CS402 – Analysis & Design of Algorithms

Topic 1: Algorithm & Analysis of Algorithm

(RGPV Exam-Oriented Complete Notes)

---

1. Introduction

Before solving any problem using a computer, we first decide **how to solve it**.

This step-by-step method is called an **Algorithm**.

Example:

Suppose you want to make tea.

Steps:

1. Boil water
2. Add tea leaves
3. Add sugar
4. Add milk
5. Serve tea

These steps form an algorithm.

Similarly, computers solve problems using algorithms.

---

2. Algorithm

Exam Definition

An Algorithm is a finite sequence of well-defined instructions used to solve a particular problem.

Easy Explanation

Algorithm = Recipe of a program.

Just like a cooking recipe tells us how to make food, an algorithm tells a computer how to solve a problem.

---

3. Characteristics of Algorithm

An algorithm must have five properties.

1. Input

Algorithm should accept input.

Example:

Two numbers A and B.

---

2. Output

Algorithm should produce output.

Example:

Sum of A and B.

---

3. Definiteness

Each step should be clear and unambiguous.

Wrong:

"Add some numbers."

Correct:

"Add A and B."

---

4. Finiteness

Algorithm must stop after finite steps.

It should not run forever.

---

5. Effectiveness

Every instruction should be simple and executable.

Diagram

Input
↓
Algorithm
↓
Output

Example Algorithm

Problem: Find sum of two numbers.

Step 1: Start

Step 2: Read A, B

Step 3: Sum = A + B

Step 4: Print Sum

Step 5: Stop

Pseudocode

BEGIN

READ A, B

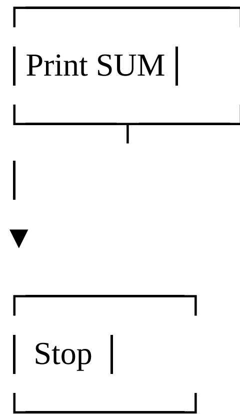
SUM ← A + B

PRINT SUM

END

Flowchart

```
┌──────────┐
│ Start │
└────┬─────┘
│
▼
┌────────────┐
│ Read A,B │
└─────┬──────┘
│
▼
┌────────────┐
│SUM=A+B │
└─────┬──────┘
│
▼
```



Designing an Algorithm

Definition

The process of creating a step-by-step solution for a problem.

Steps in Designing

Step 1

Understand the problem.

Step 2

Identify input and output.

Step 3

Develop logic.

Step 4

Write algorithm.

Step 5

Test algorithm.

Example

Problem:

Find largest of two numbers.

Input:

A, B

Output:

Largest number

Algorithm:

IF A > B

Print A

ELSE

Print B

END IF

Analysis of Algorithm

Definition

Analysis of Algorithm means evaluating performance of an algorithm.

Why Analysis is Needed?

Suppose two algorithms solve the same problem.

Which one is better?

The one that:

✔ Uses less time

✔ Uses less memory

Types of Analysis

1. Time Complexity

Measures execution time.

Question:

How many operations are performed?

Example

for(i=1;i<=n;i++)

print(i);

Loop executes n times.

Time Complexity:

O(n)

---

2. Space Complexity

Measures memory used by algorithm.

Includes:

- Variables
- Arrays
- Dynamic memory

Example

int a,b,c;

Only 3 variables.

Space Complexity:

O(1)

(Constant Space)

Why Time Complexity Matters?

Suppose:

Algorithm A → 100 operations

Algorithm B → 1,00,000 operations

For large inputs:

Algorithm A is much faster.

Therefore we compare algorithms using complexity.

Cases in Algorithm Analysis

Best Case

Minimum time required.

Example:

Searching first element.

Average Case

Average running time.

Worst Case

Maximum time required.

Example:

Searching last element.

Complexity Representation

We use:

Big O

Worst Case

Omega (Ω)

Best Case

Theta (Θ)

Exact Case

Memory Trick

O = Over Limit

Worst Case

Ω = Opening Limit

Best Case

Θ = Total Performance

Exact Case

Real-Life Example

Searching a student in class.

Best Case

Student sits on first bench.

Worst Case

Student sits on last bench.

Average Case

Student sits somewhere in middle.

Advantages of Algorithms

1. Easy Problem Solving

Provides clear steps.

---

2. Easy Debugging

Errors can be identified easily.

---

3. Language Independent

Can be implemented in any programming language.

---

4. Better Optimization

Helps select efficient solution.

Disadvantages

1. Time Consuming

Complex algorithms take time to design.

---

2. Difficult for Large Problems

Large algorithms become lengthy.

Applications

Software Development

Used before coding.

Artificial Intelligence

Decision making algorithms.

Networking

Routing algorithms.

Databases

Searching and sorting.

Comparison Table

Parameter	Algorithm	Program
Definition	Logical Steps	Implementation
Language	Independent	Language Specific
Execution	Cannot Execute Directly	Executable
Purpose	Problem Solving	Solution Implementation

Viva Questions

Q1 What is an Algorithm?

A finite sequence of instructions used to solve a problem.

Q2 What is Time Complexity?

Amount of time taken by algorithm.

Q3 What is Space Complexity?

Amount of memory used by algorithm.

Q4 What are characteristics of algorithm?

Input, Output, Definiteness, Finiteness, Effectiveness.

Q5 Why analyze algorithms?

To compare efficiency and performance.

Common Mistakes

❌ Writing vague steps.

✔ Write precise steps.

❌ Confusing program with algorithm.

✔ Algorithm = Logic

✔ Program = Code

❌ Ignoring complexity.

✔ Always mention time and space complexity.

Exam Keywords

Write these words in answers:

- Finite Sequence
- Well Defined Instructions
- Time Complexity
- Space Complexity
- Best Case
- Worst Case
- Average Case
- Efficiency
- Performance Analysis
- Optimization

One-Page Revision Sheet

Algorithm

Finite sequence of instructions.

Characteristics

1. Input
2. Output
3. Definiteness
4. Finiteness
5. Effectiveness

Analysis

Evaluate performance.

Time Complexity

Execution time.

Space Complexity

Memory used.

Cases

Best Case → Minimum Time

Average Case → Average Time

Worst Case → Maximum Time

Notations

Big O → Worst

Ω → Best

Θ → Exact

Most Important Exam Question

"Define Algorithm. Explain characteristics and analysis of algorithm with suitable example."

Topic 2: Asymptotic Notations

(RGPV Exam-Oriented Complete Notes)

---

1. Introduction

When we write an algorithm, we want to know:

- How fast is it?

- How much memory does it use?
- Is it better than another algorithm?

For small inputs, all algorithms seem fast.

For large inputs, performance matters.

Asymptotic Notations help us measure and compare algorithm efficiency for very large input sizes.

---

2. Simple Definition

Exam Definition

Asymptotic Notations are mathematical notations used to describe the growth rate of an algorithm's time and space complexity as the input size becomes very large.

Easy Explanation

Think of two bikes:

- Bike A runs at 60 km/h
- Bike B runs at 120 km/h

Bike B is faster.

Similarly, asymptotic notations help compare algorithms and determine which one is faster.

