 60 10 40
Divide into Smaller Subproblems
70 20 30 50
60 10 40
Divide
70 20
30 50
60 10
40
70
20
30
50
60
10
Conquer
Solving Subproblems to Sort the Array
20 70
30 50
10 60
40
20 30 50 70
10 40 60
Merge
Sorted Array
10 20 30 40 50 60 70

Application

- Binary Search
- Merge Sort
- Quick Sort
- Closest Pair of Points
- Strassen Matrix Multiplication
- Max-Min Problem

Recurrence Relation in Divide & Conquer

General form

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

Where

- a = number of subproblems
- n/b = size of each subproblem
- f(n) = work done outside recursion

Recurrence Relation in Divide & Conquer

- Binary Search

  $T(n)=T(n/2)+1$
- Merge Sort

  $T(n)=2T(n/2)+n$
- Quick Sort
  - Average

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

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

Binary Search

Binary Search

- Binary Search is a Divide and Conquer searching algorithm that works on **sorted data** by repeatedly comparing the target element with the middle element and reducing the search space by half.

- Instead of checking every element, Binary Search eliminates half of the remaining elements in every iteration.

Conditions to apply Binary Search Algorithm

- The data structure must be sorted.
- Access to any element of the data structure should take constant time.

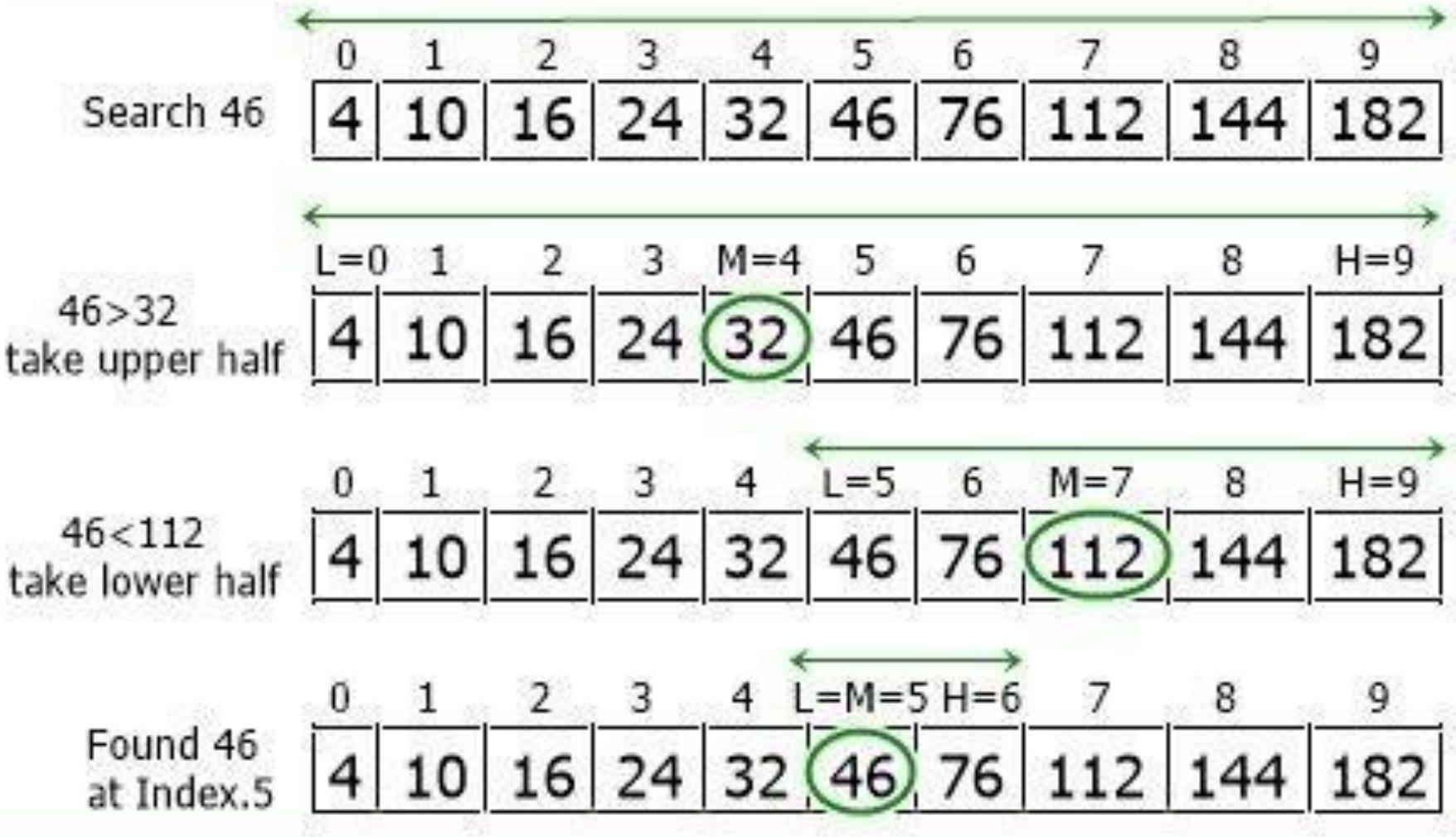
Why Binary Search is Divide & Conquer?

