SUDDHANANDA SCHOOL OF MANAGEMENT & COMPUTER SCIENCE

Lecture Notes on

DESIGN AND ANALYSIS OF ALGORITHMS (DAA)

(MCPC2001)

UNIT – I

INTRODUCTION TO ALGORITHMS AND ASYMPTOTIC ANALYSIS:

INTRODUCTION

An algorithm is a finite sequence of well-defined instructions used to solve a specific problem or perform a computation.

Algorithms form the foundation of computer programming and software development. Before writing a program, a developer designs an algorithm to describe the step-by-step solution.

DEFINITION OF ALGORITHM

An algorithm is a set of instructions that takes some input, processes it, and produces the desired output in a finite amount of time.

Example

Problem: Add two numbers.

Algorithm:

Step 1: Start

Step 2: Read A and B

Step 3: Compute Sum = A + B

Step 4: Display Sum

Step 5: Stop

CHARACTERISTICS OF AN ALGORITHM

A good algorithm must satisfy the following properties:

1. Input

An algorithm should accept zero or more inputs.

2. Output

An algorithm must produce at least one output.

3. Definiteness

Every step must be clear and unambiguous.

4. Finiteness

The algorithm must terminate after a finite number of steps.

PROF ALISHA SAHOO

5. Effectiveness

Each operation should be simple and executable.

WHY STUDY ALGORITHMS?

Algorithms help in:

1. Solving problems efficiently.
2. Reducing execution time.
3. Optimizing memory usage.
4. Improving software performance.
5. Developing scalable applications.

ALGORITHM ANALYSIS

Algorithm Analysis is the process of determining the efficiency of an algorithm.

Two important factors:

1. Time Complexity
2. Space Complexity

TIME COMPLEXITY

Time Complexity measures the amount of time required by an algorithm to execute as the input size increases.

It is represented as a function of input size n.

EXAMPLE

```
for(i=1;i<=n;i++)
{
  printf("Hello");
}
```

The loop executes n times.

Time Complexity = O(n)

SPACE COMPLEXITY

Space Complexity measures the amount of memory required by an algorithm.

It includes:

1. Input Space
2. Auxiliary Space

EXAMPLE

```
int arr[100];
```

Memory required is proportional to array size.

Space Complexity = O(n)

PROF ALISHA SAHOO

ASYMPTOTIC ANALYSIS

Asymptotic Analysis evaluates the performance of an algorithm as the input size becomes very large.

It ignores:

- Machine speed
- Programming language
- Compiler efficiency

and focuses on growth rate.

TYPES OF ASYMPTOTIC NOTATIONS

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

BIG O NOTATION

Big O represents the upper bound of an algorithm.

It describes the worst-case performance.

Example

$T(n)=3n^2+2n+1$

Ignoring constants and lower-order terms:

Time Complexity = $O(n^2)$

EXAMPLE OF BIG O

```
for(i=1;i<=n;i++)
{
  for(j=1;j<=n;j++)
  {
    printf("*");
  }
}
```

Number of operations:

$n \times n = n^2$

Time Complexity = $O(n^2)$

OMEGA (Ω) NOTATION

Omega notation represents the lower bound of an algorithm.

It describes the best-case performance.

Example:

Linear Search

PROF ALISHA SAHOO

Best Case:

Element found at first position.

$\Omega(1)$

THETA (Θ) NOTATION

Theta notation represents both upper and lower bounds.

Used when best and worst cases grow at the same rate.

Example:

```
for(i=1;i<=n;i++)
{
  printf("Hello");
}
```

$\Theta(n)$

COMPARISON OF ASYMPTOTIC NOTATIONS

Notation	Meaning
$O(n)$	Upper Bound
$\Omega(n)$	Lower Bound
$\Theta(n)$	Tight Bound

COMMON TIME COMPLEXITIES

Complexity	Name
$O(1)$	Constant
$O(\log n)$	Logarithmic
$O(n)$	Linear
$O(n \log n)$	Linearithmic
$O(n^2)$	Quadratic
$O(n^3)$	Cubic
$O(2^n)$	Exponential
$O(n!)$	Factorial

<u>CONSTANT TIME O(1)</u>

Execution time remains constant regardless of input size.

Example:

PROF ALISHA SAHOO

int x=10;

Time Complexity = O(1)

LOGARITHMIC TIME O(log n)

Occurs when problem size is reduced by half repeatedly.

Example:

Binary Search

Time Complexity = O(log n)

LINEAR TIME O(n)

Execution time increases proportionally with input size.

Example:

Linear Search

Time Complexity = O(n)

QUADRATIC TIME $O(n^2)$

Occurs with nested loops.

Example:

Bubble Sort

Time Complexity = $O(n^2)$

CUBIC TIME $O(n^3)$

Three nested loops.

Example:

Matrix Multiplication (basic approach)

Time Complexity = $O(n^3)$

EXPONENTIAL TIME $O(2^n)$

Complexity doubles with each additional input.

Example:

Recursive Fibonacci

Time Complexity = $O(2^n)$

FACTORIAL TIME O(n!)

Example:

Travelling Salesman Problem (Brute Force)

Time Complexity = O(n!)