---

3. Why Do We Use Asymptotic Notations?

Without asymptotic notation:

❌ Performance depends on:

- Computer speed
- RAM
- Programming language

With asymptotic notation:

✔ Compare algorithms fairly.

✔ Independent of hardware.

✔ Independent of programming language.

✔ Predict performance for large inputs.

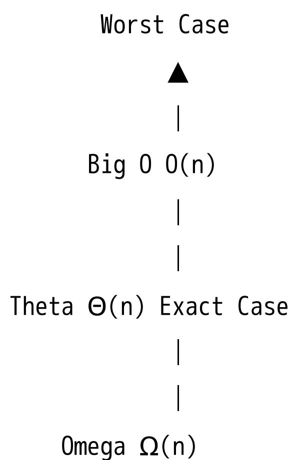
---

4. Types of Asymptotic Notations

There are three main notations:

1. Big O (O)
2. Omega (Ω)
3. Theta (Θ)

Visual Understanding



```
          |
      Best Case
```

---

5. Big O Notation

Definition

Big O notation represents the upper bound (worst-case performance) of an algorithm.

Easy Explanation

It tells:

"Maximum time an algorithm can take."

Example

Linear Search

Searching number 50 in:

10 20 30 40 50

Worst Case:

Check every element.

Operations = n

Therefore:

O(n)O(n)O(n)

Graph

Time

^

|

| /

| /

| /

| /

|/_____> n

Exam Keywords

- Upper Bound
- Worst Case
- Maximum Running Time

---

6. Omega (Ω) Notation

Definition

Omega notation represents the lower bound (best-case performance) of an algorithm.

Easy Explanation

It tells:

"Minimum time an algorithm can take."

Example

Linear Search

Array:

50 10 20 30 40

Searching 50.

Found at first position.

Operations = 1

Therefore:

Ω(1)\Omega(1)Ω(1)

Exam Keywords

- Lower Bound
- Best Case
- Minimum Running Time

---

7. Theta (Θ) Notation

Definition

Theta notation represents the exact bound of an algorithm.

Easy Explanation

It gives both:

- Upper Bound
- Lower Bound

Together.

Example

If an algorithm always performs n operations:

Then:

$\Theta(n)$

Exam Keywords

- Tight Bound
- Exact Complexity
- Precise Growth Rate

---

8. Common Complexity Classes

Constant Complexity

Example:

a = b + c

Complexity:

$O(1)$

Meaning:

Execution time remains constant.

Logarithmic Complexity

Example:

Binary Search

Complexity:

O(logn)O(\log n)O(logn)

Linear Complexity

Example:

Single Loop

for(i=1;i<=n;i++)

Complexity:

O(n)O(n)O(n)

Quadratic Complexity

Example:

Nested Loops

for(i=1;i<=n;i++)
for(j=1;j<=n;j++)

Complexity:

O(n2)O(n^2)O(n2)

Cubic Complexity

Three nested loops.

Complexity:

O(n3)O(n^3)O(n3)

---

9. Complexity Order

Fastest → Slowest

O(1)<O(logn)<O(n)<O(nlogn)<O(n2)<O(n3)<O(2n)<O(n!)O(1)<O(\log n)<O(n)<O(n\log n)<O(n^2)<O(n^3)<O(2^n)<O(n!)O(1)<O(logn)<O(n)<O(nlogn)<O(n2)<O(n3)<O(2n)<O(n!)

Memory Trick

1

↓

Log

↓

n

↓

nlogn

↓

n^2

↓

n^3

↓

2^n

↓

n!

---

10. Solved Examples

Example 1

```
for(i=1;i<=n;i++)

print(i);
```

Loop runs n times.

Complexity:

O(n)O(n)O(n)

Example 2

```
for(i=1;i<=n;i++)

for(j=1;j<=n;j++)

print(i,j);
```

Outer Loop = n

Inner Loop = n

Total Operations = n × n

Complexity:

O(n2)O(n^2)O(n2)

Example 3

Binary Search

Every step halves data.

Complexity:

O(logn)O(\log n)O(logn)

---

11. Comparison Table

Notation	Meaning	Case
O	Upper Bound	Worst
Ω	Lower Bound	Best
Θ	Exact Bound	Average/Exact

---

12. Advantages

1

Easy comparison of algorithms.

2

Independent of hardware.

3

Predicts scalability.

4

Helps choose efficient algorithms.

---

13. Disadvantages

1

Ignores constant factors.

2

Works mainly for large inputs.

3

Does not show exact running time.

---

14. Viva Questions

What is Big O?

Represents worst-case complexity.

What is Omega?

Represents best-case complexity.

What is Theta?

Represents exact complexity.

Why use asymptotic notations?

To compare algorithm efficiency.

Which is fastest?

$O(1)$

Which is slower?

$O(n!)$

---

15. Common Mistakes

❌ Writing $O(n^2)$ for a single loop.

✔ Single loop = $O(n)$

❌ Confusing O and Ω.

✔ O = Worst

✔ Ω = Best

❌ Ignoring dominant term.

Example:

$n^2 + n$

Correct:

$O(n^2)$

---

16. Memory Tricks

Big O

"O = Over"

Maximum Limit

Worst Case

Omega

"Omega Opens"

Minimum Limit

Best Case

Theta

"Theta = Total"

Exact Complexity

---

17. Exam Keywords

Write these in answers:

- Upper Bound
- Lower Bound
- Tight Bound
- Growth Rate
- Time Complexity
- Space Complexity
- Best Case
- Worst Case
- Efficiency
- Scalability

---

18. 5-Mark Answer

Define Asymptotic Notations

Asymptotic Notations are mathematical tools used to describe the performance of algorithms for large input sizes.

Types:

1. Big O – Upper Bound (Worst Case)
2. Omega – Lower Bound (Best Case)
3. Theta – Exact Bound

They help compare algorithm efficiency independent of hardware and programming language.

---

19. 7-Mark Answer

Explain Asymptotic Notations

Definition, types (O, Ω, Θ), diagrams, examples, and comparison table.

Mention:

- O(n)
- O(log n)
- $O(n^2)$

Add advantages and conclusion.

---

20. One-Page Revision Sheet

O = Worst Case

Ω = Best Case

Θ = Exact Case

Fastest

O(1)

Slowest

O(n!)

Single Loop

O(n)

Nested Loop

$O(n^2)$

Binary Search

O(log n)

Merge Sort

O(n log n)

Quick Sort

O(n log n) average

Strassen

O(n^2.81)

Most Asked Question

"Explain Big O, Omega and Theta Notations with suitable examples."

Topic 3: Heap and Heap Sort

(RGPV Exam-Oriented Complete Notes)

---

1. Introduction

Many algorithms require fast insertion, deletion, and finding the largest/smallest element.

For this purpose, a special tree structure called **Heap** is used.

Heap is mainly used in:

- Heap Sort
- Priority Queue
- CPU Scheduling
- Graph Algorithms

---

2. Heap

Exam Definition

A Heap is a Complete Binary Tree that satisfies the Heap Property.

Easy Explanation

Heap is a special binary tree where parent and child follow a specific order.