Binary Search follows the three phases of Divide & Conquer.

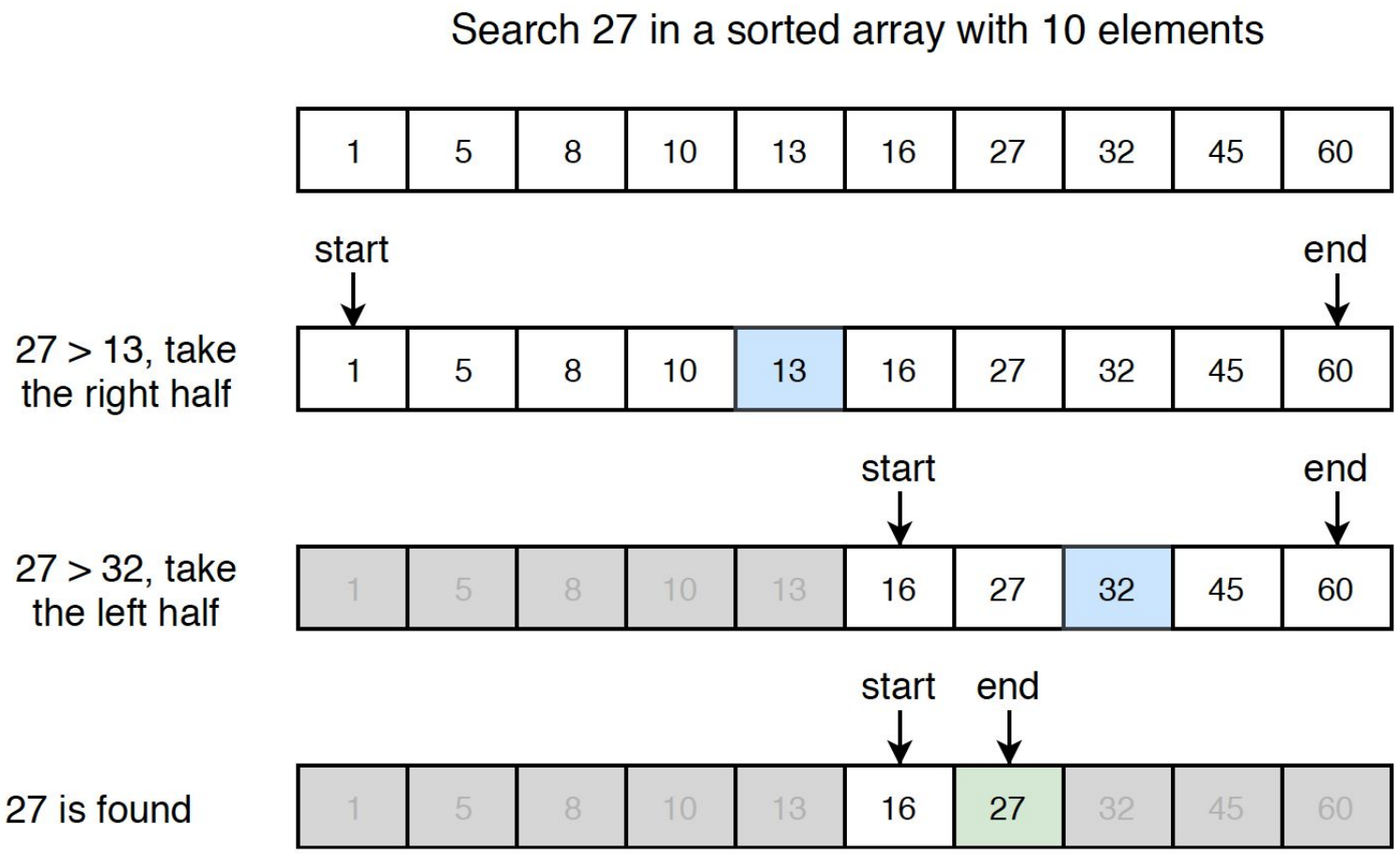
- **Divide**
  - Find the middle element.

    mid = (low + high) / 2
  - The array is divided into two halves.
- **Conquer**
  - If Key < A[mid]
    - Search the left half.
  - If Key > A[mid]
    - Search the right half.
- **Combine**
  - No combine step is required because only one half is explored.

How does Binary Search Algorithm work?



Example


Search 27 in a sorted array with 10 elements
1 5 8 10 13 16 27 32 45 60
start
end
27 > 13, take the right half
1 5 8 10 13 16 27 32 45 60
start
end
27 > 32, take the left half
1 5 8 10 13 16 27 32 45 60
start
end
27 is found
1 5 8 10 13 16 27 32 45 60

How to Implement Binary Search?