BEST, WORST AND AVERAGE CASE ANALYSIS

BEST CASE

Minimum execution time.

PROF ALISHA SAHOO

Example:

Linear Search finds element at first position.

Complexity = O(1)

WORST CASE

Maximum execution time.

Example:

Element found at last position.

Complexity = O(n)

AVERAGE CASE

Expected execution time.

Example:

Element found somewhere in middle.

Complexity = O(n)

MATHEMATICAL ANALYSIS OF LOOPS

SINGLE LOOP

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

Runs n times.

Complexity = O(n)

NESTED LOOP

```
for(i=1;i<=n;i++)
{
  for(j=1;j<=n;j++)
  {
  }
}
```

Complexity = $O(n^2)$

TRIANGULAR LOOP

```
for(i=1;i<=n;i++)
{
  for(j=1;j<=i;j++)
  {
  }
}
```

Operations:

PROF ALISHA SAHOO

1+2+3+...+n

= n(n+1)/2

Complexity = $O(n^2)$

RECURSION

Recursion is a technique where a function calls itself.

EXAMPLE

Factorial:

```
Factorial(n)
{
  if(n==1)
    return 1;

  return n*Factorial(n-1);
}
```

Factorial(5)

= 5×4×3×2×1

= 120

ADVANTAGES OF RECURSION

1. Simple code.
2. Easy problem solving.
3. Suitable for tree structures.

DISADVANTAGES OF RECURSION

1. More memory usage.
2. Slower execution.
3. Stack overflow possibility.

DIVIDE AND CONQUER STRATEGY

A powerful algorithm design technique.

Steps:

1. Divide
2. Conquer
3. Combine

EXAMPLE

Binary Search

PROF ALISHA SAHOO

Merge Sort

Quick Sort

BINARY SEARCH ALGORITHM

Binary Search works on sorted arrays.

Algorithm

Step 1: Find middle element.

Step 2: Compare key with middle.

Step 3:

- If equal → Success
- If smaller → Search left half
- If larger → Search right half

Step 4: Repeat until found.

Example

Array:

10 20 30 40 50 60 70

Search Key = 50

Middle = 40

50 > 40

Search right half

50 found.

BINARY SEARCH COMPLEXITY

Best Case:

O(1)

Worst Case:

O(log n)

LINEAR SEARCH

Linear Search examines elements one by one.

Example

Array:

10 20 30 40 50

Search Key = 40

Comparisons:

10 → 20 → 30 → 40

Found.

PROF ALISHA SAHOO

LINEAR SEARCH COMPLEXITY

Best Case:

O(1)

Worst Case:

O(n)

Average Case:

O(n)

DIFFERENCE BETWEEN LINEAR AND BINARY SEARCH

Linear Search	Binary Search
Works on unsorted data	Requires sorted data
O(n)	O(log n)
Simple	Faster
Sequential search	Divide and conquer

PROF ALISHA SAHOO

UNIT – II

DIVIDE AND CONQUER, MERGE SORT, QUICK SORT, HEAP SORT, RECURRENCE RELATIONS AND MASTER THEOREM:

INTRODUCTION TO DIVIDE AND CONQUER

Divide and Conquer is one of the most important algorithm design techniques used to solve complex problems efficiently.

The basic idea is:

1. Divide the problem into smaller sub-problems.
2. Solve each sub-problem recursively.

3. Combine the solutions to obtain the final result.

Many efficient algorithms are based on this strategy.

Examples:

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

GENERAL STRUCTURE OF DIVIDE AND CONQUER

Problem

↓

Divide

↓

Sub Problems

↓

Conquer

↓

Solutions

↓

Combine

↓

Final Solution

ADVANTAGES OF DIVIDE AND CONQUER

1. Reduces complexity.
2. Faster execution.
3. Efficient for large datasets.
4. Easy recursive implementation.
5. Suitable for parallel processing.

DISADVANTAGES

1. Uses recursion.
2. Extra memory may be required.
3. Recursive calls increase overhead.

RECURRENCE RELATION

PROF ALISHA SAHOO

A recurrence relation is an equation that expresses the running time of a recursive algorithm in terms of smaller input sizes.

It helps analyze recursive algorithms.

General Form:

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

Where:

- a = Number of sub-problems
- n/b = Size of each sub-problem
- f(n) = Cost of dividing and combining

EXAMPLE

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

This recurrence represents a simple recursive process.

SOLVING RECURRENCE RELATIONS

Methods:

1. Iteration Method
2. Substitution Method
3. Recursion Tree Method
4. Master Theorem

ITERATION METHOD

Example:

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

Expanding:

$T(n)$

$= T(n-1)+1$

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

$= T(n-3)+1+1+1$

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

Therefore:

$T(n)=O(n)$

RECURSION TREE METHOD

The recurrence is represented as a tree.

PROF ALISHA SAHOO

Each level shows the amount of work performed.

The total cost is obtained by summing all levels.