Think of a family:

Parent should either always be larger than children or always be smaller than children.

---

3. Complete Binary Tree

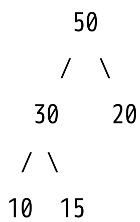
Before understanding Heap, understand Complete Binary Tree.

Definition

A Binary Tree in which all levels are completely filled except possibly the last level.

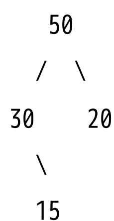
Last level is filled from left to right.

Example



✔ Complete Binary Tree

Not Complete



❌ Gap exists

---

4. Types of Heap

There are two types:

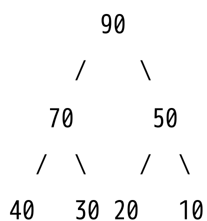
1. Max Heap
2. Min Heap

Max Heap

Definition

In Max Heap, Parent Node is always greater than or equal to Child Nodes.

Example



Condition:

90 > 70,50

70 > 40,30

50 > 20,10

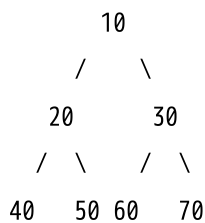
✔ Max Heap

Min Heap

Definition

In Min Heap, Parent Node is always smaller than or equal to Child Nodes.

Example



Condition:

10 < 20,30

20 < 40,50

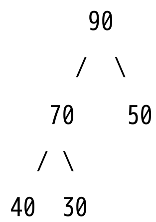
30 < 60,70

✔ Min Heap

Heap Representation in Array

Heap is stored using array.

Example:



```
       90
      /  \
    70    50
   / \
 40  30
```

Array:

```
[90,70,50,40,30]
```

Formula

If node index = i

Left Child

$2i+1$

Right Child

$2i+2$

Parent

$\lfloor (i-1)/2 \rfloor$

Heap Operations

Insertion

Steps

1. Insert at last position.
2. Compare with parent.
3. Swap if required.
4. Repeat until heap property satisfied.

Deletion

Steps

1. Delete root.
2. Move last node to root.
3. Heapify tree.
4. Restore heap property.

Heapify

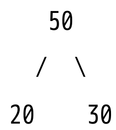
Definition

Process of converting a binary tree into a valid heap.

Example

```
    20
   /  \
 50    30
```

Swap 20 and 50



Now Max Heap formed.

Heap Sort

Definition

Heap Sort is a comparison-based sorting algorithm that uses a Heap data structure.

Easy Explanation

Heap Sort repeatedly removes the largest element from a Max Heap and places it at the end.

Working of Heap Sort

Step 1

Build Max Heap.

Step 2

Swap Root and Last Element.

Step 3

Reduce Heap Size.

Step 4

Heapify Remaining Tree.

Step 5

Repeat until all elements sorted.

Example

Array:

```
40 10 30 50 20
```

Build Max Heap

```
         50
       /    \
     40      30
    /  \
   10  20
```

Array:

```
50 40 30 10 20
```

Pass 1

Swap Root and Last

```
20 40 30 10 50
```

Heapify

```
40 20 30 10 50
```

Pass 2

Swap Root and Last

```
10 20 30 40 50
```

Heapify

```
30 20 10 40 50
```

Continue...

Final Sorted Array

```
10 20 30 40 50
```

Heap Sort Algorithm

Pseudocode

```
HEAPSORT(A)

Build_Max_Heap(A)

for i = n downto 2

swap(A[1],A[i])

heap_size--

Max_Heapify(A,1)

end
```

Flowchart

```
Start
   |
Build Max Heap
   |
Swap Root & Last
   |
Heapify
   |
Sorted ?
  / \
No  Yes
```

```
 |    |
Repeat Stop
```

Complexity Analysis

Building Heap

Complexity:

O(n)O(n)O(n)

Heapify

Complexity:

O(logn)O(\log n)O(logn)

Heap Sort

Best Case

O(nlogn)O(n\log n)O(nlogn)

Average Case

O(nlogn)O(n\log n)O(nlogn)

Worst Case

O(nlogn)O(n\log n)O(nlogn)

Why Heap Sort is Special?

Unlike Quick Sort:

✔ Worst Case also O(n log n)

No performance degradation.

Advantages

1

Guaranteed O(n log n)

2

No extra memory required

(In-place)

3

Efficient for large data

4

Useful in Priority Queues

Disadvantages

1

Not Stable

2

More difficult than Bubble Sort

3

Cache performance is lower

Applications

Priority Queue

Most common use.

CPU Scheduling

Process priority handling.

Dijkstra Algorithm

Graph processing.

Operating Systems

Task scheduling.

Heap vs Binary Search Tree

Heap	BST
Complete Binary Tree	Binary Tree
Fast access to Max/Min	Fast Searching
Used in Heap Sort	Used in Searching
Parent follows Heap Property	Left < Root < Right

Viva Questions

What is Heap?

Complete Binary Tree satisfying Heap Property.

What is Max Heap?

Parent is greater than children.

What is Min Heap?

Parent is smaller than children.

What is Heapify?

Converting a tree into heap.

Complexity of Heap Sort?

O(n log n)

Is Heap Sort Stable?

No.

Is Heap Sort In-Place?

Yes.

Common Mistakes

❌ Heap is not BST.

✔ Heap follows Parent-Child Property only.

❌ Writing $O(n^2)$ for Heap Sort.

✔ Correct:

O(n log n)

❌ Forgetting Heapify step.

✔ Always mention Heapify.

Memory Tricks

Heap

"Parent dominates children."

Max Heap

Maximum element at root.

Min Heap

Minimum element at root.

Heap Sort

Build Heap → Extract Root → Heapify → Repeat

Mnemonic:

B-E-H-R

(Build, Extract, Heapify, Repeat)

Exam Keywords

Write these keywords in answers:

- Complete Binary Tree
- Heap Property
- Max Heap
- Min Heap
- Heapify
- In-Place Sorting
- Priority Queue
- Root Node
- O(n log n)
- Comparison Based Sorting

5-Mark Answer

Explain Heap

Heap is a Complete Binary Tree that satisfies Heap Property. It can be Max Heap or Min Heap. Heap is represented using arrays and is used in Heap Sort and Priority Queues.

7-Mark Answer

Explain Heap Sort with Example

Write:

1. Definition
2. Max Heap Diagram
3. Steps of Heap Sort
4. Example
5. Algorithm
6. Complexity
7. Applications

10-Mark Topper Answer

Heap Sort

Definition: Heap Sort is a comparison-based sorting technique that uses a Max Heap.

Algorithm:

1. Build Max Heap
2. Swap Root with Last Element
3. Reduce Heap Size
4. Heapify
5. Repeat

Complexity:

- Best = O(n log n)
- Average = O(n log n)
- Worst = O(n log n)

Advantages:

- In-place
- Guaranteed Performance

Applications:

- Scheduling
- Priority Queues
- Operating Systems

Conclusion: Heap Sort is an efficient sorting algorithm with guaranteed O(n log n) complexity.

