DESIGN AND ANALYSIS OF ALGORITMS

M.Sc. Computer Science

First Year, Semester-II, Paper-II

Lesson Writers

Dr. Neelima Guntupalli
Assistant Professor
Department of CS&E
Acharya Nagarjuna University

Dr. U. Surya Kameswari
Assistant Professor
Department of CS&E
Acharya Nagarjuna University

Dr. Vasantha Rudramalla
Faculty, Department of CS&E
Acharya Nagarjuna University

Mrs. Appikatla Pushpa Latha
Faculty, Deponent of CS&E
Acharya Nagarjuna University

Editor

Dr. Kampa Lavanya
Assistant Professor
Department of CS&E
Acharya Nagarjuna University

Academic Advisor

Dr. Kampa Lavanya
Assistant Professor
Department of CS&E
Acharya Nagarjuna University

DIRECTOR, I/c.
Prof. V. Venkateswarlu
M.A., M.P.S., M.S.W., M.Phil., Ph.D.
CENTRE FOR DISTANCE EDUCATION
ACHARYA NAGARJUNA UNIVERSITY
NAGARJUNA NAGAR 522 510
Ph: 0863-2346222, 2346208
0863- 2346259 (Study Material)
Website www.anucde.info
E-mail: anucdedirector@gmail.com

M.Sc., (Computer Science) : DESIGN AND ANALYSIS OF ALGORITMS

First Edition : 2025

No. of Copies :

Published by:

Prof. V. VENKATESWARLU
Director, I/c
Centre for Distance Education,
Acharya Nagarjuna University

Printed at:

FOREWORD

Since its establishment in 1976, Acharya Nagarjuna University has been forging ahead in the path of progress and dynamism, offering a variety of courses and research contributions. I am extremely happy that by gaining 'A+' grade from the NAAC in the year 2024, Acharya Nagarjuna University is offering educational opportunities at the UG, PG levels apart from research degrees to students from over 221 affiliated colleges spread over the two districts of Guntur and Prakasam.

The University has also started the Centre for Distance Education in 2003-04 with the aim of taking higher education to the door step of all the sectors of the society. The centre will be a great help to those who cannot join in colleges, those who cannot afford the exorbitant fees as regular students, and even to housewives desirous of pursuing higher studies. Acharya Nagarjuna University has started offering B.Sc., B.A., B.B.A., and B.Com courses at the Degree level and M.A., M.Com., M.Sc., M.B.A., and L.L.M., courses at the PG level from the academic year 2003-2004 onwards.

To facilitate easier understanding by students studying through the distance mode, these self-instruction materials have been prepared by eminent and experienced teachers. The lessons have been drafted with great care and expertise in the stipulated time by these teachers. Constructive ideas and scholarly suggestions are welcome from students and teachers involved respectively. Such ideas will be incorporated for the greater efficacy of this distance mode of education. For clarification of doubts and feedback, weekly classes and contact classes will be arranged at the UG and PG levels respectively.

It is my aim that students getting higher education through the Centre for Distance Education should improve their qualification, have better employment opportunities and in turn be part of country's progress. It is my fond desire that in the years to come, the Centre for Distance Education will go from strength to strength in the form of new courses and by catering to larger number of people. My congratulations to all the Directors, Academic Coordinators, Editors and Lesson-writers of the Centre who have helped in these endeavors.

Prof. K. Gangadhara Rao
M.Tech., Ph.D.,
Vice-Chancellor I/c
Acharya Nagarjuna University.

202CP24: Design & Analysis of Algorithms

Syllabus

UNIT I

Introduction to Computer Algorithms- Algorithm Specification, Performance Analysis, Randomized algorithms

Elementary Data Structures- Stacks and Queues, Tree, Dictionaries, Priority queues, Sets and Disjoint Set Union, graphs
Learning Outcomes

UNIT - II

Divide - And - Conquer - General Method, Binary Search, Maximum and Minimum, Merge Sort, Quick Sort, Selection, Strassen's Matrix Multiplication, Convex Hull.

UNIT-III

The Greedy Method - Knapsack Problem, Tree vertex splitting, Job sequencing, with deadlines, Minimum-cost spanning trees, Optimal storage on tapes, Optimal merge pattern, Single source shortest paths.

UNIT-IV

Dynamic Programming - General method, Multistage graph, All pairs shortest path, Single-source
shortest path, Optimal Binary search trees, String Editing, 0/l Knapsack, Reliability design, The traveling salesman problem, Flow shop scheduling.
Basic Traversal and Search Techniques - Basic traversal & search techniques - Techniques for
binary trees, techniques for graphs, connected components & spanning trees, Bi-connected components & DFS.

UNIT-V

Backtracking - Back tracking - The General Method, The 8-Queens problem, Sum of subsets, Graph coloring, Hamiltonian cycle, Knapsack problem.
Branch and Bound - The method, 0/l Knapsack problem, Traveling salesperson, Efficiency considerations.

Prescribed Book

L Ellis Horwitz, Sartaj Sahani , 'Fundamentals of Computer Algorithms', Universities press, The following topics in the prescribed book Topics 1,2,3,4,5,6,7,8.

Reference Books

l. Bases S, & Gelder A.V - computer Algorithms, Addision Wesley(200)
2. Cormen TH et al - Introduction to Algorithms, pHI(200l)
3. Brassard & Bralley - Fundamentals of Algorithms, pHI(2001)

202CP24

M.Sc., (Computer Science)
MODEL QUESTION PAPER
MCS205 – DESIGN AND ANALYSIS OF ALGORITHMS

Time: 3 Hours **Max Marks:** 70

Answer ONE Question from each Unit **5 × 14 = 70 Marks**

UNIT – I

1. a) Define an algorithm. Explain the various criteria for analyzing its performance. (7M)
b) Explain the working of randomized algorithms with an example. (7M)

OR

2. a) Explain the implementation of stacks and queues using arrays. (7M)
b) Describe the operations on disjoint sets with suitable examples. (7M)

UNIT – II

3. a) Explain the divide and conquer method and write its control abstraction. (7M)
b) Solve a convex hull problem using the divide and conquer approach. Write its algorithm and analyze its time complexity. (7M)

OR

4. a) Write the quick sort algorithm and explain its partitioning process. Analyze its best and worst-case complexity. (7M)
b) Explain Strassen's matrix multiplication method with example matrices and show how it reduces computational steps. (7M)

UNIT – III

5. a) Explain the greedy method for solving the knapsack problem. (7M)
b) Construct an optimal binary search tree for the identifiers (do, if, while) given p(1:3) = (3,3,1) and q(0:3) = (2,3,1,1). (7M)

OR

6. a) Discuss the job sequencing with deadlines problem using the greedy approach. (7M)
b) Write an algorithm for single-source shortest path using Dijkstra's method. Illustrate with an example graph. (7M)

UNIT – IV

7. a) Explain the dynamic programming solution for the 0/1 Knapsack problem. Derive its recurrence relation. (7M)
b) Construct the optimal binary search tree (OBST) using given probabilities and compute its expected cost. (7M)

OR

8. a) Describe the dynamic programming solution for the Traveling Salesman Problem (TSP). (7M)
b) Explain BFS and DFS traversal techniques for graphs with suitable diagrams. (7M)

UNIT – V

9. a) Explain the branch and bound method for solving combinatorial problems. (7M)
b) Write the FIFO branch and bound algorithm and explain its working with an example. (7M)

OR

10. a) Solve the 8-Queens problem using backtracking. (7M)
b) Explain the differences between LC search, FIFO branch and bound, and bounding techniques. (7M).

CONTENTS

LESSON - 1

INTRODUCTION TO ALGORITHMS

OBJECTIVES

The objectives of this lesson are to:

- Understand what an algorithm is and why it is essential to computer science.
- Learn how algorithms are specified using pseudocode and recursive techniques.
- Evaluate the performance of algorithms based on time and space complexity.
- Grasp the concept and significance of asymptotic notation.
- Explore the nature and usefulness of randomized algorithms.

STRUCTURE

1.1 INTRODUCTION

An algorithm is a step-by-step procedure or a finite sequence of instructions designed to perform a specific task or solve a particular problem. It acts as a blueprint or a plan that tells a computer exactly what steps to follow to reach the desired output from a given input. Algorithms are fundamental to computer science and are used in all kinds of software applications.

Key Characteristics of an Algorithm:

To be considered a valid algorithm, a set of instructions must meet the following criteria:

1. **Finiteness**:
   The algorithm must always end after a finite number of steps. It should not go into an infinite loop.
2. **Definiteness**:
   Each step of the algorithm must be clear and unambiguous. There should be no confusion about what is to be done.
3. **Input**:
   An algorithm has zero or more inputs. These are the values or data that are given to the algorithm before it begins.
4. **Output**:
   The algorithm must produce at least one output, which is the result of processing the input.
5. **Effectiveness**:
   All operations in the algorithm must be simple enough to be performed exactly and in a reasonable amount of time.

Why Are Algorithms Important?

Algorithms are the foundation of computer programming. Every program you use—from a calculator app to a search engine—relies on algorithms to function. They allow computers to:

- Solve problems efficiently
- Perform tasks like sorting data, searching for information, or making decisions
- Automate repetitive and complex operations

Efficient algorithms can save time and computational resources, which is especially important when working with large data sets or real-time applications.

Simple Example of an Algorithm:

Let's take a real-world example—making a cup of tea:

1. Boil water
2. Place a tea bag in a cup
3. Pour the hot water into the cup
4. Let it steep for a few minutes
5. Remove the tea bag
6. Add sugar or milk if desired
7. Serve the tea

This is a human-level algorithm. When these steps are followed correctly, the result is a cup of tea.

Example in Computing: Find the Maximum Number in a List

Let's say we have a list of numbers and want to find the largest one:

1. Start with the first number and assume it is the largest.
2. Compare this number with the next number in the list.
3. If the next number is larger, make it the new largest number.
4. Repeat step 2 until all numbers are checked.
5. The final value is the largest number in the list.

This is a clear, finite sequence of steps that a computer can follow.

Types of Algorithms:

There are various types of algorithms based on their purpose and approach:

- **Sorting Algorithms** (e.g., Bubble Sort, Merge Sort)
- **Searching Algorithms** (e.g., Binary Search)
- **Graph Algorithms** (e.g., Dijkstra's Algorithm)
- **Recursive Algorithms** (solve problems by calling themselves)
- **Randomized Algorithms** (use randomness to make decisions)

1.2 ALGORITHM SPECIFICATION

Algorithm specification refers to the process of describing an algorithm in a clear, precise, and structured manner. It is the stage where the logic of the algorithm is formally written down so that it can be correctly implemented in a programming language. This process ensures that both humans and machines can understand what the algorithm is designed to do, how it will do it, and what result it is expected to produce.

Purpose of Algorithm Specification

The main goal of algorithm specification is to:

- Define the problem clearly
- Describe the step-by-step solution to the problem
- Ensure accuracy and completeness before implementation
- Create a blueprint for coding

It serves as a bridge between understanding a problem and actually programming its solution. Without proper specification, developers might misinterpret the logic or produce inefficient or incorrect implementations.

Components of Algorithm Specification

A complete algorithm specification generally includes the following:

1. **Input**:
The data that the algorithm receives before execution begins. The specification must define:
   - What kind of input is required
   - The format or data type (e.g., integer, list, string)
   - Constraints on the input (e.g., non-negative numbers)
2. **Output**:
The expected result after the algorithm processes the input. The specification should state:
   - What form the output will take
   - How the output relates to the input
   - Conditions that determine correctness
3. **Logic or Procedure**:
This is the core of the specification where the step-by-step process is described. The logic includes:

   - Sequence of operations
   - Decisions (e.g., if-else conditions)
   - Loops and repetitions
   - Any special techniques used (like recursion)

Ways to Specify an Algorithm

There are different methods for specifying an algorithm, depending on the purpose and the audience:

1. **Natural Language (Plain English)**:
Easy to write and understand, but can be ambiguous or incomplete. Mostly used in initial planning or communication between non-programmers.
2. **Pseudocode**:
A structured way of writing the algorithm using a mix of natural language and programming-like statements. It removes ambiguity and helps in easily converting the algorithm into code.
3. **Flowcharts**:
Diagrams that use symbols to represent steps and flow of logic. Helpful for visual learners and when explaining to a team.
4. **Mathematical Notation**:
Used in more formal or academic settings, especially for theoretical algorithms. It defines the algorithm using functions, sets, and mathematical expressions.

Example of Algorithm Specification

Problem: Find the sum of two numbers.

- **Input**: Two integers, a and b
- **Output**: An integer representing the sum of a and b
- **Procedure**:
  1. Read the value of a
  2. Read the value of b
  3. Add a and b
  4. Return the result

This specification clearly outlines what the algorithm is supposed to do and how it does it.
Importance of Specification

- Helps in avoiding errors during implementation.
- Makes collaboration easier, especially in large teams.
- Assists in debugging and optimization, as developers can refer back to the original logic.
- Ensures the solution is complete and covers all edge cases.

Algorithm specification is a vital step in software development and problem-solving. It transforms abstract ideas into concrete steps that can be executed by a machine. A well-specified algorithm not only ensures correct implementation but also improves communication, debugging, and future maintenance of the code.

1.2.1 Pseudocode Conventions

Pseudocode is a method of describing the steps of an algorithm in a way that is easy for humans to read and understand. It is written in a combination of plain English and programming-like language. Unlike actual code, pseudocode does not follow the strict syntax rules of any particular programming language. Instead, it focuses on representing the logic and flow of the algorithm in a clear and simple manner.

Pseudocode serves as a bridge between the problem statement and the actual implementation of the algorithm in a programming language. It is especially helpful during the design phase of programming because it allows developers and learners to concentrate on the algorithm's logic without worrying about programming errors or language-specific syntax.

Purpose of Using Pseudocode

1. **To plan before coding**:
Writing pseudocode helps in organizing thoughts and breaking down the problem logically.
2. **To make communication easier**:
Since pseudocode uses plain language, it is easily understandable by non-programmers, team members, and stakeholders.
3. **To reduce errors**:
Planning logic using pseudocode helps in reducing logical and structural errors during actual coding.

Key Features of Pseudocode

- It is not executed on a computer.
- There are no strict syntax rules.
- It uses simple keywords to represent control structures like conditions and loops.
- Indentation is used to show hierarchy and nested structures.
- Comments may be added using plain language to explain what a step does.

Common Pseudocode Conventions
While pseudocode is informal, using consistent and standard conventions makes it easier to understand and implement. Here are some commonly followed conventions:

1. Control Structures:

- **IF – ELSE – ENDIF**

Used for decision making.

```
IF condition THEN
   Perform some action
ELSE
   Perform another action
ENDIF
```

- **WHILE – ENDWHILE**

Used for loops that repeat while a condition is true.

```
WHILE condition DO
   Repeat this action
ENDWHILE
```

- **FOR – TO – ENDFOR**

Used for loops with a fixed number of iterations.

```
FOR i = 1 TO 10 DO
   Print i
ENDFOR
```

- **REPEAT – UNTIL**

A loop that runs at least once and stops when a condition is true.

```
REPEAT
   Read user input
UNTIL input is valid
```

2. Input and Output:

- **READ** or **INPUT** – to take input from the user.

```
READ number
```

- **PRINT** or **OUTPUT** – to display output.

```
PRINT "The sum is ", sum
```

3. Assignment Statement:

- The ← symbol or = is used to assign values to variables.

```
total ← a + b
```

Example of Pseudocode: Add Two Numbers

```
BEGIN
  READ number1
  READ number2
  sum ← number1 + number2
  PRINT "The sum is ", sum
END
```

This pseudocode takes two numbers as input, adds them, and prints the result. There is no mention of a specific programming language, but the logic is clear and can be implemented easily in any language like Python, Java, or C++.

Indentation in Pseudocode

Indentation is used to show which instructions belong together, especially inside loops or conditional blocks. It improves readability and shows the structure of the algorithm clearly.

Example:

```
IF score >= 50 THEN
   PRINT "Pass"
ELSE
   PRINT "Fail"
ENDIF
```

Without indentation, this would be harder to read and understand.

Benefits of Using Pseudocode

- Easier to understand and modify during the planning stage
- Saves time and effort before coding
- Helps in debugging logical flow without getting distracted by syntax
- Ideal for documentation and sharing algorithms with others

Pseudocode is a valuable tool for designing algorithms. By following simple and consistent conventions, it provides a way to describe logic clearly and understandably, without the complexity of real programming language syntax. It prepares the path for smooth and error-free implementation in code.

1.2.2 Recursive Algorithms

A recursive algorithm is an approach to solving a problem by having the algorithm call itself on smaller instances of the same problem. This technique is particularly useful for problems that have a natural repetitive structure, meaning they can be broken down into similar sub-problems.

Recursion is both a powerful and elegant way of thinking about problems. It allows a complex task to be solved by reducing it into simpler versions of itself until the problem becomes so simple that it can be solved directly. This direct solution is known as the **base case**.

How Recursive Algorithms Work

A recursive algorithm typically consists of two parts:

1. **Base Case**:

This is the condition under which the recursion stops. It prevents the function from calling itself indefinitely and provides a simple, directly solvable instance of the problem.

2. **Recursive Case**:

This is the part of the algorithm where the problem is divided into smaller versions of itself, and the algorithm is called again on these sub-problems.

General Structure of a Recursive Algorithm

```
function recursiveAlgorithm(input):
   if base_case_condition:
      return simple_solution
   else:
      return some_operation(recursiveAlgorithm(smaller_input))
```

Example: Factorial of a Number

The factorial of a non-negative integer n, denoted as n!, is the product of all positive integers less than or equal to n.

Mathematically:

- factorial(0) = 1 → (Base case)
- factorial(n) = n × factorial(n−1) → (Recursive case)

Recursive Algorithm in Pseudocode:

```
function factorial(n):
   if n == 0:
      return 1
   else:
      return n * factorial(n - 1)
```

Let's break down how factorial(4) would work:

- factorial(4) = 4 * factorial(3)
- factorial(3) = 3 * factorial(2)
- factorial(2) = 2 * factorial(1)
- factorial(1) = 1 * factorial(0)
- factorial(0) = 1 (Base case)

Now the results are returned back up the chain:

- factorial(1) = 1 * 1 = 1
- factorial(2) = 2 * 1 = 2
- factorial(3) = 3 * 2 = 6
- factorial(4) = 4 * 6 = 24

When to Use Recursive Algorithms

Recursive algorithms are ideal when:

- The problem can be broken down into smaller similar problems
- The problem naturally fits a divide-and-conquer strategy
- A recursive definition already exists (like factorial, Fibonacci, or tree traversal)

Other Examples of Recursive Algorithms

1. **Fibonacci Series**:
fibonacci(n) = fibonacci(n - 1) + fibonacci(n - 2)
with base cases:
fibonacci(0) = 0
fibonacci(1) = 1

2. **Tower of Hanoi**:
A mathematical puzzle where recursion simplifies the process of moving disks between rods.

3. **Binary Search** (Recursive version):
Useful for searching a sorted array by dividing it into halves.

4. **Tree Traversals**:
Recursive functions are commonly used to traverse binary trees (e.g., in-order, pre-order, post-order).

Advantages of Recursive Algorithms
- **Simplicity and readability**: Recursive solutions can be cleaner and easier to understand than their iterative counterparts.
- **Direct translation of mathematical definitions**: Many algorithms like factorial and Fibonacci follow naturally recursive definitions.

Disadvantages of Recursive Algorithms
- **Performance**: Recursion can be less efficient due to the overhead of multiple function calls.
- **Memory usage**: Each recursive call adds a new frame to the call stack, which can lead to **stack overflow** if the recursion is too deep.
- **May need optimization**: Techniques like tail recursion or memorization (storing previously computed results) can improve performance.

Recursive algorithms provide a powerful tool for solving problems by reducing them to smaller, manageable sub-problems. While they can be elegant and straightforward, they also require careful handling to avoid excessive memory use and ensure proper termination. Understanding the base case and the recursive case is crucial for writing and analyzing recursive solutions effectively.

1.3 PERFORMANCE ANALYSIS

Performance analysis is about checking how good an algorithm is. We analyze how much time and memory an algorithm needs to run, especially as the input size grows. It helps compare different algorithms and choose the best one for a particular situation.

1.3.1 Space Complexity

Space complexity is a measure of the amount of memory or space an algorithm requires to complete its execution, as a function of the size of its input. It includes both the temporary space used during computation and the space needed for the input and output.

Understanding space complexity is important when dealing with memory-limited environments such as embedded systems or mobile devices. It helps in determining whether an algorithm can be efficiently run on a particular system and how scalable it is for large inputs.

What is Included in Space Complexity?
The total space used by an algorithm typically includes:

1. **Input space**:
The memory required to store the input data. Even though the input is provided to the algorithm, it is counted in the space complexity.

2. **Auxiliary space (or extra space)**:
Additional memory used during computation, such as:
- Temporary variables
- Data structures like arrays, lists, stacks, queues
- Function call stacks (especially important in recursive algorithms)

3. **Output space**:
If the algorithm stores the result instead of printing it directly, this space is also counted.

Notation Used
Just like time complexity, space complexity is also expressed using asymptotic notation (Big O), which describes how memory usage grows with input size n.

Examples:

- **O(1)**: Constant space – the memory usage doesn't increase with input size.
- **O(n)**: Linear space – memory usage grows linearly with input size.
- **O(n²)**: Quadratic space – used when storing two-dimensional data like matrices.

Examples

Example 1: Constant Space Algorithm (O(1))
```
def add(a, b):
   result = a + b
 return result
```
This function only uses a few variables (a, b, result) and doesn't depend on the size of the input. Hence, its space complexity is **O(1)**.

Example 2: Linear Space Algorithm (O(n))
```
def copy_list(original):
   new_list = []
   for item in original:
new_list. Append(item)
   return new_list
```

Here, the algorithm creates a new list of the same size as the input. If the input list has n elements, the memory used for new_list also grows with n. Therefore, the space complexity is **O(n)**.

Example 3: Recursive Algorithm with Stack Usage
```
def factorial(n):
   if n == 0:
      return 1
   else:
      return n * factorial(n - 1)
```

In this recursive function, each call to factorial() adds a new frame to the call stack. For input n, the function will call itself n times before reaching the base case. So, the space complexity due to the call stack is O(n), even though the function doesn't create additional variables.

Why Space Complexity Matters

- **Efficiency**: Efficient use of memory leads to better performance, especially on systems with limited RAM.
- **Scalability**: Algorithms with high space complexity might not work with large input sizes.
- **Trade-offs**: Sometimes there's a trade-off between time and space. An algorithm can be made faster by using more memory, or vice versa.

Example: In searching, Hash Tables use more memory (O(n)) but allow for very fast lookups (O (1) time). On the other hand, Binary Search Trees use less memory but take longer to search (O (log n) time).The following table outlines the relationship between common space complexities and their corresponding examples:

Table 1.1 Common Types of Space Complexities

Complexity	Description	Example
O(1)	Constant space	Swapping two numbers
O(log n)	Logarithmic space	Divide-and-conquer recursion with no storage
O(n)	Linear space	Copying a list
$O(n^2)$	Quadratic space	2D array operations
O(n!)	Factorial space	Generating all permutations

Space complexity is a crucial concept in algorithm design and analysis. It tells how efficiently an algorithm uses memory, which is important for handling large inputs and ensuring that the program runs without crashing due to memory overflow. By understanding space complexity, developers can write better, more optimized, and scalable code.

1.3.2 Time Complexity

Time complexity refers to the amount of time an algorithm takes to run, based on the size of the input. It is used to evaluate the efficiency of an algorithm by showing how its running time increases as the input size (n) increases.

Rather than measuring time in seconds or milliseconds (which can vary depending on the computer or programming language), time complexity looks at how many basic operations an algorithm performs relative to the input size.

Why Time Complexity is Important

- It helps predict how the algorithm will behave with large inputs.
- It allows comparing different algorithms based on their efficiency.
- It helps in choosing the best algorithm for a particular problem and resource availability.

Understanding with an Example

Suppose there is a list of numbers, and the goal is to find a specific number in the list.

```
def search(list, target):
    for item in list:
        if item == target:
            return True
    return False
```

This function checks each item one by one.
If the list has n elements, in the worst case (if the item is not found or is at the end), it will check all n items.

So the time complexity is O(n) — this is called linear time complexity.

Types of Time Complexities

Time complexity is often expressed using Big O notation, which describes the upper bound of an algorithm's running time — the worst-case scenario.
Here are the most common types:

Table 1.2Types of Time Complexities

Time Complexity	Name	Description
O(1)	Constant time	Execution time does not depend on input size. Example: accessing an array element by index.
O(log n)	Logarithmic time	Time increases slowly even with large inputs. Example: binary search.
O(n)	Linear time	Time increases directly with input size. Example: simple search in a list.
O(n log n)	Linear arithmetic time	More than linear but better than quadratic. Example: efficient sorting algorithms like Merge Sort.
$O(n^2)$	Quadratic time	Time increases with the square of the input size. Example: bubble sort or nested loops.
$O(2^n)$	Exponential time	Time doubles with every additional input. Example: solving the traveling salesman problem by brute force.
O(n!)	Factorial time	Extremely slow for large n. Example: generating all permutations of n items.

1.3.3 Asymptotic Notation

Asymptotic notations are mathematical tools to describe the time complexity of an algorithm without considering machine-specific details. They help in understanding the growth rate of an algorithm as the input size approaches infinity.

Types of Asymptotic Notations:

1. **Big O Notation (O):** Describes the worst-case time complexity.
2. **Omega Notation (Ω):** Describes the best-case time complexity.
3. **Theta Notation (Θ):** Describes the average-case or tight bound complexity.

1. Big O Notation – O(f(n))
Let T(n) be the running time of an algorithm. If there exist constants $c > 0$ and $n_0 \geq 0$ such that:
$T(n) \leq c \cdot f(n)$ for all $n \geq n_0$
Then, $T(n) = O(f(n))$
Example:
If $T(n) = 4n + 3$, then $T(n) = O(n)$

2. Omega Notation – Ω(f(n))
If there exist constants $c > 0$ and $n_0 \geq 0$ such that:
$T(n) \geq c \cdot f(n)$ for all $n \geq n_0$
Then, $T(n) = \Omega(f(n))$

Example:
If the best case of a search occurs at the first index, the time complexity is $\Omega(1)$

3. Theta Notation – Θ(f(n))
If there exist constants $c_1 > 0$, $c_2 > 0$, and $n_0 \geq 0$ such that:
$c_1 \cdot f(n) \leq T(n) \leq c_2 \cdot f(n)$ for all $n \geq n_0$
Then, $T(n) = \Theta(f(n))$

Example:
If an algorithm always runs for $5n + 2$ steps, then $T(n) = \Theta(n)$
Theorems on Asymptotic Notation
Theorem 1 – Transitivity of Big O:

If $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then $f(n) = O(h(n))$
Theorem 2 – Sum Rule:

If $T_1(n) = O(f(n))$ and $T_2(n) = O(g(n))$, then:
$T_1(n) + T_2(n) = O(\max(f(n), g(n)))$
Theorem 3 – Product Rule:

If $T_1(n) = O(f(n))$ and $T_2(n) = O(g(n))$, then:
$T_1(n) \cdot T_2(n) = O(f(n) \cdot g(n))$
Time complexity can be summarized as

- Time complexity measures how the running time of an algorithm changes with input size.
- Asymptotic notations (O, Ω, Θ) simplify the analysis by ignoring machine-specific constants.
- Big O provides an upper bound, Omega provides a lower bound, and Theta provides an exact bound.
- These concepts are essential for analyzing and comparing algorithm performance.

1.3.4 Practical Complexities

Practical complexities go beyond theory and consider how algorithms perform in real-world scenarios. Some algorithms may have good theoretical time but are slow in practice due to constant factors or memory usage. Others with slightly worse time complexity might be faster in real usage. Practical performance is tested by running algorithms on actual data.

1.3.5 Performance Measurement

Performance measurement involves testing an algorithm by running it and measuring how much time and memory it uses. This is done using timers or profiling tools. It helps to see how an algorithm behaves with different input sizes and conditions. This is important for optimizing applications and choosing the right algorithms.

1.4 RANDOMIZED ALGORITHMS

Randomized algorithms use random numbers to make decisions during execution. This means they may give different outputs or run in different ways for the same input. These algorithms are often simpler or faster and are used in areas like computer graphics, cryptography, and optimization.

1.4.1 Basics of Probability Theory

Probability theory is the study of how likely events are to happen. In the context of algorithms, it helps calculate the chance of certain outcomes, especially when randomness is involved. Key concepts include events, probabilities, expected value, and independence. These are used to analyze the behavior and efficiency of randomized algorithms.

1.4.2 Randomized Algorithms: An Informal Description

A randomized algorithm introduces randomness into its logic. For example, instead of checking every element, it might pick some elements randomly to save time. This can make algorithms faster or simpler, though they might not always give the same answer. However, if designed well, the chance of error can be very low.

1.4.3 Identifying the Repeated Element

This is a common problem where you are given a list of elements and need to find which one appears more than once. A randomized algorithm might pick random elements and compare them, rather than scanning the whole list. While not always 100% accurate in one run, running it, multiple times increases the chances of getting the correct result.

1.4.4 Primality Testing

Primality testing checks whether a number is a prime. Randomized algorithms like the Miller-Rabin test can do this very quickly using probability. These algorithms don't always give a definite answer, but the chance of being wrong can be made extremely small, making them useful in cryptography.

1.4.5 Advantages and Disadvantages

Advantages:

- Often simpler and faster than deterministic algorithms
- Useful when exact solutions are hard or time-consuming
- Work well in practice, especially for large inputs

Disadvantages:

- May give different results in different runs
- Not always guaranteed to give the correct answer
- Harder to analyze and test for correctness

1.5 SUMMARY

An algorithm is a clear, step-by-step procedure to solve a problem or perform a task. It must be finite, unambiguous, and produce an output. Algorithms can be represented in various forms such as natural language, pseudocode, flowcharts, or mathematical notation. Pseudocode helps describe logic simply and clearly, without worrying about syntax. Recursive algorithms break problems into smaller sub-problems and are powerful but need a well-defined base case.Performance analysis evaluates algorithms using space and time complexity. Space complexity measures memory usage; time complexity estimates how execution time scales with input size. Asymptotic notations like Big O, Omega, and Theta are used for such evaluations.Randomized algorithms use randomness to solve problems, offering simplicity and speed but may produce varying results. Probability theory helps analyze their effectiveness and reliability.By mastering these concepts, learners can design, specify, and analyze algorithms to build efficient and reliable software systems.

1.6 KEY TERMS

Algorithm, Pseudocode, Recursion, Time Complexity, Space Complexity, Asymptotic Notation

1.7 REVIEW QUESTIONS

1. What is an algorithm? List and explain its key characteristics.
2. Describe different ways to specify an algorithm.
3. What is pseudocode? Mention its advantages and basic conventions.
4. Explain how recursive algorithms work using the factorial example.

5. Define space complexity. Provide one example each of O(1) and O(n) space complexity.
6. What is time complexity? Compare O(1), O(n), and O(n^2) complexities.
7. Describe the role of asymptotic notations and explain Big O, Omega, and Theta.
8. What are randomized algorithms? List their advantages and disadvantages.

1.8 Suggestive Readings

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press.
2. Dasgupta, S., Papadimitriou, C. H., & Vazirani, U. (2008). Algorithms. McGraw-Hill.
3. Sedgewick, R., & Wayne, K. (2011). Algorithms (4th ed.). Addison-Wesley.
4. Brassard, G., & Bratley, P. (1996). Fundamentals of Algorithmics. Prentice-Hall.
5. Kleinberg, J., & Tardos, E. (2005). Algorithm Design. Pearson Education.

Dr. Neelima Guntupalli

LESSON- 2

LINEAR DATA STRUCTURES

OBJECTIVES

The objectives of this lesson are to:

- Understand the basic concepts and principles of linear data structures.
- Learn the structure, operations, and applications of stacks.
- Understand the characteristics and types of queues.
- Study the implementation of stacks and queues using arrays and linked lists.
- Explore advanced queue structures such as circular and priority queues.

STRUCTURE

2.1 INTRODUCTION

A stack is a linear data structure that follows the Last-In, First-Out (LIFO) principle. This means that the last element added to the stack is the first one to be removed. You can imagine a stack as a collection of elements arranged vertically, like a stack of plates; when a new plate is added, it goes on top, and when a plate is removed, it's also taken from the top.

Stacks are often represented using an array or linked list. An array-based stack is more straightforward and requires a maximum size, while a linked-list stack allows for dynamic resizing. In stacks, all operations (such as inserting, removing, or accessing elements) occur only at one end, referred to as the **top** of the stack.

Stacks play a critical role in computer science for several reasons:

1. **Function Call Management**: In programming, particularly in recursion, stacks manage function calls. Every time a function is called, its return address, parameters, and local variables are pushed onto the stack. When the function completes, the data is popped off the stack to resume the previous state.
2. **Expression Evaluation**: Stacks are instrumental in evaluating mathematical expressions in postfix or prefix notation, especially when converting from infix to postfix.
3. **Backtracking**: Algorithms like Depth-First Search (DFS) in graphs use stacks to track nodes. Undo mechanisms in software (e.g., the "Undo" button) also rely on stacks to store recent operations, allowing reversal.
4. **Memory Management**: Stacks manage memory for temporary storage, which is automatically cleaned up in LIFO order, making it suitable for managing local variables and function calls.

The fundamental structure and operations of a stack are illustrated below in Figure 2.1.

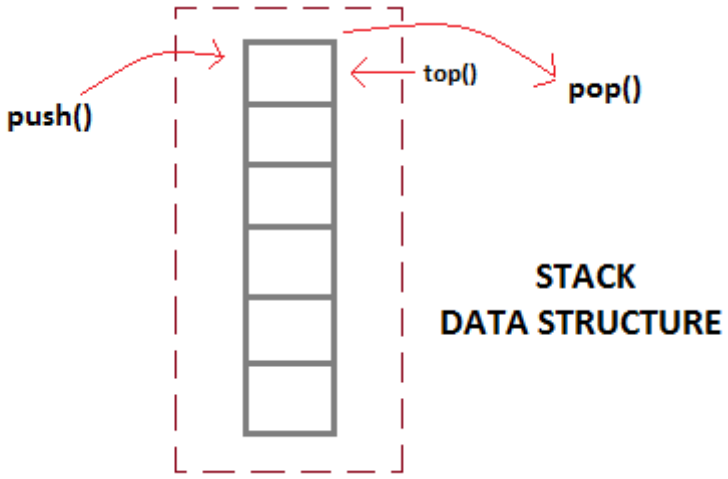


Fig 2.1 Representation of Stack

2.2 FEATURES OF STACK

Stacks are specialized data structures that follow the Last-In, First-Out (LIFO) principle, meaning the last element added is the first to be removed. This structure is commonly used in situations where temporary storage is needed and where order matters, such as in function calls, expression evaluation, and algorithmic backtracking. The main features of the stack are

- A stack is an ordered collection of elements of the same data type, arranged in a specific sequence.
- It follows the Last-In, First-Out (LIFO) or First-In, Last-Out (FILO) principle, meaning the last element added is the first to be removed.
- The Push operation adds new elements to the stack, while the Pop operation removes the top element from the stack.

- The Top is a pointer or variable that references the topmost element in the stack. Both insertion and removal of elements occur only at this end.
- A stack is in an Overflow state when it reaches its maximum capacity (FULL), and in an Underflow state when it has no elements left (EMPTY). Operations on Stacks.

2.3 OPERATIONS ON STACKS

Stacks are widely used in various applications, such as managing function calls, evaluating expressions, and backtracking algorithms. To effectively work with stacks, several basic operations are defined. The primary operations performed on stacks are:

1. **Push**: Adds an element to the top of the stack. If the stack is full, it results in a stack overflow condition.
2. **Pop**: Removes the element from the top of the stack. If the stack is empty, it results in a stack underflow condition.
3. **Top**: Retrieves the element at the top of the stack without removing it. This operation allows inspection of the top value without modifying the stack.

2.4 IMPLEMENTATION OF STACK OPERATIONS

Stack is a data structure which can be represented as an array. An array is meant to store an ordered list of elements. Using an array for representation of stack is the easiest technique to manage the data. Stack can be implemented without memory limit. But when the stacks are implemented using an array, size of the stack will be fixed. Stack can be implemented with the help of an array. In the below figure, the elements of the stack are linearly organised in the stack starting from index of 0 to n-1 or 1 to n. The array subscripts of a stack may be from 0 to n-1 or from 1 to n. A common way to represent a stack using fixed memory allocation is shown in Figure 2.2.

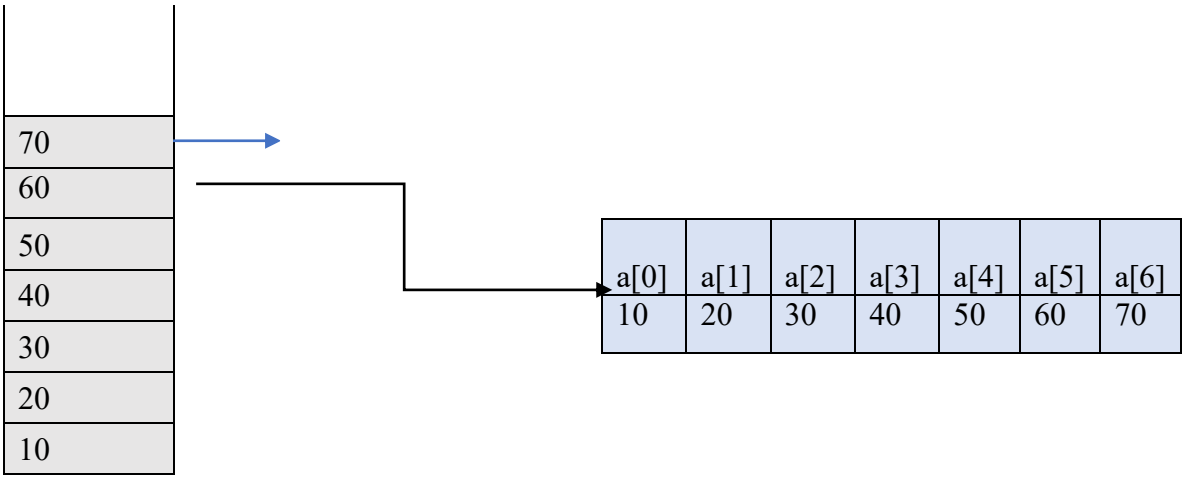


Fig 2.2 Array Representation of Stack

In the above representation, each element of the stack is stored in a node of the linked list, where each node contains two parts: the data (value of the element) and a reference (or pointer) to the next node. The top of the stack corresponds to the head of the linked list. Unlike arrays, the linked list representation of a stack does not have a fixed size, allowing it

to dynamically grow or shrink as elements are added or removed. Stack operations such as push (adding an element) and pop (removing an element) are performed by manipulating the head of the linked list. When a new element is pushed onto the stack, a new node is created, and its next pointer is set to the current top node, with the top pointer updated to the new node. Similarly, when an element is popped, the top pointer is updated to the next node, and the removed node is deallocated. This dynamic approach provides flexibility and efficient memory management while ensuring the stack operates on the Last In, First Out (LIFO) principle. In contrast to the array-based approach, a stack can be implemented using dynamic memory allocation, as shown in Figure 2.3.