It can be implemented in the following two ways

- Iterative Binary Search Algorithm
- Recursive Binary Search Algorithm

Binary Search : Iterative Algorithm

```
BinarySearch(A, key)
low = 0
high = n-1
while(low <= high)
    mid = (low + high)/2
    if A[mid] == key
        return mid
    else if key < A[mid]
        high = mid-1
    else
        low = mid+1
return -1
```

Java Implementation

```java
public static int binarySearch(int arr[], int key) {

        int low = 0;
        int high = arr.length - 1;

        while (low <= high) {

            int mid = low + (high - low) / 2;

            if (arr[mid] == key)
                return mid;

            else if (key < arr[mid])
                high = mid - 1;

            else
                low = mid + 1;
        }

        return -1;
}
```

Binary Search : Recursive

```
public static int binarySearch(int arr[], int low, int high, int key) {

    if(low > high)
        return -1;

    int mid = low + (high-low)/2;

    if(arr[mid] == key)
        return mid;

    if(key < arr[mid])
        return binarySearch(arr, low, mid-1, key);

    return binarySearch(arr, mid+1, high, key);
}
```

Complexity Analysis

- **Time Complexity:**

  -> Best Case: O(1)

  -> Average Case: O(log N)

  -> Worst Case: O(log N)

- **Auxiliary Space:** O(1), If the recursive call stack is considered then the auxiliary space will be O(log N).

Merge Sort

Merge Sort

- Merge Sort is an efficient sorting algorithm based on the **Divide and Conquer** paradigm. It divides the array into smaller subarrays until each subarray contains only one element. Then, it merges these subarrays in sorted order to produce the final sorted array.

Definition

- **Merge Sort** is a Divide and Conquer sorting algorithm that recursively divides an array into two halves, sorts each half independently, and merges the sorted halves to produce a fully sorted array.

Why is Merge Sort a Divide & Conquer Algorithm?

Merge Sort follows the three phases of Divide & Conquer.

- **Divide**
  - Split the array into two equal halves.
- **Conquer**
  - Recursively sort each half.
- **Combine**
  - Merge the two sorted halves into a single sorted array.