One-Page Revision Sheet

Heap

Complete Binary Tree + Heap Property

Types

Max Heap

Min Heap

Heapify

Convert Tree into Heap

Heap Sort Steps

Build Heap

↓

Swap Root

↓

Heapify

↓

Repeat

Complexities

Build Heap → O(n)

Heapify → O(log n)

Heap Sort → O(n log n)

Most Important Question

"Explain Heap Sort with suitable example and analyze its complexity."

Topic 4: Introduction to Divide and Conquer Technique

(RGPV Exam-Oriented Complete Notes)

---

1. Introduction

Suppose you have to search a name in a dictionary containing 1000 pages.

Will you start from page 1?

❌ No.

You open the middle page.

If the name comes before that page, search the left half.

Otherwise search the right half.

This is the idea of **Divide and Conquer**.

---

2. Definition

Exam Definition

Divide and Conquer is an algorithm design technique in which a problem is divided into smaller subproblems, solved independently, and their solutions are combined to obtain the final solution.

Easy Explanation

Instead of solving one big problem directly:

1. Divide it into smaller parts.
2. Solve each part.
3. Join the answers.

---

3. Why Do We Use Divide and Conquer?

Large problems are difficult.

Small problems are easier.

Divide and Conquer reduces complexity and improves efficiency.

---

4. Basic Steps

Every Divide and Conquer algorithm has 3 steps.

Step 1: Divide

Break the problem into smaller subproblems.

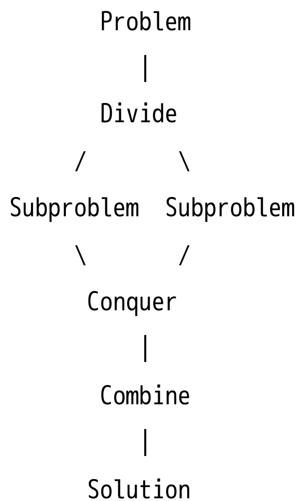
Step 2: Conquer

Solve the subproblems recursively.

Step 3: Combine

Merge solutions of subproblems.

Diagram



Memory Trick

D → C → C

Divide → Conquer → Combine

Remember:

"First Break, Then Solve, Then Join"

---

5. General Structure

Pseudocode

```
Divide_And_Conquer(problem)

IF problem is small

   Solve directly

ELSE

   Divide problem

   Solve subproblems

   Combine results

END IF
```

---

6. Real-Life Example

Suppose you have 100 exam papers.

You alone checking papers:

Slow

Divide among 5 teachers:

✔ Faster

Each teacher checks 20 papers.

Results combined later.

This follows Divide and Conquer.

---

7. Examples of Divide and Conquer Algorithms

Binary Search

Divide array into halves.

Merge Sort

Divide array and merge sorted parts.

Quick Sort

Partition around pivot.

Strassen Matrix Multiplication

Divide matrix into smaller matrices.

---

8. Binary Search as Divide & Conquer

Array:

```
10 20 30 40 50 60 70
```

Search = 60

Step 1

Middle = 40

60 > 40

Search Right Half

Step 2

Middle = 60

Found

Complexity

O(logn)O(\log n)O(logn)

---

9. Merge Sort as Divide & Conquer

Array

```
38 27 43 3
```

Divide

```
38 27 | 43 3
```

Further Divide

```
38 | 27 | 43 | 3
```

Conquer

Sort Individually

```
27 38 | 3 43
```

Combine

```
3 27 38 43
```

Complexity

$O(n\log n)$

---

10. Quick Sort as Divide & Conquer

Array

```
50 30 70 20 60
```

Pivot = 50

Divide

```
30 20 | 50 | 70 60
```

Sort Left

Sort Right

Combine

```
20 30 50 60 70
```

Complexity

Best:

O(nlogn)O(n\log n)O(nlogn)

Worst:

O(n2)O(n^2)O(n2)

---

11. Advantages

1. Faster Execution

Smaller problems are solved quickly.

---

2. Efficient Use of Recursion

Natural recursive structure.

---

3. Suitable for Parallel Processing

Subproblems can run simultaneously.

---

4. Reduces Complexity

Makes large problems manageable.

---

12. Disadvantages

1. Recursion Overhead

Uses extra memory.

---

2. Complex Implementation

Harder than simple iterative methods.

---

3. Extra Space

Merge Sort requires extra memory.

---

13. Comparison

Divide & Conquer	Brute Force
Breaks problem	Solves directly
Faster	Slower
Recursive	Usually iterative
Efficient for large data	Inefficient for large data

---

14. Viva Questions

What is Divide and Conquer?

An algorithm design technique that divides a problem into smaller subproblems and combines their solutions.

What are the steps?

1. Divide
2. Conquer
3. Combine

Give examples.

- Binary Search
- Merge Sort
- Quick Sort
- Strassen Matrix Multiplication

Why is it efficient?

Because smaller problems are easier and faster to solve.

---

15. Common Mistakes

❌ Writing only Divide and Conquer definition.

✔ Also explain:

- Divide

- Conquer
- Combine

❌ Forgetting examples.

✔ Always write:

Binary Search, Merge Sort, Quick Sort.

❌ Forgetting complexity.

✔ Mention algorithm complexities.

---

16. Exam Keywords

Write these keywords:

- Divide
- Conquer
- Combine
- Recursion
- Subproblem
- Efficiency
- Binary Search
- Merge Sort
- Quick Sort
- Recursive Algorithm

---

17. 5-Mark Answer

Explain Divide and Conquer Technique

Divide and Conquer is an algorithm design strategy in which a large problem is divided into smaller subproblems, solved independently, and combined to obtain the final solution.

Steps:

1. Divide
2. Conquer
3. Combine

Examples:

- Binary Search
- Merge Sort
- Quick Sort

Advantages:

- Faster execution
- Reduced complexity

---

18. 7-Mark Answer

Divide and Conquer Technique

Definition: Divide and Conquer is a technique that breaks a problem into smaller subproblems, solves them recursively, and combines the results.

Steps:

- Divide
- Conquer
- Combine

Diagram: (Draw D-C-C diagram)

Examples:

- Binary Search
- Merge Sort
- Quick Sort

Advantages:

- Efficient
- Suitable for recursion

Applications:

- Sorting
- Searching
- Matrix Multiplication

---

19. 10-Mark Topper Answer

Definition

Divide and Conquer is an algorithmic paradigm in which a problem is recursively divided into smaller subproblems until they become simple enough to solve directly. The solutions are then combined to form the final solution.

Steps

1. Divide
2. Conquer
3. Combine

Diagram
(Draw D-C-C diagram)

Examples

- Binary Search
- Merge Sort
- Quick Sort

- Strassen Matrix Multiplication

Advantages

- Efficient
- Recursive
- Parallelizable

Disadvantages

- Extra memory
- Recursion overhead

Conclusion
Divide and Conquer is one of the most important algorithm design techniques and forms the basis of many efficient algorithms.