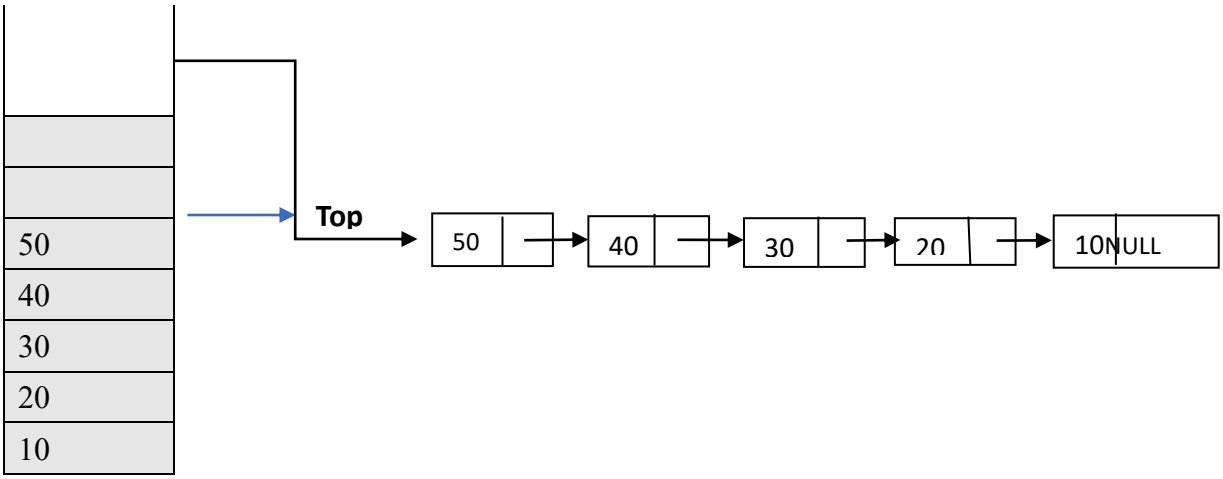


Fig 2.3 Linked List Representation of Stack

2.4.1 Push Operation

The push operation adds an element to the top of the stack. In an array-based stack, we first check if there is space available (to prevent overflow). If space is available, we increment the top pointer and add the new element at this position. If the stack is full, we display a "Stack Overflow" message.

Algorithm for an array-based stack

PUSH (stack, top, item, MAX)

Input:

- stack[] – an array representing the stack
- top – current index of the top element
- item – the element to be pushed
- MAX – maximum size of the stack
- Output:
- Updated stack and top pointer after inserting the item
- Message if the stack is full (Overflow)

1. IF top == MAX – 1
   PRINT "Stack Overflow"
   RETURN
2. top ← top + 1
3. stack[top] ← item

4. PRINT "Item pushed successfully"
5. RETURN

This algorithm checks if the stack is full before inserting the new item. If not full, it increments the top pointer and adds the item to the top of the stack.

The push function adds an element to the stack.

- First, it checks if the stack is full by comparing top with n - 1.
- If the stack is full, it prints "Stack Overflow" to indicate that no more elements can be added.
- If there's space, it increments top to the next position and places the new value at stack[top].

Finally, it prints a message to confirm that the value was added to the stack.

Algorithm for Linked List -based stack

Each node in the stack has two parts:

- data: stores the element
- next: points to the next node

The top pointer always points to the top of the stack (i.e., the most recently added node).

PUSH (Node *top, int item)

Input: item – the element to be pushed

Output: Updated stack with the item added on top

1. Create a new node Node
2. Set Node.data ← item
3. Set Node.next ← top
4. Set top ← Node
5. PRINT "Item pushed successfully"
6. RETURN

The push function adds a new element to the top of a stack implemented using a linked list. It takes a reference to the top pointer (representing the current top of the stack) and the value to be pushed as arguments. A new node is dynamically created using the new keyword, which allocates memory for the node and initializes it with the given value. The next pointer of the new node is set to point to the current top, effectively linking it to the existing stack. Finally, the top pointer is updated to reference the new node, making it the new top of the stack. This ensures that the stack maintains its Last-In, First-Out (LIFO) order, where the most recently added element is always at the top.

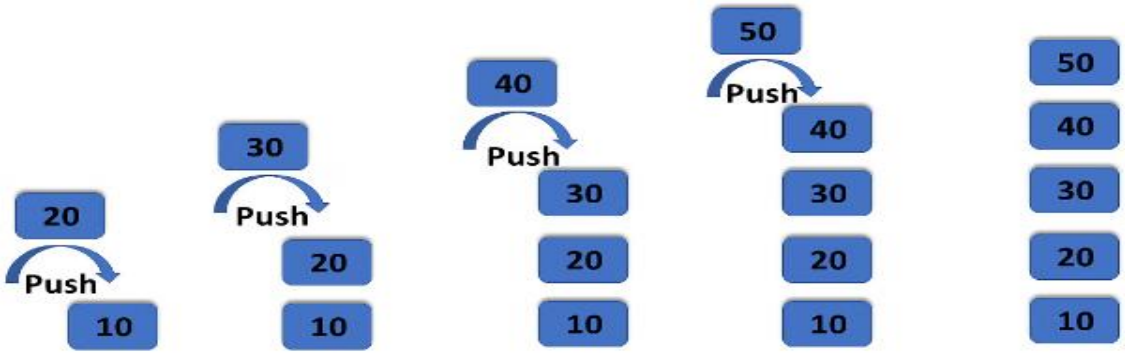


Fig 2.4 Push Operation

Figure 2.4 demonstrates a series of push operations, where elements are added sequentially to the top of the stack. The steps are explained below.

1. **Starting with an Empty Stack**: The stack begins empty, and the first element, 10, is pushed onto it, becoming the bottom element.
2. **Adding 20**: The next element, 20, is pushed onto the stack, sitting on top of 10.
3. **Adding 30**: The element 30 is pushed on top, making it the new top of the stack, with 20 and 10 below it.
4. **Adding 40**: The element 40 is pushed onto the stack, becoming the top element, while 30, 20, and 10 are below it in order.
5. **Adding 50**: Finally, 50 is pushed onto the stack, sitting at the top, with 40, 30, 20, and 10 below it.

Each "push" operation adds a new element to the top of the stack, following the Last-In, First-Out (LIFO) principle, where the most recently added element is always on top.

2.4.2 Pop Operation

The pop operation removes the element at the top of the stack. Before popping, we check if the stack is empty to avoid underflow. If it's not empty, we retrieve the value at top, decrement the top pointer, and return or display the removed element. If the stack is empty, we display a "Stack Underflow" message.

```
Algorithm POP(item)
1. if (*top == -1) then
2.     write("Stack Underflow!");
3.     return false;
4. else
5.     *item := stack[*top];
6.     *top := *top - 1;
7.     return true;
```

The pop function removes the top element from the stack.

- It first checks if the stack is empty by seeing if top is -1.
- If the stack is empty, it prints "Stack Underflow" to indicate there's nothing to remove and returns -1 as an error value.
- If the stack has elements, it retrieves the value at stack[top], decreases top by 1, and returns the removed value.
- It also prints a message confirming which value was removed from the stack.

```
Algorithm Delete(item)Using Linked List
1. if (top == NULL) then
2. write("Stack is empty!");
3. return NULL;
4. else
5. item := top → data;
6. temp := top;
7. top := top → link;
8. delete temp;
9. return top;
```

The pop function removes the top element from a stack implemented using a linked list. It first checks if the stack is empty by verifying if the top pointer is NULL. If the stack is empty, it prints "Stack Underflow" and returns -1 to indicate an error. Otherwise, it temporarily stores the current top node in a pointer temp, retrieves the value from this node, and updates the top pointer to the next node in the stack. The memory allocated for the removed node is then deallocated using free(). Finally, the function prints the value that was popped and returns it. This ensures proper memory management while maintaining the Last-In, First-Out (LIFO) principle.

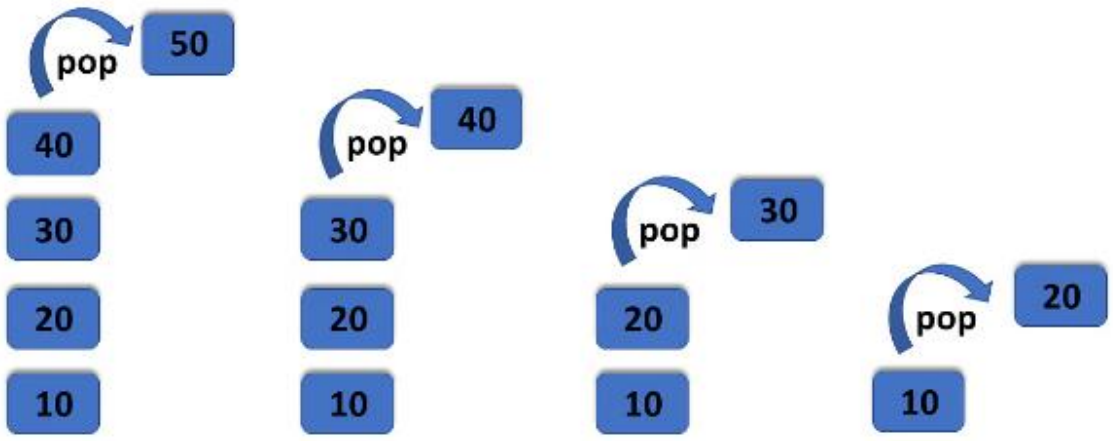


Fig 2.5 Pop Operation

Figure 2.5 illustrates the pop operation, showing how elements are removed one by one from the top, following the LIFO principle.

1. **Removing 50**: The stack starts with 50 at the top. A pop operation removes 50, leaving 40 as the new top element.
2. **Removing 40**: The next pop operation removes 40, making 30 the top element.
3. **Removing 30**: Another pop operation removes 30, leaving 20 at the top of the stack.
4. **Removing 20**: The pop operation removes 20, making 10 the last remaining element.
5. **Removing 10**: Finally, 10 is removed from the stack, leaving it empty.

Each "pop" operation removes the top element, following the Last-In, First-Out (LIFO) principle, where the most recently added element is removed first.

2.5 QUEUES

A queue is a linear data structure that follows the First In, First Out (FIFO) principle, meaning the element inserted first is removed first. It is similar to a real-world queue, such as a line at a ticket counter. Queues are widely used in various applications like task scheduling, managing resources in operating systems, and data buffering in communication systems. Basic operations include enqueue (insertion) and dequeue (removal), ensuring orderly processing of elements. Variants like circular queues, priority queues, and double-ended queues (deque) extend its functionality for specific use cases.

Queues can be implemented using either arrays or linked lists. An array-based queue uses a fixed-size array, while a linked-list queue allows dynamic resizing. In queues, elements are

added at one end (the rear) and removed from the other end (the front), making them suitable for applications where the order of processing is crucial.

2.5.1 Role and Characteristics of Queues

Queues play a critical role in computer science for several reasons:

1. **Task Scheduling:** In operating systems, queues are used to manage tasks, where each task waits in line until its turn. Job scheduling and print spooling are examples where queues are essential.
2. **Data Streaming:** Queues are commonly used to manage streaming data. As data arrives, it's added to the rear, and as it's processed, it's removed from the front.
3. **Breadth-First Search (BFS):** Queues are central to BFS algorithms, where nodes in a graph or tree are processed level by level. Each unvisited node is added to the queue and processed in FIFO order.
4. **Real-Time Data Processing:** In applications like real-time data analytics or messaging systems, queues handle data as it arrives and is processed in order, ensuring that the earliest data is handled first.

Fig 2.6 Representation of Queue

The basic structure and primary operation points (Front and Rear) of a queue are displayed in Figure 2.6.

2.5.2 Features of Queue

Queues are specialized data structures that follow the First-In, First-Out (FIFO) principle. The main features of a queue are:

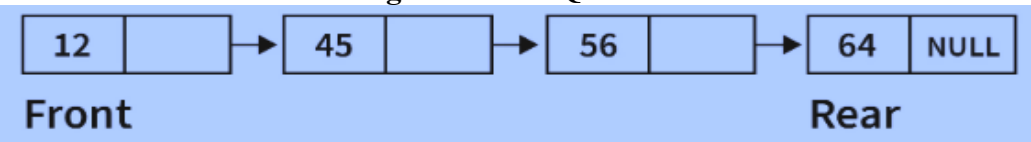
- A queue is an ordered collection of elements of the same data type, arranged in a specific sequence.
- It follows the First-In, First-Out (FIFO) principle, meaning the first element added is the first to be removed.
- The Enqueue operation adds new elements to the queue, while the Dequeue operation removes elements from the front.
- The Front points to the first element in the queue, and the Rear points to the last element. Insertion occurs at the rear, and removal occurs at the front.
- A queue is in an Overflow state when it reaches its maximum capacity (FULL) and in an Underflow state when it has no elements left (EMPTY).

2.6 TYPES OF QUEUES

Queues can be implemented in different ways, each serving various needs:

1. **Linear Queue:** A simple queue with fixed size, where elements are added at the rear and removed from the front. It is typically implemented using arrays. However, once full, it does not reuse empty spaces created by removed elements.
2. An example of a simple linear queue, where elements move in a straight line, is presented in Figure 2.7.

Fig 2.7 Linear Queue



3. **Circular Queue:** A more efficient queue that reuses empty spaces by wrapping around to the beginning of the array, making it suitable for scenarios that require dynamic resizing.

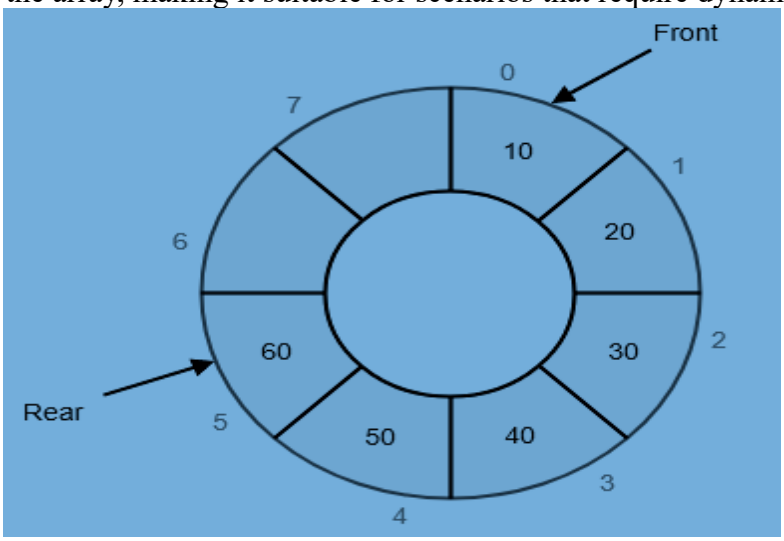


Fig 2.8 Circular Queue

A more efficient array implementation that reuses space by wrapping around the array's ends is visualized as a circular queue in Figure 2.8.

4. **Priority Queue:** Elements are removed based on priority rather than FIFO order.
5. **Deque (Double-Ended Queue):** Allows insertion and deletion from both ends, often implemented with linked lists.

2.7 QUEUE OPERATIONS

Queues are widely used in various applications such as task scheduling, data streaming, and BFS algorithms. The primary operations performed on queues are:

1. **Enqueue:** Adds an element to the rear of the queue. If the queue is full, it results in a queue overflow condition.
2. **Dequeue:** Removes the element from the front of the queue. If the queue is empty, it results in a queue underflow condition.
3. **Front:** Retrieves the element at the front of the queue without removing it. This operation allows inspection of the front value without modifying the queue.
4. **isEmpty:** Checks whether the queue is empty. This is useful for preventing underflow errors before a Dequeue operation.

5. **isFull:** Checks whether the queue is full, mainly in array-based queues, to prevent overflow errors during an Enqueue operation.
6. **Traversal:** Displays all elements in the queue from front to rear without altering the queue. This operation helps in examining the queue's contents.
7. **Search:** Searches for a specific element within the queue and returns its position relative to the front, or an indication if it's not present.

2.8 IMPLEMENTATION OF QUEUE

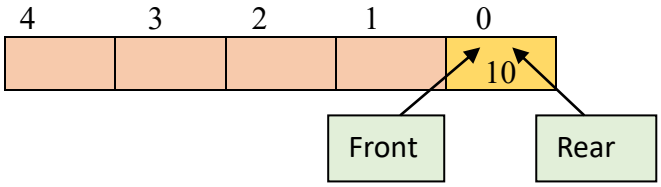
Queues can be implemented using arrays or linked lists. In both cases, the Enqueue operation adds elements at the rear, and the Dequeue operation removes elements from the front.

Array-Based Queue
In an array-based queue, two pointers, front and rear, are used to track the position of the first and last elements, respectively.

- **Array:** A fixed-size array is used to store the elements of the queue. The maximum size of this array is pre-defined.
- **Front Pointer:** This pointer represents the front of the queue, where elements are Dequeued. Initially, **front = -1**.
- **Rear Pointer:** This pointer represents the rear of the queue, where new elements are Enqueued. Initially, **rear = -1**.

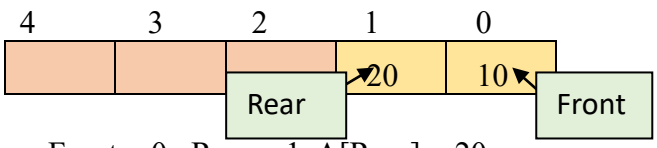
The queue operations modify these pointers to manage elements in the queue. The Enqueue and Dequeue operations increment or decrement these pointers to add or remove elements, respectively.

Example:
Let's say we have an array A of size 5 and we want to insert elements 10, 20, 30, 40, 50 in sequence. For this implementation array of size 5 is taken.
Visual Representation of the above example is

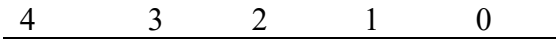


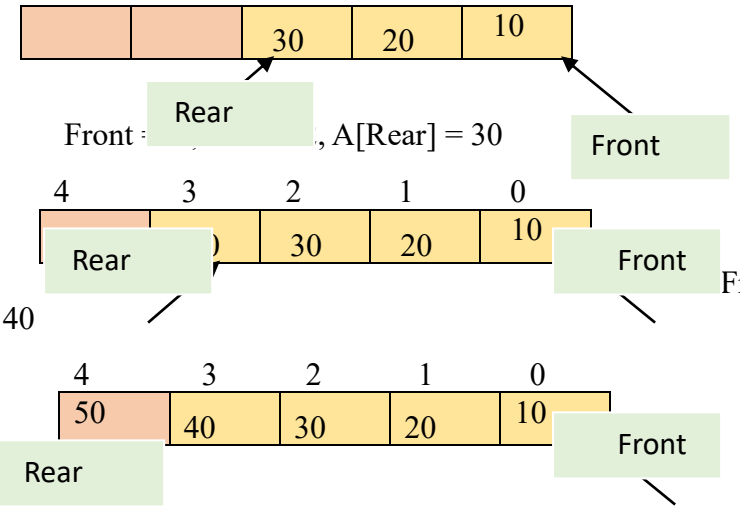
Initially, Front = Rear = -1. Before inserting the first element front and rear values should be initialized to 0.

Front = Rear = 0, A[Rear] = 10



Front = 0 , Rear = 1, A[Rear] = 20





Front = 0 , Rear = 4, A[Rear] = 50

At this point, the queue is full since rear has reached N - 1.

Linked List-Based Queue

In a **linked-list queue**, each element is stored as a node containing the data and a pointer to the next node. The **front** and **rear** pointers keep track of the first and last nodes in the queue, respectively.

- **Node:** Each element in the queue is stored as a node with two parts: the data and a pointer to the next node.
- **Front Pointer:** This pointer points to the first node in the queue.
- **Rear Pointer:** This pointer points to the last node in the queue.

In a linked-list-based queue, the Enqueue operation involves adding a new node at the end (rear) of the queue, and the Dequeue operation involves removing the node at the front. Algorithms for basicqueueoperations.

Algorithm AddQ(item)

```
   // Inserts 'item' into the circular queue q[0...n-1]
   // 'rear' points to the last inserted item, 'front' points one position counterclockwise
rear := (rear + 1) mod n   // Advance rear clockwise
   if front = rear then
write("Queue is full!")
       if front ≠ 0 then
rear := rear - 1
       else
rear := n - 1
       return false
   else
       q[rear] := item        // Insert the new item
       return true
```

Algorithm DeleteQ(item)

```
// Removes and returns the front element from the circular queue q[0...n-1]
   if front = rear then
       write("Queue is empty!")
```

```
        return false
    else
front:= (front + 1) mod n   // Advance front clockwise
item := q[front]                    // Set 'item' to the front of the queue
        return true
```

2.9 PRIORITY QUEUE

A priority queue is a special type of queue in which each element is associated with a priority, and elements are served based on their priority — not just by their order of arrival (as in a regular queue).

- In a max-priority queue, the element with the highest priority is served first.
- In a min-priority queue, the element with the lowest priority is served first.

Key Features:

- Each element has a value and a priority.
- Insertion happens at the end (like a normal queue).
- Deletion (or retrieval) happens from the position of the element with the highest (or lowest) priority, not necessarily the front.

Types of Priority Queues:

1. **Ascending Priority Queue**: Lowest priority element is served first.
2. **Descending Priority Queue**: Highest priority element is served first.

Operations in a Priority Queue

Operation	Description
Insert(item, priority)	Inserts the item with a given priority.
Delete() / Pop()	Removes and returns the item with the highest (or lowest) priority.
Peek() / Front()	Returns the item with the highest (or lowest) priority without removing it.

Implementation Methods:

- **Array or List**
- **Heap (Binary Heap)** – Most efficient (O (log n) for insertion and deletion)
- **Linked List**

Example:
Let's consider a hospital emergency room system, where patients are attended based on the severity of their condition (i.e., priority), not the order in which they arrived.

Patient Name	Condition Severity (Priority)
Alice	2
Bob	5
Charlie	3

Here, higher number = higher priority (e.g., 5 = emergency, 1 = mild)

- When the queue is processed:
  - Bob is attended first (priority 5)
  - Then Charlie (priority 3)
  - Then Alice (priority 2)

This is different from a normal queue where Alice would be served first since she arrived first.

2.10 SUMMARY

This lesson covered two fundamental linear data structures: stacks and queues. A stack operates on the Last-In, First-Out (LIFO) principle and is crucial in recursion, expression evaluation, and memory management. It can be implemented using arrays or linked lists. The key operations are push, pop, and peek (top). Queues follow the First-In, First-Out (FIFO) principle and are widely used in task scheduling, real-time processing, and graph algorithms. Queue variants include circular queues, priority queues, and double-ended queues. Implementation of queues can also be done using arrays or linked lists. Understanding these structures and their efficient implementation forms the basis for solving complex computational problems.

2.11 Key Terms

Stack, Queue, LIFO, FIFO, Push, Pop

2.12 Review Questions

1. Define a stack. What are the primary operations performed on a stack?
2. Explain the push operation with an example using an array.
3. How is a stack implemented using a linked list? Describe with algorithm.
4. Define a queue. What is the difference between a stack and a queue?
5. List and explain the different types of queues.
6. How is a circular queue different from a linear queue?
7. Explain the working of a priority queue with an example.
8. Describe the basic queue operations and their implementation using arrays.

2.13 SUGGESTIVE READINGS

1. Horowitz, E., Sahni, S., & Anderson-Freed, S. (2008). Fundamentals of Data Structures in C. University Press.
2. Lafore, R. (2002). Data Structures and Algorithms in C++. Sams Publishing.
3. Tenenbaum, A. M., Langsam, Y., & Augenstein, M. J. (1990). Data Structures Using C. Prentice-Hall.
4. Weiss, M. A. (2014). Data Structures and Algorithm Analysis in C++. Pearson.
5. Drozdek, A. (2012). Data Structures and Algorithms in C++. Cengage Learning.

Dr. Neelima Guntupalli

LESSON - 3

NON-LINEAR AND ADVANCED DATA STRUCTURES

OBJECTIVES

The objectives of this lesson are to:

- Understand the structure and types of trees, including binary trees and their terminologies.
- Explore dictionary-based structures and Binary Search Trees (BST).
- Learn the concepts of cost amortization in algorithm analysis.
- Understand the design and implementation of priority queues using heaps.
- Study disjoint set data structures and operations including union and find.

Structure

3.1 INTRODUCTION

A linear data structure is one in which the elements are arranged in a straight line or sequence. Examples of such structures include arrays, stacks, queues, and linked lists, all of which organize data in a step-by-step or linear order.

On the other hand, a non-linear data structure organizes data in a branching or hierarchical way. In these structures, data elements are connected at different levels. The most commonly used non-linear data structures are trees and graphs.

Both trees and graphs are useful when representing relationships between elements in a hierarchy. A tree is actually a specific kind of graph that follows certain rules, while a graph is more general and flexible. In a graph, nodes can have any number of connections, and cycles (loops) are allowed. Trees, however, do not allow cycles and have a clear parent-child structure.

So, while all trees are graphs, not all graphs are trees. Figure 3.1 illustrates the difference between a tree and a structure that is not a tree.

Figure 3.1 Tree vs. Not a Tree

A tree is a commonly used non-linear data structure that is applied in many areas of computer science. It organizes data in a hierarchical manner, meaning that elements are arranged at different levels. This structure is often defined recursively, as each part of the tree can also be seen as a smaller tree.

More formally, a tree can be described as a finite collection of nodes that follows these rules:

- One special node is known as the root of the tree.
- All the other nodes are divided into zero or more separate groups called subtrees.
- Each of these subtrees is also a tree, and they are directly connected to the root node.

This structure allows for efficient representation of hierarchical relationships among data items.

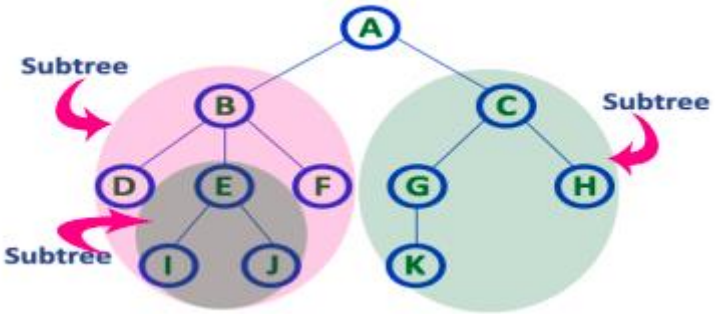


Figure 3.2 Tree Hierarchy/Subtrees

A tree is a hierarchical structure made up of nodes. At the top of the hierarchy is a special node called the root, which does not have a parent. Every other node in the tree has only one incoming link, and the node from which this link comes is called the parent node.

Each node can have multiple outgoing links to other nodes, which are known as its children. Trees follow a recursive pattern, meaning each child node can act as the root of its own smaller tree, known as a subtree.

The nodes that do not have any children are called leaf nodes, and they are found at the bottom of the tree.

3.1.1 Trees Terminology

A tree is made up of connected elements called nodes, and it is used to represent data in a hierarchical structure. Each node stores some data and also contains links to other nodes. Below are important concepts used when working with trees:

1. Node
Each element in a tree is called a node. A node holds data and can also have links to other nodes based on the tree's structure.

2. Root
The **root** is the topmost node in a tree and acts as the starting point. Every tree must have exactly one root node, and it does not have a parent. For example, in a tree where A is the root, A is the origin of that tree.

3. Edge
An edge is the connection or link between two nodes. If a tree has N nodes, it will always have N – 1 edges.

4. Parent
A parent node is a node that has a direct connection to one or more child nodes. In simple terms, if a node leads to another, the first node is considered the parent. For example, A is the parent of B, C, and D.

5. Child
A child node is a node that has a link from a parent node. Every node except the root has a parent and is considered a child. For example, H, I, and J are children of D.

6. Siblings
Siblings are nodes that share the same parent. For example, if B, C, and D all have the same parent A, they are siblings.

7. Leaf
A leaf node is a node with no children. These are also known as external nodes or terminal nodes. In other words, leaf nodes are the end points of the tree. For example, K, L, F, G, M, I, and J are all leaf nodes.

8. Internal Nodes
An internal node is a node that has at least one child. These nodes are also called non-terminal nodes. If a tree has more than one node, the root is also considered an internal node. For example, B, C, D, E, and H are internal nodes.

9. Degree
The degree of a node is the total number of its children. The degree of the tree is the highest degree among all nodes in the tree.

10. Level
Each level in a tree refers to the distance from the root node:

- The root node is at level 0.
- Its children are at level 1.
- Their children are at level 2, and so on.

Some definitions may begin with level 1 instead of 0, depending on the context.

11. Height
The height of a node is the number of edges on the longest path from that node to a leaf node. The height of the tree is the height of the root node. All leaf nodes have a height of 0.

12. Depth
The depth of a node is the number of edges from the root node to that particular node. The depth of the tree is the highest depth found among all the nodes. The root node has a depth of 0.

13. Path
A path in a tree is the sequence of nodes and the edges that connect them, from one node to another. The length of a path is the total number of edges in it. For instance, the path A → B → E → J has a length of 3.

14. Subtree
A subtree is formed by any node and all of its descendants. In simple terms, each child node forms a subtree with respect to its parent, and this structure can repeat recursively.

3.1.2 Binary Trees
A binary tree is a hierarchical, non-linear data structure in which each node has a maximum of two children, commonly known as the left child and the right child. The topmost node is alled the root, while the nodes with no children at the bottom are known as leaf nodes oreaves.

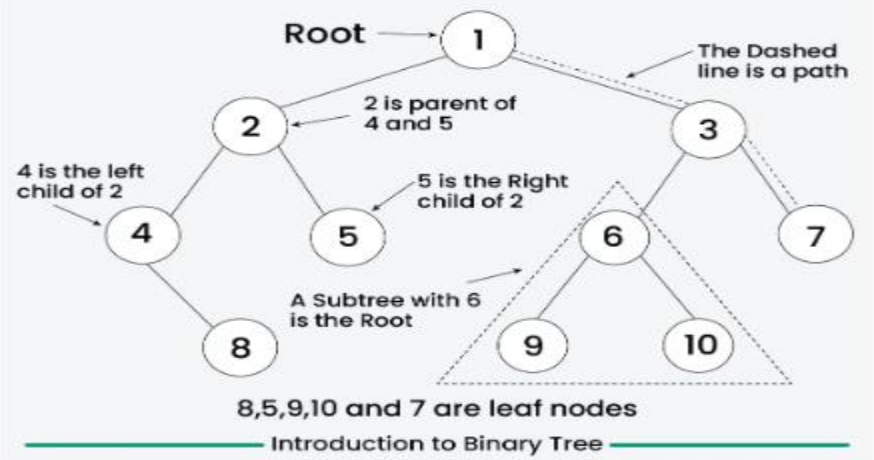


Figure 3.3 Binary Tree Example

Binary Tree Representation

Each node in a binary tree consists of three components:

- **Data**: Stores the value or information of the node.
- **Left Pointer**: Points to the left child of the node.
- **Right Pointer**: Points to the right child of the node.

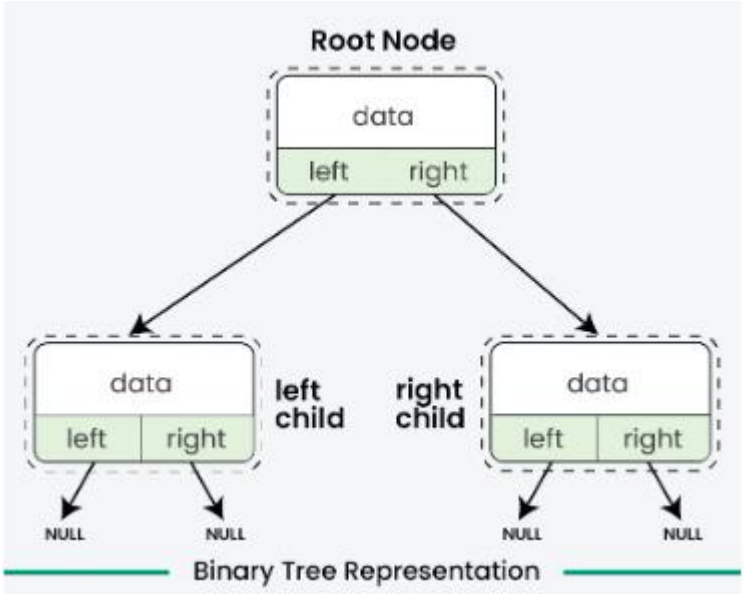


Figure 3.4 Binary Tree Node Structure

Types of binary trees in data structures
1.Full Binary Tree
A binary tree in which every node has either 0 or 2 children.
2.Complete Binary Tree
A binary tree in which all levels are completely filled except possibly the last level, and the last level has all nodes as far left as possible.
3.Perfect Binary Tree
A binary tree in which all internal nodes have exactly two children, and all leaf nodes are at the same level.
4.Balanced Binary Tree
A binary tree in which the height of the left and right subtrees of every node differ by at most 1.
5.Degenerate (or Skewed) Binary Tree.
A tree where each parent node has only one child. It can be:

- Left-skewed (each node has only a left child)
- Right-skewed (each node has only a right child)

6.Binary Search Tree (BST)
A binary tree in which the left child contains values less than the parent node and the right child contains values greater than the parent.

3.2 DICTIONARIES

In data structures, a dictionary is an abstract data type that stores data in key-value pairs, allowing efficient access, insertion, and deletion of values based on their keys. Dictionaries are widely used in computer science for searching, mapping, and organizing data quickly.

Unlike simple arrays or lists where data is accessed through indexes, dictionaries allow access through custom keys, making them suitable for applications where lookups by unique identifiers are required, such as databases, symbol tables in compilers, or caching systems.

3.2.1 Binary Search Trees

A Binary Search Tree (BST) is a special kind of binary tree that is organized in a way that makes searching, inserting, and deleting elements efficient.

Main Properties of a BST

1. Each node contains a unique key—no duplicates.
2. All keys in the left subtree of a node are smaller than the node's key.
3. All keys in the right subtree are greater than the node's key.
4. The left and right subtrees are also BSTs.

These rules help in keeping the tree ordered, making operations faster.
Searching in a BST

To search for a key x:

1. Start at the root.
2. If the tree is empty, the search fails.
3. If x equals the root's key → found!
4. If x is smaller, search the left subtree.
5. If x is larger, search the right subtree.
6. Repeat this process recursively or using a loop.

This process works efficiently in O(h) time, where h is the height of the tree.
If each node also stores the number of elements in its left subtree, you can also search by rank (like finding the 5th smallest element).
Inserting into a BST

To insert a new element x:

1. First, search for x to ensure it's not already in the tree.
2. If it's not found, insert it where the search ended—as a new leaf.
3. The new node becomes the left or right child of the last node visited, based on the comparison.
4. This insertion also takes O(h) time.

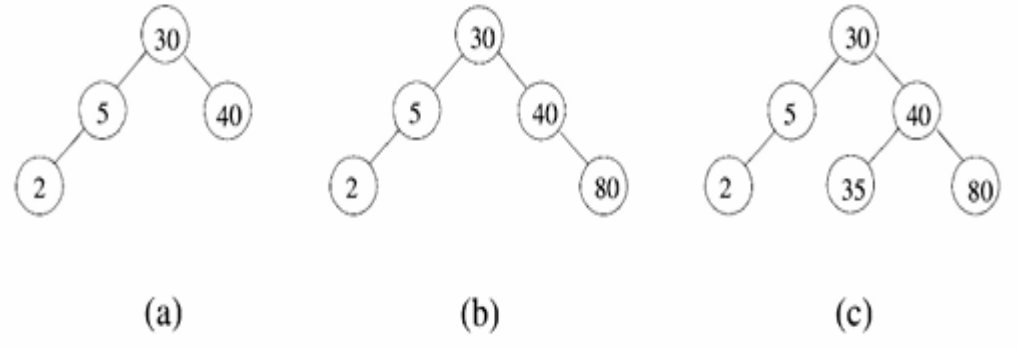


Figure 3.5 (a, b, c)BST Insertion

Algorithm Insert(x):
1. Create a new node with key = x, left child = NULL, right child = NULL
2. If the tree is empty:
    - Set the new node as the root
    - Exit
3. Set current = root
4. While current is not NULL:
    - Set parent = current
    - If x == current.key:
        - Value already exists; Do not insert
        - Exit
    - Else if x <current.key:
        - Move to left child: current = current.left
    - Else:
        - Move to right child: current = current.right
5. After loop ends:
    - If x <parent.key:
        - parent.left = new node
    - Else:
        - parent.right = new node

Deleting from a BST

There are three cases for deletion:

1. Node is a leaf (no children):

Just remove it from the tree.

2. Node has one child:

Remove the node and connect its parent to its only child.

3. Node has two children:

Find the largest node in the left subtree (in-order predecessor) or the smallest in the right subtree (in-order successor).

- Replace the value of the node to be deleted with this value.
- Then, delete that replacement node, which will be a simpler case (it will have at most one child).

Deletion also takes O(h) time.