Useful for analyzing:

- Merge Sort
- Quick Sort

MASTER THEOREM

Master Theorem provides a direct method for solving recurrence relations of the form:

$T(n)=aT\left(\frac{n}{b}\right)+f(n)$

Where:

- $a \geq 1$
- $b > 1$

CASE 1

If:

$f(n)=O(n^{\log_b(a)-\varepsilon})$

Then:

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

CASE 2

If:

$f(n)=\Theta(n^{\log_b(a)})$

Then:

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

CASE 3

If:

$f(n)=\Omega(n^{\log_b(a)+\varepsilon})$

Then:

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

IMPORTANCE OF MASTER THEOREM

1. Simplifies recurrence solving.
2. Saves calculation time.
3. Useful in divide-and-conquer algorithms.

MERGE SORT

Merge Sort is a Divide and Conquer sorting algorithm.

PROF ALISHA SAHOO

It repeatedly divides the array into two halves, sorts them, and merges them.

WORKING OF MERGE SORT

Given Array:

38 27 43 3 9 82 10

Step 1:

Divide into halves

38 27 43 3

9 82 10

Step 2:

Further divide until single elements remain.

38

27

43

3

9

82

10

Step 3:

Merge sorted elements.

27 38

3 43

9 10 82

Step 4:

Final sorted array

3 9 10 27 38 43 82

MERGE SORT ALGORITHM

```
MergeSort(A,l,r)


if(l<r)

{

  mid=(l+r)/2


  MergeSort(A,l,mid)
```

PROF ALISHA SAHOO

```
  MergeSort(A,mid+1,r)


  Merge(A,l,mid,r)
}
```

TIME COMPLEXITY OF MERGE SORT

Recurrence:

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

Using Master Theorem:

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

SPACE COMPLEXITY

$O(n)$

because extra array is required.

ADVANTAGES OF MERGE SORT

1. Stable sorting algorithm.
2. Efficient for large datasets.
3. Guaranteed $O(n \log n)$.

DISADVANTAGES

1. Extra memory required.
2. Not suitable for small arrays.

QUICK SORT

Quick Sort is another Divide and Conquer algorithm.

It selects a pivot element and partitions the array around the pivot.

WORKING OF QUICK SORT

Array:

50 20 10 40 80 60

Choose Pivot = 50

Partition:

20 10 40

50

PROF ALISHA SAHOO

80 60

Sort left and right recursively.

Final Result:

10 20 40 50 60 80

QUICK SORT ALGORITHM

```
QuickSort(A,low,high)


if(low<high)
{
  p=Partition(A)


  QuickSort(A,low,p−1)


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

PARTITION OPERATION

Partition places pivot in its correct position.

Elements smaller than pivot go left.

Elements larger than pivot go right.

TIME COMPLEXITY OF QUICK SORT

Best Case

O(n log n)

Average Case

O(n log n)

Worst Case

$O(n^2)$

Occurs when pivot selection is poor.

Example:

Already sorted array.

SPACE COMPLEXITY

O(log n)

ADVANTAGES OF QUICK SORT

1. Faster in practice.
2. No extra array required.

PROF ALISHA SAHOO

3. Efficient for large datasets.

DISADVANTAGES

1. Worst-case $O(n^2)$.
2. Performance depends on pivot selection.

COMPARISON OF MERGE SORT AND 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)	$O(n^2)$
Extra Space	O(n)	O(log n)
Stable	Yes	No

HEAP

A Heap is a complete binary tree that satisfies the Heap Property.

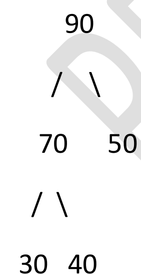
Types:

1. Max Heap
2. Min Heap

MAX HEAP

Parent node is greater than or equal to child nodes.

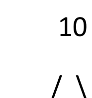
Example:



MIN HEAP

Parent node is smaller than child nodes.

Example:



PROF ALISHA SAHOO

20 30

/ \

40 50

HEAP SORT

Heap Sort uses Heap Data Structure for sorting.

Steps:

1. Build Max Heap.
2. Remove root element.
3. Place root at end.
4. Rebuild Heap.
5. Repeat until sorted.

EXAMPLE

Array:

4 10 3 5 1

Max Heap:

```
     10
    /  \
   5    3
  / \
 4   1
```

Sorted Output:

1 3 4 5 10

HEAP SORT ALGORITHM

```
BuildHeap(A)


for(i=n−1;i>=1;i--)
{
  Swap(A[0],A[i])