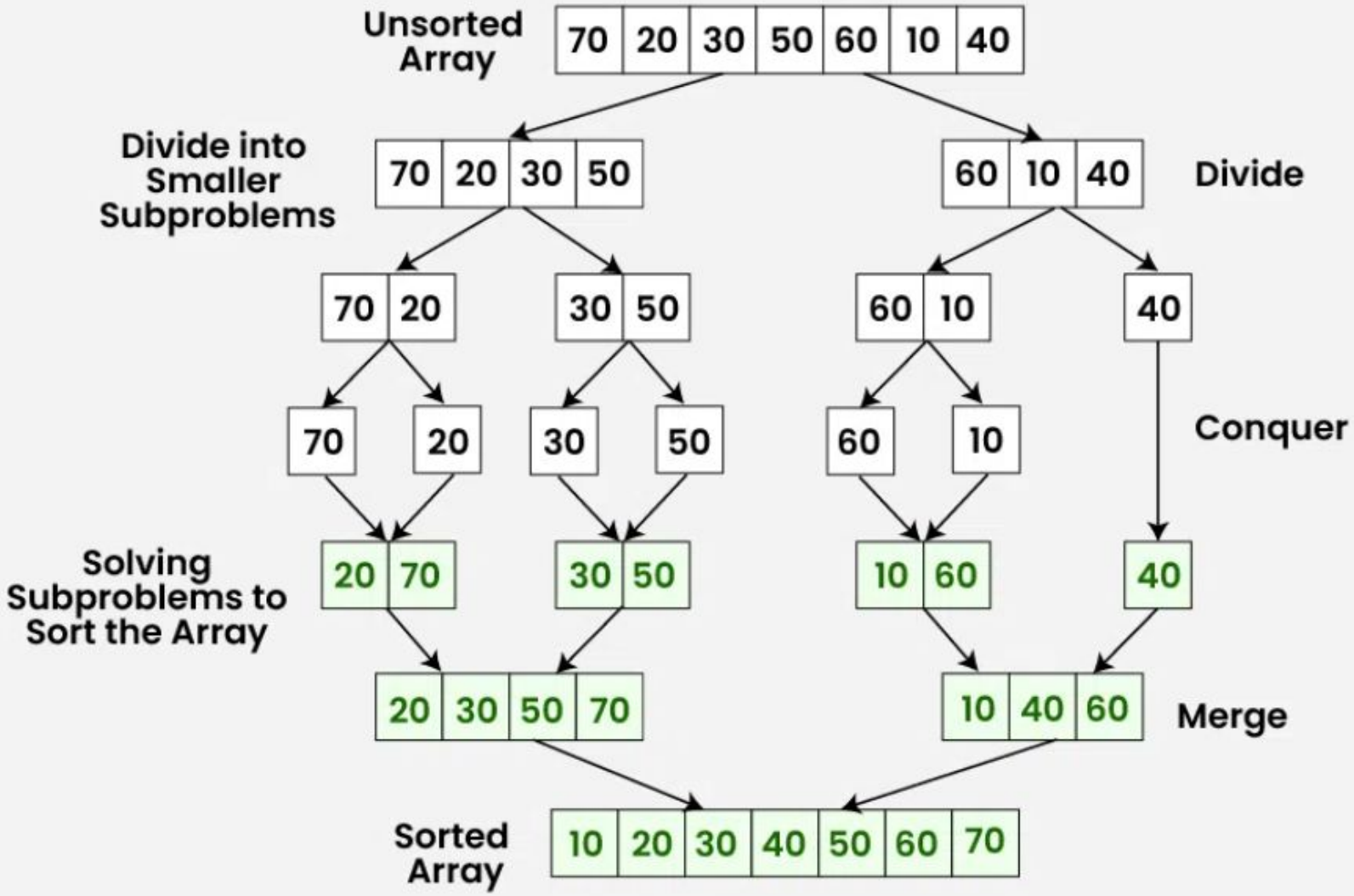
Working of Merge Sort

1. **Divide:** Divide the list or array recursively into two halves until it can no more be divided.

2. **Conquer:** Each subarray is sorted individually using the merge sort algorithm.

3. **Merge:** The sorted subarrays are merged back together in sorted order. The process continues until all elements from both subarrays have been merged.

Working of Merge Sort



Merge Sort Algorithm

Merge Sort

```
MergeSort(A, low, high)

if(low < high)
    mid = (low + high)/2

    MergeSort(A, low, mid)
    MergeSort(A, mid+1, high)

    Merge(A, low, mid, high)
```

Merge Sort Algorithm

Merge Algorithm

Merge(A, low, mid, high)

Create Left[] and Right[]

Copy left half

Copy right half

Compare elements

Insert smaller element into original array

Copy remaining elements

Return

Java Implementation

```java
public class MergeSort {

    public static void mergeSort(int arr[], int low, int high) {

        if (low < high) {

            int mid = low + (high - low) / 2;

            mergeSort(arr, low, mid);

            mergeSort(arr, mid + 1, high);

            merge(arr, low, mid, high);
        }
    }
```

Java Implementation

```java
    public static void merge(int arr[], int low, int mid, int high) {

        int n1 = mid - low + 1;
        int n2 = high - mid;

        int left[] = new int[n1];
        int right[] = new int[n2];

        for (int i = 0; i < n1; i++)
            left[i] = arr[low + i];

        for (int j = 0; j < n2; j++)
            right[j] = arr[mid + 1 + j];
```

Java Implementation

```
        int i = 0;
        int j = 0;
        int k = low;

        while (i < n1 && j < n2) {

            if (left[i] <= right[j])
                arr[k++] = left[i++];
            else
                arr[k++] = right[j++];
        }

        while (i < n1)
            arr[k++] = left[i++];

        while (j < n2)
            arr[k++] = right[j++];
    }
```

Dry run

Recurrence Relation

At every recursive call:

- Two recursive calls are made.
- Each call handles **n/2** elements.
- Merging takes **Θ(n)** time.

Therefore,

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

Base Case

$$T(1) = \Theta(1)$$

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.

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

Also Read about:

- Applications of Merge Sort
- Advantages of Merge Sort
- Disadvantages of Merge Sort

Quick Sort

Quick Sort

- Quick Sort is one of the fastest comparison-based sorting algorithms used in practice. It follows the **Divide and Conquer** paradigm by partitioning the array around a pivot element.
- Unlike Merge Sort, Quick Sort does **not require an additional merge step**. Instead, it partitions the array such that:
  - Elements smaller than the pivot are placed on the left.
  - Elements larger than the pivot are placed on the right.
  - The pivot reaches its correct sorted position.

The same process is then recursively applied to the left and right subarrays.

Why is Quick Sort a Divide & Conquer Algorithm?

Quick Sort follows the three phases of Divide & Conquer:

- **Divide**
  - Choose a pivot and partition the array into two parts.
- **Conquer**
  - Recursively sort the left and right subarrays.
- **Combine**
  - No explicit combine step is required because the partitioning process places the pivot in its correct position.