One-Page Revision Sheet

Divide & Conquer

Break → Solve → Join

Steps

1. Divide
2. Conquer
3. Combine

Examples

- Binary Search
- Merge Sort
- Quick Sort
- Strassen

Complexities

Binary Search → O(log n)

Merge Sort → O(n log n)

Quick Sort → O(n log n) Avg

Quick Sort → $O(n^2)$ Worst

Advantages

- Faster
- Recursive
- Parallel Processing

Most Important Question

🔥 **"Explain Divide and Conquer Technique with suitable examples."**

Divide & Conquer Based Algorithm

Binary Search

(RGPV Exam-Oriented Notes)

1. Introduction

Binary Search is a searching algorithm based on **Divide and Conquer** technique.

It is used to search an element in a **sorted array/list**.

Example:

Array:

```
10 20 30 40 50 60 70
```

Search = `60`

Instead of checking one by one, Binary Search checks the **middle element** and removes half of the array every time.

---

2. Definition

Exam Definition

Binary Search is an efficient searching algorithm that works on sorted data by repeatedly dividing the search interval into two halves.

Easy Explanation

Binary Search ka simple rule:

Middle check karo → left ya right half choose karo → repeat karo.

---

3. Condition for Binary Search

Binary Search works only when data is:

✅ Sorted in ascending order
or
✅ Sorted in descending order

If array is unsorted, Binary Search cannot be applied directly.

---

4. Divide and Conquer in Binary Search

Step	Binary Search Work
Divide	Array ko two halves mein divide karta hai
Conquer	Required half mein search continue karta hai
Combine	Combine step practically needed nahi hota

---

5. Working of Binary Search

Given array:

```
10 20 30 40 50 60 70
```

Search key = `60`

Step 1

Low = 0
High = 6

Middle:

```
mid = (low + high) / 2
```

Mid = 3

Element at mid = 40

Since 60 > 40, search right half.

Step 2

Low = 4
High = 6

Mid = 5

Element at mid = 60

Element found.

---

6. Diagram

```
Initial Array:

10   20   30   40   50   60   70
L              M              H

60 > 40

Right Half:

50   60   70
L    M    H

60 found
```

---

7. Algorithm

1. Start
2. Set `low = 0`, `high = n - 1`
3. Find `mid = (low + high) / 2`
4. If `A[mid] == key`, element found
5. If `key < A[mid]`, search left half
6. If `key > A[mid]`, search right half
7. Repeat until `low <= high`
8. If not found, return unsuccessful
9. Stop

---

8. Pseudocode

```
BinarySearch(A, n, 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
```

---

9. Complexity Analysis

Best Case

Element middle mein mil gaya.

Complexity:

```
O(1)
```

Worst Case

Element last division tak nahi mila.

Complexity:

```
O(log n)
```

Average Case

Complexity:

```
O(log n)
```

Space Complexity

Iterative Binary Search:

```
O(1)
```

Recursive Binary Search:

```
O(log n)
```

---

10. Why Complexity is O(log n)?

Because every step array ko half kar deta hai.

Example:

```
n → n/2 → n/4 → n/8 → ... → 1
```

So total steps:

$\log_2 n$

Therefore:

```
Time Complexity = O(log n)
```

---

11. Advantages

1. Very fast searching algorithm.
2. Better than linear search.
3. Useful for large sorted data.
4. Time complexity is only `O(log n)`.

---

12. Disadvantages

1. Works only on sorted data.
2. Not suitable for linked list.
3. Recursive version uses extra memory.

---

13. Applications

- Searching in sorted arrays
- Dictionary search
- Database indexing
- Competitive programming
- Library management systems

---

14. Binary Search vs Linear Search

Basis	Linear Search	Binary Search
Data	Sorted/Unsorted	Only Sorted
Technique	Sequential	Divide & Conquer
Best Case	O(1)	O(1)
Worst Case	O(n)	O(log n)
Speed	Slow	Fast

Basis	Linear Search	Binary Search
Suitable For	Small data	Large sorted data

---

15. Common Mistakes

❌ Binary Search works on unsorted array

✅ Correct: It works only on sorted array.

❌ Worst case is O(n)

✅ Correct: Worst case is O(log n).

❌ Forgetting low, high, mid update

✅ Always write:

- `high = mid - 1`
- `low = mid + 1`

---

16. Memory Trick

Binary Search = Middle Master

M → Middle check

L → Left if smaller

R → Right if greater

Trick:

"Middle dekho, aadha chhodo."

---

17. Exam Keywords

Write these in answer:

- Sorted Array
- Divide and Conquer
- Middle Element
- Search Interval
- Low
- High
- Mid
- O(log n)
- Efficient Searching

---

18. 5-Mark Answer

Binary Search is an efficient searching algorithm based on Divide and Conquer technique. It works only on sorted arrays. In this method, the middle element is compared with the search key. If the key is equal to the middle element, search is successful. If the key is smaller, searching continues in the left half. If the key is larger, searching continues in the right half.

Its time complexity is `O(log n)` because the search space is divided into half at every step.

---

19. 7-Mark Answer

Binary Search is a searching technique based on Divide and Conquer. It repeatedly divides the sorted array into two halves and searches only in the required half.

Algorithm:

1. Set low = 0 and high = n - 1.
2. Find mid = (low + high) / 2.
3. If A[mid] = key, return mid.
4. If key < A[mid], search left half.
5. If key > A[mid], search right half.
6. Repeat until low <= high.

Example:

Array: `10 20 30 40 50 60 70`
Key: `60`

Middle = 40
60 > 40, search right half.
New middle = 60
Element found.

Complexity:

- Best Case: `O(1)`
- Average Case: `O(log n)`
- Worst Case: `O(log n)`

Conclusion: Binary Search is faster than Linear Search but requires sorted data.

---

20. One-Page Revision Sheet

Binary Search

Definition: Searching algorithm for sorted array.

Technique: Divide and Conquer

Condition: Array must be sorted.

Formula:

```
mid = (low + high) / 2
```

Steps:

Middle check → Left/Right half → Repeat

Complexity:

Best = `O(1)`

Average = `O(log n)`

Worst = `O(log n)`

Space:

Iterative = `O(1)`

Recursive = `O(log n)`

Most Important Line:

Binary Search reduces the search space by half at every step.

Exam Question:

🔥 Explain Binary Search with algorithm, example and complexity.

Merge Sort

(RGPV Exam-Oriented Complete Notes)

---

1. Introduction

Suppose you have the array:

```
38 27 43 3 9 82 10
```

Sorting directly is difficult.

Merge Sort uses the **Divide and Conquer** technique.

Idea:

1. Divide array into smaller parts.
2. Sort each part.

3. Merge sorted parts.

---

2. Definition

Exam Definition

Merge Sort is a Divide and Conquer sorting algorithm that divides the array into smaller subarrays, sorts them recursively, and merges them to obtain the final sorted array.

Easy Explanation

Merge Sort follows:

Divide → Sort → Merge

It keeps dividing until only one element remains.

Since one element is already sorted, merging begins.

---

3. Why Merge Sort?

Normal sorting methods:

- Bubble Sort → $O(n^2)$
- Selection Sort → $O(n^2)$

Merge Sort:

O(nlogn)

Much faster for large data.

---

4. Divide and Conquer in Merge Sort

Divide

Split array into two halves.