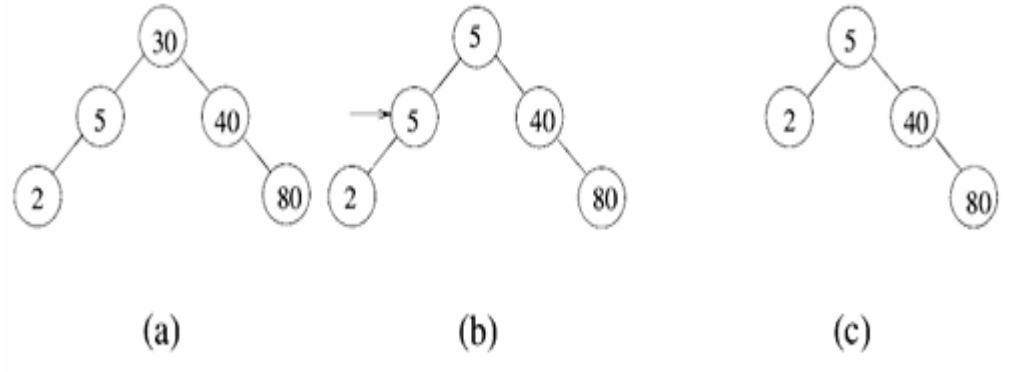


Figure 3.6 (a, b, c)BST Deletion

Height of a Binary Search Tree

- In the worst case, the BST can become a straight line (like a linked list), especially if elements are inserted in sorted order (e.g., 1, 2, 3, ..., n). Then the height becomes O(n).
- In the average case, if insertions and deletions are done randomly, the height is around O(log n).

Balanced Binary Search Trees
To avoid performance issues caused by a tall (unbalanced) tree, balanced BSTs are used. These trees maintain a height of O(log n) in all cases.

Examples of balanced BSTs:
- AVL Trees
- Red-Black Trees
- 2-3 Trees
- B-Trees

These types allow fast search, insertion, and deletion in O(log n) time.

3.2.2 Cost Amortization

Cost amortization is a technique used in algorithm analysis to determine the average performance (cost) of each operation over a sequence of operations, particularly when some individual operations are expensive but occur infrequently. It gives a more realistic measure of performance than worst-case analysis.

Why is Cost Amortization Important?

In many data structures and algorithms, most operations are inexpensive, but occasionally, an expensive operation is needed. If we only analyze the worst-case time of an individual operation, it might suggest poor performance even though the overall system performs efficiently.
Amortized analysis accounts for this by averaging the cost of operations over time.

Example: Dynamic Array Insertion

Consider a dynamic array that doubles in size when it runs out of space:
1. Most insertions take constant time O(1).
2. Occasionally, when the array is full, it must:
- Allocate a new array with double the capacity.
- Copy all existing elements to the new array, which takes O(n) time.

Although resizing is expensive, it happens infrequently. When we analyze the total cost over many operations, the average or amortized cost of each insertion remains O(1).

Techniques of Amortized Analysis:
1. **Aggregate Method:**
   - Add up the total cost of a sequence of operations.
   - Divide the total cost by the number of operations to get the average cost.

2. **Accounting Method:**
   - Assign a fixed “charge” or cost to each operation.
   - Some operations are charged more than their actual cost to “save up” for future expensive operations.
   - The extra charge is stored as "credit" and used to pay for costly operations later.
3. **Potential Method:**
   - Define a potential function that reflects the state or structure of the data.
   - The difference in potential between operations accounts for the cost.
   - Useful in more complex and mathematical scenarios.

Applications of Cost Amortization:
Amortized analysis is commonly used in:

- Dynamic arrays
- Stack operations (e.g., multi-pop)
- Self-adjusting binary search trees (e.g., splay trees)
- Disjoint-set data structures with path compression
- Hash tables with resizing

Cost amortization explains why operations such as insert, delete, or search in some data structures (like splay trees or AVL trees) remain efficient on average, even though certain operations may take longer due to restructuring or balancing. It provides a practical view of performance over a long sequence of operations.

3.3 PRIORITY QUEUES

A priority queue is an abstract data type similar to a regular queue or stack, but where each element is associated with a *priority*. Elements are dequeued based on priority rather than order of insertion.

Key Operations:

- **Insert (enqueue):** Add a new element with a specific priority.
- **Extract-Min / Extract-Max:** Remove and return the element with the highest or lowest priority (depending on whether it is a max-priority or min-priority queue).
- **Peek:** View the element with the highest or lowest priority without removing it.
- **Increase/Decrease-Key:** Change the priority of an element already in the queue.

Priority queues are commonly implemented using **heaps**, which allow efficient performance of the above operations.

3.3.1 Heaps

A heap is a special type of binary tree used to implement priority queues. It satisfies the heap property:

- **Max-Heap Property:** The value of each node is greater than or equal to the values of its children.
- **Min-Heap Property:** The value of each node is less than or equal to the values of its children.

Characteristics of Heaps:

- It is a complete binary tree, meaning all levels are filled except possibly the last, which is filled from left to right.
- Heaps are typically implemented using arrays, where for a node at index i:
  - Left child = index 2i
  - Right child = index 2i + 1
  - Parent = index i/2 (integer division)

Heap Operations:

- **Insert:** Add the element at the end and restore heap property using "heapify-up".
- **Delete (Extract-Min/Max):** Replace the root with the last element and restore heap property using "heapify-down".
- **Heapify:** A process to maintain the heap property, typically used after insertion or deletion.

3.3.2 Heap Sort

Heap sort is a sorting algorithm that follows the heap algorithm to convert a list of unsorted elements into a heap structure, and then sorts the heap. This algorithm has a time complexity of O(n log n), making it suitable for large datasets. Instead of other sorting algorithms such as quicksort or merge sort, heap sort guarantees O(n log n) performance for both best and worst cases, making it a reliable sorting method.

Data structures that satisfy the heap property are specialized binary tree-based structures. Heaps are usually represented as binary heaps and come in two main types:

- Max heap: In a max heap, the value of the parent node is greater than or equal to the values of its children. This ensures that the largest element is always at the root.
- Min heap: Nodes that have a value less than or equal to the value of their children are called min heaps. The smallest node in a min-heap is at the root.

Heaps are typically used to implement priority queues, where the highest or lowest priority element must be accessed quickly.

Algorithm for Heap Sort

HeapSort uses the Max-Heap structure to sort an array. The main idea behind HeapSort is to:
Build a Max-Heap from the input array.

- Swap the root (maximum value) with the last element in the array.
- Reduce the heap size by 1 (excluding the sorted elements at the end).
- Re-heapify the heap to restore the heap property.

Working of Heap Sort Algorithm

The Heap Sort algorithm is a comparison-based sorting technique that uses a binary heap data structure.
It works in two major phases:

1. Building the Max-Heap
2. Extracting the Root and Sorting the Array

Let's take an example of an unsorted array:
[45, 21, 67, 34, 89, 12, 56, 90, 72]

1. Building the Max-Heap

To begin, given array must be converted into a max heap — a complete binary tree where each parent node is greater than or equal to its child nodes.
In an array-based heap representation, for any node at index i:

- Left child index = 2i + 1
- Right child index = 2i + 2

The heap construction starts from the last non-leaf node, which is at index
(n/2) - 1 = (9/2) - 1 = 3.
Now, we will heapify from index 3 up to 0.

Step 1: Heapify node at index 3 (element 34)
Children: 90 (index 7), 72 (index 8).
→ The largest child is 90. Swap 34 and 90.
Array after heapify at index 3:
[45, 21, 67, 90, 89, 12, 56, 34, 72]

Step 2: Heapify node at index 2 (element 67)
Children: 12 (index 5), 56 (index 6).
→ 67 is already greater than both children, no change.
Array remains:
[45, 21, 67, 90, 89, 12, 56, 34, 72]

Step 3: Heapify node at index 1 (element 21)
Children: 90 (index 3), 89 (index 4).
→ Largest child is 90. Swap 21 and 90.
Now heapify subtree rooted at index 3.
After swap: [45, 90, 67, 21, 89, 12, 56, 34, 72]
Children of index 3 → 34, 72 → Largest = 72 → swap 21 and 72.
Array becomes:
[45, 90, 67, 72, 89, 12, 56, 34, 21]

Step 4: Heapify root node at index 0 (element 45)
Children: 90 (index 1), 67 (index 2).
→ Largest child is 90. Swap 45 and 90.
After swap: [90, 45, 67, 72, 89, 12, 56, 34, 21]
Now heapify at index 1 (45): children = 72, 89 → largest = 89. Swap 45 and 89.
Final Max-Heap:
[90, 89, 67, 72, 45, 12, 56, 34, 21]

2. Extracting the Root and Sorting the Array

Now that we have a Max-Heap, we can sort the array by repeatedly removing the root (maximum element) and rebuilding the heap.
Each removal places the largest element at the end of the array.

Step 1: Swap root (90) with last element (21).
Array → [21, 89, 67, 72, 45, 12, 56, 34, 90]
Heapify on first 8 elements.
After heapify → [89, 72, 67, 34, 45, 12, 56, 21, 90]

Step 2: Swap root (89) with last of heap (21).
Array → [21, 72, 67, 34, 45, 12, 56, 89, 90]
Heapify first 7 → [72, 45, 67, 34, 21, 12, 56, 89, 90]

Step 3: Swap root (72) with last of heap (56).
Array → [56, 45, 67, 34, 21, 12, 72, 89, 90]
Heapify first 6 → [67, 45, 56, 34, 21, 12, 72, 89, 90]
Step 4: Swap root (67) with last of heap (12).
Array → [12, 45, 56, 34, 21, 67, 72, 89, 90]
Heapify first 5 → [56, 45, 12, 34, 21, 67, 72, 89, 90]

Step 5: Swap root (56) with last of heap (21).
Array → [21, 45, 12, 34, 56, 67, 72, 89, 90]
Heapify first 4 → [45, 34, 12, 21, 56, 67, 72, 89, 90]

Step 6: Swap root (45) with last of heap (21).
Array → [21, 34, 12, 45, 56, 67, 72, 89, 90]
Heapify first 3 → [34, 21, 12, 45, 56, 67, 72, 89, 90]

Step 7: Swap root (34) with last of heap (12).
Array → [12, 21, 34, 45, 56, 67, 72, 89, 90]
The array is now completely sorted in ascending order.
Final Sorted Array
[12, 21, 34, 45, 56, 67, 72, 89, 90]

3.4 Sets and Disjoint Set Union

In this topic, we deal with how to represent and manage disjoint sets using trees.

- Disjoint sets are sets that do not share any common element.
- The elements in the sets are labeled with numbers like 1, 2, 3, ..., n.

For example, if n = 10, we could have:

Set 1: {1, 7, 8, 9}
Set 2: {2, 5, 10}
Set 3: {3, 4, 6}

Each set is stored as a tree, but instead of pointing from parent to child, here each node points to its parent. The top node (root) of each tree represents the whole set.

Operations Performed:

1. **Union(i, j)**
   - Combines two different sets into one set.
   - For example, Union(Set1, Set2) = {1, 2, 5, 7, 8, 9, 10}.
2. **Find(i)**
   - Finds the set-in which element i belongs by following the parent links until the root is found.

Union and Find Operations
We represent each element and its parent using an array p[1..n].

- If p[i] = -1, it means i is the root of its tree.
- Otherwise, p[i] gives the parent of i.

Example Tree Representation in an Array:
If we have:
Set 1: {1, 7, 8, 9}
Set 2: {2, 5, 10}
Set 3: {3, 4, 6}

Then we might represent it as:
Index (i): 1 2 3 4 5 6 7 8 9 10
p[i]: -1 5 -1 3 -1 3 1 1 1 5
To perform:

- Find(i): Keep going to the parent p[i] until you reach a root (where p[i] < 0).
- Union(i, j): Make one tree's root a child of the other tree's root by updating the p[i] array.

Problems with Simple Union and Find
Using simple Union can create tall trees. This makes Find(i) slow because it may need to climb many levels.
Example:
If you do Union(1,2), then Union(2,3), ..., up to Union(n-1, n), you'll get a long chain, and Find(n) will take O(n) time.

Optimized Union: Weighted Union
To fix the tall tree problem, we use the Weighted Union rule:

- Always attach the smaller tree under the larger tree.
- How to do it:
- Count the number of nodes in each tree (we store this as a negative number at the root in the p[i] array).
- If p[i] = -4, it means i is a root and its tree has 4 nodes.
- Now when doing Union(i, j):
- Attach the smaller tree's root to the bigger tree's root.
- Update the count (i.e., total number of nodes in the new tree).

This avoids tall trees and makes find operations faster.

Tree Height with Weighted Union

Using Weighted Union, we guarantee that the height of any tree created by unions will be at most log2(n) + 1.
So Find(i) now takes O(log n) time in the worst case, which is much better than O(n).
Further Optimization: Path Compression (Collapsing Rule)
Even better, we improve the Find(i) function using the Collapsing Rule:

- While finding the root of a node, we also update all nodes along the path to point directly to the root.

This flattens the tree so that future Find operations are much faster.

Example:
If 8 → 5 → 1 (root), after Find(8), we change it so 8 → 1, and 5 → 1.
This means that future Find(8) and Find(5) take only 1 step.

Time Complexity (Advanced Concept)
When we combine both Weighted Union and Path Compression, the total time to do f finds and u unions is almost linear:

- Time = O((n + f) * α(n))
  Where α(n) is the inverse Ackermann function, which grows extremely slowly (for all practical values of n, it's less than 5).

This means that:
"For all practical purposes, our Find and Union operations are nearly constant time."
Conclusion.

- We can efficiently manage disjoint sets using trees and arrays.
- Weighted Union prevents tall trees.
- Path Compression flattens the trees during find operations.
- Together, they ensure that even a large number of operations are very fast.

3.5 SUMMARY

This lesson introduced key non-linear and advanced data structures used in algorithm design and implementation. Trees are fundamental for representing hierarchical data and include types like binary trees and binary search trees. BSTs enable efficient search, insert, and delete operations and form the foundation of dictionary-based structures. Cost amortization offers a more realistic measure of an algorithm's average performance over a sequence of operations. Priority queues use heaps, and heap sort is an efficient comparison-based sorting technique. Disjoint set unions, optimized by weighted union and path compression, provide fast set operations. These concepts are critical for managing dynamic and hierarchical data efficiently.

3.6 Key Terms

Binary Tree, Binary Search Tree, Amortized Cost, Heap, Union-Find, Priority Queue

3.7 REVIEW QUESTIONS

1. What is the difference between linear and non-linear data structures?
2. Explain any five common terminologies used in trees.
3. Define a binary search tree. What are its properties and use cases?
4. Describe the three cases of deletion in a BST with examples.
5. What is cost amortization? How does it differ from worst-case analysis?
6. Explain the insert and extract-max operations in a max-heap.
7. Describe the working of the heap sort algorithm.
8. How do weighted union and path compression improve union-find operations?

3.8 SUGGESTIVE READINGS

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms. MIT Press.
2. Horowitz, E., Sahni, S., & Anderson-Freed, S. (2008). Fundamentals of Data Structures in C. University Press.
3. Sedgewick, R., & Wayne, K. (2011). Algorithms. Addison-Wesley.
4. Aho, A. V., Hopcroft, J. E., & Ullman, J. D. (1983). Data Structures and Algorithms. Addison-Wesley.
5. Drozdek, A. (2012). Data Structures and Algorithms in C++. Cengage Learning.

Dr. Neelima Guntupalli

LESSON-4

DIVIDE AND CONQUER TECHNIQUES

OBJECTIVES

The objectives of this lesson are to:

- Understand the concept and utility of the divide and conquer strategy.
- Learn the general method and recurrence relations.
- Apply the Master Theorem for time complexity analysis.
- Implement and compare recursive and iterative binary search.
- Understand the divide and conquer approach for finding maximum and minimum.
- Evaluate performance, memory use, and element comparisons in divide and conquer algorithms.

STRUCTURE

4.1 INTRODUCTION

The Divide and Conquer technique is a fundamental algorithm design strategy used to solve complex problems efficiently by breaking them down into smaller, manageable subproblems. Each subproblem is solved independently, often using recursion, and their solutions are then combined to form the overall solution to the original problem. This approach is particularly effective for problems where the subproblems are similar in nature to the original problem and can be solved more easily due to their reduced size. Common examples of divide and conquer algorithms include merge sort, quicksort, binary search, and Strassen's matrix multiplication. This technique not only improves performance over brute force methods but also provides a systematic way to design efficient and scalable algorithms.An algorithm design paradigm is a general approach or strategy used to solve computational problems.The following Table 4.1 compares the characteristics of several common algorithm design paradigms.

Table 4.1 Types of Paradigms

Paradigm	Description
Brute Force	Try all possible solutions and pick the best one. Simple but inefficient for large inputs. Example: Linear search, checking all subsets.
Divide and Conquer	Divide the problem into smaller subproblems, solve them independently, and combine the results. Example: Merge Sort, Binary Search.
Greedy Algorithms	Make the locally optimal choice at each step, hoping it leads to a global optimum. Example: Kruskal's and Prim's Algorithms.
Dynamic Programming (DP)	Break problems into overlapping subproblems, solve each once and store the result to avoid repeated work. Example: Fibonacci sequence with memorization, 0/1 Knapsack.
Backtracking	Try a solution, go back (backtrack) if it doesn't lead to the final answer. Example: N-Queens problem, Sudoku.
Randomized Algorithms	Use random inputs or choices in the algorithm to simplify or speed up solutions. Example: Randomized Quick Sort.

The Divide-and-Conquer paradigm is used when:

- The problem can be divided into smaller instances of the same problem.
- Solutions of subproblems can be combined to form the solution of the original problem.
- The structure of the problem naturally supports recursion.
- It fits between brute force and dynamic programming in terms of complexity and performance. It is most effective when:
- Subproblems are independent.
- The cost of combining subproblem solutions is not too high.

4.2 General Method and Recurrence Relations

The general method of the Divide and Conquer strategy involves solving a problem by dividing it into smaller subproblems, solving each subproblem independently, and then combining their solutions to form the overall result. This method relies on recursive decomposition, where the problem continues to be divided until a base case is reached that can be solved directly. To analyze the efficiency of such algorithms, recurrence relations are used. A recurrence relation expresses the total time required to solve a problem of size n in terms of the time needed to solve smaller subproblems and the time required to divide and combine them. These mathematical approaches helps in estimating the time complexity and comparing the efficiency of different divide and conquer algorithms.

The Divide and Conquer technique solve problems in three steps:

1. **Divide** – Break the original problem into smaller subproblems.
2. **Conquer** – Solve each subproblem recursively (or directly if small enough).
3. **Combine** – Merge the solutions of the subproblems to form the solution of the original problem.

This recursive breakdown continues until the base case is reached, which is simple enough to be solved directly. After solving all the small problems, the solutions are combined in a way that constructs the final result.

Recurrence Relations and Time Complexity

To analyze the performance (or time complexity) of divide and conquer algorithms, we often use recurrence relations. A recurrence relation expresses the total time taken, T(n), to solve a problem of size n, in terms of the time taken to solve smaller subproblems.

General Form of a Recurrence Relation:

When a problem of size n is divided into a subproblems, each of size n/b, and it takes f(n) time to divide and combine them, the recurrence is:

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

a = number of subproblems
n/b = size of each subproblem
f(n) = time to divide and combine

4.3 MASTER THEOREM

The Master Theoremis a powerful tool that provides a direct way to solve recurrence relations of the form:

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

Where:

- $a \geq 1$ and $b > 1$ are constants,
- f(n) is an asymptotically positive function (like n, n^2, log n, etc.)

Three Cases of the Master Theorem:

1. **Case 1**: If $f(n) = O(n^{\log_b a - \epsilon})$ for some $\epsilon > 0$,
   → Then $T(n) = \Theta(n^{\log_b a})$

   > (Divide step dominates)

2. **Case 2**: If $f(n) = \Theta(n^{\log_b a})$,
   → Then $T(n) = \Theta(n^{\log_b a} \cdot \log n)$

   > (Divide and combine balanced)

3. **Case 3**: If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some $\epsilon > 0$ **and** regularity condition holds,
   → Then $T(n) = \Theta(f(n))$

   > (Combine step dominates)

4.4 BINARY SEARCH

Binary Search is a classic example of the divide-and-conquer strategy applied to searching. It is used to efficiently find the position of a target element in a sorted list. Unlike linear search, which checks every element sequentially, binary search divides the list in half with each step, significantly reducing the number of comparisons. Binary search operates on the principle of elimination. It uses the fact that the list is already sorted and eliminates half of the remaining elements at each step. This makes it extremely efficient for large datasets.

The key difference between linear search and binary search lies in their approach and efficiency. Linear search scans each element one by one until the target is found or the list ends, making it suitable for unsorted data but inefficient for large datasets, with a worst-case time complexity of O(n). In contrast, binary search works only on sorted data and repeatedly divides the search space in half, achieving a much better time complexity of O(log n). While linear search is simpler and does not require sorted input, binary search is significantly faster when dealing with large and sorted datasets.

Steps Involved:

1. Identify the middle element of the current list or sublist.
2. Compare the target value with the middle element:
   - If equal, return the index (element found).
   - If target < middle, repeat the search in the left sub-array.
   - If target > middle, repeat the search in the right sub-array.
3. Continue the process until the target is found or the sublist size reduces to zero.

Algorithm for Iterative Binary Search

```
BinarySearch(a, n, x)
// Input: a[1..n] is a sorted array (non-decreasing order), n > 0
//  x is the element to search for
// Output: Index j such that a[j] = x, or 0 if not found

    1. low: = 1
    2. high: = n
    3. while low <= high do
    4. mid: = floor((low + high) / 2)
    5. if x < a[mid] then
    6. high:= mid - 1
    7. else if x > a[mid] then
    8. low:= mid + 1
    9. else
    10. return mid
    11. return 0
```

Algorithm for Recursive Binary Search

```
BinarySearch(a, i, l, x)
// Input: a[i..l] is a sorted array (non-decreasing order), i<= l
//        x is the element to search for
// Output: Index j such that a[j] = x, or 0 if not found
```

```
1. if i> l then
2. return 0
3. mid:= floor((i + l) / 2)
4. if x = a[mid] then
5. return mid
6. else if x < a[mid] then
7. return BinarySearch(a, i, mid - 1, x)
8. else
9. return BinarySearch(a, mid + 1, l, x)
```

Example
Consider the sorted array:

[4, 15, 17, 26, 30, 46, 48, 56, 58, 82, 90, 95]
Target: 58
Step-by-step Search:

1. Initial middle: 30 (index 4) → 58 > 30 → search in right half.
2. New range: [46, 48, 56, 58, 82, 90, 95]
   - Middle: 56 (index 7) → 58 > 56 → search in right half.
3. New range: [58, 82, 90, 95]
   - Middle: 58 (index 8) → Match found!

Time Complexity Analysis

Case	Time Complexity
Best Case	O(1)
Average Case	O(log n)
Worst Case	O(log n)

- In each step, the search space is reduced by half.
- Thus, for n elements, the maximum number of steps required is login.

T
able 4.2 Recursive vs Iterative Version

Feature	Recursive Version	Iterative Version
Approach	Uses function calls (stack frames)	Uses loops
Memory Usage	Higher due to recursion stack	Lower; no extra stack needed
Readability	Often easier to read and understand	Slightly more complex but efficient
Risk	Stack overflow for very large datasets	More control over space

Advantages of Binary Search

- Efficient for large **sorted** datasets.
- Easy to implement both recursively and iteratively.
- Much faster than linear search for large lists.

Limitations

- Only works on sorted data. Unsorted data must be sorted first (which can take O(n log n) time).
- Performance gain is less noticeable on small datasets.

Binary Search is a fundamental algorithm demonstrating the power of divide and conquer. With a logarithmic time complexity, it significantly improves search efficiency in sorted datasets. Understanding both recursive and iterative implementations helps grasp its practical applications and limitations.

Let us select the 14 entries:
-15, -6, 0, 7, 9, 23, 54, 82, 101, 112, 125, 131, 142, 151
and place them in array a[1:14], and simulate the steps that BinarySearch goes through as it searches for different values of x. We only need to track variables low, high, and mid as we go through the algorithm. We try the following x values: 151, -14,and 9. These cover two successful searches and one unsuccessful search.

Theorem : Algorithm BinarySearch(a, n, x) works correctly.

Proof: Assume all steps function correctly and comparisons such as x > a[mid] are executed accurately. Initially, low = 1, high = n, and the array is sorted in non-decreasing order, i.e., a[1] <a[2] < ... < a[n]. If n = 0, the while loop will not execute. When the loop is active, the potential positions for x are between a[low] and a[high]. If x = a[mid], the value has been found. Otherwise, the search range is reduced by adjusting low or high. If low becomes greater than high, it indicates that x is not present in the array, and the loop exits.

To comprehensively test the Binary Search algorithm, changing the array values a[1:n] is not necessary. By selecting different x values, all possible outcomes can be simulated. For successful searches, each existing element is tested. For unsuccessful searches, testing n + 1 different values is sufficient. Therefore, the total number of required test cases is 2n + 1 for any array size n.

In analyzing the performance of the Binary Search algorithm, two key aspects are considered: the frequency of certain operations and memory usage. Binary Search utilizes space for n array elements along with the variables low, high, mid, and x, resulting in a total of n + 4 variables. Regarding execution time, the best, average, and worst cases are examined. Based on the previous data set, the algorithm performs comparisons along with some arithmetic and data movement steps. The primary focus is on the number of times x is compared with elements in the array a[].

These are referred to as element comparisons. It is assumed that each if statement involves only one comparison. The number of comparisons per element is analyzed accordingly.

4.5 FINDING MAXIMUM AND MINIMUM

To find the largest (maximum) and smallest (minimum) elements in an array, we can use different approaches. Below, we explore and compare the Linear Scan method and the Divide and Conquer method using examples.

4.5.1 Linear Scan vs Divide and Conquer

Linear Scan (StraightMaxMin Method)
In the linear scan method, we start by setting both the maximum and minimum values to the first element of the array. Then, we go through each remaining element and compare it with the current max and min to update them.

Example:
Array:
a[1..14] = [-15, -6, 0, 7, 9, 23, 54, 82, 101, 112, 125, 131, 142, 151]
Steps:

- Start with: max = -15, min = -15
- After comparing with each of the remaining 13 elements:
  - Final max = 151
  - Final min = -15

This method is simple but performstwo comparisons for each of the remaining (n – 1) elements**.**

Divide and Conquer Method
In this method, we divide the array into two halves. We then recursively find the max and min in each half, and finally compare the results to find the overall max and min.

Steps:

1. If the array has only 1 element: it is both max and min.
2. If the array has 2 elements: one comparison is enough to decide max and min.
3. For larger arrays:
   - Divide the array into two halves.
   - Recursively find the max and min in each half.
   - Compare the two maxima and two minima to get the final result.

Example:

Same array:
[-15, -6, 0, 7, 9, 23, 54, 82, 101, 112, 125, 131, 142, 151]

- Divide into left and right parts: a[1..7], a[8..14]
- Recursively find max and min in each part
- Combine: max(54, 151) = 151; min(-15, 82) = -15

This method does fewer comparisons than the linear scan, especially for large arrays.

4.5.2 Number of Comparisons

Linear Scan:
Each element (after the first) is compared twice:

- One for max
- One for min

So, for n = 14 elements:
Comparisons = $2(n – 1) = 2 \times 13 = 26$
Divide and Conquer:
The number of comparisons follows a recurrence relation:
$T(n) = 2T(n/2) + 2$
For $n = 2^k$, solution is:
$T(n) = 3n/2 – 2$
For n = 14:
$T(14) \approx 19$ comparisons
So, Divide and Conquer saves about 25% of the comparisons compared to linear scan.

- Linear Scan: 26 comparisons
- Divide and Conquer: 19 comparisons
- Savings: 7 comparisons

4.5.3 Recursive Implementation

Let's take a smaller array to see how Divide and Conquer works recursively:
Array: a[1..9] = [22, 13, -5, -8, 15, 60, 17, 31, 47]

Step-by-Step Process:

1. Split a[1..9] into a[1..5] and a[6..9]
2. a[1..5] → split into a[1..3] and a[4..5]
3. Continue splitting until we get subarrays of size 1 or 2
4. Find max and min in each small subarray
5. Merge results:
   - Compare max values of subarrays to find overall max
   - Compare min values of subarrays to find overall min

Recursion Tree:

Each level splits the array further. If $n = 2^k$, there are about $\log_2 (n) + 1$ levels of recursion.

4.6 SUMMARY

The Divide and Conquer technique is an efficient algorithm design strategy that simplifies complex problems by breaking them into smaller, manageable subproblems. Each subproblem is solved independently, and their solutions are combined to obtain the final result. This method is especially effective when the problem structure supports recursion and the subproblems are independent. Common applications include binary search, merge sort, and finding maximum and minimum elements. To evaluate the efficiency of divide and conquer algorithms, recurrence relations are used, and their solutions can be derived using the Master Theorem. Binary search demonstrates the power of this approach by reducing the search time in sorted arrays to logarithmic complexity. Similarly, the divide and conquer method for finding maximum and minimum reduces the number of comparisons compared to the linear scan method, making it optimal for large datasets. Although recursion adds overhead in terms of memory, the divide and conquer strategy remains a preferred choice for solving a wide range of computational problems efficiently.

4.7 KEY TERMS

Divide and Conquer, Recurrence Relation, Binary Search, Master Theorem, Recursive Algorithm, Time Complexity

4.8 REVIEW QUESTIONS

1. What is the divide and conquer strategy? Explain its steps.
2. Compare brute force, divide and conquer, and dynamic programming.
3. Write the iterative and recursive versions of Binary Search.
4. Analyze the time complexity of binary search in different cases.
5. How does divide and conquer help in finding max and min efficiently?
6. Derive and explain the recurrence relation used for divide and conquer max-min.

4.9 SUGGESTIVE READINGS

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms. MIT Press.
2. Horowitz, E., Sahni, S. (2008). Fundamentals of Data Structures in C. University Press.
3. Weiss, M. A. (2014). Data Structures and Algorithm Analysis in C++. Pearson.
4. Lafore, R. (2002). Data Structures and Algorithms in C++. Sams Publishing.

Dr. Neelima Guntupalli

LESSON-5
SORTING ALGORITHMS USING DIVIDE AND CONQUER

OBJECTIVES

After completing this lesson, the learner will be able to:

- Understand the working principles of Merge Sort and Quick Sort.
- Implement recursive and iterative versions of Merge Sort.
- Analyze the time and space complexity of Merge Sort and Quick Sort.
- Explain the partitioning process in Quick Sort.
- Compare Merge Sort and Quick Sort, including their advantages and drawbacks.
- Apply insertion sort to improve the performance of divide and conquer-based sorting algorithms.

STRUCTURE

5.1 INTRODUCTION

Sorting is a fundamental operation in computer science used to arrange elements in a specific order, typically numerical or lexicographical. Efficient sorting is crucial for optimizing the performance of various algorithms that require sorted data, such as search algorithms and database systems. Among the different sorting techniques, those based on the divide and conquer paradigm offer a powerful approach to solving the sorting problem with improved performance.

Divide and conquer sorting algorithms operate by breaking down the input list into smaller sublists, sorting these sublists recursively, and then combining the results to produce the final sorted list. This approach not only simplifies the sorting task but also improves efficiency through structured recursion and minimization of redundant operations.

Two of the most prominent sorting algorithms that use this method are Merge Sort and Quick Sort**.** Merge Sort guarantees consistent performance with a worst-case time complexity of O(n log n)**,** while Quick Sort typically performs faster in practice due to in-place partitioning.

Both algorithms benefit from enhancements like insertion sort for small subarrays and randomized pivot selection, which further optimize their real-world efficiency.
These lessons explore the principles, implementation, and optimizations of Merge Sort and Quick Sort, providing a comprehensive understanding of sorting using divide and conquer.

5.2 MERGE SORT

Merge Sort serves as a strong example of the divide-and-conquer technique in algorithm design. It is particularly known for maintaining a worst-case time complexity of *O(n log n)*, which is considered efficient for sorting operations.

The fundamental idea involves breaking the input array into two halves. Each half is sorted independently using recursive calls, and then the two sorted halves are merged together to form a single, fully sorted array. The merging process is deferred until the recursion has broken the array down into the smallest possible units—typically subarrays of one element.

This recursive breakdown continues until each element stands alone, after which merging begins. During merging, elements from both sides are compared and placed into an auxiliary array in sorted order. Once all elements are merged, the result is copied back into the original array.

Merge Sort Algorithm

```
MergeSort(low, high)
{
If low < high:
        mid ← (low + high) / 2
MergeSort(low, mid)
MergeSort(mid + 1, high)
Merge(low, mid, high)
}
```

Merge Procedure

```
Merge(low, mid, high)
{
h ← low, i ← low, j ← mid + 1
While h ≤ mid and j ≤ high:
        If a[h] ≤ a[j]:
                b[i] ← a[h];
                h ← h + 1
        Else:
        b[i] ← a[j];
                j ← j + 1
i ← i + 1
}
Copy any remaining elements from a[h...mid] or a[j... high] into b
Copy b[low...high] back to a[low...high]
```

Example

Take an array of ten numbers:
[310, 285, 179, 652, 351, 423, 861, 254, 450, 520]

The MergeSort algorithm begins by dividing this array into two equal parts:

- First subarray: a[1:5] = [310, 285, 179, 652, 351]
- Second subarray: a[6:10] = [423, 861, 254, 450, 520]

The first part a[1:5] is further divided into:

- a[1:3] = [310, 285, 179]
- a[4:5] = [652, 351]

Then, a[1:3] is split again into:

- a[1:2] = [310, 285]
- a[3:3] = [179]

Now, a[1:2] is split into two single-element subarrays:

- a[1:1] = [310]
- a[2:2] = [285]

At this point, the merging process starts. Until now, no actual data movement has happened. The recursive function internally keeps track of these small subarrays.

Visually, the array structure looks like this now:
(310 | 285 | 179 | 652, 351 | 423, 861, 254, 450, 520)
Here, the vertical bars indicate the boundaries between subarrays.
The first merge operation is performed between a[1] and a[2]. The result is:
(285, 310 | 179 | 652, 351 | 423, 861, 254, 450, 520)
Now, a[3] = 179 is merged with [285, 310], resulting in:
(179, 285, 310 | 652, 351 | 423, 861, 254, 450, 520)
Next, the elements a[4] and a[5] i.e. [652, 351] are merged to give:
(179, 285, 310 | 351, 652 | 423, 861, 254, 450, 520)
Then, the two sorted arrays [179, 285, 310] and [351, 652] are merged into one:
(179, 285, 310, 351, 652 | 423, 861, 254, 450, 520)
This completes the sorting of the first half of the array. The algorithm now begins processing the second half.

The second half [423, 861, 254, 450, 520] goes through similar steps:

- First, 423 and 861 are merged
- Then, 254 is merged into them
- The result becomes [254, 423, 861]
- Separately, 450 and 520 are merged into [450, 520]
- Then, [254, 423, 861] and [450, 520] are merged together:

(179, 285, 310, 351, 652 | 254, 423, 450, 520, 861)
Finally, the two fully sorted halves:

- [179, 285, 310, 351, 652]
- [254, 423, 450, 520, 861]

are merged one last time, giving the completely sorted array:
(179, 254, 285, 310, 351, 423, 450, 520, 652, 861)

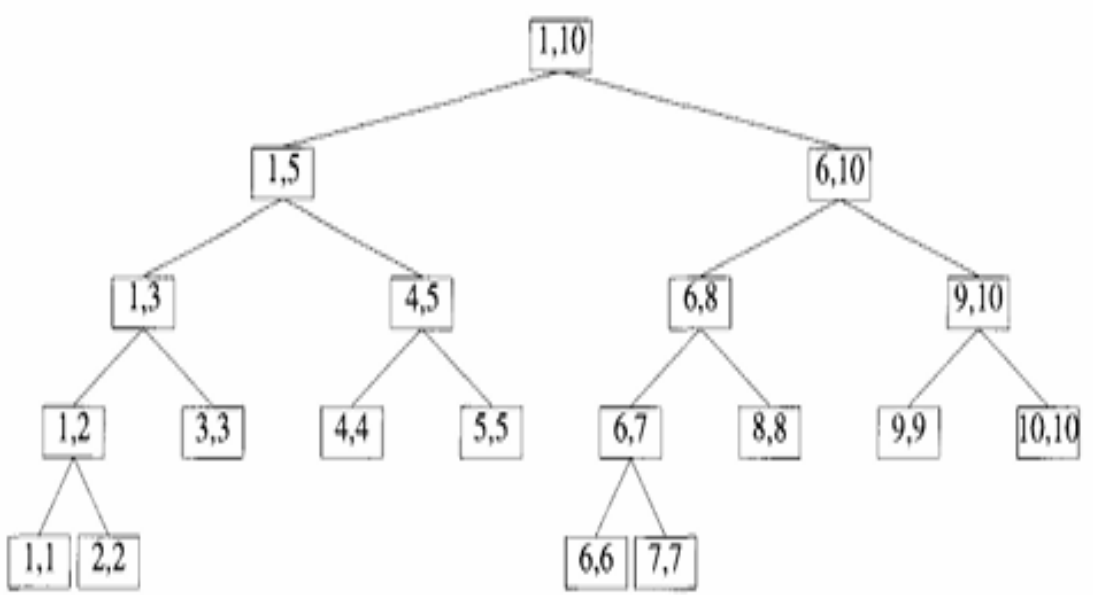


Figure 3.1 Tree of calls of MergeSort

Recursive Call Tree for MergeSort

A tree diagram in Figure 3.1 represents how the recursive calls were made. Each node in the tree shows the values of low and high, which define the subarray currently being worked on. The division continues until each subarray contains just one element.