Steps of Quick Sort

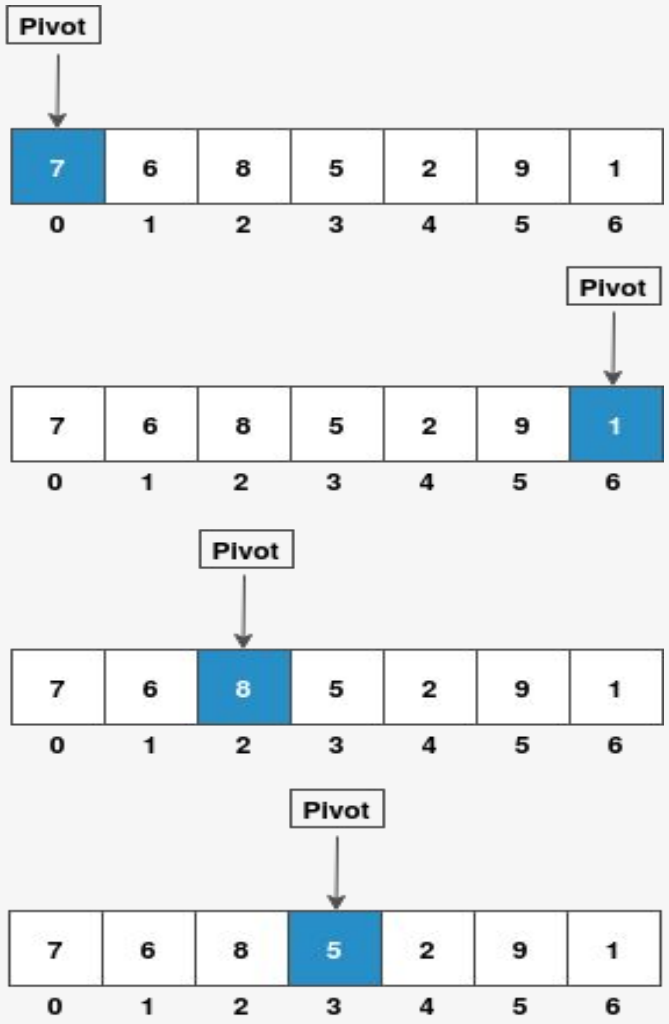
- **Step 1**
  - Choose a Pivot.
- **Step 2**
  - Partition the array.
- **Step 3**
  - Place the pivot in its correct position.
- **Step 4**
  - Recursively sort the left subarray.
- **Step 5**
  - Recursively sort the right subarray.

Choice of Pivot

The performance of Quick Sort depends on the pivot selection.

There are many different choices for picking pivots.

- First Element
- Last Element
- Middle Element
- Random Pivot
- Median of Three



Partition Algorithm

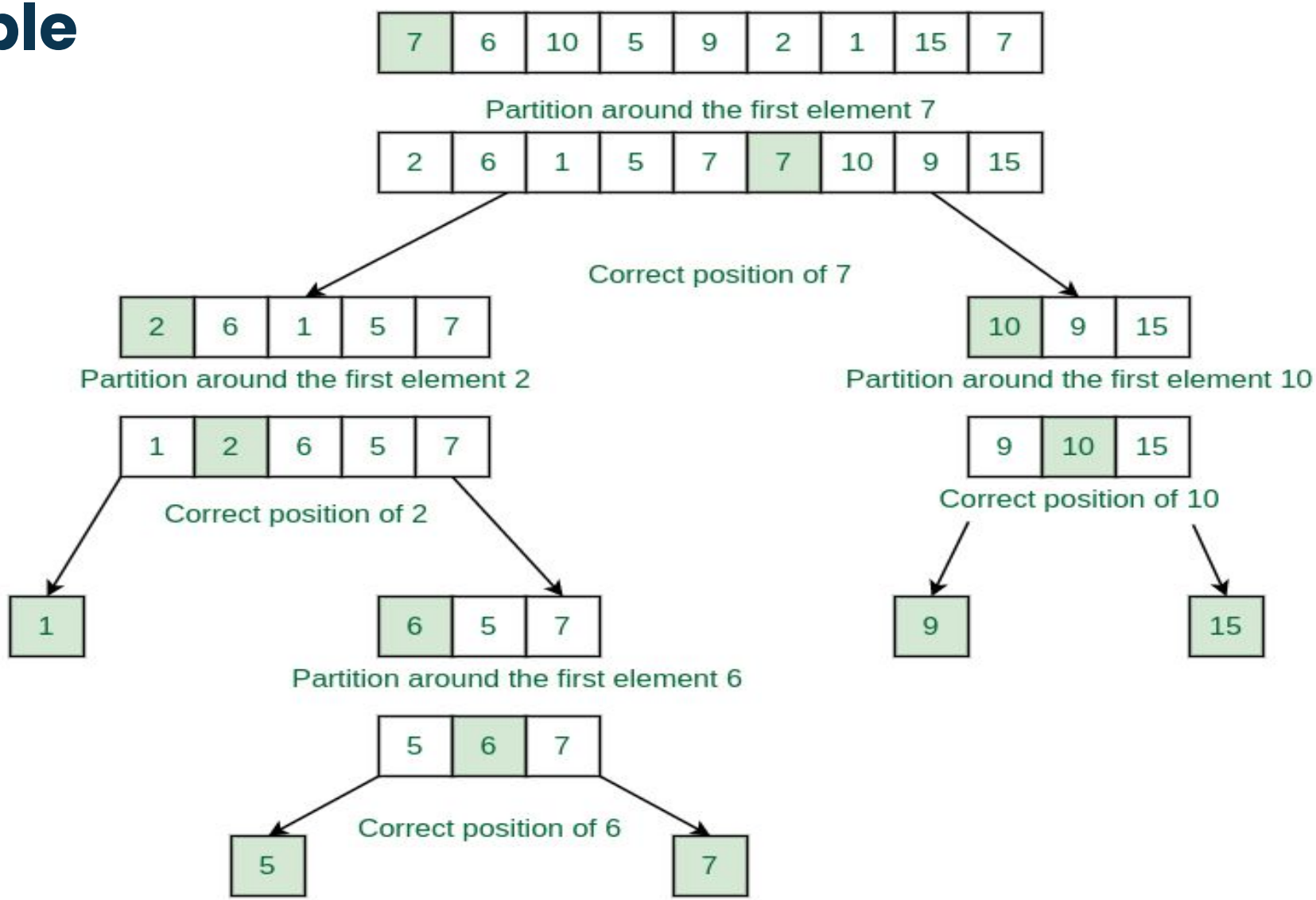
- The key process in **quickSort** is a **partition().**
- Partitioning rearranges the array so that:

  Elements < Pivot | Pivot | Elements > Pivot

After partitioning:

- The pivot is in its final sorted position.
- Left subarray contains smaller elements.
- Right subarray contains larger elements.

Example


7 6 10 5 9 2 1 15 7
Partition around the first element 7
2 6 1 5 7 7 10 9 15
Correct position of 7
2 6 1 5 7
Partition around the first element 2
10 9 15
Partition around the first element 10
1 2 6 5 7
Correct position of 2
9 10 15
Correct position of 10
1
6 5 7
9
15
Partition around the first element 6
5 6 7
Correct position of 6
5
7

Quick Sort Algorithm

```
QuickSort(A, low, high)

if(low < high)

    p = Partition(A, low, high)

    QuickSort(A, low, p-1)

    QuickSort(A, p+1, high)
```

Partition Algorithm

```
Partition(A, low, high)

    pivot = A[low]
    i = low + 1
    j = high
    while(true)
        while(i <= high AND A[i] <= pivot)
            i++
        while(j >= low AND A[j] > pivot)
            j--
        if(i < j)
            swap(A[i], A[j])
        else
            break
    swap(A[low], A[j])
    return j
```

Java Implementation

```java
public class QuickSort {

    public static void quickSort(int arr[], int low, int high) {
        if (low < high) {
            int pivotIndex = partition(arr, low, high);

            quickSort(arr, low, pivotIndex - 1);
            quickSort(arr, pivotIndex + 1, high);
        }
    }
```

```java
public static int partition(int arr[], int low, int high) {

    int pivot = arr[low];      // First element as pivot
    int i = low + 1;
    int j = high;

    while (i <= j) {

        while (i <= high && arr[i] <= pivot) {
            i++;
        }

        while (j >= low && arr[j] > pivot) {
            j--;
        }

        if (i < j) {
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }

    // Place pivot at its correct position
    int temp = arr[low];
    arr[low] = arr[j];
    arr[j] = temp;

    return j;
}
```

Dry run

Recurrence Relation

Best Case

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

- Using the Master Theorem,
  - $T(n) = \Theta(n \log n)$

Average Case

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

- Time Complexity
  - $\Theta(n \log n)$

Worst Case

- The pivot is always the smallest or largest element, creating one empty partition and one partition of size n−1.
- $T(n) = T(n-1) + \Theta(n)$

- Using substitution,
  - $T(n)=\Theta(n^2)$

Complexity Analysis of Quick Sort

Time Complexity:

- **Best Case:** ($\Omega(n \log n)$), Occurs when the pivot element divides the array into two equal halves.
- **Average Case** ($\theta(n \log n)$), On average, the pivot divides the array into two parts, but not necessarily equal.
- **Worst Case:** ($O(n^2)$), Occurs when the smallest or largest element is always chosen as the pivot (e.g., sorted arrays).

Auxiliary Space:

- **Worst-case scenario: O(n)** due to unbalanced partitioning leading to a skewed recursion tree requiring a call stack of size O(n).
- **Best-case scenario: O(log n)** as a result of balanced partitioning leading to a balanced recursion tree with a call stack of size O(log n).

Also Read About

- Advantages
- Disadvantages
- Applications

Comparison: Merge Sort vs Quick Sort

Feature	Merge Sort	Quick Sort
Technique	Divide & Conquer	Divide & Conquer
Best Case	$\Theta(n \log n)$	$\Theta(n \log n)$
Average Case	$\Theta(n \log n)$	$\Theta(n \log n)$
Worst Case	$\Theta(n \log n)$	$\Theta(n^2)$
Space Complexity	$\Theta(n)$	$\Theta(\log n)$ average
Stable	Yes	No
In-place	No	Yes
Suitable for Linked Lists	Yes	No
Suitable for Arrays	Good	Excellent

Max – Min Problem

Max-Min Problem

Finding the **maximum** and **minimum** elements in an array is one of the most common problems in Computer Science.