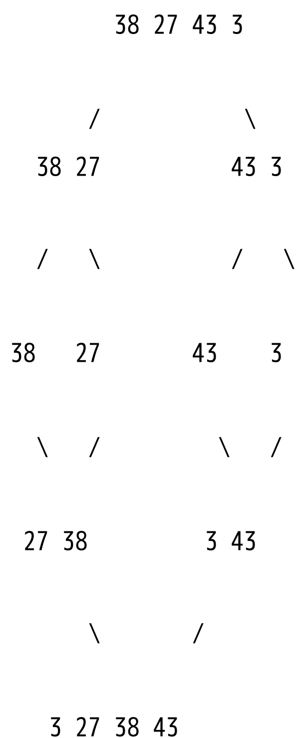
Conquer

Recursively sort both halves.

Combine

Merge sorted halves.

Diagram



---

5. Working Example

Array:

```
38 27 43 3
```

Step 1: Divide

```
38 27 | 43 3
```

Step 2: Divide Again

```
38 | 27 | 43 | 3
```

Now every element is separate.

Step 3: Merge

Merge:

```
38 + 27
```

Result:

```
27 38
```

Merge:

```
43 + 3
```

Result:

```
3 43
```

Step 4: Final Merge

```
27 38
3 43
```

After merging:

```
3 27 38 43
```

Sorted Array obtained.

---

6. Algorithm

Merge Sort Algorithm

```
MergeSort(A, l, r)

IF l < r

   mid = (l+r)/2

   MergeSort(A,l,mid)

   MergeSort(A,mid+1,r)

   Merge(A,l,mid,r)

END IF
```

Merge Procedure

```
Compare elements

Place smaller element first

Continue until all elements merged
```

---

7. Flowchart

```
Start
   |
Divide Array
   |
Subarray Size = 1 ?
   |
Yes ----> Merge
   |
No
   |
Divide Again
   |
Repeat
   |
Sorted Array
   |
Stop
```

---

8. Dry Run

Array:

```
8 4 2 6
```

Divide:

```
8 4 | 2 6
```

Further Divide:

```
8 | 4 | 2 | 6
```

Merge:

```
4 8 | 2 6
```

Final Merge:

```
2 4 6 8
```

---

9. Complexity Analysis

Recurrence Relation

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

Best Case

O(nlogn)

Average Case

O(nlogn)

Worst Case

$O(n \log n)$

Space Complexity

$O(n)$

Extra memory required.

Why Complexity is O(n log n)?

At each level:

Work = n

Levels = log n

Therefore:

$n \times \log n = O(n \log n)$

---

10. Advantages

1

Fast and efficient.

2

Guaranteed O(n log n).

3

Stable Sorting Algorithm.

(Equal elements preserve order.)

4

Good for large datasets.

5

Useful in external sorting.

---

11. Disadvantages

1

Requires extra memory.

2

Recursive calls increase overhead.

3

Not in-place sorting.

---

12. Applications

Database Systems

Sorting records.

External Sorting

Large files.

Data Processing

Large datasets.

Operating Systems

File sorting.

---

13. Merge Sort vs Quick Sort

Feature	Merge Sort	Quick Sort
Technique	Divide & Conquer	Divide & Conquer
Best Case	O(n log n)	O(n log n)
Average Case	O(n log n)	O(n log n)
Worst Case	O(n log n)	$O(n^2)$
Stable	Yes	No
Extra Memory	Required	Less
In-place	No	Yes

---

14. Viva Questions

What is Merge Sort?

A Divide and Conquer sorting algorithm.

Why is it called Merge Sort?

Because sorted subarrays are merged.

Is Merge Sort Stable?

Yes.

Is Merge Sort In-place?

No.

Complexity of Merge Sort?

$O(n \log n)$

Space Complexity?

$O(n)$

---

15. Common Mistakes

❌ Writing $O(n^2)$

✔ Correct:

$O(n \log n)$

❌ Forgetting merge step.

✔ Divide + Merge both important.

❌ Writing In-place sorting.

✔ Merge Sort requires extra memory.

---

16. Memory Trick

Merge Sort

D → D → M

Divide

↓

Divide

↓

Merge

Mnemonic:

"Break Everything, Then Join Smartly."

---

17. Exam Keywords

Write these words:

- Divide and Conquer
- Recursive Sorting
- Merge Procedure
- Stable Sorting
- O(n log n)
- Recursion
- Subarrays
- Sorted Merge
- Extra Memory
- Efficient Sorting

---

18. 5-Mark Answer

Explain Merge Sort

Merge Sort is a Divide and Conquer sorting algorithm. It divides an array into smaller subarrays, sorts them recursively, and merges them to obtain the final sorted array.

Steps:

1. Divide
2. Sort
3. Merge

Complexity:

O(n log n)

Advantages:

- Stable
- Efficient

---

19. 7-Mark Answer

Explain Merge Sort with Example

Definition

Merge Sort is a Divide and Conquer sorting algorithm.

Example

Array:

38 27 43 3

Divide:

38 27 | 43 3

Further Divide:

38 | 27 | 43 | 3

Merge:

27 38 | 3 43

Final:

3 27 38 43

Complexity

Best = O(n log n)

Average = O(n log n)

Worst = O(n log n)

Applications

Database systems and large data processing.

---

20. 10-Mark Topper Answer

Merge Sort

Definition

Merge Sort is a recursive Divide and Conquer sorting algorithm.

Algorithm

1. Divide array into halves.
2. Sort both halves recursively.
3. Merge sorted halves.

Diagram

(Draw divide-merge tree)

Complexity

Best=Average=Worst=O(nlogn)

Space Complexity:

O(n)

Advantages

- Stable
- Efficient

Applications

- Databases
- External Sorting
- Large Files

Conclusion

Merge Sort is one of the most efficient sorting algorithms with guaranteed O(n log n) performance.

One-Page Revision Sheet

Merge Sort

Technique:

Divide & Conquer

Steps

Divide

↓

Sort

↓

Merge

Recurrence

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

Complexity

Best = $O(n \log n)$

Average = $O(n \log n)$

Worst = $O(n \log n)$

Space = $O(n)$

Stable?

Yes

In-place?

No

Applications

Database Sorting

External Sorting

Most Important Question

Quick Sort

(RGPV Exam-Oriented Complete Notes)

---

1. Introduction

Suppose we have an array:

```
50 30 70 20 60 10 80
```

Instead of repeatedly merging like Merge Sort, Quick Sort chooses a special element called **Pivot** and places it in its correct position.

After that:

- Smaller elements go left.
- Larger elements go right.

Then the same process is repeated recursively.

---

2. Definition

Exam Definition

Quick Sort is a Divide and Conquer sorting algorithm that selects a pivot element and partitions the array into two subarrays such that elements smaller than the pivot are placed on the left and

larger elements on the right.

Easy Explanation

Quick Sort ka simple rule:

Choose Pivot → Partition → Sort Left → Sort Right

---

3. Why is it called Quick Sort?

Because in practice it is one of the fastest sorting algorithms.

Average complexity:

O(nlogn)O(n\log n)O(nlogn)

---

4. Divide and Conquer in Quick Sort

Divide

Choose Pivot and divide array.