  Heapify(A)
}
```

TIME COMPLEXITY OF HEAP SORT

Building Heap:

PROF ALISHA SAHOO

O(n)

Heapify Operations:

O(log n)

Total Complexity:

O(n log n)

SPACE COMPLEXITY

O(1)

In-place sorting.

ADVANTAGES OF HEAP SORT

1. No extra memory required.
2. Guaranteed O(n log n).
3. Efficient for priority queues.

DISADVANTAGES

1. Not stable.
2. Slower than Quick Sort in practice.

COMPARISON OF SORTING ALGORITHMS

Algorithm	Best Case	Average Case	Worst Case
Merge Sort	O(n log n)	O(n log n)	O(n log n)
Quick Sort	O(n log n)	O(n log n)	$O(n^2)$
Heap Sort	O(n log n)	O(n log n)	O(n log n)

PROF ALISHA SAHOO

UNIT – III

GREEDY METHOD, KNAPSACK PROBLEM, HUFFMAN CODING, MINIMUM SPANNING TREE AND SHORTEST PATH ALGORITHMS:

INTRODUCTION TO GREEDY METHOD

The Greedy Method is an algorithm design technique that builds a solution step by step by choosing the locally optimal solution at each stage.

The greedy approach makes the best possible choice at the current step without considering future consequences.

The main idea is:

"Choose the best now and hope it leads to the best overall solution."

CHARACTERISTICS OF GREEDY ALGORITHMS

A problem can be solved using a greedy approach if it satisfies:

1. Greedy Choice Property

A globally optimal solution can be obtained by making locally optimal choices.

2. Optimal Substructure

The optimal solution contains optimal solutions to subproblems.

GENERAL STEPS OF GREEDY METHOD

Problem

PROF ALISHA SAHOO

Select Best Choice

Add to Solution

Check Feasibility

Repeat

Optimal Solution

ADVANTAGES OF GREEDY METHOD

1. Simple implementation.
2. Faster execution.
3. Less memory requirement.
4. Suitable for optimization problems.

DISADVANTAGES OF GREEDY METHOD

1. Does not always give optimal solutions.
2. Problem-specific approach.
3. Not suitable for all optimization problems.

APPLICATIONS OF GREEDY METHOD

1. Fractional Knapsack Problem
2. Huffman Coding
3. Prim's Algorithm
4. Kruskal's Algorithm
5. Dijkstra's Algorithm

FRACTIONAL KNAPSACK PROBLEM

The Fractional Knapsack Problem is one of the most important applications of Greedy Method.

PROBLEM STATEMENT

A knapsack has capacity W.

Each item has:

- Weight
- Profit

Goal:

Maximize profit by selecting items.

PROF ALISHA SAHOO

Unlike 0/1 Knapsack:

Items can be divided into fractions.

EXAMPLE

Item	Profit	Weight
A	60	10
B	100	20
C	120	30

Knapsack Capacity = 50

STEP 1: CALCULATE PROFIT/WEIGHT RATIO

Item	Ratio
A	6
B	5
C	4

STEP 2: SELECT ITEMS

Take A completely.

Weight = 10

Profit = 60

Remaining Capacity = 40

Take B completely.

Weight = 20

Profit = 100

Remaining Capacity = 20

Take 20/30 part of C.

Profit =

120 × (20/30)

= 80

TOTAL PROFIT

60 + 100 + 80

= 240

Maximum Profit = **240**

PROF ALISHA SAHOO

FRACTIONAL KNAPSACK ALGORITHM

1. Calculate Profit/Weight ratio.
2. Sort items in descending order.
3. Select items with highest ratio first.
4. Continue until knapsack is full.

TIME COMPLEXITY

O(n log n)

Due to sorting.

HUFFMAN CODING

Huffman Coding is a Greedy Algorithm used for data compression.

It assigns shorter binary codes to frequently occurring characters and longer codes to less frequent characters.

Developed by:

David Huffman (1952)

APPLICATIONS OF HUFFMAN CODING

1. ZIP Files
2. JPEG Images
3. MP3 Compression
4. PDF Compression

EXAMPLE

Character Frequencies:

Character	Frequency
A	5
B	9
C	12
D	13
E	16
F	45

STEPS OF HUFFMAN CODING

Step 1

Create leaf nodes.

PROF ALISHA SAHOO

A=5

B=9

C=12

D=13

E=16

F=45

Step 2

Select two minimum frequencies.

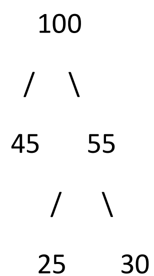
5 + 9 = 14

Step 3

Create new node.

Repeat until one tree remains.

HUFFMAN TREE



(Partial representation)

ADVANTAGES OF HUFFMAN CODING

1. Reduces storage.
2. Faster transmission.
3. Lossless compression.

DISADVANTAGES

1. Tree construction overhead.
2. Not suitable for equal frequencies.

MINIMUM SPANNING TREE (MST)

PROF ALISHA SAHOO

A Minimum Spanning Tree is a spanning tree of a connected weighted graph having minimum total edge weight.

SPANNING TREE

A spanning tree:

- Contains all vertices.
- Has no cycles.
- Contains exactly:

n − 1 edges

Where n = Number of vertices.

EXAMPLE GRAPH

```
A ----- B
| \     |
|  \    |
|   \   |
C ----- D
```

A spanning tree connects all vertices without cycles.

PROPERTIES OF MST

1. Connected graph.
2. No cycles.
3. Minimum edge cost.
4. Contains n−1 edges.

<u>PRIM'S ALGORITHM</u>

Prim's Algorithm is a Greedy Algorithm used to find MST.

WORKING PRINCIPLE

1. Start with any vertex.
2. Select minimum weight edge.
3. Add new vertex.
4. Repeat until all vertices included.

EXAMPLE

Graph:

```
A--2--B
|     |
```

PROF ALISHA SAHOO

3   1

| |

C--4--D

Step 1

Start from A

Select:

A-B = 2

Step 2

Select:

B-D = 1

Step 3

Select:

A-C = 3

MST COST

2 + 1 + 3

= 6

PRIM'S ALGORITHM COMPLEXITY

Using Priority Queue:

O(E log V)

Where:

- E = Number of edges
- V = Number of vertices

KRUSKAL'S ALGORITHM

Kruskal's Algorithm is another Greedy Algorithm for MST.

WORKING PRINCIPLE

1. Sort edges by weight.

PROF ALISHA SAHOO

2. Select smallest edge.
3. Avoid cycles.
4. Continue until MST formed.

EXAMPLE

Edges:

A-B = 2

B-D = 1

A-C = 3

C-D = 4

SORT EDGES

1,2,3,4

Select:

1,2,3

MST Cost:

6

TIME COMPLEXITY

O(E log E)

DIFFERENCE BETWEEN PRIM'S AND KRUSKAL'S

Prim's Algorithm	Kruskal's Algorithm
Vertex based	Edge based
Starts from vertex	Starts from edges
Suitable for dense graph	Suitable for sparse graph
Uses priority queue	Uses sorting

SHORTEST PATH PROBLEM

The objective is to find the minimum distance between two vertices.

Applications:

- GPS Navigation
- Network Routing
- Transportation Systems

DIJKSTRA'S ALGORITHM

PROF ALISHA SAHOO

Dijkstra's Algorithm finds shortest paths from a source vertex to all other vertices.

Developed by:

Edsger Dijkstra

WORKING PRINCIPLE

1. Select source vertex.
2. Assign distance 0.
3. Assign infinity to others.
4. Select nearest unvisited vertex.
5. Update distances.
6. Repeat.

EXAMPLE

Graph:

A ---4--- B

| |

2 5

| |

C ---1--- D

Source = A

STEP 1

Distance(A)=0

B=∞

C=∞

D=∞

STEP 2

Update:

C=2

B=4

STEP 3

Select C

Update:

D=3

PROF ALISHA SAHOO

FINAL DISTANCES

Vertex	Distance
A	0
B	4
C	2
D	3

TIME COMPLEXITY

Using Priority Queue:

O(E log V)

ADVANTAGES OF DIJKSTRA'S ALGORITHM

1. Efficient shortest path calculation.
2. Widely used in networks.
3. Suitable for positive edge weights.

LIMITATIONS

1. Cannot handle negative edge weights.
2. Less suitable for dynamic graphs.

GREEDY ALGORITHM VS DYNAMIC PROGRAMMING

Greedy Method	Dynamic Programming
Local optimum choice	Global optimum approach
Faster	Slower
Less memory	More memory
Simpler	Complex

REAL-LIFE APPLICATIONS

Fractional Knapsack

- Cargo Loading
- Resource Allocation

Huffman Coding

- File Compression

PROF ALISHA SAHOO

- Image Compression

Prim's & Kruskal's

- Telephone Networks
- Road Construction

Dijkstra's Algorithm

- GPS Navigation
- Internet Routing

PROF ALISHA SAHOO

UNIT – IV

DYNAMIC PROGRAMMING, BACKTRACKING AND BRANCH & BOUND:

INTRODUCTION TO DYNAMIC PROGRAMMING

Dynamic Programming (DP) is an algorithm design technique used to solve optimization and decision problems by breaking them into smaller overlapping subproblems and storing their solutions.

It was introduced by **Richard Bellman** in the 1950s.

The basic idea is:

Solve each subproblem only once and store its result for future use.

This avoids repeated computations and improves efficiency.

PRINCIPLES OF DYNAMIC PROGRAMMING

Dynamic Programming is based on two important properties:

1. Optimal Substructure

A problem has optimal substructure if an optimal solution can be obtained from optimal solutions of its subproblems.

Example

Shortest Path Problem

The shortest path from A to C through B consists of:

- Shortest path from A to B
- Shortest path from B to C

2. Overlapping Subproblems

The same subproblems are solved repeatedly.

Dynamic Programming stores their results and reuses them.

APPROACHES OF DYNAMIC PROGRAMMING

1. Top-Down Approach (Memoization)

Uses recursion and stores results.

Example:

Factorial
Fibonacci

2. Bottom-Up Approach (Tabulation)

Builds solution from smaller subproblems.

PROF ALISHA SAHOO

Generally more efficient.

FIBONACCI USING DYNAMIC PROGRAMMING

Fibonacci Series:

0 1 1 2 3 5 8 13 ...

Recurrence Relation:

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

Base Cases:

F(0)=0

F(1)=1

PROBLEM WITH SIMPLE RECURSION

For calculating F(5):

```
F(5)
 ├── F(4)
 |    ├── F(3)
 |    └── F(2)
 └── F(3)
```

Notice:

F(3) is calculated multiple times.

This causes inefficiency.

DYNAMIC PROGRAMMING SOLUTION

Store computed values in an array.

F[0]=0

F[1]=1