The problem can be solved using different approaches:

- Brute Force (Linear Scan)
- Divide & Conquer

Although both approaches have **O(n)** time complexity, the Divide & Conquer approach reduces the number of comparisons, making it more efficient.

Problem Statement

Given an array of **n** elements, find:

- The maximum element.
- The minimum element.

Example

Input

A = {25, 17, 40, 9, 31, 55, 12}

Output

Maximum = 55

Minimum = 9

Brute Force Approach

Traverse the array from left to right.

For each element,

- Compare it with the current maximum.
- Compare it with the current minimum.

Algorithm

```
max = A[0]
min = A[0]
for each element
    if element > max
        max = element
    if element < min
        min = element
```

Complexity

Time Complexity

O(n)

Comparisons

2(n-1)

Why Divide & Conquer?

Instead of comparing every element with both maximum and minimum,
We divide the array into two halves.
Each half independently finds

- Maximum
- Minimum

Finally,

only **two comparisons** are needed to combine both answers.

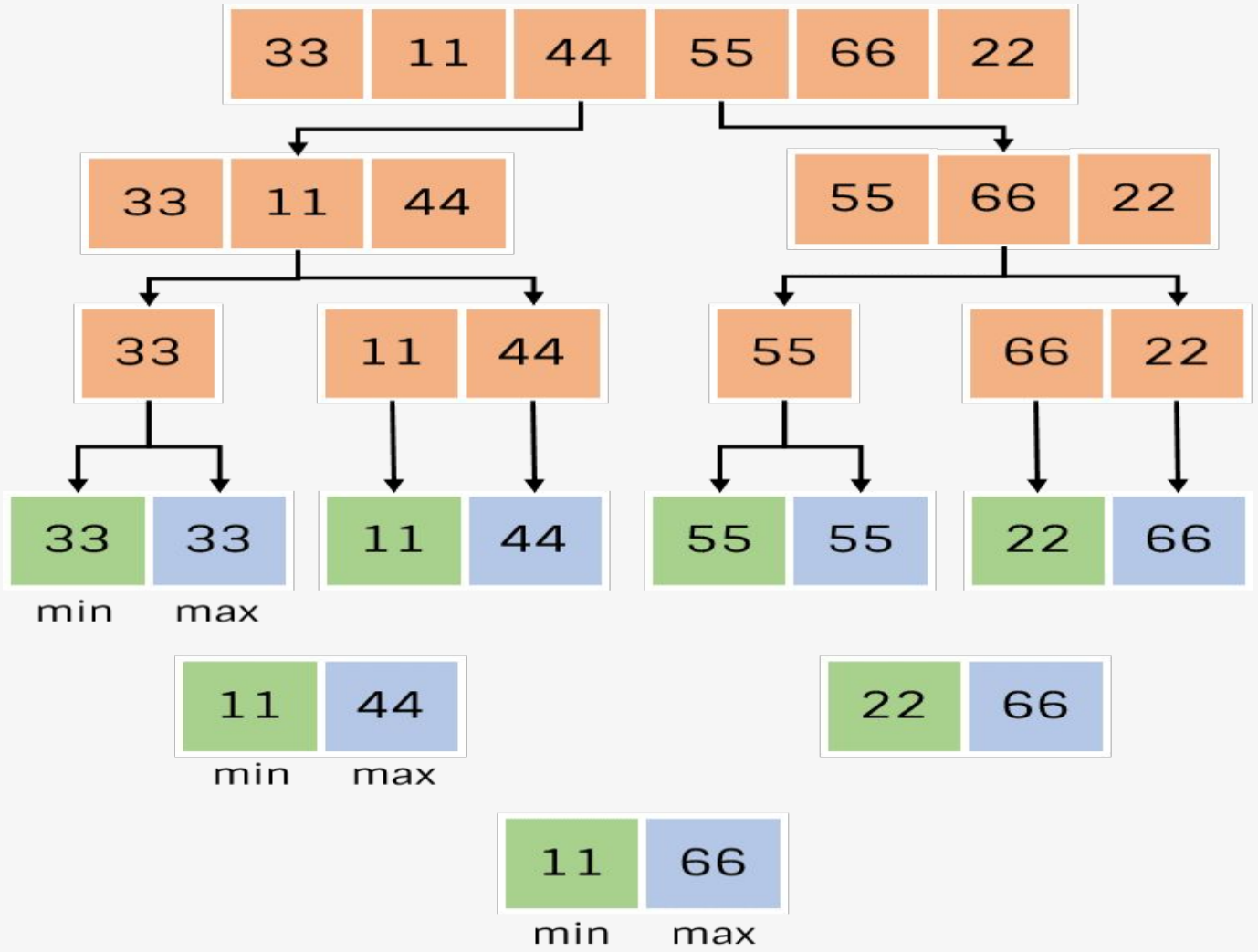
Thus, fewer comparisons are performed than the brute-force approach.