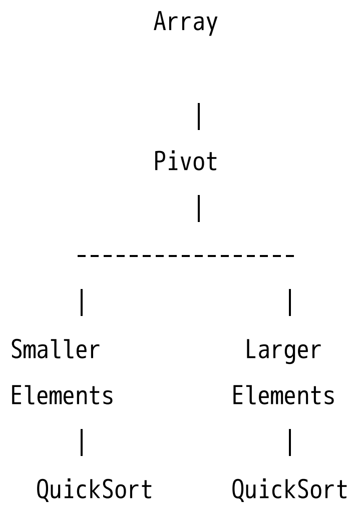
Conquer

Recursively sort left and right parts.

Combine

No merging required.

Diagram



---

5. Pivot

Pivot is the element around which partitioning is performed.

Common choices:

- First Element
- Last Element
- Middle Element
- Random Element

Example

Array:

```
50 30 70 20 60
```

Pivot = 50

After partition:

```
30 20 | 50 | 70 60
```

Now 50 is in its correct position.

---

6. Working Example

Array:

```
50 30 70 20 60
```

Pass 1

Pivot = 50

Partition:

```
30 20 | 50 | 70 60
```

Sort Left

Array:

```
30 20
```

Pivot = 30

Result:

```
20 30
```

Sort Right

Array:

```
70 60
```

Pivot = 70

Result:

```
60 70
```

Final Sorted Array

```
20 30 50 60 70
```

---

7. Algorithm

Steps

1. Choose Pivot.
2. Partition array.
3. Place pivot in correct position.
4. Recursively sort left subarray.
5. Recursively sort right subarray.
6. Stop when subarray size becomes 1.

Pseudocode

```
QuickSort(A, low, high)

IF low < high

    pivot = Partition(A,low,high)

    QuickSort(A,low,pivot-1)

    QuickSort(A,pivot+1,high)

END IF
```

Partition Procedure

```
Choose Pivot
```

```
Move smaller elements left

Move larger elements right

Place pivot at correct position
```

Flowchart

```
Start
   |
Choose Pivot
   |
Partition Array
   |
Pivot Fixed
   |
Sort Left Part
   |
Sort Right Part
   |
Array Sorted?
   |
 Yes
   |
 Stop
```

---

8. Dry Run

Array:

```
8 4 2 6
```

Pivot = 8

Partition:

```
4 2 6 | 8
```

Pivot = 4

Partition:

```
2 | 4 | 6
```

Final:

```
2 4 6 8
```

---

9. Complexity Analysis

Best Case

When pivot divides array equally.

O(nlogn)

Average Case

O(nlogn)

Worst Case

Occurs when pivot is always smallest or largest element.

Example:

```
10 20 30 40 50
```

Already sorted.

Complexity:

O(n^2)

Space Complexity

O(logn)

Recursive stack.

Why Worst Case $O(n^2)$?

Suppose:

```
10 20 30 40 50
```

Choose first element as pivot.

Each time:

Only one element is fixed.

Remaining array size decreases by 1.

Operations:

```
n + (n-1) + (n-2) + ...
```

Result:

$O(n^2)$

---

10. Advantages

1

Very fast in practice.

2

In-place sorting.

(No extra array needed)

3

Cache friendly.

4

Widely used.

---

11. Disadvantages

1

Worst case $O(n^2)$.

2

Not Stable.

3

Performance depends on pivot selection.

---

12. Applications

Database Systems

Sorting records.

Operating Systems

Scheduling.

Large Data Processing

Efficient sorting.

Search Engines

Index management.

---

13. Quick Sort vs Merge Sort

Feature	Quick Sort	Merge Sort
Technique	Divide & Conquer	Divide & Conquer
Best Case	O(n log n)	O(n log n)
Average Case	O(n log n)	O(n log n)
Worst Case	$O(n^2)$	O(n log n)
Stable	No	Yes
Extra Memory	No	Yes
In-place	Yes	No
Practical Speed	Faster	Slightly Slower

---

14. Viva Questions

What is Quick Sort?

A Divide and Conquer sorting algorithm.

What is Pivot?

Special element used for partitioning.

Is Quick Sort Stable?

No.

Is Quick Sort In-place?

Yes.

Average Complexity?

$O(n \log n)$

Worst Complexity?

$O(n^2)$

---

15. Common Mistakes

❌ Forgetting pivot.

✔ Always mention pivot.

❌ Writing worst case $O(n \log n)$.

✔ Correct:

$O(n^2)$

❌ Confusing Quick Sort with Merge Sort.

✔ Quick Sort partitions.

✔ Merge Sort merges.

---

16. Memory Tricks

Quick Sort

PPSR

Pivot

↓

Partition

↓

Sort Left

↓

Sort Right

Remember

"Choose Pivot → Everything revolves around Pivot."

---

17. Exam Keywords

Write these keywords:

- Divide and Conquer
- Pivot Element
- Partitioning
- Recursive Sorting
- In-place Sorting
- O(n log n)
- $O(n^2)$
- Efficient Sorting
- Recursion
- Subarray

---

18. 5-Mark Answer

Explain Quick Sort

Quick Sort is a Divide and Conquer sorting algorithm that selects a pivot element and partitions the array into smaller and larger elements. The process is recursively repeated until the array becomes sorted.

Complexity:

- Best = O(n log n)
- Average = O(n log n)
- Worst = $O(n^2)$

---

19. 7-Mark Answer

Explain Quick Sort with Example

Definition

Quick Sort is a Divide and Conquer sorting algorithm.

Example

Array:

```
50 30 70 20 60
```

Pivot = 50

Partition:

```
30 20 | 50 | 70 60
```

Sort left and right parts recursively.

Final:

```
20 30 50 60 70
```

Complexity

Best = O(n log n)

Average = O(n log n)

Worst = $O(n^2)$

Applications

Database systems and large-scale sorting.

---

20. 10-Mark Topper Answer

Quick Sort

Definition

Quick Sort is a Divide and Conquer sorting algorithm that uses a pivot element for partitioning.

Algorithm

1. Choose Pivot.
2. Partition Array.
3. Sort Left Part.
4. Sort Right Part.
5. Repeat recursively.

Diagram

(Draw Pivot-Partition Diagram)

Complexity

Best=O(nlogn)

Average=O(nlogn)

Worst=O(n^2)

Advantages

- Fast
- In-place

Disadvantages

- Worst case $O(n^2)$
- Not Stable

Conclusion

Quick Sort is one of the fastest practical sorting algorithms and is widely used due to its excellent average-case performance.

One-Page Revision Sheet

Quick Sort

Technique:

Divide & Conquer

Steps

Pivot

↓

Partition

↓

Sort Left

↓

Sort Right

Complexities

Best = O(n log n)

Average = O(n log n)

Worst = $O(n^2)$

Space = O(log n)

Stable?

No

In-place?

Yes

Most Important Concept

Pivot Selection

Most Important Question

Explain Quick Sort with suitable example and analyze its complexity.

Strassen's Matrix Multiplication

(RGPV Exam-Oriented Complete Notes)

---

1. Introduction

Normally, two matrices are multiplied using the traditional method.