```
for(i=2;i<=n;i++)
{
  F[i]=F[i-1]+F[i-2]
}
```

TIME COMPLEXITY

O(n)

PROF ALISHA SAHOO

Space Complexity:

O(n)

0/1 KNAPSACK PROBLEM

The 0/1 Knapsack Problem is one of the most important applications of Dynamic Programming.

PROBLEM STATEMENT

A knapsack can carry W units of weight.

Given:

Item	Weight	Profit
1	2	12
2	1	10
3	3	20
4	2	15

Find maximum profit without exceeding capacity.

CHARACTERISTICS

Each item:

- Either selected (1)
- Or not selected (0)

Fractional selection is not allowed.

DYNAMIC PROGRAMMING FORMULATION

If weight exceeds capacity:

K[i][w]=K[i-1][w]

Otherwise:

K[i][w]=max(

K[i-1][w],

Profit+K[i-1][w-weight]

)

TIME COMPLEXITY

O(nW)

Where:

- n = Number of items
- W = Capacity

PROF ALISHA SAHOO

ADVANTAGES OF DYNAMIC PROGRAMMING

1. Efficient solution.
2. Avoids repeated calculations.
3. Suitable for optimization problems.
4. Reduces execution time.

DISADVANTAGES

1. Higher memory usage.
2. Difficult implementation.

BACKTRACKING

Backtracking is an algorithm design technique used to solve problems incrementally.

If a solution path becomes invalid, the algorithm goes back (backtracks) and tries another path.

BASIC IDEA

Choose

↓

Explore

↓

Valid?

↓

Yes → Continue

No → Backtrack

CHARACTERISTICS OF BACKTRACKING

1. Recursive technique.
2. Generates solutions systematically.
3. Prunes invalid solutions.
4. Suitable for combinatorial problems.

APPLICATIONS OF BACKTRACKING

1. N-Queens Problem
2. Graph Coloring
3. Hamiltonian Cycle
4. Sudoku Solver
5. Knight's Tour

N-QUEENS PROBLEM

PROF ALISHA SAHOO

PROBLEM STATEMENT

Place N queens on an N × N chessboard such that:

- No two queens share the same row.
- No two queens share the same column.
- No two queens share the same diagonal.

4-QUEENS SOLUTION

Q . . .

. . Q .

. Q . .

. . . Q

Where:

Q = Queen

ALGORITHM

1. Place queen in first row.
2. Move to next row.
3. Check safety.
4. If safe, place queen.
5. If not safe, backtrack.
6. Continue until solution found.

TIME COMPLEXITY

Approximately:

O(N!)

GRAPH COLORING PROBLEM

Assign colors to vertices of a graph such that:

- Adjacent vertices have different colors.

Goal:

Use minimum number of colors.

EXAMPLE

A ----- B

| |

| |

D ----- C

PROF ALISHA SAHOO

Possible Coloring:

A = Red

B = Blue

C = Red

D = Blue

HAMILTONIAN CYCLE

A Hamiltonian Cycle is a cycle that visits every vertex exactly once and returns to the starting vertex.

EXAMPLE

A → B → C → D → A

Every vertex visited once.

ADVANTAGES OF BACKTRACKING

1. Systematic solution search.
2. Reduces unnecessary computations.
3. Solves constraint satisfaction problems.

DISADVANTAGES

1. High time complexity.
2. Not suitable for very large inputs.

BRANCH AND BOUND

Branch and Bound is an optimization technique used for solving combinatorial optimization problems.

Unlike Backtracking:

- Backtracking searches feasible solutions.
- Branch and Bound searches optimal solutions.

BASIC IDEA

1. Divide problem into subproblems.
2. Compute bounds.
3. Eliminate subproblems that cannot produce better solutions.
4. Continue until optimal solution found.

TERMINOLOGIES

Branching

Creating subproblems.

Bounding

Calculating upper/lower bounds.

PROF ALISHA SAHOO

Pruning

Discarding useless branches.

APPLICATIONS OF BRANCH AND BOUND

1. Travelling Salesman Problem (TSP)
2. 0/1 Knapsack
3. Assignment Problem
4. Job Scheduling

TRAVELLING SALESMAN PROBLEM (TSP)

PROBLEM STATEMENT

A salesman must visit all cities exactly once and return to the starting city.

Goal:

Find minimum cost route.

EXAMPLE