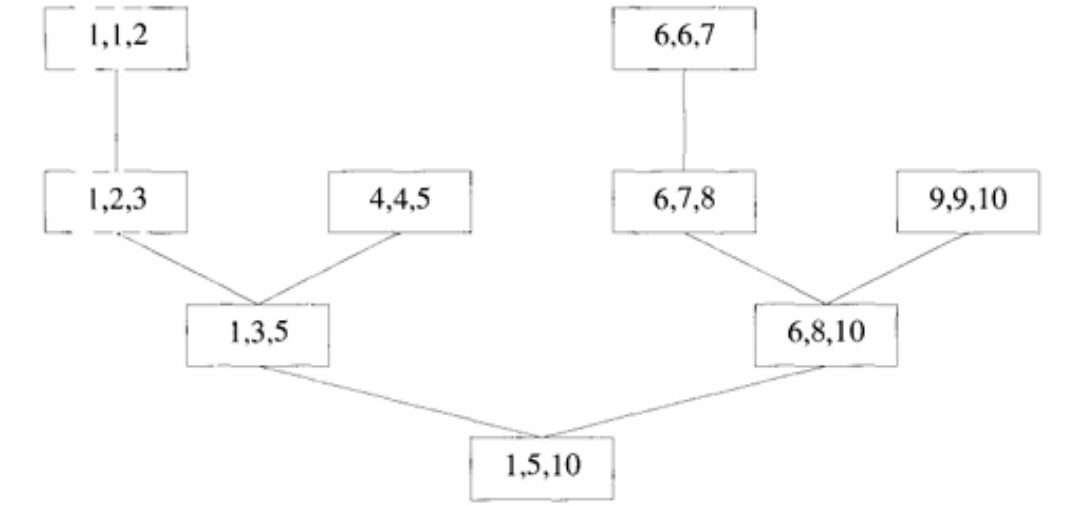


Figure3.2 Treeof callsof Merge

Another tree diagram in Figure 3.2explains how calls to the Merge function are made. For example, one node merges a[1:2] with a[3], which was shown earlier.

Time Complexity of MergeSort

If the time taken to merge is proportional to the size of the array (i.e. linear), then the time required to sort n elements follows the recurrence:

$T(n) = 2T(n/2) + cn$ for $n > 1$

$T(1) = a$ where a is a constant

Solving this gives:

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

5.2.1 Refinements in MergeSort

Even though MergeSort is efficient in terms of time, it has some drawbacks. These issues can be solved with improvements, though the overall time complexity remains O(n log n). Also, as explained in Chapter 10, no comparison-based sorting algorithm can perform better in the worst case.

One main issue is space usage. MergeSort uses an extra array of size n, which means it needs twice as much space (i.e., 2n memory locations). This is because it's difficult to merge two sorted arrays in-place without extra space.

Using Links to Save Space

Instead of copying data into another array during merging, an extra field (like a pointer or index) can be added to each record (element). This field will "link" elements together in sorted order. So during merging, only the links (not the data itself) are rearranged. Since links are smaller than actual records, this method saves space.

An auxiliary array link[1:n] is created, which holds integers from 0 to n. These numbers act like pointers to elements in a[]. A list is formed by chaining these pointers and ends with a 0.

Example of Linked Lists in MergeSort:
Suppose link[] looks like:
link: [1]=6, [2]=4, [3]=7, [4]=1, [5]=3, [6]=0, [7]=8, [8]=0

This forms two linked lists:

- Q = (2 → 4 → 1 → 6)
- R = (5 → 3 → 7 → 8)

These lists represent sorted subsets of a[1:8]. This means:

- a[2] <a[4] <a[1] <a[6]
- a[5] <a[3] <a[7] <a[8]

Problem of Recursion and Stack Space

Another concern is the extra stack space used due to recursion. Since the array is always divided in half, the maximum stack depth is proportional to log n. This need arises because of the top-down approach.

This problem can be solved by designing a bottom-up version of MergeSort (iterative), which avoids recursion. Exercises at the end of the chapter suggest how this can be done.

Use Insertion Sort for Small Subarrays
When the size of the subarray is very small (e.g., less than 16), recursive calls take more time than the actual sorting. To optimize, a different sorting algorithm can be used for these small subarrays.

Insertion Sort works well on small datasets. Its logic is:

```
for j = 2 to n:
  place a[j] in its correct position in a[1:j-1]
```

This algorithm performs very fast when there are only a few elements, though for large values of n, it becomes slower ($O(n^2)$).

Time Complexity of Insertion Sort

- **Worst-case time:**

If the array is in reverse order, the inner loop runs j times for each j,
so total time $=\sum j$ from 2 to $n = n(n+1)/2 - 1 \rightarrow O(n^2)$

- **Best-case time:**

If the array is already sorted, the inner loop is skipped, so time = O(n)

MergeSort with Linked Lists and Insertion Sort

The improved version uses:

- MergeSort for large subarrays
- Insertion Sort for subarrays smaller than 16
- Linked lists (using link[]) to avoid moving records

The function MergeSort1(low, high) is initially called with the array a[1:n] and a link[1:n] array set to zero.

It returns a pointer to the beginning of a linked list of indices that represents the array sorted in increasing order.

The Insertion Sort algorithm (Algorithm 3.9 in the book) is slightly modified to work on subarrays a[low: high] and update the link[] list instead of changing the array directly. This modified version is called InsertionSort1.

The merging of lists is handled by a revised function called Merge1, which links sorted sublists without copying records.

5.3 QUICK SORT

The divide-and-conquer approach can also be used to create another efficient sorting algorithm called Quick Sort, which is different from Merge Sort.

In Merge Sort, the array a[1:n] is divided at the middle into two parts, and each part is sorted independently. After that, the two sorted parts are merged together. But in Quick Sort, the array is divided into two parts in such a way that no merging is needed after sorting.

This is done by rearranging the elements in the array a[1:n] so that all elements from a[1] to a[m] are less than or equal to a certain element, and all elements from a[m+1] to a[n] are greater than or equal to it. In this way, the two sides can be sorted independently without merging them later.

This rearrangement is done by picking one of the elements in the array, let's call it t = a[s], and reordering the rest of the elements so that:

- All elements before t are less than or equal to it
- All elements after t are greater than or equal to it

This step is known as partitioning.

Partition Function

```
Algorithm Partition(a, m, p)
// Within a[m], a[m + 1], ..., a[p − 1] the elements are
// rearranged in such a manner that if initially t = a[m],
// then after completion a[q] = t for some q between m
// and p − 1, a[k] ≤ t for m ≤ k < q, and a[k] ≥ t
// for q < k < p. q is returned. Set a[p] = ∞.
{
    v := a[m]; i := m; j := p;
    repeat
    {
        repeat
            i := i + 1;
        until (a[i] ≥ v);

        repeat
            j := j − 1;
        until (a[j] ≤ v);

        if (i < j) then Interchange(a, i, j);

    } until (i ≥ j);

    a[m] := a[j]; a[j] := v; return j;
}

Algorithm Interchange(a, i, j)
// Exchange a[i] with a[j].
{
    p := a[i];
    a[i] := a[j]; a[j] := p;
}
```

The Partition function performs the rearrangement in-place—this means it does not use extra arrays. It works on the section of the array from a[m] to a[p–1] and assumes that a[m] is used as the pivot (the element to be placed in its correct position).

If m = 1 and p - 1 = n, then the element at a[p] (which is one past the end of the array) should be defined and must be greater than or equal to all elements in the array a[1: n]. This condition ensures proper working of the function. Although a[m] is used as the pivot here for simplicity, in actual practice, other choices (like a random element) are usually better.
The function Interchange(i, j) is used to swap the elements a[i] and a[j].

```
Algorithm QuickSort(p, q)
// Sorts the elements a[p], ..., a[q] which reside in the global
// array a[1 : n] into ascending order; a[n + 1] is considered to
// be defined and must be ≥ all the elements in a[1 : n].
{
    if (p < q) then  // If there are more than one element
    {
        // divide P into two subproblems.
            j := Partition(a, p, q + 1);
                // j is the position of the partitioning element.
        // Solve the subproblems.
            QuickSort(p, j − 1);
            QuickSort(j + 1, q);
        // There is no need for combining solutions.
    }
}
```

The QuickSort algorithm is used to sort elements in an array between two positions, p and q, in ascending order. It works by repeatedly dividing the array into smaller parts and sorting them independently. The array a[1:n] is the global array being sorted, and one extra element, a[n+1], is assumed to be a very large value (like infinity). This extra element helps avoid going out of bounds during the partitioning step.

First, the algorithm checks if there are more than one elements to sort by comparing p and q. If p is less than q, it means the subarray has at least two elements and sorting is required. If not, the function simply ends because no sorting is needed for a single element.

Next, the array is divided into two parts using the Partition function. This function rearranges the elements so that all smaller elements come before a chosen pivot element, and all larger elements come after it. The pivot is placed in its correct position, and that position is returned as j.

After partitioning, the QuickSort algorithm is called again on the two parts of the array: from position p to j-1 and from j+1 to q. These recursive calls continue until all subarrays are sorted. Importantly, there's no need to combine the results later, because partitioning already puts the pivot in its correct place.

Example 3.9 – How Partition Works

Let’s take an array of 9 elements and apply the partition function:
[65, 70, 75, 80, 85, 60, 55, 50, 45]
Here, a[1] = 65 is the pivot (partitioning element), and the function is called as Partition(a, 1, 10). The element at a[10] is assumed to be +∞ (a value greater than all others). The goal is to place 65 in its correct position, so that all smaller or equal values are before it and all larger values are after it.

Here’s how the array changes step-by-step:
1. **Initial:**
65 70 75 80 85 60 55 50 45 +∞

2. **Step 1 (Swap a[2] and a[9]):**
65 45 75 80 85 60 55 50 70 +∞
3. **Step 2 (Swap a[3] and a[8]):**
65 45 50 80 85 60 55 75 70 +∞
4. **Step 3 (Swap a[4] and a[7]):**
65 45 50 55 85 60 80 75 70 +∞
5. **Step 4 (Swap a[5] and a[6]):**
65 45 50 55 60 85 80 75 70 +∞
6. **Step 5 (Final step – Swap a[1] and a[5]):**
60 45 50 55 65 85 80 75 70 +∞

Here, 65 is now in its correct sorted position (5th smallest), and all elements have been partitioned around it. Note: the other elements are not sorted yet, but they are on the correct side of the pivot.

Applying Partition to Full Quick Sort

Using this method of partitioning, Quick Sort continues by sorting the two parts separately using more calls to the Partition function. This is how a full Quick Sort algorithm works.

Time Analysis of Quick Sort
Let's understand how much time Quick Sort takes.
Let C(n) be the number of comparisons made during the sorting of n elements. It can be assumed that:

- All n elements are different.
- The pivot (partition element) is equally likely to be any element in the array.

Worst-Case Analysis
Let Cw(n) be the worst-case number of comparisons.
In each call to Partition, the number of comparisons is at most p - m + 1.
At level 1 of recursion:

- 1 call to Partition(a, 1, n + 1) is made.
- Total elements compared: n
- At level 2:
- At most 2 calls to Partition are made.
- Total elements compared: n - 1

This continues, and at each new level, one less element is compared (since the pivot element is eliminated from the next level).
So the worst-case number of comparisons is:
Cw(n) = sum of values from 2 to n = $O(n^2)$
This shows that in the worst case, Quick Sort makes about n^2 comparisons.

Average-Case Analysis
Now consider the average number of comparisons, denoted $C_A(n)$.
If the pivot is equally likely to be any element in the subarray, then:
The two parts after partitioning will be:

- a[m:j] and a[j+1: p-1] for some $m < j < p$

From this, the average-case recurrence becomes:
$C_A(n) = n + 1 + (1/n) * \Sigma [C_A(k-1) + C_A(n-k)]$
where k = 1 to n
This means:

- First call to Partition makes n + 1 comparisons
- The remaining work is done on the two subarrays

Also:
$C_A(0) = C_A(1) = 0$
By solving this recurrence (details involve multiplying both sides, subtracting successive equations, and simplifying), it results in:
$C_A(n) < 2(n + 1)[\log_e(n + 2) – \log_e(2)] = O(n \log n)$
So, the average case performance is O(n log n), which is very good.

Stack Space Usage in Quick Sort
The recursive calls in Quick Sort use stack space. In the worst case, the recursion depth can go as deep as n – 1. This happens when the pivot always ends up being the smallest element in the array.

This means the algorithm could use O(n) stack space in the worst case.
However, this space can be reduced to O(log n) by improving the algorithm:

- Always sort the smaller of the two subarrays first.
- Replace the second recursive call with a simple jump (loop).

These changes lead to a more efficient version of Quick Sort (Algorithm 3.14 in the book).
Let S(n) be the maximum stack space needed. Then:
$S(n) < 2 + S((n – 1)/2) < 2 \log n$
This confirms the maximum stack space is only O(log n).

Using Insertion Sort with Quick Sort

Quick Sort is a very efficient sorting algorithm for large arrays. However, when the number of elements to sort becomes very small—typically less than 10 or 16—Quick Sort becomes less efficient. This is because its recursive calls and overhead (like function calls and partitioning) take more time than the actual sorting when the size is small.

On the other hand, Insertion Sort is a simple algorithm that works very well for small arrays. It goes through the array, picks one element at a time, and places it in the correct position in the already sorted part of the array. For very small arrays (like 5 to 15 elements), Insertion Sort is actually faster than Quick Sort, even though its worst-case time complexity is higher.
Because of this, many optimized versions of Quick Sort use a smart approach: When Quick Sort breaks the array into very small parts during recursion—say, when the subarray has fewer than 16 elements (i.e., when q - p < 16)—it stops using Quick Sort and switches to Insertion Sort to sort that small part. This change reduces unnecessary overhead and speeds up the final sorting.

So, the overall idea is to combine the speed of Quick Sort for large data with the simplicity and efficiency of Insertion Sort for small data. This hybrid method improves the real-world performance of Quick Sort, especially when dealing with large arrays that are broken into many smaller parts.

Randomized Quick Sort Algorithm

```
Algorithm RQuickSort(p, q)
// Sorts elements a[p] to a[q] in ascending order
{
   if (p < q) then
   {
      if ((q - p) > 5)
         // Randomly choose a pivot
         Interchange a[p] with a[Random() mod (q - p + 1) + p];
         // Partition the array around pivot
j := Partition(a, p, q + 1);
      // Recursively sort the two parts
RQuickSort(p, j - 1);
RQuickSort(j + 1, q);
   }
}
```

Here, a random pivot helps prevent the worst-case scenario (like when the array is already sorted).

Time Complexity of Quick Sort

To understand its performance, let A(n) be the average time taken to sort n elements using Randomized Quick Sort.

Since any position between 1 and n can become the pivot with equal chance, the recurrence relation for A(n) is:

$A(n) = (1/n) * \sum [A(k-1) + A(n-k)] + n + 1 \quad (\text{for } k = 1 \text{ to } n)$

This simplifies to:

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

So, on average, Quick Sort is just as efficient as Merge Sort. However, in the worst case (like sorted input with poor pivot selection), its time can degrade to:

$O(n^2)$

Using a random pivot helps avoid this scenario in most real-world cases.

Performance Comparison

The below table shows how Quick Sort and Randomized Quick Sort performed on a workstation for different values of n. For input arrays [1, 2, ..., n] (already sorted), the comparison was:

n	QuickSort Time (ms)	RQuickSort Time (ms)
1000	195.5	9.4
2000	759.2	21.0
3000	1728	30.5
4000	3165	41.6
5000	4829	52.8

These results show that Randomized Quick Sort performs much faster than standard Quick Sort, especially on inputs that are already sorted. The reason is that regular Quick Sort performs $O(n^2)$ comparisons in the worst case, while the randomized version avoids this behavior by choosing a pivot randomly.

How to Improve Quick Sort Further

The performance of Quick Sort can be improved in several ways. One technique is to pick a small number of elements (e.g., 11) randomly from the array, sort them, and use the median

as the pivot. This median is likely to be close to the actual middle of the full array and leads to better partitioning.

Another advanced method involves:

1. Picking a random sample of size s from the array.
2. Sorting the sample using Heap Sort or Merge Sort.
3. Using the sorted sample to divide the full array into s+1 parts.
4. Sorting each part separately using Quick Sort or another method.

This general idea ensures even partitioning, and the overall time remains close to:
O(n log n + O(n log n))
which is very efficient and close to the theoretical lower bound of sorting using comparisons.

5.5 SUMMARY

Sorting plays an important role in improving the efficiency of many algorithms and applications. This lesson covered two popular sorting algorithms that use the divide and conquer approach: Merge Sort and Quick Sort. Merge Sort offers consistent and predictable performance with a time complexity of O(n log n) and is especially effective for sorting linked lists and external files due to its merging process, though it requires extra space. Techniques such as linked list-based sorting and the use of insertion sort for small subarrays help reduce this space usage. Quick Sort is a fast in-place sorting algorithm that uses partitioning to rearrange elements. While it performs well on average, it may become inefficient if the pivot is poorly selected. Enhancements like random pivot selection, median-of-sample pivoting, and hybrid combinations with insertion sort address this issue. Together, these methods demonstrate the effectiveness of divide and conquer in designing high-performance algorithms and highlight the importance of recursion, memory management, and optimization strategies.

5.6 KEY TERMS

Divide and Conquer, Merge Sort, Quick Sort, Partitioning, Pivot, Insertion Sort, Randomized Quick Sort

5.7 Review Questions

1. What is the core principle of divide and conquer in sorting?
2. How does Merge Sort work, and why is it suitable for linked list sorting?
3. Explain the working of the Partition function in Quick Sort.
4. Why is Quick Sort generally faster in practice despite a worse worst-case time complexity than Merge Sort?
5. What role does Insertion Sort play in optimizing Merge Sort and Quick Sort?

5.8 SUGGESTIVE READINGS

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms *(3rd ed.). MIT Press.*
2. Horowitz, E., Sahni, S., & Rajasekaran, S. (2007). *Fundamentals of Computer Algorithms*. Universities Press.
3. Sedgewick, R., & Wayne, K. (2011). *Algorithms* (4th ed.). Addison-Wesley.
4. Aho, A. V., Hopcroft, J. E., & Ullman, J. D. (1983). Data Structures and Algorithms. Addison-Wesley.
5. Weiss, M. A. (2014). Data Structures and Algorithm Analysis *in C++* (4th ed.). Pearson.

Dr. U.Surya Kameswari

LESSON-6
GEOMETRIC ALGORITHMS OF DIVIDE AND CONQUER

OBJECTIVES

After completing this lesson, the learner will be able to:

- Understand the divide-and-conquer approach in matrix multiplication and geometric algorithms.
- Implement Strassen's matrix multiplication and analyze its efficiency.
- Explain and compare different convex hull algorithms like QuickHull, Graham's Scan, and the Divide-and-Conquer approach.
- Use geometric primitives such as orientation tests and determinants for computational geometry.
- Evaluate the time complexity and real-world application of geometric algorithms.

STRUCTURE

6.1 INTRODUCTION

Divide and conquer is a strategy that solves complex problems by breaking them into smaller subproblems, solving each one individually, and then combining their results. This method is commonly used in sorting algorithms, but it is also effective in solving geometric problems. This lesson focuses on how the divide-and-conquer approach can be applied to matrix multiplication and geometric computations, both of which are important in areas such as graphics, navigation systems, and robotics.

The lesson begins with Strassen's Matrix Multiplication, an improved method that reduces the number of multiplication steps in large matrix calculations. It then explains the convex hull problem**,** which involves finding the smallest convex shape that covers a set of points on a plane. Three main algorithms are discussed: QuickHull**,** Graham's Scan, and a divide-and-

conquer-based convex hull algorithm**.** These methods rely on simple geometric operations such as angle sorting and orientation tests. By using these techniques, geometric problems can be solved more efficiently and accurately.

6.2 STRASSEN'S MATRIX MULTIPLICATION

Matrix multiplication is a core operation in many algorithms. Given two $n \times n$ matrices A and B, the product matrix C = AB is also of size $n \times n$. Each element C(i,j) is calculated using the formula:

$C(i, j) = \Sigma A(i, k) \times B(k, j)$, for k = 1 to n

This traditional method requires:

n^3 multiplications, since each of the n^2 elements in matrix C requires n multiplications.

Time Complexity: $O(n^3)$

Divide-and-Conquer Matrix Multiplication

If n is a power of 2, the matrix A and B are divided into four equal submatrices of size $n/2 \times n/2$:

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

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

Then, matrix C = AB is computed as:

$C_{11} = A_{11} \times B_{11} + A_{12} \times B_{21}$
$C_{12} = A_{11} \times B_{12} + A_{12} \times B_{22}$
$C_{21} = A_{21} \times B_{11} + A_{22} \times B_{21}$
$C_{22} = A_{21} \times B_{12} + A_{22} \times B_{22}$

This method performs 8 recursive multiplications and some additions:

$T(n) = 8T(n/2) + c \cdot n^2$

Time Complexity: $O(n^3)$

Strassen's Optimization

Strassen proposed a more efficient way by reducing the number of multiplications from 8 to 7. He introduced the following seven new matrices:

$P = (A_{11} + A_{22}) \times (B_{11} + B_{22})$
$Q = (A_{21} + A_{22}) \times B_{11}$
$R = A_{11} \times (B_{12} - B_{22})$
$S = A_{22} \times (B_{21} - B_{11})$
$T = (A_{11} + A_{12}) \times B_{22}$
$U = (A_{21} - A_{11}) \times (B_{11} + B_{12})$
$V = (A_{12} - A_{22}) \times (B_{21} + B_{22})$

Now the final submatrices of C are computed as:

$C_{11} = P + S - T + V$
$C_{12} = R + T$
$C_{21} = Q + S$
$C_{22} = P + R - Q + U$

This gives a new recurrence:
$T(n) = 7T(n/2) + c \cdot n^2$
Time Complexity: $O(n^{\log_2 7}) \approx O(n^{2.81})$

Strassen's matrix multiplication significantly improves performance by reducing the number of multiplication steps, which are more computationally expensive than additions. This algorithm laid the foundation for future developments in fast matrix multiplication methods used in high-performance computing.

Table 6.1 Differences Between Standard and Strassen's Matrix Multiplication

Aspect	Conventional Matrix Multiplication	Strassen's Matrix Multiplication
Basic Idea	Multiply each element of rows with columns and sum up.	Uses a divide-and-conquer approach by breaking matrices into submatrices and combining results cleverly.
Time Complexity	$O(n^3)$	Approximately $O(n^{2.81})$
Number of Multiplications (for 2×2 matrices)	8	7
Number of Additions/Subtractions	4 (for 2×2)	18 (for 2×2)
Efficiency	Simple and straightforward; not optimized for large matrices.	More efficient for large matrices due to fewer multiplications.
Recursion	Not recursive; applies direct multiplication.	Recursive; divides matrices until reaching base size.
Memory Usage	Less memory, as no extra space is needed for submatrices.	Requires extra space for temporary matrices during recursion.
Implementation	Easier to implement and debug.	More complex and requires careful handling of matrix sizes.
Suitability	Better for small matrices or when simplicity is preferred.	Better for large matrices where performance is critical.
Numerical Stability	More stable; fewer rounding errors.	Slightly less stable due to increased additions/subtractions.

6.3 CONVEX HULL

The QuickHull algorithm is a popular method for computing the convex hull of a set of points in two-dimensional space. It is conceptually similar to the well-known Quicksort algorithm and employs a divide-and-conquer strategy to solve the problem efficiently.

Given a set of points in the plane, the convex hull is the smallest convex polygon that encloses all the points. The QuickHull algorithm finds this hull by recursively identifying the "extreme" points that form the outer boundary.
Key Ideas

1. Start with Extremes:
Identify the points with the minimum and maximum x-coordinates. These two points, say A and B, will definitely be part of the convex hull. Draw a line between them—this line divides the remaining points into two subsets:
   - Points that lie above the line AB
   - Points that lie below the line AB

2. Recursive Construction:
For each subset:
   - Find the point C that is farthest from the line AB. Point C, along with A and B, forms a triangle. All points inside this triangle cannot be part of the convex hull and are discarded.
   - The problem is then recursively reduced to finding convex hulls for the regions outside triangle ABC, i.e., on either side of lines AC and CB.
   - This process is repeated for each region until no points are left outside.

3. Base Case:
If no points are left on one side of a line segment, that segment is part of the convex hull.
Geometric Tools Used
QuickHull uses geometric primitives like:
- Orientation test (using determinants) to find which side of a line a point lies on.
- Distance from a point to a line to identify the farthest point.
- Collinearity checks to handle edge cases.
- Time Complexity
- The average case time complexity of QuickHull is O(n log n), similar to other efficient convex hull algorithms.
- The worst-case complexity is $O(n^2)$, which can occur when all points lie on the convex hull (e.g., when points form a circle).

6.3.1 Geometric Primitives

To determine the relative position of a point q with respect to a line formed by two other points p_1 and p_2, a special calculation called the determinant is used. This determinant helps tell whether q lies to the left or right of the line segment connecting p_1 and p_2:

- If the determinant is positive, then q is located to the left of the directed line from p_1 to p_2.
- If the determinant is negative, then q is to the right of the line.
- If the determinant is zero, all three points (p_1, p_2, and q) lie on a straight line—they are said to be colinear.

This concept is very useful in geometry. For example, if you want to find out whether a point p is inside a triangle formed by three other points (p_1, p_2, and p_3) arranged in clockwise order, this determinant method can help. Point p lies inside the triangle if it is to the right of each of the three edges:

1. Edge from p_1 to p_2
2. Edge from p_2 to p_3
3. Edge from p_3 to p_1

This way, using three right-turn checks (based on determinants), one can confirm if p is within the triangle.

In addition to orientation, the determinant can also be used to compute the signed area of the triangle formed by three points: (x_1, y_1), (x_2, y_2), and (x_3, y_3). The signed area is equal to half of the determinant's value. A positive value means the triangle is oriented counterclockwise, while a negative value means clockwise.

Now consider a more general problem. Suppose there is a convex polygon Q defined by n vertices $(p_1, p_2, ..., p_n)$ listed in clockwise order, and we are given another point p. The task is to determine whether p lies inside this polygon Q or outside it.

To do this, imagine a horizontal line h that extends infinitely in both directions (from $-\infty$ to ∞) and passes through the point p. There are two cases to consider:

1. The line h does not cross any edge of the polygon Q.
In this situation, it is clear that the point p lies outside the polygon.

2. The line h intersects some edges of Q.
In this case, count how many times line h intersects the edges of the polygon to the left of point p.
   - If h intersects a vertex (i.e., a corner point of the polygon), it is counted as two intersections (since it touches two edges).
   - After counting the total number of intersections to the left of p:
     - If the count is even, the point p lies outside the polygon.
     - If the count is odd, then p lies inside the polygon.

This entire check—whether p is inside or outside the convex polygon—can be performed in O(n) time, where n is the number of edges or vertices of the polygon.

6.3.2 The QuickHull Algorithm

The QuickHull algorithm is a geometric method used to find the convex hull of a set of points in a 2D plane. A convex hull is the smallest convex polygon that can enclose all the given points, like a rubber band stretched around them.

QuickHull is similar in concept to QuickSort. Just as QuickSort picks a pivot and partitions data around it, QuickHull picks two extreme points and recursively finds the outer boundary (convex hull) on either side.

Step-by-Step Explanation of QuickHull.

Step 1: Identify Extreme Points
Start with a set X containing n points in the 2D plane.

- Find the point p_1 with the smallest x-coordinate (the leftmost point).
- Find the point p_2 with the largest x-coordinate (the rightmost point).

These two points are guaranteed to be on the convex hull, because no other point can be further left or right. Think of them as the "anchors" of the hull.

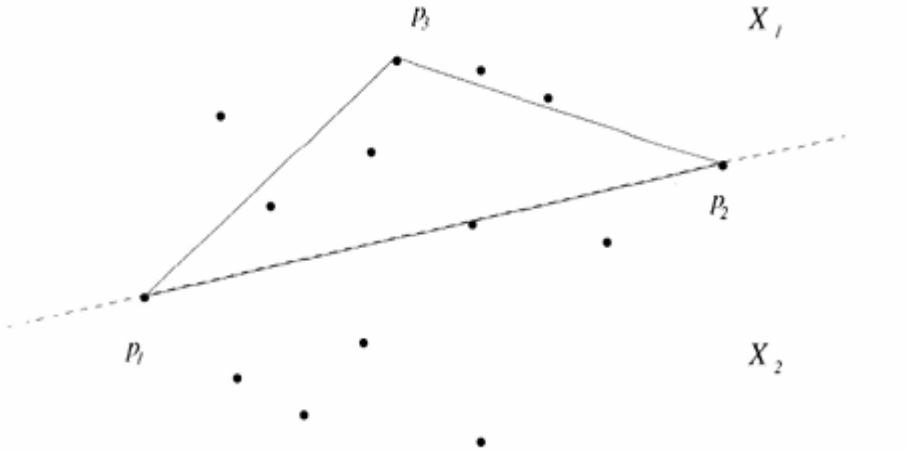


Figure 6.1 Identifying a point on the convexhull of Xi

Step 2: Divide the Point Set

Draw a straight line between p_1 and p_2. This line divides the rest of the points into two sets:

- X_1 (Upper Set): Points that lie above he line segment from p_1 to p_2.
- X_2 (Lower Set): Points that lie below the line segment from p_1 to p_2.
- Both X_1 and X_2 include p_1 and p_2, because they are endpoints of the dividing line and part of the final hull.

Step 3: Recursive Construction

Now the convex hulls of X_1 and X_2 are constructed separately using a recursive helper function (often named Hull):

Upper Hull Construction (from X_1):

- Find the point in X_1 that is farthest from the line segment (p_1, p_2). This point must lie on the convex hull.
- Form a triangle using this point and p_1 and p_2. All points inside this triangle can be discarded.
- Repeat the process on the remaining points on the left of the line (p_1 to farthest point) and the left of the line (farthest point to p_2).
- Lower Hull Construction (from X_2):
- The same logic is applied on the lower half (X_2), to find the lower convex boundary.

Step 4: Combine the Hulls

Finally, combine the points found in the upper hull and lower hull to form the complete convex hull that surrounds all the given points.

Summary of QuickHull

Step	Description
Find Extremes	Locate the leftmost and rightmost points (p_1 and p_2).
Divide Points	Separate remaining points into two sets based on their position to (p_1, p_2).
Recursive Hulls	Recursively find the farthest points from lines and remove interior ones.
Combine Results	Merge upper and lower hulls to form the final convex boundary.

Time Complexity

- Best/Average case: O(n log n)
- Worst case: O(n^2), especially if all points lie on the convex hull (e.g., all in a circle)

Real-World Use

QuickHull is efficient and often used in practical computational geometry applications such as:

- Image processing
- Pathfinding in 2D space
- Collision detection
- Boundary detection in GPS data or spatial datasets

6.3.3 Graham's Scan

Graham's Scan is an efficient algorithm used to find the convex hull of a given set of points in the 2D plane. The convex hull is the smallest convex polygon that can contain all the given points.

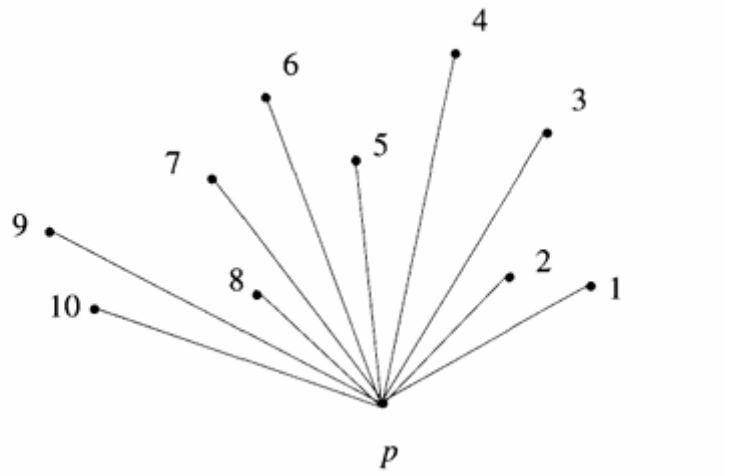


Figure 6.2 Graham's scan algorithm

In the above figure it can be seen that

- A central point labeled p, from which 10 points (labeled 1 to 10) radiate outward like spokes on a wheel.
- The points are ordered based on the angle they form with point p and the positive x-axis. This angular sorting is the first and most crucial step in Graham's Scan.

Step-by-Step Explanation of Graham's Scan:

1. Choose the Starting Point

First, the algorithm selects a starting point p from the set of points S. This is the point with the lowest y-coordinate.

- If multiple points have the same y-coordinate, the leftmost one (smallest x-coordinate) is chosen.

In the diagram, point p is this chosen point.

2. Sort All Other Points

Next, all other points are sorted based on the angle they make with point p and the positive x-axis.

- This sorting is already shown in the diagram: the points are numbered from 1 to 10 in counterclockwise angular order.
- For example, point 1 makes the smallest angle with the x-axis, while point 10 makes the largest.

This sorting helps ensure that when the points are connected in this order, the outermost boundary (the convex hull) can be constructed efficiently.

3. Begin Scanning and Constructing the Hull

The algorithm now scans through the sorted list, starting from point p and considers three points at a time:

- Let the three current points be p_1, p_2, and p_3.
- Initially, $p_1 = p$, p_2 = point 1, and p_3 = point 2 (as per the diagram).

4. Check for Left or Right Turn

The next step is to determine whether the three points p_1, p_2, and p_3 form a left turn or a right turn. This is done using a determinant test or cross product.

- Left Turn:
  If the angle from p_1 to p_2 to p_3 curves counterclockwise, then this is part of the convex hull. The algorithm keeps all three points and moves forward:

$p_1 \leftarrow p_2$, $p_2 \leftarrow p_3$, $p_3 \leftarrow$ next point

- Right Turn:
  If the angle turns clockwise, then point p_2 must be inside the hull, not on the boundary.
  This point is removed (deleted), and the algorithm steps one point backward:

$p_1 \leftarrow$ previous point of p_1

This check is repeated for every three consecutive points as the algorithm progresses through the sorted list.

5. Continue Until All Points Are Processed

This scanning and elimination process continues until:

- All points have been checked.
- The final point connects back to the starting point **p**, forming a closed polygon.

The remaining points form the **convex hull** in order.

How It Works in the Diagram

Let's assume the scan starts from p, and the order is:

$p \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow \ldots \rightarrow 10$

- The algorithm will check each triplet (p, 1, 2), (1, 2, 3), (2, 3, 4), and so on.

- At any point, if a right turn is detected, the middle point is removed, because it lies inside the potential convex shape.
- Eventually, only the outermost points (that form the convex boundary) will remain.

Time Complexity

- Sorting points by angle: **O(n log n)**
- Scanning and building the hull: **O(n)**

So the total time complexity is:**O(n log n)**

Step	Description
Select base point p	Lowest y-coordinate (leftmost if tie)
Sort other points	Based on angle with x-axis
Scan with 3-point check	Use left/right turn test to include or discard points
Build convex hull	Final remaining points after scan form the convex polygon boundary

6.3.4 Divide-and-Conquer Algorithm Convex Hull

In this section, a simple divide-and-conquer algorithm is presented, called DCHull, which also takes O(n log n) time and computes the convex hull in clockwise order.

Given a set of n points, like in the case of QuickHull, the problem is reduced to finding the upper hull and lower hull separately and then combining them. Since the computations for both are similar, the discussion is limited to computing the upper hull.

The divide-and-conquer algorithm partitions X into two nearly equal halves based on the x-coordinate values of points using the median x-coordinate as the splitter. Upper hulls are computed recursively for the two halves. These two hulls are then merged by finding the line of tangent (a straight line connecting one point from each of the two hulls such that all the points lie on one side of the line).

Let p_1 and p_2 be the points with the smallest and largest x-coordinate values. All the points to the left of the segment (p_1, p_2) are separated from those on the right. These are called the input set X.

Sorting the input points based on their x-coordinates takes O(N log N) time and is done once. The sorted list is divided into two halves. Upper hulls H_1 and H_2 are computed for each half. A tangent line is found in O(log N) time. Points on H_1 to the right of the tangent and points on H_2 to the left are discarded. The rest, along with the tangent, form the upper hull.

If T(N) is the run time of the recursive algorithm on N points, the recurrence relation is:

$$T(N) = 2T(N/2) + O(\log^2 N)$$

which solves to:

$$T(N) = O(N)$$

Thus, the total time is dominated by the initial sorting step, which is O(N log N)

6.4 SUMMARY

This lesson explored advanced divide-and-conquer applications in matrix operations and computational geometry. Strassen's matrix multiplication improves efficiency by reducing multiplications. Convex hull algorithms like QuickHull, Graham's Scan, and a pure divide-and-conquer method demonstrate how recursive thinking simplifies complex spatial problems. These techniques are vital in graphics, robotics, and geographic systems, showcasing how divide-and-conquer enables efficient, scalable solutions.

6.5 KEY TERMS

Strassen's Algorithm, Convex Hull, QuickHull, Graham's Scan, Determinant Test, Tangent Line

6.6 REVIEW QUESTIONS

1. What is the main advantage of Strassen's matrix multiplication over the standardmethod?
2. How does the QuickHull algorithm use divide and conquer?
3. What role does the determinant play in geometric algorithms?
4. Explain the process of Graham's Scan and its time complexity.
5. How does the divide-and-conquer convex hull algorithm achieve O(n log n) time complexity?

6.7 SUGGESTIVE READINGS

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms *(*3rd ed.). MIT Press.
2. de Berg, M., van Kreveld, M., Overmars, M., & Schwarzkopf, O. (2008). Computational Geometry: Algorithms and Applications (3rd ed.). Springer.
3. Preparata, F. P., &Shamos, M. I. (1985). Computational Geometry: An Introduction. Springer.
4. O'Rourke, J. (1998). Computational Geometry in C (2nd ed.). Cambridge University Press.

Dr. U. Surya Kameswari

LESSON-7

GREEDY METHOD: CORE CONCEPTS

OBJECTIVES

After completing this lesson, the learner will be able to:

- Understand the general approach and principles behind the greedy method.
- Recognize when greedy algorithms can produce optimal solutions.
- Apply greedy strategies to problems like the fractional knapsack, activity selection, Huffman coding, and graph-based algorithms.
- Differentiate between greedy, dynamic programming and brute force methods.
- Analyze and compare various greedy strategies.

STRUCTURE

7.1 INTRODUCTION

The greedy method is a widely used technique in algorithm design. It builds up a solution step-by-step by choosing the best available option at each stage without reconsidering previous decisions. This method works well for problems that exhibit the greedy-choice property and optimal substructure.

In greedy algorithms, the input is processed piece by piece. At each stage, a local optimal choice is made, aiming to find a global optimum. However, not all problems guarantee that a locally optimal choice will lead to the best overall result. For problems like the fractional knapsack or minimum spanning tree, the greedy method does work efficiently and provides an optimal result. This lesson explores the theory and application of greedy algorithms, focusing on real-world problems and multiple selection strategies.