Divide & Conquer Strategy

The algorithm follows three steps.

- **Divide**
  - Split the array into two equal halves.
- **Conquer**
  - Recursively find:
    - Maximum
    - Minimum
  - for each half.
- **Combine**
  - Compare
    - Left Maximum with Right Maximum
    - Left Minimum with Right Minimum
  - to obtain the final maximum and minimum.

Working Example



Algorithm

```
MaxMin(A, low, high)
if(low == high)
    max = min = A[low]
else if(high == low + 1)
    if(A[low] < A[high])
        min = A[low]
        max = A[high]
    else
        min = A[high]
        max = A[low]
```

Algorithm

```
else

    mid = (low + high)/2

    Find Max-Min of Left Half

    Find Max-Min of Right Half

    max = maximum(leftMax,rightMax)

    min = minimum(leftMin,rightMin)
```

Java Implementation

```java
public class MaxMin {

    public static int[] findMaxMin(int arr[], int low, int high) {

        // Base Case 1
        if (low == high) {
            return new int[]{arr[low], arr[low]}; // {min, max}
        }

        // Base Case 2
        if (high == low + 1) {
            if (arr[low] < arr[high])
                return new int[]{arr[low], arr[high]};
            else
                return new int[]{arr[high], arr[low]};
        }

        int mid = (low + high) / 2;

        int[] left = findMaxMin(arr, low, mid);
        int[] right = findMaxMin(arr, mid + 1, high);

        int min = Math.min(left[0], right[0]);
        int max = Math.max(left[1], right[1]);

        return new int[]{min, max};
    }
```

Recurrence Relation

For every recursive call:

- Two recursive calls are made.
- Each recursive call processes **n/2** elements.
- Combining takes constant time.

Therefore,

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

Complexity Analysis

Time Complexity

Case	Complexity
Best	Θ(n)
Average	Θ(n)
Worst	Θ(n)

Space Complexity

Recursive call stack

Θ(log n)

Number of Comparisons

Method	Comparisons
Brute Force	$2(n-1)$
Divide & Conquer	Approximately $(3n/2)-2$

Strassen Matrix Multiplication

Strassen Matrix Multiplication

Matrix multiplication is one of the most important operations in Computer Science.

It is widely used in:

- Artificial Intelligence
- Computer Graphics
- Image Processing
- Machine Learning
- Robotics

The conventional matrix multiplication algorithm requires many multiplication operations.

In **1969**, **Volker Strassen** developed a faster algorithm using the **Divide & Conquer** technique.

What is Matrix Multiplication?

Suppose

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

and

$$B = \begin{bmatrix} e & f \\ g & h \end{bmatrix}$$

Then

$$C = A \times B$$

What is Matrix Multiplication?

- where

$$C = \begin{bmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{bmatrix}$$

Conventional Matrix Multiplication

- For an **n × n** matrix
  - Each element requires **n multiplications**
  - Total elements = **n^2**

Therefore

Time Complexity

$$O(n^3)$$

Need for Strassen Algorithm

- The conventional method performs

$$8$$

- multiplications for multiplying two **2 × 2** matrices.
- Multiplication is computationally more expensive than addition.
- Strassen reduced the number of multiplications from **8 to 7**.
- This reduction improves the overall complexity for large matrices.

Divide and Conquer Strategy

The algorithm follows three steps.

- **Divide**
  - Divide each matrix into four equal-sized submatrices.

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

$$B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

- **Conquer**
  - Instead of performing **8 recursive multiplications**, compute only **7 products**.

Seven Products

- Strassen computes

$$M1 = (A_{11} + A_{22})(B_{11} + B_{22})$$

$$M2 = (A_{21} + A_{22})B_{11}$$

$$M3 = A_{11}(B_{12} - B_{22})$$

$$M4 = A_{22}(B_{21} - B_{11})$$

$$M5 = (A_{11} + A_{12})B_{22}$$

$$M6 = (A_{21} - A_{11})(B_{11} + B_{12})$$

$$M7 = (A_{12} - A_{22})(B_{21} + B_{22})$$

Strassen Matrix Multiplication

$p1 = a(f - h)$
$p3 = (c + d)e$
$p5 = (a + d)(e + h)$
$p7 = (a - c)(e + f)$

$p2 = (a + b)h$
$p4 = d(g - e)$
$p6 = (b - d)(g + h)$

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

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

X , Y 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

Complexity Analysis

- **Time Complexity :** $O(n^{2.808})$, the algorithm first checks if the size of the matrices is 1, and if so, returns the result of a standard matrix multiplication. Otherwise, it divides the matrices into 4 submatrices and performs 7 matrix multiplications recursively. Finally, it combines the results of the multiplications to obtain the final result.

- **Space complexity :** $O(n^2)$

Thanks!

Do you have any questions?

prakash.pandey40626@paruniversity.ac.in
pandeyprakashkumar17@gmail.com
+91 9572573780