```
    A
  /  \
 10   15
 /      \
B---20---C
```

Possible Tours:

A → B → C → A

A → C → B → A

Choose minimum cost route.

BRANCH AND BOUND APPROACH

Step 1:

Generate possible tours.

Step 2:

Calculate lower bound.

Step 3:

Prune expensive routes.

Step 4:

Select minimum cost tour.

PROF ALISHA SAHOO

DIFFERENCE BETWEEN BACKTRACKING AND BRANCH & BOUND

Backtracking	Branch & Bound
Finds feasible solution	Finds optimal solution
Uses constraint checking	Uses bounds
May not find optimum	Finds optimum
Decision problems	Optimization problems

COMPARISON OF DYNAMIC PROGRAMMING, BACKTRACKING AND BRANCH & BOUND

Feature	Dynamic Programming	Backtracking	Branch & Bound
Stores solutions	Yes	No	No
Optimization	Yes	Limited	Yes
Recursive	Sometimes	Yes	Yes
Memory Usage	High	Low	Medium
Speed	Fast	Slow	Moderate

REAL-LIFE APPLICATIONS

Dynamic Programming

- GPS Navigation
- Resource Allocation
- Inventory Management

Backtracking

- Sudoku Solver
- Puzzle Games
- Timetable Generation

Branch and Bound

- Airline Scheduling
- Vehicle Routing
- Network Optimization

ADVANTAGES OF DAA TECHNIQUES

1. Efficient problem solving.
2. Reduced execution time.

PROF ALISHA SAHOO

3. Better optimization.
4. Suitable for large applications.

PROF ALISHA SAHOO

UNIT – V

COMPUTATIONAL COMPLEXITY, P, NP, NP-COMPLETE, NP-HARD AND APPROXIMATION ALGORITHMS:

INTRODUCTION TO COMPUTATIONAL COMPLEXITY

Computational Complexity is the study of the resources required by an algorithm to solve a problem.

The main resources considered are:

1. Time
2. Memory

Complexity Theory helps classify problems according to how difficult they are to solve.

WHY STUDY COMPLEXITY?

Complexity analysis helps:

- Compare algorithms.
- Determine feasibility of solutions.
- Identify difficult problems.
- Design efficient algorithms.

COMPLEXITY OF PROBLEMS

Problems are generally classified into:

Easy Problems

Hard Problems

Very Hard Problems

In Computer Science, these are represented by:

- P
- NP
- NP-Complete
- NP-Hard

DETERMINISTIC ALGORITHM

A Deterministic Algorithm follows exactly one sequence of steps for a given input.

It always produces the same output.

Example

Binary Search

Linear Search

Merge Sort

NON-DETERMINISTIC ALGORITHM

A Non-Deterministic Algorithm can explore multiple possibilities simultaneously.

It is a theoretical concept used in complexity analysis.

PROF ALISHA SAHOO

CLASS P

P stands for:

Polynomial Time

P is the class of problems that can be solved by a deterministic algorithm in polynomial time.

DEFINITION

A problem belongs to class P if it can be solved in:

$O(n)$

$O(n^2)$

$O(n^3)$

$O(n^k)$

for some constant k.

EXAMPLES OF P PROBLEMS

Linear Search

Complexity:

$O(n)$

Binary Search

Complexity:

$O(\log n)$

Merge Sort

Complexity:

$O(n \log n)$

Dijkstra's Algorithm:

Complexity:

$O(E \log V)$

CHARACTERISTICS OF P

1. Efficiently solvable.
2. Practical implementation possible.
3. Polynomial-time solutions exist.

CLASS NP

NP stands for:

Non-Deterministic Polynomial Time

A problem belongs to NP if its solution can be verified in polynomial time.

IMPORTANT POINT

PROF ALISHA SAHOO

For NP problems:

Finding the solution may be difficult.

Checking the solution is easy.

EXAMPLE

Sudoku Puzzle

Finding solution:

Difficult

Checking solution:

Easy

EXAMPLE OF NP PROBLEM

Suppose someone gives a route for Travelling Salesman Problem.

Checking whether:

- Every city visited once.
- Total cost is correct.

can be done quickly.

Thus verification is easy.

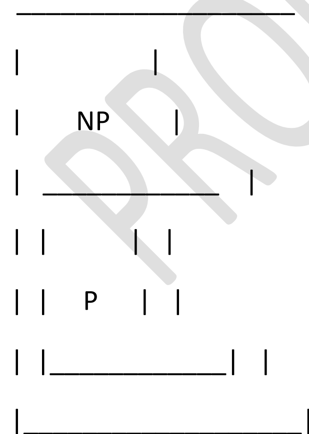
RELATIONSHIP BETWEEN P AND NP

Every problem in P is also in NP.

Therefore:

P ⊆ NP

DIAGRAM



THE P VS NP PROBLEM

One of the most important unsolved problems in Computer Science.

Question:

Is P = NP ?

PROF ALISHA SAHOO

No proof currently exists.

IF P = NP

Then:

- Every difficult problem becomes easy.
- Cryptography becomes vulnerable.
- Optimization becomes easier.

IF P ≠ NP

Then:

- Some problems remain difficult forever.

Most researchers believe:

P ≠ NP

NP-COMPLETE PROBLEMS

NP-Complete problems are the hardest problems in NP.

If one NP-Complete problem is solved in polynomial time, then all NP problems can be solved in polynomial time.

CONDITIONS FOR NP-COMPLETE

A problem must satisfy:

Condition 1

Problem belongs to NP.

Condition 2