7.2 GENERAL APPROACH OF GREEDY METHOD

The greedy method is one of the most intuitive and straightforward design techniques used in algorithm development. It is widely applicable across many types of problems, particularly those involving a set of inputs where the objective is to identify a subset that satisfies specific constraints.

- A feasible solution is any subset of inputs that meets the problem's constraints.
- An optimal solution is a feasible solution that either maximizes or minimizes a given objective function.

While it is often easy to determine a feasible solution, finding an optimal one may not be as obvious.

The greedy method proposes that an algorithm can be constructed in multiple stages, each processing one input at a time. During each stage, a decision is made about whether to include the current input in the partially built solution. This decision is made based on a selection procedure, which typically relies on some optimization measure — such as the value of the objective function.

If including the current input would result in an infeasible solution, then it is excluded. Otherwise, it is added to the solution.

Note: For many problems, there may be multiple plausible optimization criteria, but only some may lead to an optimal or near-optimal solution. In many cases, greedy algorithms result in suboptimal solutions, especially when the problem structure does not guarantee optimality through local decisions.

This approach is formally described using the subset paradigm, which fits problems where the goal is to choose an optimal subset of items. A control abstraction for such problems uses the following functions:

- Select: Selects and removes an input item from the list a[] and assigns it to variable x.
- Feasible(x): A Boolean function that returns true if including x keeps the solution feasible.
- Union(x): Combines x with the current solution and updates the objective function.

Once a specific problem is chosen, the functions Select, Feasible, and Union must be properly implemented to reflect the problem’s requirements.

7.3 APPLICATIONS OF THE GREEDY METHOD

The Greedy Method is a fundamental design technique used in algorithm development. It is widely appreciated for its simplicity and speed, especially when a problem allows making local optimal choices that lead to a global optimum. In the context of Design and Analysis of Algorithms, the greedy method has numerous applications across optimization problems.

Below is a detailed and structured explanation of the Applications of the Greedy Method:

1. Fractional Knapsack Problem

- Problem: Given a set of items, each with a weight and value, and a knapsack with a weight limit, select items (or fractions of them) to maximize total value.
- Greedy Strategy: Choose items based on the maximum value-to-weight ratio (pi/wi).
- Why Greedy Works: Because fractional items are allowed, always taking the "most profitable per unit weight" item ensures maximum value.
- Result: Yields optimal solution.

2. Activity Selection Problem

- Problem: Given n activities with start and finish times, select the maximum number of activities that can be performed by a single person without time conflicts.
- Greedy Strategy: Always pick the activity that finishes earliest (i.e., has the smallest finishing time).
- Why Greedy Works: Finishing early allows more activities to be scheduled later.
- Result: Yields optimal solution.

3. Job Sequencing with Deadlines

- Problem: Given n jobs with deadlines and profits, schedule the jobs to maximize profit. Only one job can be executed at a time and each takes a single time unit.
- Greedy Strategy: Sort jobs by profit in decreasing order and place each in the latest available slot before its deadline.
- Why Greedy Works: Greedily choosing high-profit jobs and placing them optimally gives near-optimal results.
- Result: Provides optimal or close-to-optimal solution in O(n log n) time.

4. Huffman Coding

- Problem: Given a set of characters with frequencies, construct a binary prefix code (Huffman Tree) to minimize the total encoding cost.
- Greedy Strategy: At each step, combine the two least frequent nodes.
- Why Greedy Works: Merging the least frequent symbols early minimizes their depth in the tree.
- Result: Guarantees optimal prefix code, widely used in file compression.

5. Prim's Algorithm for Minimum Spanning Tree (MST)

- Problem: Find a subset of edges that connect all vertices in a graph with the minimum total edge weight and no cycles.
- Greedy Strategy: Start from any node, and always add the smallest weight edge that connects a visited node to an unvisited one.
- Why Greedy Works: Greedily expanding the tree using the least-cost edge maintains the MST property.
- Result: Always finds the optimal MST.

6. Kruskal's Algorithm for MST

- Problem: Same as above, but works differently.
- Greedy Strategy: Sort all edges and add them in increasing order of weight, avoiding cycles using disjoint set union.
- Why Greedy Works: Edge-by-edge inclusion based on weight ensures minimal total cost without forming cycles.
- Result: Always returns an optimal MST.

7. Dijkstra's Algorithm for Single-Source Shortest Path

- Problem: Given a graph with non-negative edge weights, find the shortest path from a source node to all other nodes.
- Greedy Strategy: Repeatedly pick the nearest unvisited node and update paths.
- Why Greedy Works: Once a node's shortest path is found, it will never be improved.
- Result: Produces optimal shortest paths for non-negative weights.

8. Egyptian Fraction Representation

- Problem: Represent a given fraction as the sum of unique unit fractions (fractions with numerator 1).
- Greedy Strategy: Always subtract the largest possible unit fraction from the current fraction.
- Why Greedy Works: This leads to a quick, unique representation in many cases.
- Result: Yields a valid representation, though not always minimal.

9. Coin Change Problem (Limited to Specific Denominations)

- Problem: Find the minimum number of coins to make a given amount using given denominations.
- Greedy Strategy: Always choose the largest denomination coin less than the remaining amount.
- Why Greedy Sometimes Fails: Only works optimally for coin systems like U.S. currency (canonical systems), not all.
- Result: Works optimally only for specific denominations.

10. Optimal Merge Patterns

- Problem: Given n files with different sizes, combine them into one file using minimum total cost (where cost = sum of file sizes being merged).
- Greedy Strategy: Always merge the two smallest files.
- Why Greedy Works: Minimizes the accumulated cost due to smaller intermediate merges.
- Result: Produces minimum total merging cost.

Key Characteristics of Greedy Algorithms

- Local Optimization: Makes decisions based only on immediate information.
- No Backtracking: Once a decision is made, it is never changed.
- Efficiency: Generally faster than dynamic programming; often O(n log n) or O(n).
- Correctness: Only works when the problem exhibits the Greedy Choice Property and Optimal Substructure.

When Greedy Method Works Best

The problem must have:

- Greedy-choice property: A global optimum can be arrived at by choosing local optimums.

- Optimal substructure: An optimal solution to the problem contains optimal solutions to subproblems.

The greedy method is a powerful tool in algorithm design, especially when problems allow for locally optimal choices to lead to globally optimal solutions. It has widespread applications in resource allocation, network design, scheduling, and compression algorithms. However, it's important to analyze the problem's properties before applying a greedy solution, as not all problems are suited for this approach.

7.4 FRACTIONAL KNAPSACK PROBLEM

The Knapsack Problem is a classic optimization problem in computer science and operations research. It involves selecting a subset of items, each with an associated weight and profit, to place into a container (referred to as a knapsack) with a limited capacity. The goal is to maximize the total profit without exceeding the knapsack's weight limit.

In the Fractional Knapsack Problem, it is permissible to include fractions of items, rather than requiring the selection of entire items. This allows a more flexible and continuous decision-making process, where a portion of an item can be selected based on available capacity.

Why Use the Greedy Method for the Fractional Knapsack Problem?

The greedy method is particularly effective for solving the fractional knapsack problem because the problem satisfies two key properties:

1. Greedy-choice property:
A global optimum can be arrived at by making a series of locally optimal (greedy) choices. At each step, selecting the item (or fraction) with the highest profit-to-weight ratio (p_i/w_i) leads toward the best solution.

2. Optimal substructure:
The optimal solution to the problem contains optimal solutions to subproblems. Once a portion of the capacity is filled optimally, the remaining problem is of the same type with reduced capacity and a subset of items.

Due to these properties, the greedy approach — which selects items in order of decreasing profit-to-weight ratio — produces an optimal solution for the fractional version in an efficient manner, typically in O(n log n) time (due to sorting).

However, it is important to note that the greedy method does not guarantee an optimal solution for the 0/1 knapsack problem, where only whole items can be selected.
Application of the greedy method to the Knapsack Problem.

The Knapsack Problem involves a container (referred to as a knapsack) that has a maximum capacity m. There are n objects available, where each object i is associated with:

- A weight w_i
- A profit p_i

In the fractional knapsack variant, it is allowed to include fractions of objects in the knapsack. For each object i, a fraction x_i is selected such that:

$$0 \leq x_i \leq 1$$

The objective is to maximize the total profit earned from the selected items, while ensuring that the total weight does not exceed the knapsack's capacity m.

To achieve this, the greedy method is applied by selecting items based on their profit-to-weight ratio (p_i/w_i) in descending order. At each step, the item offering the highest profit per unit weight is chosen and included fully if possible. If the remaining capacity is insufficient, a suitable fraction of the item is included to exactly fill the knapsack.

This greedy approach guarantees an optimal solution for the fractional knapsack problem due to its greedy-choice property and optimal substructure.

However, there are different ways to decide which item to include next. Given:

- Number of items n=3,
- Knapsack capacity m=20,
- Profits (p1,p2,p3)=(25,24,15)
- Weights (w1,w2,w3)=(18,15,10)

Solution	**(x_1, x_2, x_3)**	**$\sum w_i x_i$**	**$\sum p_i x_i$**
1	(1/2, 1/3, 1/4)	16.5	24.25
2	(1, 2/15, 0)	20	28.2
3	(0, 2/3, 1)	20	31.0
4	(0, 1, 1/2)	20	31.5

Strategy 1: Fractional parts of objects

In Solution 1, the variable values are given as (x_1, x_2, x_3) = (1/2, 1/3, 1/4). These represent partial or fractional selections of three items or decision variables — a common setup in optimization or linear programming problems (for example, in the fractional knapsack problem).

- The term $\sum w_i x_i = 16.5$ represents the total weight contributed by these fractional selections. Each variable xix_ixi is multiplied by its corresponding weight wiw_iwi, and the results are summed up. The total of 16.5 indicates that the combination of chosen fractions uses 16.5 units of available capacity.
- The term $\sum p_i x_i = 24.25$ represents the total profit (or value) obtained by including these fractions. Each variable xix_ixi is multiplied by its profit pip_ipi, and the results are summed to yield the total benefit of 24.25.

Hence, Solution 1 indicates that by taking ½ of item 1, ⅓ of item 2, and ¼ of item 3, the system achieves a total weight of 16.5 and a total profit of 24.25.
This kind of solution typically arises in heuristic search or optimization-based AI systems, where the goal is to maximize profit or minimize cost while satisfying constraints (such as capacity or weight limits).

Strategy 2: Select the Object with the Highest Profit First

This strategy involves selecting items based on their total profit (pi) — in descending order.

- In the above problem, object 1 has the highest profit of 25, so it is added first to the knapsack. This uses up part of the knapsack's capacity, and we earn a profit of 25.
- After this, only 2 units of capacity remain in the knapsack.
- The next object, object 2, has a profit of 24, but it weighs 15 units, which is more than the remaining capacity. So, we take only a fraction of it.
- We include 2/15 of object 2, which exactly fills the knapsack and adds a profit of 3.2, leading to a total profit of 28.2.

Even though this strategy sounds reasonable (pick the most valuable item first), it gives a suboptimal solution in this case. That's why it is still called a greedy method, but it doesn't always give the best answer.

Also, just changing the last step to pick an object that gives the biggest profit possible with the remaining capacity does not guarantee an optimal solution.

Strategy 3: Select Objects with the Smallest Weights First

In this approach, we try to use the knapsack capacity slowly by choosing objects with smaller weights (wi) first.

- The idea is: by picking light items, we may be able to fit more objects in the knapsack and earn more total profit.
- However, in Example 4.1, this also results in a suboptimal solution. Although capacity is used more gradually, the profits increase too slowly, and we still don't reach the maximum possible profit.

So, again, this method doesn't work well in all situations.

Strategy 4: Choose Based on Profit per Unit Weight (pi/wi)

This is the best greedy approach for the fractional knapsack problem.

- Here, we calculate the profit-to-weight ratio for each item, i.e., pi/wi, and sort the items in non-increasing order of this ratio.
- We then pick items in that order:
  - Start with the item that gives the most profit per unit of weight.
  - Add it fully if possible; otherwise, add a fraction that fits.
    - This strategy tries to get the most profit for each unit of capacity used, thus achieving a balance between earning profit and using capacity.

This approach is used in the algorithm called Greedy Knapsack .
If the objects are already sorted by pi/wi, this method can be implemented very efficiently — in O(n) time (not counting the sorting step).

Summary of the Three Greedy Strategies

When solving the knapsack problem using greedy methods, there are three main criteria to decide which item to choose next:

1. Maximum total profit (p_i)
   - Simple but not optimal.
2. Minimum weight (w_i)
   - Uses capacity slowly but earns less profit.
3. Maximum profit per weight (p_i/w_i)
   - Best balance and yields the optimal solution for the fractional knapsack.

Once a strategy (optimization measure) is chosen, the greedy method proceeds step-by-step, making decisions based only on current choices and not revisiting past ones. For the fractional knapsack problem, Strategy 3 (p_i/w_i) is the correct and optimal greedy strategy.

Another Example

A container has a maximum weight capacity of 50 kilograms. There are three items available, each with an associated profit and weight. A fraction of any item can be included in the container if necessary.

The objective is to determine the combination of items (including fractions, if required) that maximizes the total profit, while ensuring the total weight does not exceed the container's capacity.

Item	Profit (₹)	Weight (kg)
1	60	10
2	100	20
3	120	30

Step-by-Step Greedy Approach

Step 1: Compute Profit-to-Weight Ratio

This ratio indicates the profit earned per unit weight of each item. It helps in determining which item offers the most value for the least weight.

$$\text{Profit-to-Weight Ratio} = \frac{\text{Profit}}{\text{Weight}}$$

Item	Profit	Weight	Profit/Weight Ratio
1	60	10	6.0
2	100	20	5.0
3	120	30	4.0

Step 2: Sort Items by Ratio (Descending Order)

Items are sorted in descending order of their profit-to-weight ratio:

1. Item 1 (ratio = 6.0)
2. Item 2 (ratio = 5.0)
3. Item 3 (ratio = 4.0)

Step 3: Select Items for the Container
The container has a remaining capacity of 50 kg.
Item 1

- Weight: 10 kg
- Entire item fits
- Remaining capacity = 50 − 10 = 40 kg
- Profit earned = 60

Item 2

- Weight: 20 kg
- Entire item fits
- Remaining capacity = 40 − 20 = 20 kg
- Profit earned = 100
- Cumulative profit = 60 + 100 = ₹160

Item 3

- Weight: 30 kg
- Only 20 kg of capacity is available
- A fraction of the item is selected:

Fraction taken= 20 /30 = 2/3

- Profit earned = (2/3) × 120 = 80
- Container is now fully filled
- Total profit = 160 + 80 = 240

Final Result

Maximum Profit=240
This example demonstrates the use of the greedy method for solving the fractional knapsack problem:

- Items are selected based on profit per unit weight.
- The greedy strategy of always choosing the item with the highest profit-to-weight ratio results in an optimal solution.
- This approach is efficient and suitable when fractions of items are allowed.

This method works optimally for the fractional version of the knapsack problem but not necessarily for the 0/1 knapsack problem, where only whole items can be selected.

7.5 SUMMARY

The greedy method is an efficient design technique that works by making a series of locally optimal choices. It is effective when problems meet specific conditions like the greedy-choice property and optimal substructure. Many well-known algorithms such as Prim's, Kruskal's, and Dijkstra's use greedy strategies to find optimal solutions.

The fractional knapsack problem serves as a key example where the greedy approach works perfectly. However, for problems like the 0/1 knapsack, greedy strategies may produce suboptimal results. Selecting the right optimization measure is crucial for ensuring solution accuracy.

7.6 KEY TERMS

Greedy Algorithm, Fractional Knapsack, Optimal Substructure, Greedy-Choice Property, Profit-to-Weight Ratio, Huffman Coding

7.7 REVIEW QUESTIONS

1. What is the basic idea behind the greedy method?
2. Explain why greedy strategy gives an optimal solution for the fractional knapsack problem.
3. Give an example where a greedy algorithm fails to produce an optimal result.
4. How do Prim's and Kruskal's algorithms apply the greedy approach?
5. What conditions must be satisfied for a greedy algorithm to work correctly?

7.8 SUGGESTIVE READINGS

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press.
2. Horowitz, E., Sahni, S., & Rajasekaran, S. (2007). *Fundamentals of Computer Algorithms*. Universities Press.
3. Kleinberg, J., & Tardos, É. (2006). *Algorithm Design*. Pearson.
4. Dasgupta, S., Papadimitriou, C. H., & Vazirani, U. V. (2008). *Algorithms*. McGraw-Hill.

Dr. U. Surya Kameswari

LESSON - 8
TREE AND SCHEDULING PROBLEMS

OBJECTIVES

After completing this lesson, the learner will be able to:

- Understand the types and characteristics of tree and scheduling problems in algorithm design.
- Explain the concept and application of the Tree Vertex Splitting Problem.
- Solve Tree Vertex Splitting using greedy methods.
- Analyze and implement Job Sequencing with deadlines to maximize profit.
- Compare greedy strategies used in tree and scheduling-based optimization problems.

STRUCTURE

8.1 INTRODUCTION

In the field of algorithm design, many real-world computational problems can be broadly categorized into two main types: tree-based problems and scheduling-based problems. These types of problems appear frequently across various domains such as computer networks, operating systems, compiler design, project management, and resource allocation. Solving these problems efficiently is crucial in both theoretical and practical applications.

A common characteristic of both tree and scheduling problems is the need to make decisions that optimize a particular objective, such as minimizing cost, maximizing profit, reducing time, or improving resource utilization. These decisions must often be made step by step, considering certain constraints or structural limitations. As the complexity of such problems increases, it becomes essential to adopt efficient and intelligent algorithmic techniques to find solutions that are not only correct but also optimal or near-optimal in terms of performance.

One of the most widely used and efficient techniques in such cases is the greedy method. The greedy method is a problem-solving paradigm that builds up a solution piece by piece, always choosing the option that offers the most immediate benefit at each stage. In simple terms, it makes a decision that appears to be the best at the moment, with the hope that this will lead to a globally optimal solution.

The greedy strategy is particularly powerful when the problem satisfies two important conditions:

1. Greedy-choice property: A global optimal solution can be constructed by making a series of local optimal choices.
2. Optimal substructure: An optimal solution to the problem contains optimal solutions to its subproblems.

When these properties are met, the greedy method can often produce efficient algorithms with significantly reduced time complexity compared to brute-force or exhaustive methods. Unlike methods such as dynamic programming, which may require re-evaluating earlier choices or storing intermediate results, greedy algorithms typically do not backtrack or revise past decisions. This makes them faster and easier to implement, although not always suitable for every problem.

In algorithmic problem-solving, recognizing when the greedy method is appropriate is a key skill. When applied correctly, it can provide elegant and optimal solutions to a wide variety of problems, especially those involving optimization under constraints.

This lesson explores the application of the greedy method in solving two important types of algorithmic problems: tree-based and scheduling-based. The greedy method is an approach that builds a solution incrementally by selecting the most advantageous option available at each step, without revisiting or altering previous choices. It is particularly effective for problems where locally optimal decisions lead to a globally optimal result.

To illustrate this strategy, the lesson will cover two representative problems:

1. Tree Vertex Splitting
2. Job Sequencing with Deadlines

These problems showcase how greedy techniques can be utilized to derive efficient and often optimal solutions in structured and time-constrained scenarios.

8.2 TREE VERTEX SPLITTING

In many applications, a tree data structure is used to represent a network, where edges are labeled with numerical values known as weights. A tree that has weights assigned to its edges is called a weighted tree.

A weighted tree can be used to model real-world distribution networks. For example, in an electrical distribution system or an oil pipeline network:

- Nodes in the tree represent receiving stations (places where electricity or oil is delivered).
- Edges represent the transmission lines (paths through which electricity or oil flows).

During the transmission process, it is common for some loss to occur. In the case of electrical networks, this may be a drop in voltage, and in oil pipelines, it could be a drop in pressure. These losses are significant because if the total loss along a path becomes too high, the end receiver may not receive the required quality or quantity of the resource.

To control these losses, boosters are used. Boosters are devices that help amplify or restore the transmission signal (voltage, pressure, etc.) back to an acceptable level. However, boosters cannot be placed just anywhere—they are only allowed to be placed at the nodes of the tree (not along the edges).

The Tree Vertex Splitting Problem (TVSP) arises in this context. Given a weighted tree representing the transmission network, and a maximum allowable loss or tolerance level, the goal is to determine the optimal placement of boosters so that no path in the network exceeds the allowed loss.

Formally, the Tree Vertex Splitting Problem can be defined as follows:

Let
T=(V,E,w)be a weighted, directed tree, where:

- V is the set of vertices (nodes)
- E is the set of edges
- w is the weight function that assigns a weight to each edge

Specifically, for an edge between node i and node j, the weight is denoted as:w (i,j)
This weight represents the loss incurred when transmitting from node i to node j.

It is important to note that:

- The weight w(i, j) is defined only if (i, j) is an edge in the tree.
- If (i', j) is not an edge, then w(i', j) is undefined.

A source vertex in the tree is a node with no incoming edges (in-degree zero), and a sink vertex is a node with no outgoing edges (out-degree zero).

For any path P from a source to a sink, the delay d(P) is calculated as the sum of the weights of all edges on that path. The delay of the entire tree T, denoted d(T), is the maximum delay among all source-to-sink paths in the tree.

The main objective of the Tree Vertex Splitting Problem is to identify a booster placement strategy that ensures no path in the tree has a delay greater than the allowed tolerance level, while also minimizing the number of booster placements or satisfying other cost-related constraints.

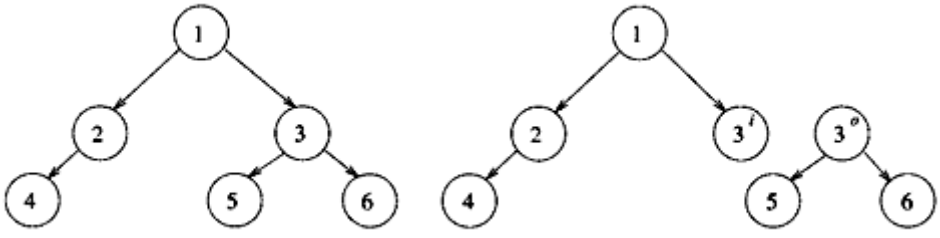


Figure 8.1. A tree before and after splitting the node 3

Figure 8.1 demonstrates the concept of vertex splitting on a small tree structure.

- On the left side, node 1 is the root, and it connects to nodes 2 and 3. Node 3 has two children: nodes 5 and 6.
- On the right side, the result of splitting node 3 is shown:

- Node 3 has been split into 3′ and 3″.
- Node 3′ retains the connection with node 1 and connects to node 5.
- A new node, 3″, is introduced to handle node 6, and it is made a child of 3′.

This transformation helps reduce the degree of node 3 and potentially improves path delay control in scenarios like network transmission.

In Figure 8.1, a tree is shown before and after breaking node 3 into two parts. Originally, node 3 was connected to two other nodes, 5 and 6. After splitting, node 3′ handles one connection (5), and passes the rest to a new helper node 3″, which connects to 6. This makes the tree more manageable and efficient.

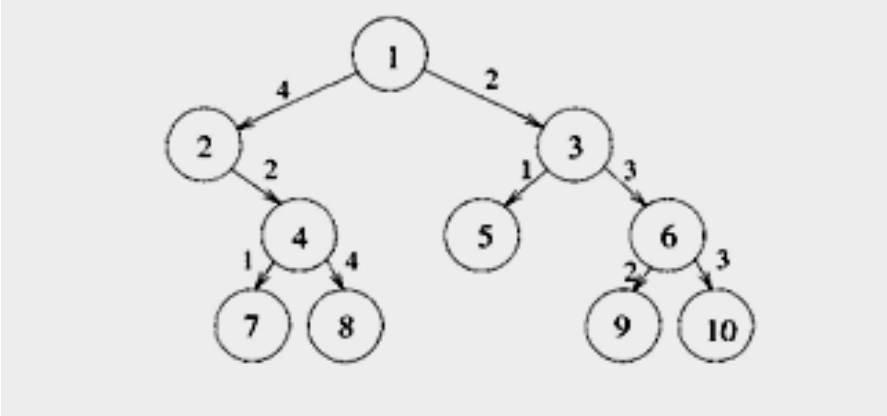


Figure 8.2. An example tree

Figure 8.2 presents a larger weighted tree structure used to illustrate the Tree Vertex Splitting Problem (TVSP).

- Node 1 is the root, and all edges have weights indicating **losses** (e.g., (1, 2) has weight 4, (3, 6) has weight 3).
- Nodes such as 2, 3, 4, and 6 are intermediate nodes, each with multiple children.
- Leaf nodes include 5, 7, 8, 9, and 10.
- The edge weights represent delays or transmission losses along the paths from root to leaves.
- Certain paths like $1 \rightarrow 2 \rightarrow 4 \rightarrow 7$ and $1 \rightarrow 3 \rightarrow 6 \rightarrow 10$ may exceed a maximum loss tolerance, depending on a given limit.

This tree acts as the input network for identifying which nodes need to be split to keep losses under control.

In Figure 8.3, a more complex tree is shown where each line between the nodes has a number representing how much loss or delay happens during transmission. Some nodes have too many branches, and some paths are too long in terms of total loss. This structure needs improvement to avoid crossing the loss limit.

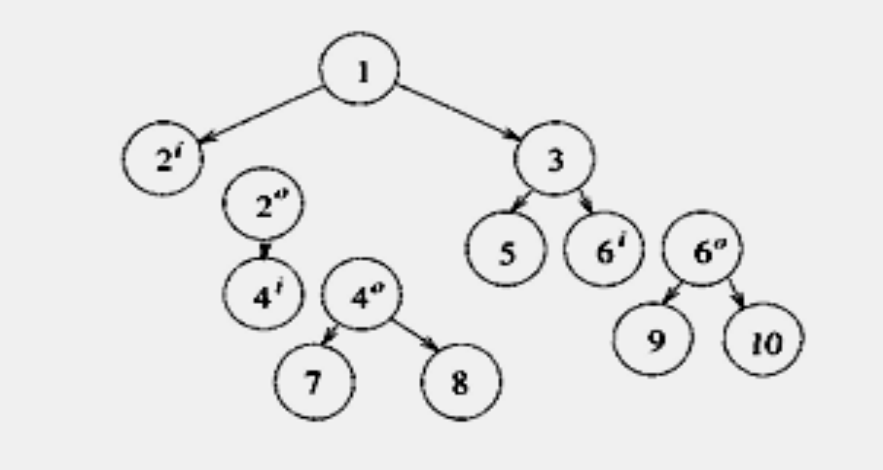


Figure.8.3The final tree after splitting the nodes2,4, and 6

The above figure shows the resulting tree after applying the greedy strategy to solve the Tree Vertex Splitting Problem.

- Nodes 2, 4, and 6 are the ones identified for splitting because they either have:
  - Too many child nodes (high out-degree), or
  - Appear in paths where the delay exceeds the allowable threshold.
- Each split creates:
  - 2′ and 2″ in place of node 2
  - 4′ and 4″ in place of node 4
  - 6′ and 6″ in place of node 6

- These new nodes reorganize the children to spread the load and reduce the maximum delay in the tree.

This transformation leads to a more balanced structure with shorter or controlled path delays, meeting the loss tolerance requirement.

Figure 8.3 shows how the original tree from Figure 8.2 looks after fixing the problems by splitting certain overloaded nodes. Nodes 2, 4, and 6 were doing too much work or causing long delays, so they were each split into two parts. This makes the tree better organized and ensures that no path has too much loss.

Algorithm for Tree Vertex Splitting

```
tvs(T) // T is the current node
{
   if (T is a leaf) then
      d[T] := 0;  // Set delay of leaf node to 0
   else
   {
      for each child v of T do
      {
tvs(v);  // Recursively call tvs for child v
         d[T] := max{ d[T], d[v] + w(T, v) };
      }
   }
```

```
    if ((T is not the root) and (d[T] + w(parent(T), T) > 6)) then
    {
       write(T);     // Output the node T to be split
       d[T] := 0;     // Reset delay after placing booster
    }
}
```

The algorithm is designed to solve the Tree Vertex Splitting Problem (TVSP) using a greedy approach. It determines which nodes need to be split (or where boosters need to be placed) so that no path from the root to any leaf exceeds a delay (loss) tolerance of 6.

Terminology:

- T: Current node being processed.
- v: A child of node T.
- d[T]: Maximum delay from the current node T down to its farthest leaf.
- w(T, v): Weight (delay/loss) on the edge from node T to its child v.
- parent(T): The parent of node T.

Working Step-by-Step:

1. Base Case (Leaf Node):
   - If T is a leaf node (no children), then the delay d[T] is set to 0 because there is no path beyond it.
2. Recursive Case (Internal Node):
   - For each child v of the current node T:
     - Recursively call tvs(v) to calculate the delay of the child.
     - Update d[T] as the maximum of its current value and d[v] + w(T, v). This calculates the maximum delay from T to any of its leaves via v.
3. Check for Splitting (Booster Placement):
   - After all child delays are computed, the algorithm checks if:
     - Node T is not the root, and
     - The delay from its parent to this node plus the current delay d[T] exceeds 6.
   - If so, it prints the node T (i.e., marks it for splitting or booster placement) and resets d[T] to 0.
     - This reset means the delay counting restarts from here because a booster (split) was placed.

The greedy part comes in how the algorithm:

- Processes children before parents (post-order traversal),
- Places a booster (split) only when absolutely needed (when delay > 6),
- Minimizes the number of boosters by splitting only those nodes which cause the delay to exceed the threshold.

This algorithm ensures that no path in the tree will have a delay greater than 6. It works efficiently using recursion and local greedy decisions to solve the Tree Vertex Splitting Problem.

Algorithm of TVS for the specialcaseof binary trees

```
Algorithm TVS(i, δ)
// Determine and output a minimum cardinality split set.
// The tree is realized using the sequential representation.
// Root is at tree[1]. N is the largest number such that
// tree[N] has a tree node.
{
    if (tree[i] ≠ 0) then // If the tree is not empty
        if (2i > N) then d[i] := 0; // i is a leaf.
        else
        {
            TVS(2i, δ);
            d[i] := max(d[i], d[2i] + weight[2i]);
            if (2i + 1 ≤ N) then
            {
                TVS(2i + 1, δ);
                d[i] := max(d[i], d[2i + 1] + weight[2i + 1]);
            }
        }
        if ((tree[i] ≠ 1) and (d[i] + weight[i] > δ)) then
        {
            write (tree[i]); d[i] := 0;
        }
}
```

This algorithm solves the Tree Vertex Splitting Problem (TVSP) for a binary tree stored in a sequential array (like in heaps). The goal is to determine the minimum number of nodes to split (or place boosters on) so that no root-to-leaf path has a total delay (sum of edge weights) greater than a given threshold (e.g., 5).

Terminology and Structure:

- tree[i]: Represents the value at the i-th position in the tree array.
- N: The last index in the array (total number of nodes).
- d[i]: The maximum delay from node i to its farthest descendant.
- weight[i]: The weight (or delay/loss) of the edge from the parent of i to node i.
- i: Current node index in the tree array.

Step-by-Step Working:
Step 1: Base Case — Leaf Node

- If 2 * i> N, node i is a leaf node (has no children).
- The delay d[i] is set to 0 because there is no path going downward.

Step 2: Recursive Case — Internal Node

- If node i has a left child (at index 2i), recursively compute its delay.
- Update d[i] as the maximum of:
    - Its current delay, and
    - The delay of the left child + the weight of the edge to it.
    - If node i also has a right child (at index 2i + 1), repeat the same steps.

This process ensures that d[i] holds the maximum delay to any leaf in the subtree rooted at node i.

Step 3: Decision to Split

- After calculating the delay at node i, check if:
    - The node is not the root ($i \neq 1$)
    - The total delay from the parent to this node (d[i] + weight[i]) exceeds the allowed tolerance (in this case, 5)
- If both conditions are true:
    - Output node i as one that needs to be split or boosted.
    - Reset d[i] to 0 to indicate a fresh start in delay counting from that node onward.

Why This Algorithm Works (Greedy Strategy):

- The tree is processed in post-order traversal (children before parent).
- At each step, a local greedy decision is made:
    - Only split a node when the delay becomes unacceptable.
    - This minimizes the number of splits and ensures that the constraint is satisfied.

This algorithm efficiently finds the minimum cardinality split set in a binary tree with sequential representation. It ensures that no path delay exceeds the given limit, making it suitable for optimizing delay-sensitive tree structures like distribution networks or hierarchical routing systems.

8.3 JOB SEQUENCING WITH DEADLINES

A set of n jobs is given, where each job i has an associated deadline d_i (a positive integer) and a profit p_i (also a positive integer). The profit p_i is earned only if the job is completed on or before its deadline.

Each job takes exactly one unit of time to complete, and there is only one machine available to process the jobs. This means only one job can be worked on at any given time.
A feasible solution is defined as a subset of jobs, say J, where every job in J can be completed by its respective deadline. The total value of this feasible solution is the sum of the profits of all jobs in the subset, represented as:

$$\sum_{i \in J} p_i$$

An optimal solution is a feasible subset of jobs that gives the maximum possible total profit.
Since this problem is about choosing an optimal subset from a given set based on constraints (deadlines), it is categorized under the subset paradigm, which is often addressed using greedy methods.

Example : Job Sequencing with Deadlines – Step-by-Step Explanation
Step 1: Problem Definition
Given:

Number of jobs (n) = 4

- Profits: $(p_1, p_2, p_3, p_4) = (100, 10, 15, 27)$
- Deadlines: $(d_1, d_2, d_3, d_4) = (2, 1, 2, 1)$

Each job takes 1 unit of time to complete
Only one job can be scheduled at a time
Objective: Maximize total profit by scheduling jobs before their deadlines.
This problem is solved using a greedy method:

1. Sort all jobs in decreasing order of profit.
2. Try to assign each job to the latest possible available time slot ≤ its deadline.
3. If the slot is already taken, move backward and try earlier slots.
4. If no slot is available, skip the job.

Sorted Job List (by Profit):

Order	Job	Profit	Deadline
1	1	100	2
2	4	27	1
3	3	15	2
4	2	10	1

Available Slots:
Since the **maximum deadline is 2**, there are **2 slots**:

- Slot 1 → Time 1
- Slot 2 → Time 2

Step-by-Step Job Assignment:
Try Job 1 (Profit: 100, Deadline: 2)

- Try slot 2 (latest ≤ 2) → **Available** → Assign Job 1 to Slot 2

Schedule: Slot 2 = Job 1
Try Job 4 (Profit: 27, Deadline: 1)

- Try slot 1 → **Available** → Assign Job 4 to Slot 1

Schedule: Slot 1 = Job 4, Slot 2 = Job 1
Try Job 3 (Profit: 15, Deadline: 2)

- Try slot 2 → Taken
- Try slot 1 → Taken

No slot available → **Skip**
Try Job 2 (Profit: 10, Deadline: 1)

- Try slot 1 → Taken

No slot available → **Skip**

Step 2: List of Feasible Solutions and Their Profits

Feasible Solution (Jobs)	Processing Sequence	Total Profit
(1, 2)	2, 1	110
(1, 3)	1, 3 or 3, 1	115
(1, 4)	4, 1	127
(2, 3)	2, 3	25
(3, 4)	4, 3	42
(1)	1	100
(2)	2	10

(3)	3	15
(4)	4	27

Step 3: Conclusion

Among all the feasible solutions listed above, the solution (1, 4) with processing order (4, 1) gives the maximum total profit of 127. This is the optimal solution.

Greedyalgorithmfor sequencingunit timejobswith deadlines and profits

```
Algorithm JS(d, j, n)
// d[i] ≥ 1, 1 ≤ i ≤ n are the deadlines, n ≥ 1. The jobs
// are ordered such that p[1] ≥ p[2] ≥ ··· ≥ p[n]. J[i]
// is the ith job in the optimal solution, 1 ≤ i ≤ k.
// Also, at termination d[J[i]] ≤ d[J[i + 1]], 1 ≤ i < k.
{
    d[0] := J[0] := 0; // Initialize.
    J[1] := 1; // Include job 1.
    k := 1;
    for i := 2 to n do
    {
        // Consider jobs in nonincreasing order of p[i]. Find
        // position for i and check feasibility of insertion.
        r := k;
        while ((d[J[r]] > d[i]) and (d[J[r]] ≠ r)) do r := r − 1;
        if ((d[J[r]] ≤ d[i]) and (d[i] > r)) then
        {
            // Insert i into J[ ].
            for q := k to (r + 1) step −1 do J[q + 1] := J[q];
            J[r + 1] := i; k := k + 1;
        }
    }
    return k;
}
```

The algorithm JS(d, j, n) addresses the problem of scheduling a set of n jobs, where each job has an associated deadline and profit. The goal is to determine a subset of jobs that can be completed within their deadlines while maximizing the total profit. Each job takes exactly one unit of time to execute, and only one machine is available for processing jobs.

The jobs are assumed to be pre-sorted in non-increasing order of their profits. That is, the jobs are arranged such that p[1] ≥ p[2] ≥ ... ≥ p[n], where p[i] is the profit of job i. The algorithm selects jobs for inclusion in the schedule based on this ordering and ensures that each job is scheduled on or before its deadline.

At the beginning of the algorithm, two arrays are initialized. The array d stores the deadlines of the jobs, while the array J will be used to store the sequence of selected jobs that form the optimal solution. The first job (with the highest profit) is included in the schedule by default. That is, J[1] := 1, and the schedule size counter k is initialized to 1.

The algorithm then proceeds to iterate through the remaining jobs from index 2 to n. For each job i, it attempts to determine the best possible position in the current schedule where job i can be inserted without violating its deadline or displacing other jobs beyond their deadlines. To find a suitable insertion position for job i, the algorithm sets r := k, representing the current last position in the schedule. It then moves backward through the array to find the position r such that the deadline of job at J[r] is not greater than d[i], and the deadline of job i is greater than r. This search is done using a while loop:

while ((d[J[r]] > d[i]) and (d[J[r]] ≠ r)) do r:= r - 1;

If the condition d[J[r]] ≤ d[i] and d[i] > r is satisfied, job i can be inserted at position r + 1. In order to insert the job at this position, all jobs currently occupying positions from r + 1 to k are shifted one position to the right to make space. This is done using a for loop:

for q:= k to (r + 1) step -1 do J [q + 1]:= J[q];

After shifting, job i is inserted at J[r + 1] and the schedule size counter k is incremented by 1. This process ensures that the schedule remains feasible and maintains the constraint that all scheduled jobs meet their deadlines.