For two n × n matrices:

```
A × B = C
```

Traditional multiplication requires:

O(n^3)

operations.

When matrix size becomes very large, this method becomes slow.

To reduce computation time, **Volker Strassen** introduced a faster Divide and Conquer algorithm called **Strassen's Matrix Multiplication**.

---

2. Definition

Exam Definition

Strassen's Matrix Multiplication is a Divide and Conquer algorithm used to multiply two matrices more efficiently than the conventional matrix multiplication method.

Easy Explanation

Normal matrix multiplication performs:

8 multiplications for a 2×2 matrix block.

Strassen discovered that only:

7 multiplications

are enough.

This reduces overall complexity.

---

3. Why Do We Need Strassen's Algorithm?

Traditional Method

For n × n matrix:

O(n^3)

Strassen Method

For n × n matrix:

O(n^2.81)

Since:

```
2.81 < 3
```

Strassen is faster for large matrices.

---

4. Divide and Conquer Concept

Strassen uses:

Divide

Split matrix into four equal submatrices.

Conquer

Perform multiplication recursively.

Combine

Merge results to form final matrix.

Diagram

Matrix A

```
A = | A11  A12 |
    | A21  A22 |
```

Matrix B

```
B = | B11  B12 |
    | B21  B22 |
```

Each matrix is divided into 4 blocks.

---

5. Traditional Matrix Multiplication

For 2×2 matrices:

```
C11 = A11B11 + A12B21

C12 = A11B12 + A12B22

C21 = A21B11 + A22B21

C22 = A21B12 + A22B22
```

Requires:

8 Multiplications

4 Additions

Problem

Multiplication is expensive.

Need fewer multiplications.

---

6. Strassen's Idea

Instead of 8 multiplications,

Use only 7 multiplications.

Let:

M1

(A11+A22)(B11+B22)

M2

(A21+A22)B11

M3

A11(B12-B22)

M4

A22(B21-B11)

M5

(A11+A12)B22

M6

(A21-A11)(B11+B12)

M7

```
(A12-A22)(B21+B22)
```

---

7. Final Matrix Calculation

C11

```
M1 + M4 - M5 + M7
```

C12

```
M3 + M5
```

C21

```
M2 + M4
```

C22

```
M1 - M2 + M3 + M6
```

Memory Trick

Remember:

First create:

```
M1 M2 M3 M4 M5 M6 M7
```

Then:

```
C11 = M1+M4-M5+M7

C12 = M3+M5

C21 = M2+M4

C22 = M1-M2+M3+M6
```

For exams, memorize these four formulas.

---

8. Algorithm

Steps

1. Divide matrices into 4 blocks.
2. Calculate M1 to M7.

3. Calculate C11, C12, C21, C22.
4. Combine submatrices.
5. Obtain final matrix.

Pseudocode

```
Strassen(A,B)

Divide A into A11,A12,A21,A22

Divide B into B11,B12,B21,B22

Compute M1 to M7

Compute C11,C12,C21,C22

Combine C matrix

Return C
```

---

9. Complexity Analysis

Recurrence Relation

```
T(n)=7T(n/2)+O(n²)
```

Using Master Theorem

Complexity becomes:

O(n^2.81)

Comparison

Method	Complexity
Traditional	$O(n^3)$
Strassen	$O(n^{2.81})$

Why Faster?

Traditional:

8 recursive multiplications.

Strassen:

Only 7 recursive multiplications.

One multiplication removed at every level.

Huge saving for large matrices.

---

10. Advantages

1

Faster than conventional multiplication.

2

Uses Divide and Conquer.

3

Efficient for large matrices.

4

Reduces multiplication operations.

---

11. Disadvantages

1

More complex to implement.

2

Additional additions/subtractions required.

3

Not efficient for very small matrices.

4

Numerical stability issues.

---

12. Applications

Scientific Computing

Large matrix calculations.

Computer Graphics

Transformation matrices.

Machine Learning

Large matrix operations.

Image Processing

Matrix transformations.

---

13. Traditional vs Strassen

Feature	Traditional	Strassen
Technique	Direct	Divide & Conquer
Multiplications	8	7
Complexity	$O(n^3)$	$O(n^{2.81})$
Speed	Slower	Faster
Implementation	Easy	Complex

---

14. Viva Questions

What is Strassen Algorithm?

Fast matrix multiplication algorithm.

Who proposed it?

Volker Strassen.

Technique used?

Divide and Conquer.

Number of multiplications?

7

Complexity?

$O(n^{2.81})$

Why faster?

Uses 7 multiplications instead of 8.

---

15. Common Mistakes

❌ Writing complexity $O(n^3)$

✔ Correct:

$O(n^{2.81})$

❌ Forgetting M1–M7.

✔ Memorize formulas.

❌ Forgetting Divide & Conquer.

✔ Always mention it.

---

16. Memory Tricks

Strassen

"8 se 7"

One multiplication saved.

Complexity

Remember:

```
3 → 2.81
```

Traditional → Strassen

Shortcut

7 Multiplications = Fast Multiplication

---

17. Exam Keywords

Write these keywords:

- Divide and Conquer
- Matrix Multiplication
- Recursive Algorithm
- Submatrices
- M1–M7
- $O(n^{2.81})$
- Reduced Multiplications
- Efficient Computation
- Volker Strassen

---

18. 5-Mark Answer

Explain Strassen's Matrix Multiplication

Strassen's Matrix Multiplication is a Divide and Conquer algorithm used to multiply matrices efficiently. It reduces the number of multiplications from 8 to 7 and improves complexity from $O(n^3)$ to $O(n^{2.81})$.

Advantages:

- Faster
- Efficient for large matrices

---

19. 7-Mark Answer

Explain Strassen Matrix Multiplication with Complexity

Definition

Strassen's algorithm is a Divide and Conquer matrix multiplication algorithm.

Steps

1. Divide matrices.
2. Calculate M1–M7.
3. Compute C11–C22.
4. Combine results.

Complexity

O(n^2.81)

Advantages

- Faster than traditional method.

Applications

- Scientific computing
- Graphics

---

20. 10-Mark Topper Answer

Strassen Matrix Multiplication

Definition

Strassen's algorithm is a fast Divide and Conquer matrix multiplication algorithm proposed by Volker Strassen.

Working

- Divide matrices into four blocks.
- Compute M1–M7.
- Compute C11, C12, C21, C22.
- Combine results.

Complexity

$T(n)=7T(n/2)+O(n^2)$

Result:

$O(n^{2.81})$

Advantages

- Faster
- Reduced multiplications

Disadvantages

- Complex implementation

Conclusion

Strassen's algorithm improves matrix multiplication efficiency by reducing recursive multiplications from 8 to 7.

One-Page Revision Sheet

Strassen Matrix Multiplication

Technique:

Divide & Conquer

Key Idea

8 Multiplications → 7 Multiplications

Complexity

Traditional:

O(n^3)

Strassen:

O(n^2.81)

Recurrence

T(n)=7T(n/2)+O(n2)

Most Important Line

Strassen reduces the number of multiplications from 8 to 7.

Most Important Question

🔥 Explain Strassen Matrix Multiplication and analyze its complexity.