Every NP problem can be reduced to it in polynomial time.

EXAMPLES OF NP-COMPLETE PROBLEMS

1. SAT (Boolean Satisfiability)
2. 3-SAT
3. Clique Problem
4. Vertex Cover Problem
5. Hamiltonian Cycle Problem
6. Subset Sum Problem

SATISFIABILITY (SAT) PROBLEM

PROF ALISHA SAHOO

Given a Boolean expression:

(A OR B)

AND

(B OR C)

Determine whether there exists an assignment of variables that makes expression true.

SAT was the first NP-Complete problem.

HAMILTONIAN CYCLE PROBLEM

A Hamiltonian Cycle visits every vertex exactly once and returns to starting vertex.

Example:

A → B → C → D → A

Finding such a cycle is NP-Complete.

SUBSET SUM PROBLEM

Given numbers:

2, 4, 7, 10

Target:

11

Check whether a subset exists whose sum equals 11.

Solution:

4 + 7 = 11

This problem is NP-Complete.

NP-HARD PROBLEMS

NP-Hard problems are at least as difficult as NP-Complete problems.

They may or may not belong to NP.

DEFINITION

A problem is NP-Hard if every NP problem can be reduced to it in polynomial time.

IMPORTANT POINT

NP-Hard problems do not necessarily have solutions that can be verified in polynomial time.

EXAMPLES OF NP-HARD PROBLEMS

1. Travelling Salesman Optimization Problem
2. Halting Problem
3. Job Shop Scheduling
4. Integer Programming

PROF ALISHA SAHOO

TRAVELLING SALESMAN PROBLEM (OPTIMIZATION VERSION)

Given:

Several cities.

Objective:

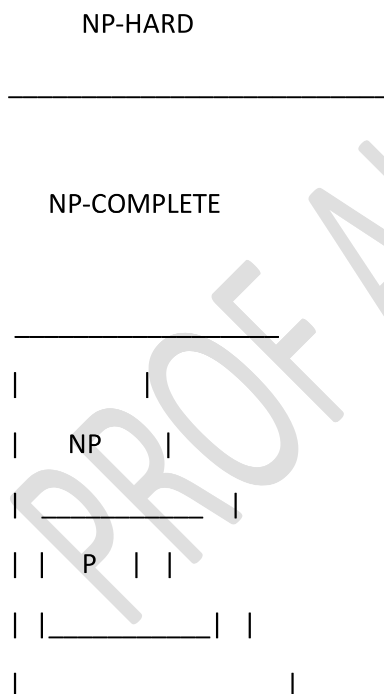
Find minimum cost tour visiting every city exactly once.

Finding optimal route is NP-Hard.

DIFFERENCE BETWEEN NP-COMPLETE AND NP-HARD

NP-Complete	NP-Hard
Belongs to NP	May not belong to NP
Solution verifiable in polynomial time	Verification may be difficult
Subset of NP	Broader category
Example: SAT	Example: Halting Problem

RELATIONSHIP AMONG P, NP, NP-COMPLETE AND NP-HARD



POLYNOMIAL TIME REDUCTION

Reduction is used to transform one problem into another.

If Problem A can be converted into Problem B in polynomial time, then:

$A \leq_p B$

Meaning:

PROF ALISHA SAHOO

B is at least as difficult as A.

IMPORTANCE OF REDUCTION

1. Proves NP-Completeness.
2. Compares difficulty of problems.
3. Helps classify problems.

APPROXIMATION ALGORITHMS

Approximation Algorithms are used for NP-Hard problems.

They produce near-optimal solutions in reasonable time.

NEED FOR APPROXIMATION ALGORITHMS

For NP-Hard problems:

Exact solutions require enormous computation.

Approximation algorithms provide acceptable solutions quickly.

CHARACTERISTICS

1. Faster execution.
2. Near-optimal solutions.
3. Practical implementation.
4. Useful for large datasets.

VERTEX COVER PROBLEM

Given a graph:

Select minimum vertices covering all edges.

Finding exact solution:

NP-Complete

Approximation algorithm provides near-optimal solution.

APPROXIMATION RATIO

Measures quality of approximation.

Formula:

Approximation Ratio

=

Approximate Solution

Optimal Solution

Smaller ratio indicates better performance.

PROF ALISHA SAHOO

RANDOMIZED ALGORITHMS

Randomized Algorithms use random numbers during execution.

Output or execution time depends on random choices.

TYPES OF RANDOMIZED ALGORITHMS

1. Las Vegas Algorithm

Always produces correct answer.

Running time varies.

Example:

Randomized Quick Sort

2. Monte Carlo Algorithm

Execution time fixed.

May produce incorrect answer occasionally.

Example:

Probabilistic Testing

ADVANTAGES OF RANDOMIZED ALGORITHMS

1. Faster execution.
2. Simpler design.
3. Useful for large-scale problems.

DISADVANTAGES

1. Non-deterministic behavior.
2. Results may vary.

COMPLEXITY CLASSES SUMMARY

Class	Meaning
P	Solvable in polynomial time
NP	Verifiable in polynomial time
NP-Complete	Hardest problems in NP
NP-Hard	At least as hard as NP-Complete

REAL-LIFE APPLICATIONS

NP-Complete Problems

- Network Design

PROF ALISHA SAHOO

- Timetable Scheduling
- Resource Allocation

NP-Hard Problems

- Airline Scheduling
- Vehicle Routing
- Supply Chain Optimization

Approximation Algorithms

- Google Maps Route Planning
- Logistics Management
- Cloud Computing

ADVANTAGES OF COMPLEXITY THEORY

1. Helps classify problems.
2. Guides algorithm design.
3. Identifies computational limits.
4. Supports optimization techniques.

LIMITATIONS

1. Some problems remain unsolved.
2. Exact solutions may be impossible.
3. High computational requirements.

PROF ALISHA SAHOO