At the end of the algorithm, the value k is returned, which represents the total number of jobs selected in the optimal solution. The job sequence J[1] to J[k] contains the indices of the scheduled jobs in the order they should be processed.

This algorithm maintains feasibility by checking deadline constraints before inserting each job and optimizes for profit by always considering jobs in order of highest profit first. It follows the greedy strategy of making a locally optimal choice at each step, which leads to a globally optimal solution for this specific problem structure.

Fast Job Scheduling

Fast Job Scheduling is an improved or optimized approach to solving the Job Sequencing with Deadlines problem, particularly when the number of jobs is large, and the naive greedy method becomes inefficient. It builds on the foundation of the greedy strategy but focuses on reducing the time complexity of scheduling decisions.

Context with Respect to the Algorithm JS(d, j, n)

The algorithm JS(d, j, n) discussed earlier attempts to insert each job into the schedule by checking feasible time slots in reverse order, and when insertion is allowed, it shifts all the affected jobs. While this method correctly solves the problem, it may become computationally expensive in worst-case scenarios, particularly because of repeated shifting operations and linear scanning for each job.

The time complexity of the basic greedy algorithm is approximately $O(n^2)$ in the worst case:

- For every job, the algorithm may need to scan and shift up to n elements.
- This becomes a performance bottleneck when n is large.

Fast Job Scheduling – Definition and Concept
Fast Job Scheduling refers to an optimized method for solving the job sequencing problem where:

- The objective remains the same: maximize total profit while meeting job deadlines.
- However, it employs efficient data structures such as the Disjoint Set Union (DSU) or Union-Find to handle slot allocation quickly.

Instead of scanning available time slots linearly, Fast Job Scheduling uses DSU to:

- Efficiently find the latest available slot ≤ deadline for each job in near constant time (amortized).
- Avoid unnecessary shifting of jobs by directly tracking free slots.

How Fast Job Scheduling Works

1. Sort the jobs in descending order of profit.
2. Initialize a Disjoint Set data structure to represent available time slots.
3. For each job i, use DSU to find the latest available slot ≤ d[i].
4. If a slot is found:
    - Assign the job to that slot.
    - Mark the slot as filled by updating the parent in the disjoint set.

The find and union operations in DSU can be done in $O(\alpha(n))$ time per operation, where $\alpha(n)$ is the inverse Ackermann function, which grows extremely slowly and is practically constant for all realistic input sizes.

Thus, Fast Job Scheduling reduces the overall time complexity from $O(n^2)$ to $O(n \log n)$ due to:

1. Sorting step ($O(n \log n)$)
2. Slot allocation using DSU ($O(n\, \alpha(n)) \approx O(n)$)
3. Benefits of Fast Job Scheduling
4. Efficient for large datasets.
5. Avoids unnecessary array shifts.
6. Provides optimal solutions in significantly less time.
7. Maintains the greedy-choice and optimal substructure properties.

Fast Job Scheduling is an enhancement of the classic greedy algorithm for job sequencing. It replaces linear slot-search and shifting operations with faster slot-allocation techniques using disjoint sets, resulting in efficient scheduling with minimal time overhead. This approach is especially important when job counts are high or when job scheduling is a part of real-time or high-performance systems.

8.4 SUMMARY

Tree and scheduling problems require optimal decision-making under constraints. Greedy methods provide a practical approach when local decisions lead to globally optimal outcomes. In Tree Vertex Splitting, greedy algorithms reduce delay by splitting overloaded nodes. In Job Sequencing with Deadlines, sorting by profit and assigning jobs strategically results in profit maximization. Efficient techniques like DSU enhance performance further.

8.5 Key Terms

Weighted Tree, Booster, Delay Constraint, Binary Tree Array, Profit Scheduling, DSU, Optimal Substructure

8.6 Review Questions

1. What is the goal of the Tree Vertex Splitting Problem?
2. Explain how the greedy method is used to place boosters in a transmission tree.
3. What are the two properties required for greedy algorithms to work effectively?
4. How does the job sequencing algorithm ensure deadlines are met?
5. Describe the purpose of using Disjoint Set Union in Fast Job Scheduling.

8.7 SUGGESTIVE READINGS

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press.
2. Horowitz, E., Sahni, S., & Rajasekaran, S. (2007). Fundamentals of Computer Algorithms. Universities Press.
3. Kleinberg, J., & Tardos, É. (2006). Algorithm Design. Pearson.
4. Dasgupta, S., Papadimitriou, C. H., & Vazirani, U. V. (2008). Algorithms. McGraw-Hill.

Dr. U. Surya Kameswari

LESSON-9

STORAGE AND MERGE OPTIMIZATION

OBJECTIVES

After completing this lesson, the learner will be able to:

- Understand the importance of optimizing storage and data processing.
- Apply greedy strategies to solve Optimal Storage on Tapes and Optimal Merge Patterns.
- Calculate Mean Retrieval Time (MRT) and Total Retrieval Time (TD).
- Design binary merge trees to minimize record moves.
- Explain the working and efficiency of Huffman coding.

STRUCTURE

9.1 INTRODUCTION

In the field of algorithm design and analysis, a significant category of problems revolves around the optimization of data organization, access, and processing. These problems are vital because, in practical computing systems, the efficiency with which data is handled can directly impact the overall performance, resource utilization, and responsiveness of software and hardware components. Among such optimization problems, two particularly important and classical examples are the Optimal Storage on Tapes and Optimal Merge Patterns problems.

Both of these problems deal with the central challenge of determining an optimal ordering or sequencing of operations. Specifically, the objective is to minimize the total cumulative cost—whether that cost is in terms of access time, processing time, or computation cost—by making intelligent decisions about how data should be arranged or combined. These problems do not focus on modifying the data itself but rather on determining the most efficient way to handle or structure the data operations.

In many real-world applications, the cost of sequential access or processing is not uniform, and accessing or processing certain data earlier or later can have a large impact on total system performance. For example, when data is stored on a medium like a magnetic tape,

which allows only sequential access, the position of each file on the tape affects how long it takes to retrieve it. Files that are accessed more frequently should ideally be placed closer to the beginning of the tape to minimize average retrieval time. This leads to the problem known as Optimal Storage on Tapes. Though traditional magnetic tapes are less commonly used today, the same principle applies to any sequential-access system or even modern hierarchical storage systems where read latency can vary.

Similarly, in Optimal Merge Patterns, the problem arises when multiple sorted sequences (such as files or data segments) need to be combined into a single sorted sequence. Each merge operation incurs a cost proportional to the size of the sequences being merged. If large sequences are merged early, they create even larger sequences, which, when merged later, lead to even higher costs. Hence, the order of merge operations becomes crucial. The goal is to determine a merge sequence that leads to minimum total merging cost. This problem is highly relevant in scenarios such as external sorting algorithms, Huffman coding in data compression, and compilers where syntax trees or instruction streams are combined.

The importance of these problems lies in their practical relevance and their demonstration of fundamental algorithmic principles, particularly those associated with greedy strategies. Both problems show how choosing locally optimal steps—such as storing the most frequently accessed file first, or always merging the two smallest sequences—can often lead to globally optimal solutions. They also highlight how problem constraints and structure influence the design of efficient algorithms.

In summary, the problems of Optimal Storage on Tapes and Optimal Merge Patterns illustrate the broader class of optimization problems where the goal is not to compute a value, but to determine the best possible structure or order of operations to minimize an accumulated cost. They continue to serve as fundamental examples in algorithm courses due to their conceptual clarity, ease of implementation, and deep relevance to practical computing systems.

9.2 OPTIMAL STORAGE ON TAPES

In many computing systems, especially older ones, data such as programs or files were stored on tapes. A tape is a sequential access device, meaning to access a program stored at the end of the tape, the system must first pass through all the programs before it. Because of this, the time it takes to access a program depends on its position on the tape. If a program is stored near the beginning, it can be accessed quickly. If it is stored near the end, access will take longer.

Now, imagine there are multiple programs, each with a certain length (size). The objective is to store these programs on a tape in an order that minimizes the average time taken to access any program. This average is referred to as the Mean Retrieval Time (MRT).

If all programs are accessed equally often, then the MRT is calculated based on how far each program is from the start of the tape, considering the total size of programs that come before it.

This problem is known as the Optimal Storage on Tape problem. It requires finding the best order to store the programs so that the total retrieval time is minimized.

Consider the 3 programs with lengths:

- Program 1 = 5
- Program 2 = 10
- Program 3 = 3

There are 3! = 6 possible ways to arrange these programs on the tape. For each arrangement, we calculate the total retrieval time (also called d(I)). This value is the sum of the times needed to retrieve each program in that order.
For instance:

Ordering: 1, 2, 3

- Time to access Program 1 = 5
- Time to access Program 2 = 5 (Program 1) + 10 = 15
- Time to access Program 3 = 5 + 10 + 3 = 18
- Total retrieval time = 5 + 15 + 18 = 38

Ordering: 3, 1, 2

- Time to access Program 3 = 3
- Time to access Program 1 = 3 + 5 = 8
- Time to access Program 2 = 3 + 5 + 10 = 18
- Total retrieval time = 3 + 8 + 18 = 29

The ordering that gives the smallest total retrieval time is 3, 1, 2, with a total of 29. This is the optimal arrangement.

The possible feasible solutions to the problem are listed below.

ordering I	$d(I)$		
$1,2,3$	$5+5+10+5+10+3$	$=$	38
$1,3,2$	$5+5+3+5+3+10$	$=$	31
$2,1,3$	$10+10+5+10+5+3$	$=$	43
$2,3,1$	$10+10+3+10+3+5$	$=$	41
$3,1,2$	$3+3+5+3+5+10$	$=$	29
$3,2,1$	$3+3+10+3+10+5$	$=$	34

A simple greedy strategy can be used to solve this problem. It says that to reduce total retrieval time, store the shortest programs first. That means, sort the list of programs in increasing order of their lengths and place them on the tape in that order.

This works because:

- Short programs take less time to pass through.
- Placing them earlier ensures that longer programs appear later, reducing the time they contribute to the total.

So, for the example above, sorting the lengths:

- (3, 5, 10) → Programs 3, 1, 2

This is the optimal order, and the greedy method gives the correct answer.

This strategy is very efficient. The programs can be sorted in O(n log n) time using any standard sorting algorithm such as merge sort or heap sort.

Extension to Multiple Tapes

In more complex systems, programs may need to be stored across multiple tapes, not just one. Suppose there are m tapes available.

The task is to distribute the programs across the tapes in such a way that the total retrieval time for all tapes combined is minimized. This is called Total Retrieval Time (TD).

To solve this, a greedy rule is used:

1. First, sort the programs by length in increasing order.
2. Distribute the programs one by one across the tapes in a round-robin fashion (cycling through each tape in turn).
3. For example:
   - First m programs go to tapes T_0, T_1, ..., T_{m-1}
   - Next m programs again go to T_0, T_1, ..., T_{m-1}
   - Continue this until all programs are assigned.

On each tape, the programs are stored in increasing order of their lengths (as they were already sorted). This ensures that the retrieval time remains low on every tape.

This strategy does not require knowledge of the actual lengths while distributing and works efficiently with time complexity O(n). The correctness of this method is backed by a theorem, which proves it leads to optimal storage.

The Optimal Storage on Tapes problem is solved efficiently using a greedy strategy:

- For a single tape, sort programs by length and store them in that order.
- For multiple tapes, distribute them round-robin after sorting.

The approach ensures minimal average retrieval time and uses simple operations like sorting and sequential assignment. It is an excellent example of the ordering paradigm in greedy algorithm design.

Algorithm for Assigning programs to tapes

```
Algorithm Store(n, m)
// n is the number of programs
// m is the number of tapes
{
j := 0; // Start from the first tape
   for i := 1 to n do
   {
        Write("Append program", i, "to permutation for tape", j);
        j := (j + 1) mod m;
   }
}
```

This algorithm is designed for the multi-tape version of the Optimal Storage on Tapes problem. It distributes n programs across m tapes, ensuring a balanced and efficient arrangement.

Input:

- n: Total number of programs to be stored.
- m: Total number of tapes available.

Working Mechanism:

1. The algorithm initializes a counter j to 0. This variable keeps track of the current tape number where the next program will be stored.
2. A for loop runs from i = 1 to i = n, where each iteration represents assigning one program.
3. Inside the loop:

   - The algorithm assigns program i to tape j.
   - After assignment, it updates j using the formula:
     j := (j + 1) mod m
   - This ensures that the program is stored on the next tape in a cyclic (round-robin) fashion.
   - When j reaches m, it resets to 0, continuing the cycle.

Output:

The output is a print statement (or logical assignment) that shows:
"Append program i to permutation for tape j"
This means that program i will be added to the list of programs assigned to tape j.
Example:
Suppose n = 6 and m = 3.

The algorithm would distribute programs as follows:

Program (i)	Assigned Tape (j)
1	0
2	1
3	2
4	0
5	1
6	2

This round-robin assignment ensures that:

- Programs are evenly spread across all tapes.
- Each tape receives approximately n/m programs.
- It supports efficient retrieval and load balancing.

The Store(n, m) algorithm distributes programs across multiple tapes using a modulo-based round-robin strategy. It ensures that no single tape is overloaded, and the order of distribution respects the sequence in which the programs appear (assuming they are already sorted by increasing length). This approach is simple, fast (linear in time), and forms the basis of an optimal strategy for multi-tape storage systems.

9.3 OPTIMAL MERGE PATTERNS

When dealing with sorted data, it is often necessary to merge multiple sorted files into one final sorted file. If there are only two sorted files, say with n and m records respectively, they can be merged in O(n + m) time. However, when there are more than two files, merging becomes more complex, as multiple pairwise merges need to be performed until a single sorted file is produced.

For example, if there are four sorted files: x1, x2, x3, and x4, there are many ways to merge them:

1. One way is to:
   - Merge x1 and x2 → result is y1
   - Merge y1 and x3 → result is y2
   - Merge y2 and x4 → final merged file
2. Another way is to:
   - Merge x1 and x2 → y1
   - Merge x3 and x4 → y2
   - Merge y1 and y2 → final merged file

Each of these merge patterns will result in a different total number of record moves, and some may take significantly more time than others. The challenge is to find the merge pattern that requires the fewest total record moves, or comparisons. This problem is known as the Optimal Merge Pattern Problem, and it fits into the ordering paradigm because it involves choosing the best sequence (or order) of merges.

Example:
The files x_1, x_2, and x_3 are three sorted files of length 30, 20, and 10 records each.Merging x_1 and x_2 requires 50 record moves. Merging the result with x_3 requires another 60 moves. The total number of record moves required to merge the three files this way is 110.If, instead, merge x_2 and x_3 (taking 30 moves) and then merge the result with x_1 (taking 60 moves),the total record moves made is only 90. Hence, the second merge pattern is faster than the first.

Merge Pattern 1:
1. Merge x_1 and x_2 →
Cost = 30 + 20 = 50 moves
2. Merge the result (size 50) with x_3 →
Cost = 50 + 10 = 60 moves
Total cost = 50 + 60 = 110 moves

Merge Pattern 2:
1. Merge x_2 and x_3 →
Cost = 20 + 10 = 30 moves
2. Merge the result (size 30) with x_1 →
Cost = 30 + 30 = 60 moves
Total cost = 30 + 60 = 90 moves

The second merge pattern is more efficient, requiring only 90 moves compared to 110 moves in the first pattern. This demonstrates that the order of merging plays a crucial role in

minimizing the total merge cost.Choosing the right sequence of merges is essential to achieving the optimal merge pattern.

When merging two sorted files, say one with n records and another with m records, the total number of record moves required is approximately n + m. This cost arises because all the records from both files need to be scanned and merged into one.

Now, suppose we have more than two files that need to be merged into a single sorted file. There are multiple ways in which these files can be pairwise merged, and each merge pattern results in a different total number of record moves. The goal is to determine an optimal merge pattern—that is, the merge sequence that results in the fewest total record moves.

To achieve this, we follow a greedy selection criterion, which is simple yet effective:

At each step, merge the two smallest-sized files available.

Example: Merging Five Files
Consider five files:

- x_1, x_2, x_3, x_4, x_5
- File sizes: 20, 30, 10, 5, and 30 respectively

Using the greedy rule, we proceed as follows:

1. **Merge x_4 and x_3**
   Files: 5 + 10 → Result = z_1 of size 15
2. **Merge z_1 and x_1**
   Files: 15 + 20 → Result = z_2 of size 35
3. **Merge x_2 and x_5**
   Files: 30 + 30 → Result = z_3 of size 60
4. **Merge z_2 and z_3**
   Files: 35 + 60 → Result = z_4 of size 95

Total record moves =
15 (step 1) + 35 (step 2) + 60 (step 3) + 95 (step 4) = 205
This merge pattern is found to be optimal for this instance, as it minimizes the total number of record moves.

Definition: Two-Way Merge Pattern

This type of merging process—where at each step only two files are merged—is known as a two-way merge pattern. All steps involve exactly two input files, and the process continues until a single final file is obtained.

Binary Merge Tree Representation
A two-way merge pattern can be represented using a binary merge tree, where:

- Leaf nodes (drawn as squares) represent the original input files. These are called external nodes.
- Internal nodes (drawn as circles) represent intermediate merged files formed during the process.
- Each internal node has exactly two children, representing the two files it was created from.
- The value inside each node is the number of records in that file.

This tree visually models the entire merging process.

Tracking Record Moves

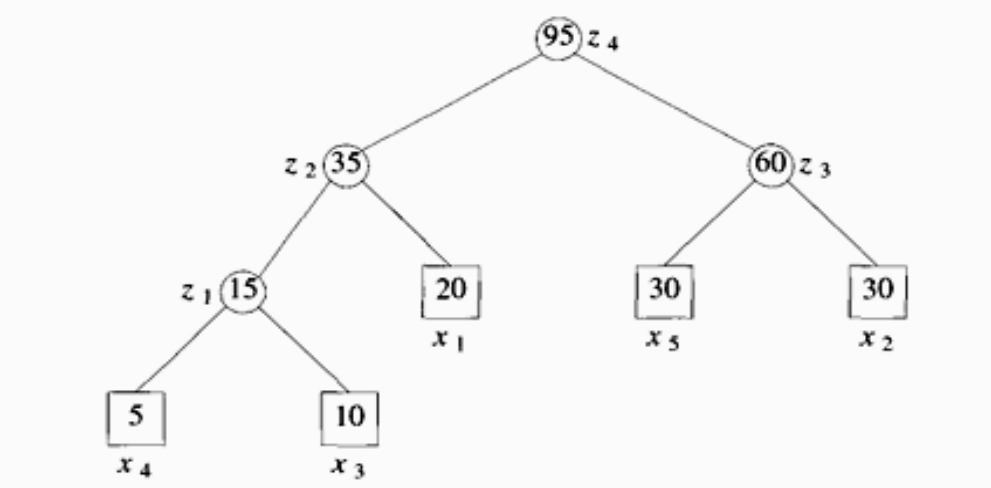


Figure 9.1Binary merge tree representing a merge pattern

The number of times each file's records are moved corresponds to the depth of its external node in the binary merge tree

.

For example, if file x_4 is merged:

- First to form z_1
- Then z_1 is merged into z_2
- Finally, z_2 is merged into z_4

Then, x_4 appears at depth 3 from the root, and its records are moved 3 times.

To calculate the total number of record moves, use the formula:

$$\text{Total moves} = \sum_{i=1}^{n} (d_i \times q_i)$$

Where:

- d_i = distance of external node i from the root (depth)
- q_i = number of records in file x_i

This value is called the weighted external path length of the merge tree.

Goal of the Optimal Merge Pattern

The aim is to find a binary merge tree (i.e., a two-way merge pattern) that has the minimum weighted external path length, which corresponds to the least total record movement cost.

Steps to generatea two-way mergetree

```
treenode = record {
    treenode * lchild; treenode * rchild;
    integer weight;
};
```

```
treenode = record {
    treenode * lchild; treenode * rchild;
    integer weight;
};

Algorithm Tree(n)
// list is a global list of n single node
// binary trees as described above.
{
    for i := 1 to n - 1 do
    {
        pt := new treenode; // Get a new tree node.
        (pt → lchild) := Least(list); // Merge two trees with
        (pt → rchild) := Least(list); // smallest lengths.
        (pt → weight) := ((pt → lchild) → weight)
                + ((pt → rchild) → weight);
        Insert(list, pt);
    }
    return Least(list); // Tree left in list is the merge tree.
}
```

The Tree function implements the greedy rule to generate an optimal two-way merge tree for n input files. Here's how it works:

1. Input is a list of n trees, where each tree initially contains just one node.
   - This node is an external node representing a file.
   - Each node has three fields:
   - lchild: pointer to the left child
   - rchild: pointer to the right child
   - weight: size of the file (number of records)
   - Initially, lchild and rchild are set to 0, and weight is set to the file's length.
2. During the algorithm, each tree in the list evolves:
   - A tree's root node's weight becomes the sum of the weights of all files it represents.
3. The algorithm uses two utility functions:
   - Least(list):
     - Finds and removes from the list the tree with the smallest root weight.
     - Returns a pointer to that tree.
   - Insert(list, t):
     - Inserts tree t (with a new root) back into the list.
4. The process continues until only one tree remains in the list.

This final tree is the optimal binary merge tree representing the best sequence of merges.
The correctness of the algorithm is proved by the theorem, which states that the tree generated by the Tree function produces the minimum total record moves, and thus constructs an optimal merge pattern.

9.4 HUFFMAN CODES

Huffman coding is a widely used technique in data compression and efficient message encoding. It is based on constructing a binary tree where each external (leaf) node represents a unique message or symbol.

Purpose of Huffman Coding

Suppose there is a set of messages:

$M_1, M_2, ..., M_{n+1}$.

Each message needs to be transmitted using a binary code (a sequence of 0s and 1s). The goal is to assign binary codes to the messages such that:

1. No code is a prefix of another (ensuring the codes are uniquely decodable).
2. The average number of bits used to represent the messages is as small as possible.

Binary Decode Tree

To decode messages efficiently, a binary decode tree is used. In this tree:

- Each external node corresponds to a message.
- Internal nodes represent decision points in the decoding process.
- Moving left in the tree corresponds to reading a 0, and moving right corresponds to reading a 1.

So, the path from the root to an external node defines the binary code for that message.
For example, if the path to message M_1 is left-left-left, the code for M_1 would be 000.

Huffman Codes from the Tree

The binary strings formed by traversing the decode tree (from root to external nodes) are called Huffman Codes.
These codes are:

- Shorter for more frequent messages.
- Longer for less frequent messages.

This makes the coding efficient, because commonly used messages consume fewer bits, which is ideal for compression.

Cost of Decoding

The cost of decoding a message is proportional to the number of bits in its code. This is the same as the depth (or distance from the root) of the external node in the decode tree.
Let

- q_i be the relative frequency (or probability) that message M_i will be transmitted.
- d_i be the depth of the node corresponding to message M_i in the tree.

Then, the expected decode time (or average number of bits per message) is:

$$\text{Expected decode time} = \sum_{i=1}^{n+1} q_i \cdot d_i$$

This is the same as the weighted external path length of the decode tree.

Optimization Goal

To minimize the expected decode time, we need to construct a binary tree that has the smallest possible weighted external path length. Huffman's algorithm is specifically designed to do this.
Therefore:

- The Huffman code minimizes the average number of bits per message.
- It also minimizes the decoding time, making it both optimal and efficient.

Huffman coding is a perfect example of how greedy algorithms and binary trees are applied to real-world problems like compression. By assigning shorter codes to more frequent messages and longer codes to rare ones, Huffman coding achieves optimal encoding that reduces the size of transmitted or stored data without losing any information.

9.5 SUMMARY

The problems of Optimal Storage on Tapes **and** Optimal Merge Patterns emphasize the power of greedy algorithms in structuring data for efficient retrieval and processing. Both problems are rooted in ordering and sequencing, where early optimal choices lead to globally optimal outcomes. Huffman coding extends this logic to data compression, ensuring minimal decoding cost through binary tree construction. These problems not only demonstrate key algorithmic principles but also have real-world applications in storage systems, compiler design, and communication technologies.

9.6 KEY TERMS

Mean Retrieval Time, Round Robin Assignment, Record Move Cost, Greedy Algorithm, Binary Decode Tree, Prefix-Free Code

9.7 REVIEW QUESTIONS

1. What is Mean Retrieval Time, and how is it minimized in tape storage?
2. Describe the greedy strategy used for Optimal Storage on Tapes.
3. Explain the concept of Optimal Merge Patterns with an example.
4. How is a Binary Merge Tree constructed, and what does its cost represent?
5. What are Huffman codes, and how do they achieve data compression?
6. Describe how the greedy strategy is used in building Huffman Trees.

9.8 SUGGESTIVE READINGS

1. Horowitz, E., Sahni, S., & Rajasekaran, S. (2007). Fundamentals of Computer Algorithms. Universities Press.
2. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press.
3. Kleinberg, J., & Tardos, É. (2006). Algorithm Design. Pearson.
4. Dasgupta, S., Papadimitriou, C. H., & Vazirani, U. V. (2008). Algorithms. McGraw-Hill.

Dr. Vasantha Rudramalla

LESSON-10

GREEDY TECHNIQUES IN GRAPH ALGORITHMS

OBJECTIVES

After completing this lesson, the learner will be able to:

- Understand the concept of spanning trees and their applications.
- Apply greedy methods to find Minimum-Cost Spanning Trees.
- Compare and implement Prim's and Kruskal's algorithms.
- Identify the subset paradigm in graph algorithms.
- Evaluate the time complexity and use cases of MST algorithms.

STRUCTURE

10.1 INTRODUCTION

Graphs are widely used in computer science to model relationships in networks. A spanning tree of a graph is a subgraph that connects all vertices with the minimum number of edges and no cycles. The Greedy method is used in graph algorithms to ensure local optimum choices result in a globally optimal solution. In this lesson, Minimum-Cost Spanning Tree (MST) problems are solved using Prim's and Kruskal's greedy algorithms.

10.2 SPANNING TREES AND APPLICATIONS

Definition 4.1
Let G = (V, E) be an undirected, connected graph.
A subgraph t = (V, E′) of G is called a spanning tree of G if t is a tree.
This means:

- t includes all the vertices of G (it spans G),
- t is connected,
- t has no cycles (a tree by definition).

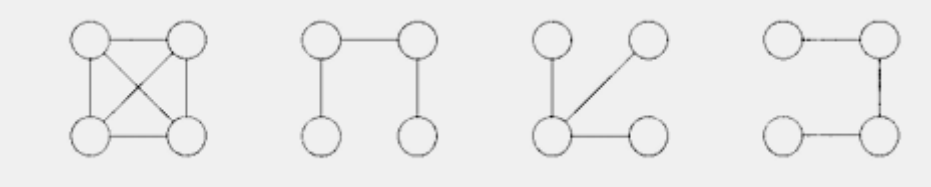


Figure 10.1 An undirected graph and three of its spanningtrees

The above figure shows a complete graph on four nodes, along with three different spanning trees of that graph. Each spanning tree connects all the vertices without forming any cycles, using exactly n − 1 edges, where n is the number of vertices.

Applications of Spanning Trees
Spanning trees are widely used in various real-world and theoretical applications. Below are some important uses:

1. Electrical Networks and Independent Circuit Equations
Spanning trees can be used to derive a set of independent circuit equations for an electric network. The process works as follows:

- First, identify a spanning tree in the network.
- Let B be the set of edges not included in the spanning tree.
- Add one edge from set B at a time to the spanning tree. Each time an edge is added, it forms a unique cycle.
- These cycles are used to derive circuit equations, one for each edge in B.
- These cycles are independent, meaning:
- No cycle in the set can be created by combining other cycles in the set.
- Each cycle includes one edge from B that is not shared with any other cycle.

Therefore, the circuit equations formed this way are also independent. Moreover, the set of cycles formed by this method is known as a cycle basis of the graph. This means that all possible cycles in the graph can be generated by taking linear combinations of the cycles in the cycle basis.

2. Minimally Connected Subgraphs
Another important application comes from the fact that a spanning tree is a minimal subgraph of G that:

- Contains all vertices of G, and
- Is connected.

A minimal subgraph means it has the fewest number of edges necessary to connect all nodes. In any connected graph with n vertices, a spanning tree has exactly n − 1 edges. This is the minimum number of edges needed to keep all vertices connected without creating cycles.

3. Connecting Cities or Communication Networks
Spanning trees are especially useful when:

- Vertices represent cities, and
- Edges represent possible communication links between cities.
- In such a case:
- The minimum number of links needed to connect all cities is n − 1.

- Each spanning tree of G represents a different valid way to connect the cities using exactly n − 1 links.

10.3 WEIGHTED GRAPHS AND MST PROBLEM

Weighted Graphs and Cost Optimization
In many real-life applications, the edges of a graph have weights.

These weights can represent:
- Cost of construction
- Distance between cities
- Time delay, and so on.

When dealing with a weighted graph, the objective is to:
- Find a set of connections (edges) that form a tree, and
- Have the minimum total weight (or cost).

This gives rise to the Minimum-Cost Spanning Tree Problem.
Note:
- If a selection of edges includes a cycle, then it is not a tree.
- In that case, removing any edge from the cycle will reduce the total cost and still keep the graph connected.
- Hence, the minimum-cost connection must always be a spanning tree.

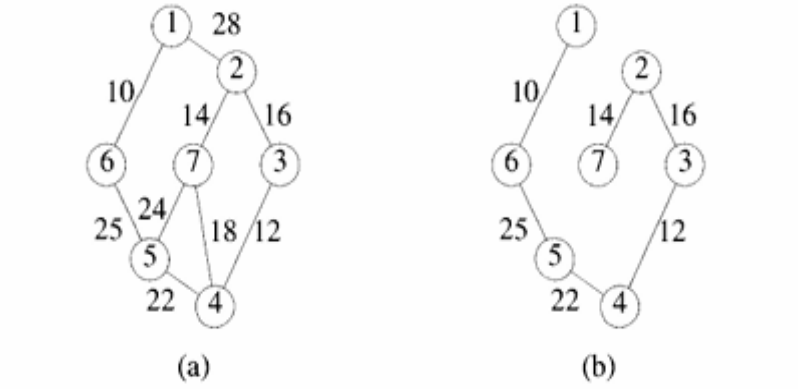


Figure 10.2 A graph and its minimum cost spanning tree

The above figure illustrates:
- A graph with weighted edges, and
- One of its minimum-cost spanning trees, showing how the best set of connections is selected.

Subset Paradigm Classification
Since finding a minimum-cost spanning tree involves:
- Selecting a subset of edges from the full set of edges,
- While maintaining connectivity and minimizing cost,

This problem fits into the subset paradigm in algorithm design.

Greedy Method for Minimum-Cost Spanning Tree: Two Interpretations

In the greedy approach, the next edge to be added to the spanning tree is the one that results in the smallest increase in the total cost. There are two primary ways to interpret and implement this rule:

1. Prim's Interpretation – Tree-Growing Method
Approach:
- Start with an arbitrary node (or the smallest-weight edge).
- Grow the tree by adding the minimum-weight edge that connects a vertex already in the tree to a vertex outside the tree.
- Ensure that the tree remains connected and acyclic.

Criterion:
Choose the minimum-cost edge (u, v) such that:
- One endpoint is in the growing tree,
- The other is not yet included, and
- Adding this edge does not create a cycle.

This is the method described earlier and is used in Prim's Algorithm.

2. Kruskal's Interpretation – Edge-Selection Method
Approach:
- Consider all edges of the graph, sorted in increasing order of cost.
- Starting from an empty set, add the next smallest edge to the spanning tree if and only if it does not create a cycle.
- Repeat until (n – 1) edges are selected (where n is the number of vertices).

Criterion:

Among all remaining edges, choose the one with minimum cost that does not form a cycle when added to the current set of selected edges.

This is the second interpretation of the greedy method and is used in Kruskal's Algorithm.

10.4 PRIM'S ALGORITHM

In the field of graph theory and algorithm design, Prim's Algorithm is a classical greedy approach used to find a Minimum-Cost Spanning Tree (MST) for a connected, undirected, and weighted graph.
A spanning tree of a graph is a subgraph that:
- Includes all the vertices of the graph,
- Is connected, and
- Contains no cycles.

A Minimum-Cost Spanning Tree is a spanning tree in which the sum of the weights (or costs) of the edges is as small as possible.

Prim's Algorithm constructs such a tree by starting with a single vertex and gradually adding edges to expand the tree, always choosing the minimum-weight edge that connects a vertex in the tree to a vertex outside the tree.

Definition of Prim's Algorithm
Prim's Algorithm is a greedy algorithm that finds a minimum spanning tree for a weighted undirected graph. The algorithm works by:

At each step, selecting the edge with the minimum weight that connects a vertex already in the tree to a vertex not yet included, and adding this edge to the tree.
It continues this process until all vertices are included in the tree and exactly (n – 1) edges have been added (where n is the number of vertices in the graph).

Working Principle

- Let G = (V, E) be a connected, undirected graph with weights on edges.
- Initialize a set T to store the edges of the minimum spanning tree.
- Start from any arbitrary vertex v.
- At each step, choose the edge (u, v) such that:
  - u is in the current tree (T)
  - v is not in the tree
  - (u, v) has the smallest cost among all such edges
- Add vertex v and edge (u, v) to the tree.
- Repeat until all vertices are included in the tree.

Data Structures Used

To implement Prim's Algorithm efficiently, especially in large graphs, the following data structures are often used:

- Priority Queue (Min-Heap): to quickly extract the next minimum-cost edge.
- Array near[]: to track the nearest vertex in the MST for each vertex not yet included.
- Boolean visited[] or MST[]: to mark which vertices are already part of the spanning tree.

Time Complexity

The time complexity depends on the implementation:

Data Structure	Time Complexity
Adjacency Matrix + Array	$O(V^2)$
Min-Heap + Adjacency List	O(E log V)

Where V is the number of vertices and E is the number of edges.

Applications of Prim’s Algorithm

Prim's Algorithm is widely used in:

1. Network Design:

Designing least-cost communication, electrical, or water distribution networks.

2. Telecommunication Routing:

Building backbones of wired and wireless networks.

3. Cluster Analysis:

In machine learning and pattern recognition, MST helps in hierarchical clustering.

4. Approximation Algorithms:

Used as part of approximation algorithms for NP-hard problems like the Traveling Salesman Problem (TSP).

5. Map Navigation and Road Design:

Planning the most efficient roads between cities or regions.

6. Circuit Design:

Connecting components with minimal total wiring.

Properties of the Resulting MST

- The resulting spanning tree is unique if all edge weights are distinct.
- It contains exactly (V − 1) edges.

- It ensures minimum total edge cost among all possible spanning trees.
- It is acyclic and spans all vertices.

Advantages and Limitations

Advantages:

- Simple and intuitive.
- Efficient for dense graphs when using adjacency matrix.
- Guarantees a minimum-cost spanning tree.

Limitations:

- May not be as efficient as Kruskal's Algorithm for sparse graphs unless implemented with a priority queue.
- Requires the graph to be connected; otherwise, the MST cannot be formed.

```
Algorithm Prim(E, cost, n, t)
// E is the set of edges in G. cost[1 : n, 1 : n] is the cost
// adjacency matrix of an n vertex graph such that cost[i, j] is
// either a positive real number or ∞ if no edge (i, j) exists.
// A minimum spanning tree is computed and stored as a set of
// edges in the array t[1 : n − 1, 1 : 2]. (t[i, 1], t[i, 2]) is an edge in
// the minimum-cost spanning tree. The final cost is returned.
{
    Let (k, l) be an edge of minimum cost in E;
    mincost := cost[k, l];
    t[1, 1] := k; t[1, 2] := l;
    for i := 1 to n do  // Initialize near.
        if (cost[i, l] < cost[i, k]) then near[i] := l;
        else near[i] := k;
    near[k] := near[l] := 0;
    for i := 2 to n − 1 do
    { // Find n − 2 additional edges for t.
        Let j be an index such that near[j] ≠ 0 and
        cost[j, near[j]] is minimum;
        t[i, 1] := j; t[i, 2] := near[j];
        mincost := mincost + cost[j, near[j]];
        near[j] := 0;
        for k := 1 to n do // Update near[ ].
            if ((near[k] ≠ 0) and (cost[k, near[k]] > cost[k, j]))
                then near[k] := j;
    }
    return mincost;
}
```

The goal of Prim's algorithm is to construct a Minimum-Cost Spanning Tree (MST) from a connected, undirected, weighted graph. The algorithm grows the MST by selecting the lowest-cost edge that connects a vertex inside the tree to a vertex outside the tree.

Inputs and Outputs

Inputs:

- E: Set of edges in the graph.
- cost[1:n]: A **cost matrix** (or adjacency matrix) representing edge weights.
    - If there is no edge between vertices i and j, cost[i][j] = ∞.
    - If there is an edge, it stores the actual cost.
- n: Number of vertices.

- t [1:n−1][1:2]: A 2D array to store the edges of the Minimum Spanning Tree.

Output:

- mincost: Total cost of the minimum spanning tree.
- t[i][1] and t[i][2]: Stores the edges in the MST.

Step-by-Step Execution

Step 1: Select Initial Minimum-Cost Edge

Let (k, l) be an edge of minimum cost in E;

```
mincost := cost[k][l];
t[1,1]:= k; t[1,2] := l;
```

- Find the edge with the **lowest cost** in the entire graph. This is the starting point for the MST.
- Add this edge to the MST (t[1,1] and t[1,2]).
- Initialize mincost with this edge's weight.

Step 2: Initialize the near[] Array

```
for i := 1 to n do
        if (cost[i][l] < cost[i][k]) then near[i] := l
        else near[i] := k;
near[k] := near[l] := 0;
```

- The near[] array keeps track of which vertex (among those currently in the tree) is closest to vertex i.
- For each vertex i, compare its distance to the two endpoints k and l of the initial edge. Assign near[i] to whichever is closer.
- Set near[k] and near[l] to 0 to mark that these two vertices are already included in the MST.

Step 3: Loop to Add Remaining (n − 2) Edges

for i := 2 to n − 1 do

- Since we already added one edge, we need **(n – 2)** more edges to complete the MST.

Step 4: Select the Next Minimum-Cost Edge

Let j be an index such that near[j] ≠ 0 and cost[j][near[j]] is minimum;

- Among all vertices not yet in the MST (i.e., near[j] ≠ 0), find the one that is closest to the current MST.
- This vertex and the edge connecting it to the tree will be added next.

Step 5: Add the Edge to the MST

```
t[i,1] := j; t[i,2] := near[j];
mincost := mincost + cost[j][near[j]];
near[j] := 0;
```

- Add the edge from vertex j to its nearest vertex in the tree into the MST.
- Update the mincost by adding the cost of this edge.
- Mark vertex j as now included in the tree (near[j] := 0).

Step 6: Update near[] Array

```
for k := 1 to n do
  if ((near[k] ≠ 0) and (cost[k][near[k]] > cost[k][j]))
```

then near[k] := j;

For every vertex k not yet in the MST:

- Check whether the newly added vertex j is closer to k than the currently stored near[k].
- If so, update near[k] to j.

Step 7: Return the Result

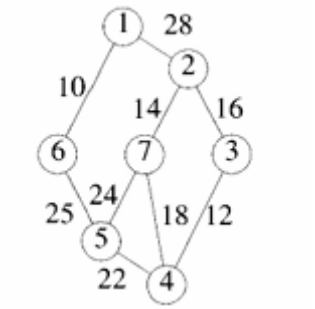
return mincost;

After n−1 edges have been added, return the mincost as the total weight of the Minimum-Cost Spanning Tree.

Summary of Prim's Algorithm Behavior

1. Starts from the lowest-cost edge.
2. Repeatedly adds the nearest vertex to the growing tree.
3. Keeps track of the closest neighbor for each vertex using near[].
4. Ensures no cycles are formed and the cost remains minimum.
5. Stops when (n−1) edges are selected, forming a complete MST.

Example



Given Graph

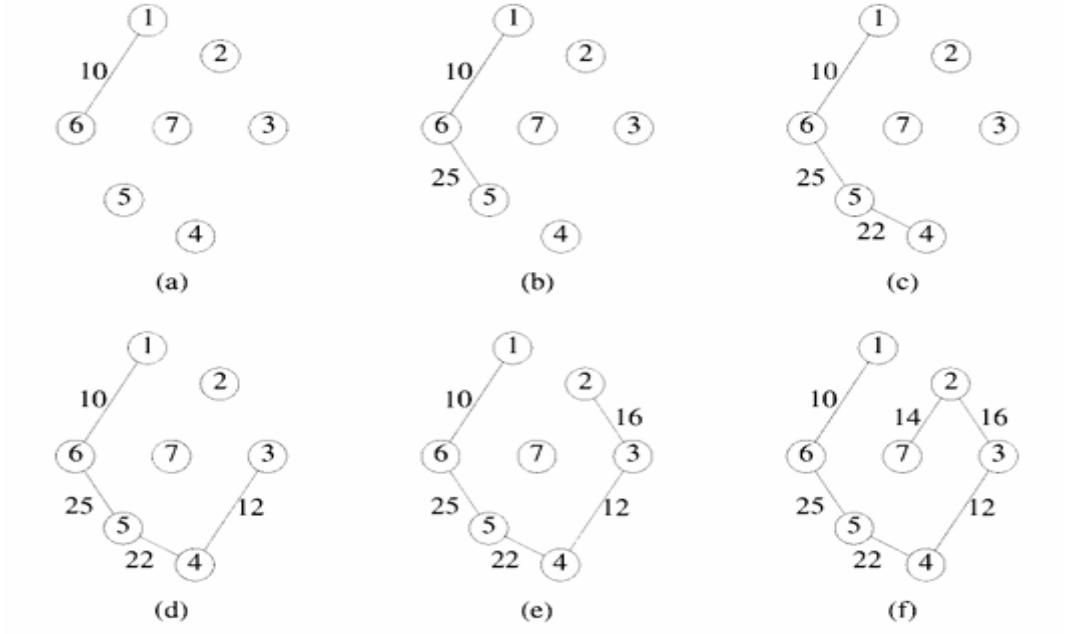


Figure 10.3 Stages in Prim's algorithm

10.5 KRUSKAL'S ALGORITHM

Kruskal's Algorithm is a well-known greedy algorithm used in graph theory to find the Minimum Spanning Tree (MST) of a connected, undirected graph. The MST is a subset of edges that connects all the vertices in the graph with the minimum possible total edge weight, and without forming any cycles.

Named after Joseph Kruskal, who introduced the method in 1956, the algorithm operates based on a simple but effective strategy: always choosing the lowest-cost connection available that links different parts of the network. This approach ensures that each new connection added contributes to a more efficient structure while avoiding any circular paths that would introduce redundancy.

Kruskal's Algorithm is especially suitable for sparse graphs, where the number of connections (edges) is relatively low compared to the number of nodes (vertices). Its edge-based nature means that the algorithm does not depend on the order or position of nodes but instead evaluates the costs of connections directly.

In practical scenarios, this algorithm is widely applied in areas that require the construction of cost-effective networks. These include the development of roadways, communication networks, electrical grid systems, and more. It is also used in clustering algorithms, image processing, and various optimization problems in computer science and operations research.

Kruskal's Algorithm plays a key role in ensuring minimum cost connectivity across systems, making it an essential tool in both academic and real-world problem-solving.

```
Algorithm Kruskal(E, cost, n, t)
// E is the set of edges in G. G has n vertices. cost[u, v] is the
// cost of edge (u, v). t is the set of edges in the minimum-cost
// spanning tree. The final cost is returned.
{
    Construct a heap out of the edge costs using Heapify;
    for i := 1 to n do parent[i] := -1;
    // Each vertex is in a different set.
    i := 0; mincost := 0.0;
    while ((i < n - 1) and (heap not empty)) do
    {
        Delete a minimum cost edge (u, v) from the heap
        and reheapify using Adjust;
        j := Find(u); k := Find(v);
        if (j ≠ k) then
        {
            i := i + 1;
            t[i, 1] := u; t[i, 2] := v;
            mincost := mincost + cost[u, v];
            Union(j, k);
        }
    }
    if (i ≠ n - 1) then write ("No spanning tree");
    else return mincost;
}
```

This algorithm finds the Minimum Spanning Tree (MST) of a graph, meaning it connects all the vertices together with the least total edge cost and no cycles.
The above algorithm can be explained step wise as below

Input Parameters:

- E: The set of all edges in the graph.
- cost[u, v]: The weight or cost of the edge between nodes u and v.
- n: Number of vertices.
- t: The final list of selected edges that form the Minimum Spanning Tree.

1. Build a Min-Heap of Edges (Heapify):

All the edges are arranged in a heap (priority queue) based on their cost, with the **smallest cost edge at the top**.
Construct a heap out of the edge costs using Heapify;
This helps efficiently pick the edge with the smallest cost at each step.

2. Initialize Disjoint Sets for Vertices:

Each vertex starts in its own set using a parent array initialized to -1.
for i := 1 to n do parent[i] := -1;
This is for cycle detection using the Disjoint Set Union (Union-Find) technique.

3. Initialize Variables:

i := 0; mincost := 0.0;

- i keeps track of how many edges have been added to the MST.
- mincost stores the total cost of the MST.

4. Main Loop (Build MST):

while ((i< n - 1) and (heap not empty)) do
Keep repeating until:

- i edges have been added (MST needs n-1 edges), AND
- The heap still has edges to check.

5. Pick the Minimum Edge and Check:

Delete a minimum cost edge (u, v) from the heap;
Get the next lowest-cost edge from the heap.
Then find the root parents of u and v:
j := Find(u); k := Find(v);
If j ≠ k (i.e., u and v are in different sets), they are not yet connected:

- Add this edge to MST.
- Increase the edge count (i).
- Add the cost to the total (mincost).
- Merge the sets (Union(j, k)).

```
if (j ≠ k) then
{
i := i + 1;
t[i, 1] := u; t[i, 2] := v;
mincost := mincost + cost[u, v];
Union(j, k);
}
```

If they are already connected (same set), skip the edge to avoid a cycle.

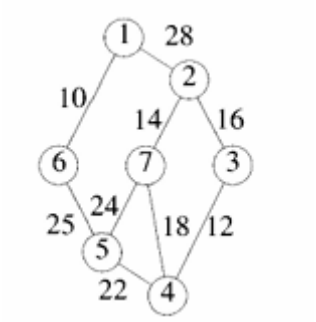
6. FINAL CHECK:

If MST has less than n-1 edges, it means the graph is disconnected:
if (i ≠ n - 1) then write ("No spanning tree");
Otherwise, return the total minimum cost:
else return mincost;

CONCLUSION (WHAT IT DOES):

- Connects all vertices using the **cheapest edges possible**.
- Ensures **no cycles** using Union-Find.
- Stops when the MST is complete or the heap is empty.
- Returns the total cost of the Minimum Spanning Tree.

Example



Given Graph

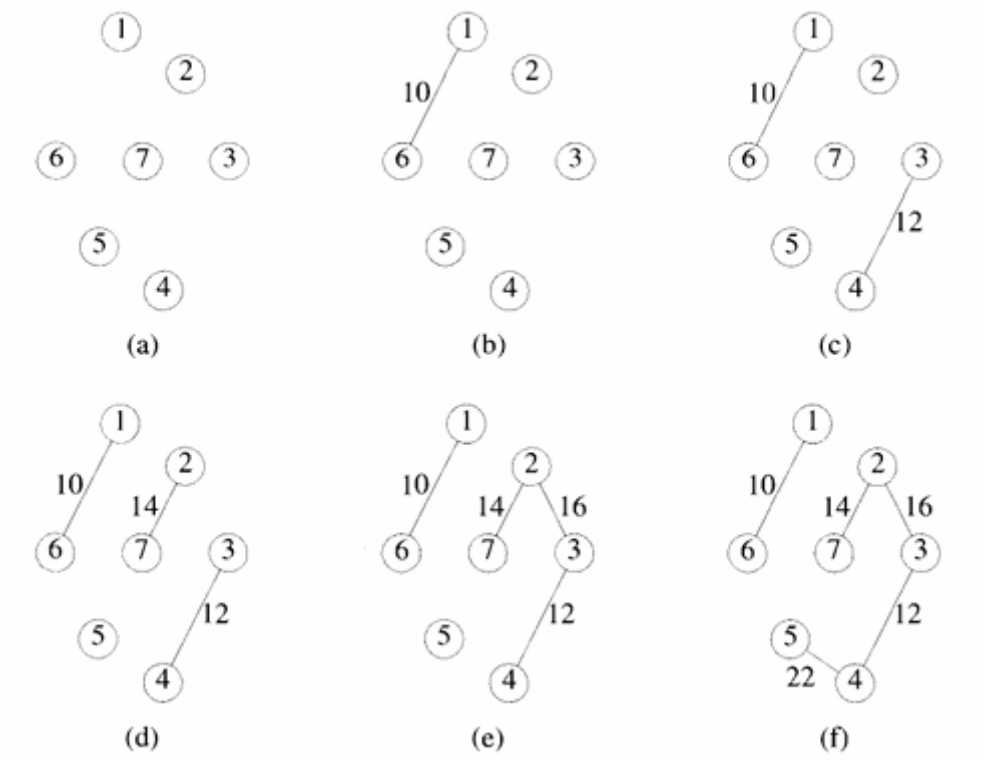


Figure 10.4 Stages in Kruskal's algorithm

Step-by-step Execution

(a) Initial State
- All nodes are isolated, no edges included yet.
- The algorithm will now start picking edges in order of increasing weight.

(b) Add edge (1–6) with weight 10
- Smallest weight edge.
- Connects nodes 1 and 6.
- No cycle is formed.
- Included in MST.

(c) Add edge (3–4) with weight 12
- Next smallest edge.
- Connects nodes 3 and 4.
- No cycle.
- Included in MST.

(d) Add edge (2–7) with weight 14
- Next smallest edge.
- Connects nodes 2 and 7.
- No cycle.
- Included in MST.

(e) Add edge (2–3) with weight 16
- Connects two different components: (2,7) and (3,4).
- No cycle.
- Included in MST.

(f) Add edge (4–5) with weight 22
- Connects node 5 to the (2–3–4–7) component.
- No cycle.
- Included in MST.

At this point:
- Nodes 1–6, 2–7, 3–4–5 are all connected.
- Total 6 nodes connected → still need one more edge.

Remaining edges (not added):
- (5–7): 24 → forms a cycle.
- (5–6): 25 → would create a cycle.
- (1–2): 28 → would also create a cycle.

So, **algorithm stops** once **(n-1 = 7-1 = 6)** edges are added.

Final MST includes edges:

1. (1–6): 10
2. (3–4): 12
3. (2–7): 14
4. (2–3): 16
5. (4–5): 22
6. (Add one more to connect all 7 nodes)

The 6th edge needed is likely selected as per the next lowest-weight edge that connects remaining disjoint sets. From diagram **(f)**, it appears all are connected, meaning the MST is complete.

Total Minimum Cost of the MST = 10 + 12 + 14 + 16 + 22 = 74

10.6 SUMMARY

This lesson explored the concept of spanning trees and their applications in network design. It covered the use of greedy methods for solving the Minimum-Cost Spanning Tree problem, focusing on two classical algorithms: Prim's Algorithm and Kruskal's Algorithm. While Prim's method grows the MST from a vertex, Kruskal's method constructs it by selecting edges in ascending order of cost. Both ensure minimal total weight and acyclic connectivity, with implementation efficiency varying by graph structure.

10.7 KEY TERMS

Spanning Tree, MST, Prim's Algorithm, Kruskal's Algorithm, Greedy Method, Union-Find

10.8 REVIEW QUESTIONS

1. What is a spanning tree? Explain its properties.
2. How does Prim's algorithm utilize the greedy approach?
3. Compare and contrast Prim's and Kruskal's algorithms.
4. What data structures are typically used in Prim's algorithm?
5. What is the subset paradigm, and how does it apply to MST problems?

10.9 Suggestive Readings

1. Horowitz, E., Sahni, S., & Rajasekaran, S. Fundamentals of Computer Algorithms. Universities Press.
2. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. Introduction to Algorithms. MIT Press.
3. Kleinberg, J., & Tardos, É. Algorithm Design. Pearson.
4. Dasgupta, S., Papadimitriou, C. H., & Vazirani, U. V. Algorithms. McGraw-Hill.

Dr. Vasantha Rudramalla

LESSON-11
FUNDAMENTALS OF DYNAMIC PROGRAMMING

OBJECTIVES

After completing this lesson, the learner will be able to:

- Understand the principles of dynamic programming and its distinction from greedy methods.
- Formulate and solve problems using bottom-up and recursive approaches.
- Apply the principle of optimality to optimization problems.
- Implement dynamic programming techniques in problems like 0/1 Knapsack, Shortest Path, Optimal BST, and String Editing.
- Derive recurrence relations and base cases for solving real-world problems.

Structure

11.1 Introduction to Dynamic Programming

Dynamic programming (DP) is a powerful method for solving optimization problems by breaking them into simpler subproblems. It works when the solution can be represented as a sequence of decisions, and where decisions made early affect future choices.

11.2 Applications of Dynamic Programming

Dynamic Programming (DP) is a powerful algorithmic technique for solving optimization problems by breaking them down into overlapping subproblems and solving each subproblem only once. This is typically achieved through recursion, memorization, and bottom-up approaches.

Dynamic programming is a systematic method for solving complex problems by breaking them down into simpler subproblems. It is applicable when the solution to a problem can be formulated as a sequence of decisions, where each decision affects the subsequent choices and the final outcome.

Many computational problems can be effectively solved using this technique. The following examples illustrate scenarios where dynamic programming applies.

Example 11.1 – Knapsack Problem

The 0/1 Knapsack Problem involves selecting items with specific values and weights in such a way that the total value is maximized without exceeding a given weight capacity. This problem can be expressed as a sequence of binary decisions: for each item $x_i$the decision is whether to include it (1) or exclude it (0).

The objective function is to maximize the total value $\sum v_i x_i$, subject to the constraint$\sum w_{i,} x_i \leq m$, where v_i andw_i represent the value and weight of the i^{th} item, respectively, and m is the capacity of the knapsack. The goal is to find the sequence of decisions that yields the optimal total value while respecting the weight constraint.

Example 11.2 – Optimal Merge Patterns

This problem involves merging a collection of files such that the total merging cost is minimized. At each step, the task is to decide which two files should be merged. The optimal sequence is one that results in the lowest total cost of merging all files.

Each decision affects future merging options, and therefore the sequence of decisions must be carefully chosen to minimize the cumulative cost. Dynamic programming is suitable here because it can track partial merges and reuse previously computed solutions.

Example 11.3 – Shortest Path in a Directed Graph

In the shortest path problem from vertex iii to vertex jjj in a directed graph, the path can be represented as a sequence of vertices. The task is to determine the second vertex, the third, and so on, until vertex j is reached. An optimal decision sequence yields the path with the minimum total weight or cost.

However, this problem cannot always be solved optimally through local decisions alone, as will be further clarified in the following discussion.

Greedy Method Limitation in Decision Making

For certain problems, decisions made purely on the basis of local information may not lead to an optimal overall result. This limitation distinguishes greedy algorithms from dynamic programming.

Example 11.4 – Greedy Limitation in Shortest Path Selection

Consider the problem of finding the shortest path from vertex iii to vertex j, where the set Ai contains vertices directly adjacent to iii. Choosing the second vertex in the path solely based on proximity or minimum edge cost does not guarantee that the complete path will be optimal. This is because future decisions depend on the current one, and an early suboptimal choice can lead to a non-optimal final solution.

In contrast, when the objective is to compute the shortest paths from a single source vertex iii to all other vertices in the graph, correct local decisions can be made iteratively, as demonstrated by algorithms such as Dijkstra's.

11.2.1 Role of Dynamic Programming

For problems where an optimal solution cannot be found using a step-by-step greedy approach, an alternative is to consider all possible decision sequences and select the best among them. However, this brute-force approach is typically infeasible due to time and space constraints.

Dynamic programming provides a more efficient strategy by eliminating sequences that cannot lead to an optimal solution. This is achieved by using the principle of optimality, which allows partial results to be stored and reused, significantly reducing the number of sequences to consider.

Principle of Optimality:

An optimal sequence of decisions possesses the property that, regardless of the initial state and the first decision, the remaining decisions must form an optimal sequence for the resulting subproblem.

This principle enables dynamic programming to systematically construct optimal solutions by solving subproblems and combining their results.

Distinction Between Greedy and Dynamic Programming Methods

- **Greedy Method**: Generates only one decision sequence based on immediate benefits without considering future consequences.
- **Dynamic Programming**: Explores multiple decision sequences and discards those containing suboptimal subsequences, as they cannot lead to the global optimum when the principle of optimality is applicable.

Example 11.5 – Verification of the Principle of Optimality (Shortest Path)
Consider the shortest path problem again. Suppose the shortest path from vertex i to vertex j is $i \rightarrow i_1 \rightarrow i_2 \rightarrow \ldots \rightarrow j$.

If the first decision leads from i to i_1, the state of the problem becomes finding the shortest path from i1to j. Assume $i_1 \rightarrow i_2 \rightarrow \ldots \rightarrow j$ is not the shortest path from i_1 to j, and instead a shorter path exists as $i_1 \rightarrow r_1 \rightarrow r_2 \rightarrow \ldots \rightarrow j$.

By substituting the sub-path, the total path becomes $i \rightarrow i_1 \rightarrow r_1 \rightarrow r_2 \rightarrow \ldots \rightarrow j$, which would be shorter than the original, contradicting the assumption of optimality.
Thus, the sub-path $i_1 \rightarrow i_2 \rightarrow \ldots \rightarrow j$ must be optimal, confirming that the principle of optimality holds in this case.

Example 11.6 – 0/1 Knapsack and Principle of Optimality
The 0/1 knapsack problem is a variation of the general knapsack problem where each item can either be selected (1) or not selected (0) — partial inclusion is not allowed.

Let the function KNAP (z, j, y) represent the subproblem of selecting from items z through j such that their total weight does not exceed y. The goal is to maximize the total value of selected items.

The complete problem is therefore represented by:

KNAP(1, n, m)

which denotes selecting from all items 1 to n with total capacity m.
Let the optimal solution be a sequence:

$y_1, y_2, ..., y_n$

where each yi∈ {0, 1} represents whether item i is selected (1) or not (0).
Now consider two cases based on the value of y1:

Case 1: $y_1 = 0$
If item 1 is **not selected**, then the remaining items y2, y3, ..., yn must form an optimal solution for the subproblem:
KNAP(2, n, m)
If they do not, then a better sequence exists, contradicting the optimality of the full sequence.

Case 2: $y_1 = 1$
If item 1 **is selected**, then the remaining capacity becomes m - w1 (where w1 is the weight of item 1), and the remaining items must solve the subproblem:

KNAP(2, n, m - w_1)

Let the optimal solution for this subproblem be:

$z_2, z_3, ..., z_n$

with the value:

$\sum$ (from i=2 to n) pi * zi

If this value plus p1 (value of item 1) is greater than the earlier considered solution, then the new sequence:

1, z2, z3, ..., zn

is a better overall solution. This confirms that the **principle of optimality** holds.

Generic Formulation Using State and Decisions
Let S0 be the initial problem state, and suppose n decisions di need to be made, where $1 \leq i \leq n$. For each decision di, let:

D1 = {r1, r2, r3, ...}

be the set of all possible choices.
Let Si be the new state resulting after making decision di. If Fi is the optimal sequence of decisions for the state Si, then the overall optimal sequence for S0 is:

ri⊕ Fi

That is, the decision ri concatenated with the optimal sequence for the new state Si.

Example 11.7 – Shortest Path Revisited

Let Ai be the set of all vertices adjacent to vertex i. For each vertex k ∈ Ai, let Fk be the shortest path from k to the destination vertex j.
Then the shortest path from i to j is simply the shortest of the paths:
{i → k → Fk}, for all k ∈ Ai
This again follows the **principle of optimality**, where the optimal solution includes optimal subpaths.

Example 11.8 – Recursive Formulation for 0/1 Knapsack

Let gj(y) be the value of the optimal solution to the subproblem:

KNAP(j + 1, n, y)

This function represents the maximum value that can be obtained from items j+1 through n with a remaining capacity of y.
The complete solution is then:

$g_0(m)$

representing the maximum value from item 1 to n with total capacity m.
Let the possible decisions for item 1 (x1) be 0 (exclude) or 1 (include). Then, from the principle of optimality, the recursive relation is:

$g_0(m) = \max\{g_1(m), g_1(m - w_1) + p_1\}$

Where:

- $g_1(m)$ is the value if item 1 is excluded,
- $g_1(m - w1) + p1$ is the value if item 1 is included.

This recursive formulation allows dynamic programming to compute the optimal solution efficiently.

Equation Summary

KNAP(1, n, m)
$g_0(m) = \max\{g_1(m), g_1(m - w_1) + p_1\}$
Let gj(y) be the value of the optimal solution to KNAP(j + 1, n, y)

Example 11.9 – Shortest Path with an Intermediate Vertex

Consider the shortest path problem from a source vertex i to a destination vertex j. Suppose that a vertex k lies somewhere along this shortest path. Then, the full path from iii to j can be divided into two parts:

1. A shortest path from i to k
2. A shortest path from k to j

Let the complete path be:

$$i \to i_1 \to i_2 \to \cdots \to k \to p_1 \to p_2 \to \cdots \to j$$

In this case:

- The subpath$i \to i_1 \to i_2 \to \cdots \to k$must be the shortest path from i to k.
- The subpath$k \to p_1 \to p_2 \to \cdots \to j$must be the shortest path from k to j.

This is a direct application of the principle of optimality, which states that every subpath of a shortest path must itself be a shortest path.
To express this relationship mathematically

$$d(i, j) = d(i, k) + d(k, j)$$

Where:

- d(i,j) is the shortest distance from vertex i to vertex j,
- d(i,k)is the shortest distance from i to the intermediate vertex k.
- d(k,j)is the shortest distance from k to j.

This equation holds true only if vertex k lies on the shortest path from i to j, confirming the path can be split into two optimally solved subproblems.

Example 11.10

Let the solution to the 0/1 Knapsack problem be represented by a binary sequence:

$$y1,\ y2,\ \ldots,\ yn$$

where each yi∈{0,1} indicates whether item i is included (1) or excluded (0) from the optimal solution.
Now consider any index j, where 1≤j<n. The sequence can be split into two subsequences:
1.The prefix y1, y2, …, yj must be the optimal solution to the subproblem:

$$\text{KNAP}\left(1,\ j,\ \sum_{i=1}^{j} w_i y_i\right)$$

That is, selecting items from 1 to j with total weight equal to the actual selected weight

$$\sum_{i=1}^{j} w_i y_i.$$

2.The suffix y_{j+1}, y_{j+2}, …, $y_n$must be the optimal solution to the subproblem:

$$\text{KNAP}\left(j+1,\ n,\ m-\sum_{i=1}^{j} w_i y_i\right)$$

This means selecting items from j+1 to n using the remaining capacity after the first j items are considered.

Implication for Recurrence

This decomposition of the optimal solution supports the principle of optimality, which states that any subsequence of an optimal decision sequence must itself be optimal for its corresponding subproblem.
Hence, the recurrence relation for the maximum value $g_i(y)$, obtainable from items i+1 through nnn with capacity y, is given by:

$$g_i(y) = \max\{g_{i+1}(y),\ g_{i+1}(y - w_{i+1}) + p_{i+1}\}$$

Where:

- $g_{i+1}(y)$ represents the value when item $i+1$ is **excluded**,
- $g_{i+1}(y - w_{i+1}) + p_{i+1}$ represents the value when item $i+1$ is **included**, assuming $y \geq w_{i+1}$.

Base Case

The recurrence is anchored with the following base condition:

$$g_n(y) = 0, \quad \text{for all } y \geq 0$$

$$g_n(y) = -\infty, \quad \text{for all } y < 0$$

These reflect:
No value can be obtained when no items are left to choose (i.e., from index n+1 onward),
Capacity cannot be negative.

By solving the recurrence from i =n−1 down to i =0, the optimal value for the entire problem is given by:

$$g_0(m)$$

This value represents the maximum total value that can be obtained by selecting items from the full set 1 to n without exceeding the knapsack capacity m.

11.3 0/1 Knapsack Problem

Given:

- Number of items, n = 3
- Weights: $w_1 = 2$, $w_2 = 3$, $w_3 = 4$
- Profits: $p_1 = 1$, $p_2 = 2$, $p_3 = 5$
- Capacity, m = 6

Objective: Compute $g_0(6)$, the maximum profit that can be obtained with a knapsack of capacity 6.

Step 1: Compute $g_3(y)$ (Base Case)
Since no items are left after item 3, the base case is:

- $g_3(y) = 0$, for all $y \geq 0$
- $g_3(y) = -\infty$, for all $y < 0$

Step 2: Compute $g_2(6)$ and $g_2(3)$
$g_2(6) = \max \{ g_3(6), g_3(6 - w_3) + p_3 \}$
$g_2(6) = \max \{ g_3(6), g_3(2) + 5 \}$
$g_2(6) = \max \{ 0, 0 + 5 \} = 5$
$g_2(3) = \max \{ g_3(3), g_3(3 - 4) + 5 \}$
$g_2(3) = \max \{ 0, g_3(-1) + 5 \}$
$g_2(3) = \max \{ 0, -\infty \} = 0$

Step 3: Compute $g_1(6)$ and $g_1(4)$
$g_1(6) = \max \{ g_2(6), g_2(6 - w_2) + p_2 \}$
$g_1(6) = \max \{ g_2(6), g_2(3) + 2 \}$
$g_1(6) = \max \{ 5, 0 + 2 \} = 5$
Now compute $g_1(4)$. For that, first compute $g_2(4)$ and $g_2(1)$:

$g_2(4) = \max \{ g_3(4), g_3(4 - 4) + 5 \}$
$g_2(4) = \max \{ 0, 0 + 5 \} = 5$
$g_2(1) = \max \{ g_3(1), g_3(1 - 4) + 5 \}$
$g_2(1) = \max \{ 0, -\infty \} = 0$
Now compute:
$g_1(4) = \max \{ g_2(4), g_2(1) + 2 \}$
$g_1(4) = \max \{ 5, 0 + 2 \} = 5$

Step 4: Compute $g_0(6)$

$g_0(6) = \max \{ g_1(6), g_1(6 - w_1) + p_1 \}$
$g_0(6) = \max \{ g_1(6), g_1(4) + 1 \}$
$g_0(6) = \max \{ 5, 5 + 1 \} = 6$

Final Answer:

$g_0(6) = 6$

Example 11.11

Shortest Path (Backward Construction Using Principle of Optimality)
Let P_j be the set of vertices that are adjacent to vertex j, i.e., there is a direct edge from each vertex k in P_j to j. Formally,
$k \in P_j$ if and only if $(k, j) \in E(G)$
For each vertex $k \in P_j$, let F_k be the shortest path from vertex i to vertex k.
According to the principle of optimality, the shortest path from vertex i to vertex j is the shortest among all possible paths formed by going from i to some vertex k (where $k \in P_j$), followed by the edge (k, j).

In mathematical form:
Shortest path from i to j = shortest of all paths $\{F_k$ followed by $(k, j)\}$, where $k \in P_j$

Explanation

To derive this formulation, the process begins at the destination vertex j, and examines the last decision made in the path. That decision must involve using one of the incoming edges to j, i.e., an edge of the form (k, j), where $k \in P_j$.
Thus, the problem is solved by working backward from vertex j, identifying all possible predecessors k, and choosing the one that minimizes the total path cost from i to j.

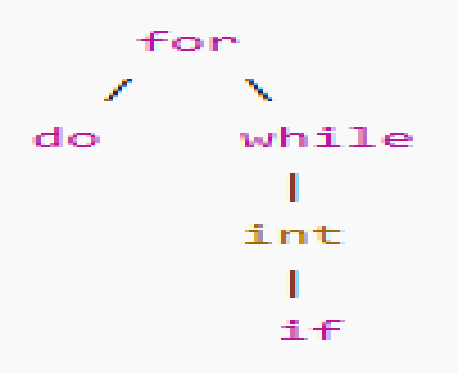
11.4 OPTIMAL BINARY SEARCH TREES

An optimal binary search tree (OBST) is a binary search tree designed to minimize the expected cost of searching for elements (either successfully or unsuccessfully), given known probabilities of access.

Let's say a fixed set of sorted identifiers is:
{for, do, while, int, if}

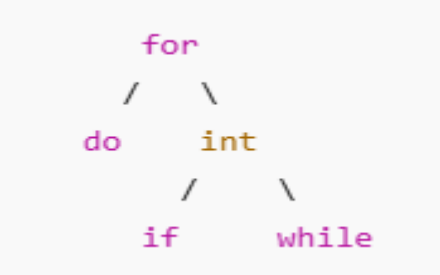
Different binary search trees can be constructed from the same set of identifiers. However, not all trees perform equally in terms of search efficiency. An optimal tree is the one that gives the minimum average search time, based on how often each identifier is searched.

Tree (a) Structure:



- Here, the search path for some elements is longer.
- For example:
  - for: 1 comparison
  - do: 2 comparisons
  - while: 2 comparisons
  - int: 3 comparisons
  - if: 4 comparisons
- **Average comparisons = (1 + 2 + 2 + 3 + 4) / 5 = 12 / 5 = 2.4**

Tree (b) Structure:



- The search paths are more balanced.
- For example:
  - for: 1 comparison
  - do: 2 comparisons
  - int: 2 comparisons
  - if: 3 comparisons
  - while: 3 comparisons
- **Average comparisons = (1 + 2 + 2 + 3 + 3) / 5 = 11 / 5 = 2.2**

So, tree (b) is more efficient on average.
Tree (a) is a less optimal tree with longer search paths for some elements, while Tree (b) is a more optimal tree that provides shorter and more balanced search paths.

External Nodes & Search Failures
To also account for unsuccessful searches (searches for values not in the tree), the tree is augmented with external nodes (represented as squares in the second diagram).

- Each external node is placed where a pointer to a child is missing.

- There are n + 1 external nodes if there are n internal (actual data) nodes.
- External nodes represent the locations where a failed search would end.

Expected Cost Calculation

Let:

- p(i) be the probability of successfully searching for identifier a_i
- q(i)be the probability of unsuccessful searches falling between $a_i$anda_{i+1}

The expected cost E of a binary search tree is given by:

$E = \Sigma [p(i) \times level(a\_i)] + \Sigma [q(i) \times (level(E\_i) - 1)]$

Where:

- level(a_i) is the depth of the node containing identifier a_i (starting at 1)
- level(E_i) is the level of the external node (representing failure after a_i)

Goal of OBST

The goal is to arrange nodes such that the expected cost E is minimized. This requires considering:

- How often each identifier is searched
- How often a search fails between identifiers
- The resulting depth of each node in the tree

The Optimal Binary Search Tree is the one that minimizes the total expected cost using the formula above.

- Tree (b) is optimal compared to tree (a) because it requires fewer comparisons on average.
- External nodes in the second set of diagrams allow modeling unsuccessful searches.
- OBST construction is important when search probabilities are known.
- The structure of the tree must balance depth and frequency of access for both successful and unsuccessful searches.

Example Problem

Let

p(1:4) = (3, 3, 1, 1)
q(0:4) = (2, 3, 1, 1, 1)

These values represent the successful search probabilities (p) and unsuccessful search probabilities (q) for the identifiers (do, if, int, while).

Note: The p's and q's have been multiplied by 16 for convenience.

Given:

Identifiers (sorted):

- do, if, int, while → Let these be: a_1 = do, a_2 = if, a_3 = int, a_4 = while

Probabilities:

1. Successful search probabilities (p(i)) for i = 1 to 4 (multiplied by 16):
   i. p(1:4) = (3, 3, 1, 1)
2. Unsuccessful search probabilities (q(i)) for i = 0 to 4 (also multiplied by 16):
   q(0:4) = (2, 3, 1, 1, 1)

These values will be used in the OBST dynamic programming formulation.

Definitions of Terms:

Let's define:

- w(i, j): Sum of probabilities from i+1 to j (i.e., total weight between i and j)

$$w(i,j) = \sum_{k=i+1}^{j} p(k) + \sum_{k=i}^{j} q(k)$$

- c(i, j): Minimum cost of OBST that stores keys a_{i+1} to a_j
- r(i, j): Root index k that gives minimum cost for subproblem (i, j)

Initialization:
For all i:

- w(i,i)=q(i)
- c(i,i)=0
- r(i,i)=0

So:

i	w(i,i)	c(i,i)	r(i,i)
0	2	0	0
1	3	0	0
2	1	0	0
3	1	0	0
4	1	0	0

Using Equation (from OBST Algorithm):

$$c(i,j) = w(i,j) + \min_{i<k\leq j} (c(i,k-1) + c(k,j))$$

Let us now compute the cost and root for subtrees of size 1, that is j= i +1:
Step-by-Step Calculations:
1. For (i = 0, j = 1):

$$w(0,1) = p(1) + q(0) + q(1) = 3 + 2 + 3 = 8$$

$$c(0,1) = w(0,1) + \min(c(0,0) + c(1,1)) = 8 + (0 + 0) = 8$$

$$r(0,1) = 1 \quad \text{(only choice)}$$

2. For (i = 1, j = 2):

$$w(1,2) = p(2) + q(1) + q(2) = 3 + 3 + 1 = 7$$

$$c(1,2) = w(1,2) + \min(c(1,1) + c(2,2)) = 7 + (0 + 0) = 7$$

$$r(1,2) = 2$$

3. For (i = 2, j = 3):

$$w(2,3) = p(3) + q(2) + q(3) = 1 + 1 + 1 = 3$$

$$c(2,3) = w(2,3) + \min(c(2,2) + c(3,3)) = 3 + (0 + 0) = 3$$

$$r(2,3) = 3$$

4. For (i = 3, j = 4):

$$w(3,4) = p(4) + q(3) + q(4) = 1 + 1 + 1 = 3$$

$$c(3,4) = w(3,4) + \min(c(3,3) + c(4,4)) = 3 + (0 + 0) = 3$$

$$r(3,4) = 4$$

Using All These to Build OBST

Now that we know:

- Which keys produce minimum cost
- Which index acts as root for each sub-tree

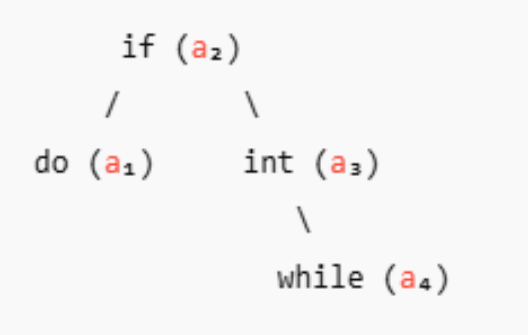
The final OBST is built using the r(i,j) values.
From the bottom of the matrix:

- For the full tree from a_1 to a_4 (i=0, j=4),

r(0,4) = 2 → Root is a_2 (if)

- Then recursively:
  - Left subtree (i=0, j=1): r(0,1) = 1 → Root is a_1 (do)
  - Right subtree (i=2, j=4): r(2,4) = 3 → Root is a_3 (int)
  - Right child: a_4 (while) from r(3,4)

Final OBST Structure



OBST reduces average search cost by arranging keys based on frequency (probability).
Dynamic programming matrices (w, c, r) are filled bottom-up.
The tree with the minimum expected cost is built using root decisions stored in r(i,j).

11.5 STRING EDITING

String editing (also called edit distance or string alignment) is a classic problem in computer science where the goal is to transform one string into another using a sequence of edit operations. Each operation has an associated cost, and the objective is to minimize the total cost.
This is often used in:

- Spelling correction
- DNA sequence analysis
- Text similarity (e.g., in plagiarism checkers)

Basic Edit Operations

1. Insertion – Add a character
2. Deletion – Remove a character
3. Substitution – Replace one character with another

Each operation may have a cost. For example:

- Cost of insert = 1
- Cost of delete = 1
- Cost of substitute = 2

You can also define different costs based on the problem.

Example 1: Converting "cat" to "cut"
Let:
- Source string: **"cat"**
- Target string: **"cut"**

To convert "cat" → "cut", only one operation is needed:
- Substitute 'a' with 'u'

Cost = 1 substitution = 2
Minimum edit distance = 2

Example 2: Converting "sunday" to "saturday"
Let:
- Source string: "sunday"
- Target string: "saturday"

One possible way:
1. Insert 'a' after 's' → saunday (cost = 1)
2. Insert 't' after 'sa' → satunday (cost = 1)
3. Replace 'n' with 'r' → saturday (cost = 2)

Total cost = 1 + 1 + 2 = 4
Minimum edit distance = 4
Dynamic Programming Approach
Let:
- A = string of length m
- B = string of length n

Define dp[i][j] as the minimum edit distance to convert the first i characters of A to the first j characters of B.

Recurrence relation:

If A[i] == B[j]
dp[i][j] = dp[i-1][j-1]
Else:
dp[i][j] = min(
  dp[i-1][j] + cost of deletion, // delete A[i]
  dp[i][j-1] + cost of insertion, // insert B[j]
  dp[i-1][j-1] + cost of substitution // replace A[i] with B[j]
)

Base cases:
dp[0][j] = j * insertion cost
dp[i][0] = i * deletion cost
Applications of String Editing
- **Spell checkers** (finding the closest dictionary word)
- **Bioinformatics** (comparing DNA sequences)
- **Natural Language Processing** (string similarity, fuzzy matching)
- **Version control systems** (comparing file differences)

String editing is a fundamental problem that uses dynamic programming to solve real-world tasks like spelling correction and sequence alignment. By defining costs for insertion, deletion, and substitution, the algorithm finds the most efficient way to transform one string into another.

11.7 SUMMARY

Dynamic Programming (DP) is a problem-solving approach that breaks complex optimization problems into simpler subproblems and reuses their results to build the overall solution efficiently.It is applicable to problems that satisfy the principle of optimality, such as shortest path, knapsack, OBST, and string editing.Unlike greedy algorithms, DP ensures global optimality by systematically considering all decision sequences.Its use of recurrence relations, memoization, and tabulation allows solving problems with overlapping subproblems in polynomial time.

11.8 KEY TERMS

Dynamic Programming , Principle of Optimality , Recurrence Relation , Knapsack , Shortest Path , Optimal BST , Edit Distance, Memoization, Bottom-Up Approach

11.9 REVIEW QUESTIONS

1. Define dynamic programming and explain its difference from the greedy approach.
2. What is the principle of optimality? Explain with an example.
3. Derive the recurrence relation for the 0/1 Knapsack problem.
4. Explain how the principle of optimality applies to the shortest path problem.
5. Describe the process of constructing an Optimal Binary Search Tree (OBST).
6. Write the recurrence relation for the string editing problem.

11.10 SUGGESTIVE READINGS

1. Horowitz, E., Sahni, S., & Rajasekaran, S. *Fundamentals of Computer Algorithms*, University Press.
2. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. *Introduction to Algorithms*, MIT Press.
3. Aho, A. V., Hopcroft, J. E., & Ullman, J. D. *Design and Analysis of Computer Algorithms*, Addison-Wesley.
4. Kleinberg, J., & Tardos, É. *Algorithm Design*, Pearson Education.

Dr. Vasantha Rudramalla

LESSON-12

GRAPH OPTIMIZATION

The objectives of this lesson are to:

- Understand the concept and structure of multistage graphs.
- Apply dynamic programming to solve shortest path problems in multistage graphs.
- Use Floyd-Warshall algorithm for computing shortest paths between all pairs of nodes.
- Solve the single-source shortest path problem using Dijkstra's and Bellman-Ford algorithms.
- Analyze the use of these techniques in practical optimization problems.

STRUCTURE

12.1 INTRODUCTION

In computer science, many real-world problems can be represented as graphs where nodes represent entities and edges represent relationships or transitions with associated costs. Optimization in such graphs often requires finding the most efficient way to travel from one node to another or to compute shortest paths across multiple nodes. This lesson explores advanced graph-based optimization techniques using Dynamic Programming (DP). It introduces multistage graphs for structured path finding, applies the Floyd-Warshall algorithm for computing all-pairs shortest paths, and discusses the single-source shortest path problem using both Dijkstra’s and Bellman-Ford algorithms. Through definitions, worked examples, and visual illustrations, learners will gain a deep understanding of how to model and solve optimization problems in graphs using algorithmic strategies.

12.2 MULTISTAGE GRAPHS

A multistage graph is a specialized form of a directed and weighted graph designed specifically for solving optimization problems, most notably those involving the computation of shortest paths **or** minimum-cost paths. In this structure, the entire set of vertices is divided into multiple stages**,** where each stage represents a specific step or level in a process, such as a time interval, a phase in a workflow, or a decision point in a sequence.

Within a multistage graph**,** nodes (vertices) are organized such that all edges are directed from one stage to the next**,** meaning a node in stage *i* can only connect to a node in stage *i+1*. This directional constraint ensures that paths proceed in a forward progression through the stages, avoiding cycles or backward traversal. Each edge carries a weight or cost, representing some measurable quantity such as distance, time, or expense incurred when moving from one node to another.

The primary objective when using a multistage graph is to identify the path from the starting node (source) in the first stage to the terminal node (destination) in the final stage that minimizes the total cost**.** This structured nature of multistage graphs makes them particularly well-suited for applying dynamic programming techniques, as the problem can be systematically broken down into smaller subproblems**,** where the optimal solution to each stage contributes to the overall optimal path.

12.2.1 DEFINITION AND CHARACTERISTICS

A multistage graph is a directed graph G=(V,E), where the set of verticesV is partitioned into k stages (where k≥2), such that:

- There is a source node in stage 1 and a destination node in stage k.
- All edges go from a node in stage iii to a node in stage i+1.
- Each edge has a weight or cost associated with it.

Key Characteristics:

1. **Stages:** Vertices are grouped into stages (e.g., Stage 1, Stage 2, ..., Stage k).
2. **Directed Edges:** Edges are only between consecutive stages, i.e., from Stage iii to Stage i+1i+1i+1.
3. **Weights:** Each edge has a cost (e.g., distance, time, etc.).
4. **Goal:** Typically, to find the minimum cost path from the source node (stage 1) to the destination node (stage k).

The multistage graph is primarily used in Dynamic Programming to find optimal paths through stages, as it allows breaking down a problem into subproblems.

12.2.2 Example with Stage-wise Cost Computation

Imagine a travel route where you need to move through several cities from start to end, and each city falls into a different stage (based on time, position, or process level). A multistage graph can model the problem of choosing the path with minimum cost (e.g., least travel time or expense) across these cities.

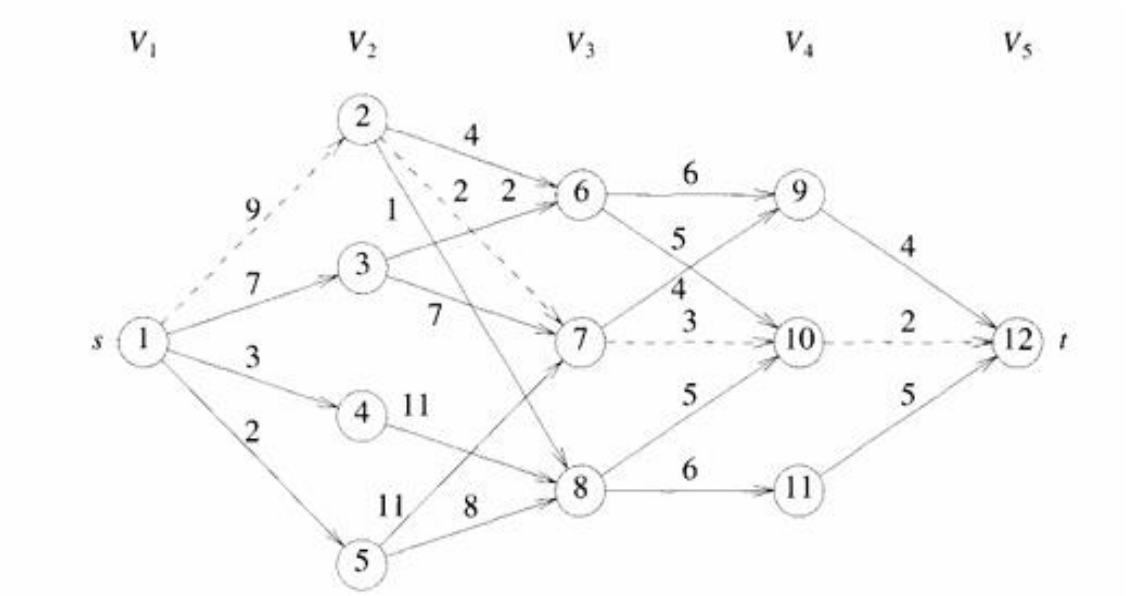


Figure 12.1 Five-stage graph

The provided graph is a multistage directed graph, divided into 5 stages V1,V2,V3,V4,V5 ,where:

- V1={1}: Start node S
- V2={2,3,4,5}
- V3={6,7,8}
- V4={9,10,11}
- V5={12}: End node t

Each edge (i,j) has a cost c(i,j), and the aim is to compute the minimum cost path from the source node S to the destination node t.
Let:

- cost(i, j) = minimum cost of reaching the destination t from node j in stage i

The recurrence relation is:

$$\text{cost}(i,j) = \min_{(j,l)\in E} \{c(j,l) + \text{cost}(i+1,l)\}$$

Where:

- c(j,l) is the cost of the edge from node j to node l.
- cost(k,t)=0 because cost to destination from destination itself is 0

Stage-wise Cost Calculation Using the Graph
We compute from last stage to first (backward traversal):
Stage V_4 to V_5 (i = 4)
Destination is node 12, so:

- cost(4,9)=c(9,12)=4
- cost(4,10)=c(10,12)=2
- cost(4,11)=c(11,12)=5

Stage V_3 to V_4 (i = 3)
Now use:

$$\text{cost}(3, j) = \min\{c(j, l) + \text{cost}(4, l)\}$$

- $\text{cost}(3, 6) = \min\{6 + \text{cost}(4, 9), 5 + \text{cost}(4, 10)\} = \min\{6 + 4, 5 + 2\} = \min\{10, 7\} = 7$
- $\text{cost}(3, 7) = \min\{4 + \text{cost}(4, 9), 3 + \text{cost}(4, 10)\} = \min\{4 + 4, 3 + 2\} = \min\{8, 5\} = 5$
- $\text{cost}(3, 8) = \min\{5 + \text{cost}(4, 10), 6 + \text{cost}(4, 11)\} = \min\{5 + 2, 6 + 5\} = \min\{7, 11\} = 7$

Stage V_2 to V_3 (i = 2)

- $\text{cost}(2, 2) = 4 + \text{cost}(3, 6) = 4 + 7 = 11$
- $\text{cost}(2, 3) = \min\{2 + \text{cost}(3, 6), 2 + \text{cost}(3, 7)\} = \min\{2 + 7, 2 + 5\} = \min\{9, 7\} = 7$
- $\text{cost}(2, 4) = 11 + \text{cost}(3, 7) = 11 + 5 = 16$
- $\text{cost}(2, 5) = \min\{8 + \text{cost}(3, 8), 11 + \text{cost}(3, 8)\} = \min\{8 + 7, 11 + 7\} = \min\{15, 18\} = 15$

Stage V_1 to V_2 (i = 1)

- $\text{cost}(1, 1) = \min\{9 + \text{cost}(2, 2), 7 + \text{cost}(2, 3), 3 + \text{cost}(2, 4), 2 + \text{cost}(2, 5)\}$

$= \min\{9 + 11, 7 + 7, 3 + 16, 2 + 15\} = \min\{20, 14, 19, 17\} = 14$

Final Result

- Minimum cost from source (node 1) to destination (node 12): 14
- Optimal path can be backtracked from computed values.

The diagram illustrates a multistage graph, where using dynamic programming (forward or backward approach), the minimum cost path from source to destination is computed efficiently. Each stage reduces the problem into smaller subproblems, enabling optimal solutions through recursive cost evaluations.

```
Algorithm FGraph(G, k, n, p)
// The input is a k-stage graph G = (V, E) with n vertices
// indexed in order of stages. E is a set of edges and c[i, j]
// is the cost of ⟨i, j⟩. p[1 : k] is a minimum-cost path.
{
    cost[n] := 0.0;
    for j := n − 1 to 1 step −1 do
    { // Compute cost[j].
        Let r be a vertex such that ⟨j, r⟩ is an edge
        of G and c[j, r] + cost[r] is minimum;
        cost[j] := c[j, r] + cost[r];
        d[j] := r;
    }
    // Find a minimum-cost path.
    p[1] := 1; p[k] := n;
    for j := 2 to k − 1 do p[j] := d[p[j − 1]];
}
```

12.3 All Pairs Shortest Paths

The All-Pairs Shortest Path (APSP) problem aims to find the shortest path distances between every pair of vertices in a weighted graph. For a graph G=(V,E) with n vertices, the task is to compute the minimum cost to go from every vertex u to every vertex v.

Applicable Graph Type:

- Directed or undirected.
- Edge weights can be positive, zero, or negative.
- Negative-weight cycles are not allowed**.**

12.3.1 Floyd-Warshall Algorithm

This is a classic dynamic programming algorithm to solve the APSP problem efficiently for dense graphs.

Notation:

Let:

- $A^{(k)}[i][j]$: Shortest distance from vertex i to vertex j, considering only vertices {1, 2, ….,k} as intermediate points.

Recurrence Relation:

$$A^{(k)}[i][j] = \min\left(A^{(k-1)}[i][j],\ A^{(k-1)}[i][k] + A^{(k-1)}[k][j]\right)$$

Initial Condition (k = 0):

$$A^{(0)}[i][j] = \begin{cases} 0 & \text{if } i = j \\ w(i,j) & \text{if there is an edge from } i \text{ to } j \\ \infty & \text{if no edge exists from } i \text{ to } j \end{cases}$$

Where:

- w(i,j) is the direct edge weight from node i to node j.
- ∞ denotes no direct connection between i and j.

Algorithm Steps:

1. Initialize the matrix $A^{(0)}$ with edge weights.
2. For each vertex k from 1 to n, update:

$$A^{(k)}[i][j] = \min\left(A^{(k-1)}[i][j],\ A^{(k-1)}[i][k] + A^{(k-1)}[k][j]\right)$$

Time and Space Complexity:

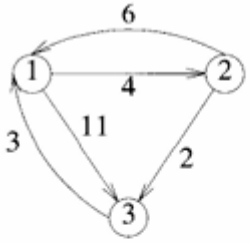
- **Time:** $O(n^3)$
- **Space:** $O(n^2)$

Matrix A[i][j] gives the shortest path cost from vertex i to vertex j.

```
Algorithm AllPaths(cost, A, n)
// cost[1 : n, 1 : n] is the cost adjacency matrix of a graph with
// n vertices; A[i, j] is the cost of a shortest path from vertex
// i to vertex j. cost[i, i] = 0.0, for 1 ≤ i ≤ n.
{
    for i := 1 to n do
        for j := 1 to n do
            A[i, j] := cost[i, j]; // Copy cost into A.
    for k := 1 to n do
        for i := 1 to n do
            for j := 1 to n do
                A[i, j] := min(A[i, j], A[i, k] + A[k, j]);
}
```

Function to compute lengths of shortest paths

12.3.2 Example with Step by Step Matrices



(a) Example digraph

A^0	1	2	3
1	0	4	11
2	6	0	2
3	3	∞	0

(b) A^0

A^1	1	2	3
1	0	4	11
2	6	0	2
3	3	7	0

(c) A^1

A^2	1	2	3
1	0	4	6
2	6	0	2
3	3	7	0

(d) A^2

A^3	1	2	3
1	0	4	6
2	5	0	2
3	3	7	0

(e) A^3

Figure 12.2: Application of Floyd-Warshall Algorithm on a Directed Graph

The above figure shows how the Floyd-Warshall Algorithm is applied to find the shortest path between all pairs of vertices in a directed graph (digraph), using a dynamic programming approach with intermediate matrices A^0,A^1,A^2,A^3.

1. Graph Overview (a) Example Digraph

Vertices: 1, 2, 3

Edges and weights:

- $1 \rightarrow 2 = 4$
- $1 \rightarrow 3 = 11$
- $2 \rightarrow 1 = 6$
- $2 \rightarrow 3 = 2$
- $3 \rightarrow 1 = 3$

2. Initial Matrix A^0(b)

This is the adjacency matrix of direct edge weights.

From\To	1	2	3
1	0	4	11
2	6	0	2
3	3	∞	0

$A^0[i][j]$=w(i,j) if edge exists; otherwise ∞
Diagonal entries are 0 (distance to itself)

3. Matrix A^1(c)
We consider **vertex 1** as an intermediate vertex.

- Formula:

$$A^1[i][j] = \min(A^0[i][j],\ A^0[i][1] + A^0[1][j])$$

Let's apply for some entries:
$A^1[3][2] = \min(\infty, A^0[3][1] + A^0[1][2]) = \min(\infty, 3 + 4) = 7$
Other values update only if shorter paths through vertex 1 are found.
Resulting matrix:

From\To	1	2	3
1	0	4	11
2	6	0	2
3	3	7	0

4. Matrix A^2 (d)
We now consider vertices 1 and 2 as intermediate nodes.
Formula:

$$A^2[i][j] = \min(A^1[i][j],\ A^1[i][2] + A^1[2][j])$$

Examples:

$$A^2[1][3] = \min(11, A^1[1][2] + A^1[2][3]) = \min(11, 4 + 2) = 6$$

Final matrix:

From\To	1	2	3
1	**0**	**4**	**6**
2	**6**	**0**	**2**
3	**3**	**7**	**0**

5. Matrix A^3 (e)
We now consider all three vertices (1, 2, 3) as possible intermediate vertices.

- Formula:

$$A^3[i][j] = \min(A^2[i][j],\ A^2[i][3] + A^2[3][j])$$

Example:

$$A^3[2][1] = \min(6, A^2[2][3] + A^2[3][1]) = \min(6, 2 + 3) = 5$$

From\To	1	2	3
1	**0**	**4**	**6**
2	**5**	**0**	**2**
3	**3**	**7**	**0**

Final Result:

The matrix A^3 gives the shortest path distances between all pairs of vertices in the graph.
The algorithm is summarized as below.

- Start with the adjacency matrix A^0.
- Use dynamic programming to update paths considering one more vertex at each step.
- The result after all iterations A^ngives the final shortest path between all pairs.
- Floyd-Warshall algorithm works even if the graph has negative edges (but not negative cycles).

12.4 SINGLE-SOURCE SHORTEST PATHS: GENERAL WEIGHTS

The Single-Source Shortest Path (SSSP) problem is a common problem in graph theory where the goal is to find the shortest path from one specific starting point (called the source node) to every other node in the graph. The graph used in this problem is weighted, meaning each connection (or edge) between two nodes has a cost or value assigned to it. This cost could represent things like distance, time, or money. These edge weights can be positive, zero, or even negative, which is why the problem is said to involve general weights. The aim is to calculate the minimum total cost to reach each node from the source, taking into account all possible paths through the network.

12.4.1 PROBLEM DEFINITION AND APPLICATIONS

Types of Graphs Covered

- Directed or undirected graphs.
- Edge weights can be:
  - Positive (common case)
  - Zero
  - Negative (must be handled carefully)
- The graph must not contain negative weight cycles that are reachable from the source (because shortest path is undefined in such cases).

Applications

- Navigation systems (e.g., GPS shortest routes)
- Routing in communication networks
- Project scheduling with task durations and dependencies
- Financial systems to detect arbitrage opportunities (if negative edges exist)

Key Algorithms to Solve SSSP with General Weights
There are two primary algorithms depending on the type of edge weights:
A. Dijkstra's Algorithm (for non-negative weights)

- Works only with non-negative edge weights.
- Uses a greedy strategy and a priority queue (min-heap).
- Time Complexity:
  - $O(V^2)$ for a simple implementation.
  - $O((V+E)\log V)$ using a binary heap.
- Fast and efficient when all edge weights are ≥ 0.

B. Bellman-Ford Algorithm (for general weights including negatives)

- Handles negative edge weights.
- Detects negative weight cycles.
- Time Complexity: O(V·E)
- Relaxation is applied repeatedly for all edges.

12.4.2 Bellman-Ford Algorithm

Goal:

Compute dist[v]: the shortest distance from source s to each vertex v.

Initialization:

Set $\text{dist}[s] = 0$

Set $\text{dist}[v] = \infty$ for all $v \neq s$

For each edge (u,v) with weight w(u,v), update:

$$\text{if } \text{dist}[u] + w(u,v) < \text{dist}[v], \text{ then set } \text{dist}[v] = \text{dist}[u] + w(u,v)$$

Procedure:

- Repeat the relaxation step for all edges V−1 times (where V = number of vertices)
- After that, check for negative weight cycles by one more iteration: if further relaxation is possible, a negative cycle exists.

Example

Let the graph have 4 vertices and edges:

- $(1 \to 2, w = 4)$
- $(1 \to 3, w = 5)$
- $(2 \to 3, w = -3)$
- $(3 \to 4, w = 2)$

Apply Bellman-Ford from source S=1:

- Initially:

$$\text{dist}[1] = 0, \text{dist}[2] = \infty, \text{dist}[3] = \infty, \text{dist}[4] = \infty$$

After first iteration:

- $\text{dist}[2] = 4$
- $\text{dist}[3] = 5$
- Update via edge $(2 \to 3)$: $\text{dist}[3] = 1$
- Then $\text{dist}[4] = 3$

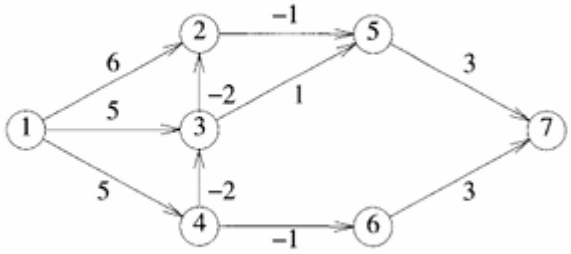
After 3 iterations, shortest paths from source to all other vertices are finalized.

Summary Table of Algorithms

Algorithm	Edge Weights	Time Complexity	Detects Negative Cycle
Dijkstra	Non-negative only	O((V+E)logV)	No
Bellman-Ford	General weights	O(V·E)	Yes

The Single-Source Shortest Path with General Weights is an essential graph problem in real-world applications. For non-negative weights, Dijkstra's algorithm is efficient and widely used. For graphs that may contain negative weights (but no negative cycles), the Bellman-

Ford algorithm provides a reliable solution. Understanding when and how to use each algorithm is critical for solving shortest path problems in various domains.



(a) A directed graph

	$dist^k[1..7]$						
k	1	2	3	4	5	6	7
1	0	6	5	5	∞	∞	∞
2	0	3	3	5	5	4	∞
3	0	1	3	5	2	4	7
4	0	1	3	5	0	4	5
5	0	1	3	5	0	4	3
6	0	1	3	5	0	4	3

(b) $dist^k$

Figure 12.3 Bellman-Ford Algorithm Applied on a Directed Graph with General Weights

The above diagram shows an application of the Bellman-Ford Algorithm on a directed graph with general weights (including negative edges) to find the shortest path from vertex 1 to all other vertices.

(a) Directed Graph

The graph consists of 7 vertices (1 to 7) with the following weighted directed edges (some weights are negative):

Edge	Weight
1 → 2	6
1 → 3	5
1 → 4	5
2 → 5	-1
3 → 2	-2
3 → 5	1
4 → 3	-2
4 → 6	-1
5 → 7	3
6 → 7	3

(b) Table of $dist^k[1..7]$ – Iterative Updates

This table shows how shortest path estimates from source vertex 1 to all other vertices evolve after each iteration k (relaxing all edges). It is a step-by-step log of the distance array maintained by the Bellman-Ford algorithm.

Let $dist^k[v]$ represent the shortest known distance from node 1 to v after k iterations.

Iteration 0 (Initial distances):

- Distance to source (1) = 0
- All other nodes = ∞

Vertex	1	2	3	4	5	6	7
k = 0	0	∞	∞	∞	∞	∞	∞

| k = 1 | 0 | 6 | 5 | 5 | ∞ | ∞ | ∞ |

Iteration 1:
Relax all edges once.

- 1 → 2: dist[2] = min(∞, 0 + 6) = 6
- 1 → 3: dist[3] = 5
- 1 → 4: dist[4] = 5

| k = 1 | 0 | 6 | 5 | ∞ | ∞ | ∞ |

Iteration 2:

- 1 →3 → 2: 5 + (–2) = 3 → dist[2] = 3
- 1 →3 → 5: 5 + 1 = 6 → dist[5] = 6
- 1 →4 → 3: 5 + (–2) = 3 → dist[3] = 3
- 1 →4 → 6: 5 + (–1) = 4 → dist[6] = 4

| k = 2 | 0 | 3 | 3 | 5 | 6 | 4 | ∞ |

Iteration 3:

- (1 →4 → 3) →2 : 3 + (-2) = dist[2] = 1
- (1 →3 → 2) → 5: 3 + (–1) = 2 → dist[5] = 2
- (1 →4 → 6) → 7: 4 + 3 = 7 → dist[7] = 7

| k = 3 | 0 | 1 | 3 | 5 | 2 | 4 | 7 |

Iteration 4:
No further updates to 1-5

- (1 →4 → 3→ 2 → 5) → 7: 0 + 3 = 3 (no change since 5 < 7)

| k = 4 | 0 | 1 | 3 | 5 | 0 | 4 | 3 |

Iteration 5:
No updates.
| k = 5 | 0 | 1 | 3 | 5 | 0 | 4 | 3 |

Iteration 6:
Stable again.
| k = 6 | 0 | 1 | 3 | 5 | 0 | 4 | 3 |
Bellman-Ford requires up to V−1=6 iterations for 7 vertices.
If no update occurs in an iteration, the algorithm may stop early.
After 6 iterations, all paths are finalized.

12.4 SUMMARY

Graph optimization techniques using dynamic programming are essential in solving real-world problems involving routing, scheduling, and resource allocation. Multistage graphs divide problems into stages for efficient backward evaluation. The Floyd-Warshall algorithm enables solving the all-pairs shortest path problem efficiently for dense graphs, while Bellman-Ford and Dijkstra's algorithms offer solutions for single-source shortest path problems depending on weight conditions. Understanding when to apply each technique is crucial for effective algorithm design and real-world system optimization.

12.5 KEY TERMS

Multistage Graph, Dynamic Programming, Floyd-Warshall, Bellman-Ford, Dijkstra's Algorithm, Shortest Path

12.6 REVIEW QUESTIONS

1. What is a multistage graph? Explain its key properties and use in dynamic programming.
2. Write and explain the recurrence relation used for solving shortest paths in multistage graphs.
3. How does the Floyd-Warshall algorithm solve the all-pairs shortest path problem? Illustrate with an example.
4. Compare and contrast Dijkstra's and Bellman-Ford algorithms. When would you use one over the other?
5. What are the time and space complexities of Floyd-Warshall and Bellman-Ford algorithms?
6. Describe a real-world application of graph optimization using dynamic programming.

12.7 SUGGESTIVE READINGS

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press.
2. Dasgupta, S., Papadimitriou, C. H., & Vazirani, U. (2008). Algorithms. McGraw-Hill.
3. Sedgewick, R., & Wayne, K. (2011). Algorithms (4th ed.). Addison-Wesley.
4. Kleinberg, J., & Tardos, E. (2005). Algorithm Design. Pearson Education.
5. Brassard, G., & Bratley, P. (1996). Fundamentals of Algorithmics. Prentice-Hall.

Dr. Vasantha Rudramalla

LESSON-13

COMPLEX DP PROBLEMS

OBJECTIVES

The objectives of this lesson are to:

- Understand the 0/1 Knapsack Problem and how dynamic programming is used to solve it.
- Learn the principles of system reliability optimization through Reliability Design.
- Explore the use of dynamic programming in solving the Travelling Salesman Problem (TSP).
- Analyze the Flow Shop Scheduling problem and how optimal job sequences are determined.
- Apply state-space and subset-based DP techniques for solving real-world optimization problems.

STRUCTURE

13.1 INTRODUCTION

This lesson explores advanced optimization problems that can be effectively solved using Dynamic Programming (DP) techniques. It covers classical NP-hard problems such as the 0/1 Knapsack Problem**,** Reliability Design**,** Travelling Salesman Problem (TSP)and Flow Shop Scheduling**.** These problems appear frequently in operations research, logistics, computer

networks, and manufacturing systems. Dynamic programming provides a systematic way to break down such problems into smaller subproblems, apply the principle of optimality, and build optimal solutions efficiently.

13.2 0/1KNAPSACK PROBLEM

The 0/1 Knapsack Problem is a classic optimization problem in computer science and operations research. It is called 0/1 because each item can either be included (1) or not included (0) in the knapsack — partial selection is not allowed.

It is typically solved using Dynamic Programming (DP) due to the problem's overlapping subproblems and optimal substructure properties.

13.2.1 PROBLEM STATEMENT

Given:

- A set of n items
- Each item i has:
  - a weight w_i
  - a value v_i
- A knapsack with maximum capacity W

OBJECTIVE:

Maximize the total value of selected items such that the total weight does not exceed the knapsack's capacity.

Decision Rule:
Include or exclude each item (0 or 1 choice).

Characteristics:

- 0/1 Constraint: Each item is chosen at most once.
- Discrete: No fractional values allowed (unlike Fractional Knapsack).
- NP-complete: Cannot be solved in polynomial time using brute-force for large inputs, but DP provides an efficient solution.

13.2.2 Dynamic Programming Approach

Let:

- S be the solution space
- P be the total profit
- W be the total weight
- Pi, Wi be the profit and weight of item i

Then, the state transition can be represented as:

$$S = \{(P, W) \mid (P - P_i, W - W_i) \in S\}$$

Subject to:

$$W \leq \text{Capacity}$$

This equation expresses that a current state (P,W) in the solution set S can be derived by including item i, provided the resulting state (P−Pi,W−Wi) is already a valid state in S.

- The constraint W≤Capacity ensures that we do not exceed the knapsack's limit.

This representation is part of backward state-space generation, where the process begins from the base case and builds the set of all reachable valid states.

13.2.3 State-Space Representation and Example

Given a 0/1 knapsack problem with the following parameters:

- Number of items: n=3
- Weights of items: w=(2,3,4)
- Profits of items: p=(1,2,5)
- Knapsack capacity: m=6

State-Space Representation

Let $S_k$denote the set of all feasible (profit, weight) pairs after considering the first k items. Each pair (P,W)$\in S_k$ represents a possible total profit P achievable with total weight W, using a subset of the first k items.

The construction of S_k_ follows a backward state-space generation approach, beginning from the base case:

$$S0=\{(0,0)\}$$

The objective is to determine the maximum profit that can be obtained by selecting a subset of items such that the total weight does not exceed the capacity mmm, and each item is either selected completely or not at all.

Step-by-Step Construction of Feasible States:
Step 1: After Item 1 (weight = 2, profit = 1)
New state generated from S_0:

$$(0+1, 0+2) = (1,2)$$

$$S_1 = \{(0,0), (1,2)\}$$

Step 2: After Item 2 (weight = 3, profit = 2)
From existing states in S_1, the following new states are added:

$$(0+2, 0+3) = (2,3), \quad (1+2, 2+3) = (3,5)$$

$$S_2 = \{(0,0), (1,2), (2,3), (3,5)\}$$

Step 3: After Item 3 (weight = 4, profit = 5)
New states generated:

$$(0+5, 0+4) = (5,4), \quad (1+5, 2+4) = (6,6), \quad (2+5, 3+4) = (7,7), \quad (3+5, 5+4) = (8,9)$$

$$S_3 = \{(0,0), (1,2), (2,3), (3,5), (5,4), (6,6), (7,7), (8,9)\}$$

Purging Rule:
The purging rule eliminates all pairs (P,W) where:

1. $W > m$, since such weights exceed the knapsack capacity and are irrelevant to the problem.
2. A pair (P_i, W_i) is dominated by another pair (P_j, W_j) if $W_j \leq W_i$ and $P_j \geq P_i$.

Applying the purging rule to S_3, the pairs $(7, 7)$ and $(8, 9)$ are removed because they violate the capacity constraint $W > 6$.

The pruned set becomes:

$$S_3 = \{(0, 0), (1, 2), (2, 3), (3, 5), (5, 4), (6, 6)\}$$

The maximum profit achievable within the knapsack capacity m=6 is determined by selecting the pair with the highest profit value such that W≤6 from S3. The optimal pair is (6,6), indicating that a total profit of 6 is achievable with a total weight of 6.

The 0/1 knapsack problem can be efficiently solved using backward state-space dynamic programming. Each state represents a feasible combination of total profit and weight for a subset of items. By applying pruning rules during the state generation process, infeasible and dominated solutions are discarded, thereby optimizing the computation. The final result corresponds to the highest profit among the valid pairs within the final state set Sn, where n is the total number of items.

13.3 RELIABILITY DESIGN

Reliability Design refers to the process of designing systems that are capable of performing their intended function consistently over time without failure. In computing and engineering, it specifically involves selecting and organizing components in such a way that the overall system reliability is maximized, often under certain constraints such as cost, weight, or power consumption.

In the context of Dynamic Programming, Reliability Design is treated as an optimization problem where the objective is to determine the most reliable configuration of components while meeting system-level constraints.

Reliability Design is widely used in areas such as:

- Computer hardware systems
- Communication networks
- Aircraft and automotive systems
- Power systems and control engineering

In such domains, failure of components can lead to critical system failures. Hence, designing for maximum reliability is essential.

Consider a system composed of n devices, connected in series.

Let r_i represent the reliability of device Di, i.e., the probability that the device works correctly. The overall system reliability in a series connection is the product of individual device reliabilities:

$$R_{\text{system}} = \prod_{i=1}^{n} r_i$$

Even if each device has high reliability (e.g., r_i=0.99), the system's overall reliability can drop significantly due to the multiplicative nature of the formula. For example:

If $n = 10$ and each $r_i = 0.99$, then:

$$R_{\text{system}} = (0.99)^{10} \approx 0.904$$

This shows that the system is less reliable than its individual components.To improve reliability, each stage (or device type) can have multiple copies of the same device, connected in parallel using a switching circuit. The switching mechanism selects a functioning device among the available ones.

- If stage i contains m_i copies of device Di, the probability that all fail is:

$$(1 - r_i)^{m_i}$$

- Thus, the reliability of stage i becomes:

$$\phi_i(m_i) = 1 - (1 - r_i)^{m_i}$$

In real systems, this may be slightly lower due to imperfect switching circuits or correlation between device failures.

The goal is to determine the optimal number of copies mim_imi to use for each device Di, such that the overall system reliability is maximized:

$$\text{Maximize} \quad \prod_{i=1}^{n} \phi_i(m_i)$$

The total cost does not exceed a given maximum C.Each unit of device i has a cost c_i, so the total cost is:

$$\sum_{i=1}^{n} c_i \cdot m_i \leq C$$

Additionally:

- $m_i \geq 1$
- $m_i$must be an integer (no partial devices allowed)

13.4 Travelling Salesman Problem

The Travelling Salesman Problem (TSP) is a classic combinatorial optimization problem. It involves a salesman who must visit n cities, starting from a designated city, visiting each city exactly once, and returning to the starting city, while minimizing the total travel cost.
TSP is known to be NP-hard, but it can be solved using dynamic programming for small values of nnn efficiently.

13.4.1 Problem Definition and Use Case

Given:
A set of n cities.

A cost matrix c_{ij}, where c_{ij} is the cost to travel from city i to city j.
The objective is to find the minimum cost tour that starts at city 1, visits every other city exactly once, and returns to city 1.
Dynamic Programming Formulation:
Let:
V be the set of all cities {1,2,...,n}
S⊂V: a subset of cities that includes city 1
g(i,S): the minimum cost of a path that starts at city iii, visits all cities in set SSS, and ends at city 1
Base Case:
If only two cities are involved, the direct cost from one to the other and back is:

$$g(i, \{1, i\}) = c_{i1}$$

Recursive Formula:

For $S \subseteq V$, $i \in S$, and $i \neq 1$:

$$g(i, S) = \min_{j \in S,\, j \neq i} \{c_{ji} + g(j, S - \{i\})\}$$

This formula states that the optimal path to city i from a set S can be computed by taking the minimum over all possible cities j that can reach i, plus the cost of an optimal path ending at j through the remaining cities S−{i}.

To compute the minimum cost tour starting and ending at city 1:

$$\text{Minimum Tour Cost} = \min_{2 \le k \le n} \{c_{k1} + g(k, V - \{1\})\}$$

This gives the total cost of completing the tour by returning from city k to the starting city 1.

The dynamic programming solution to the Travelling Salesman Problem builds the solution step-by-step by solving smaller subproblems involving subsets of cities. The approach ensures optimality by using the principle of optimality, and although the time complexity is exponential in n, it is significantly better than brute-force enumeration for small values of n.
Example

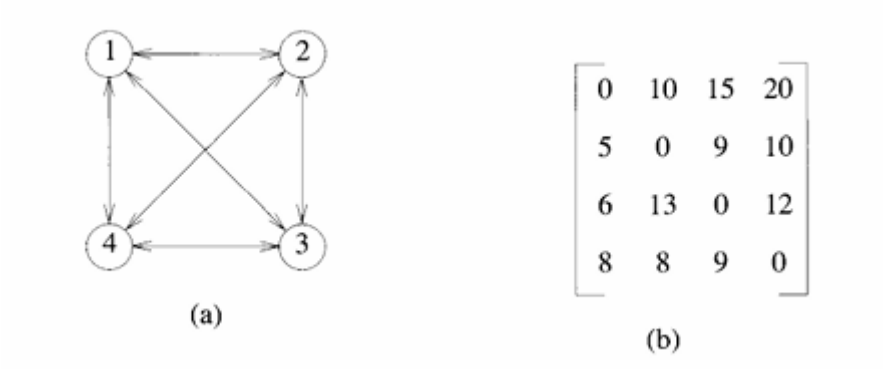


Figure 13.1 Complete Directed Graph

Figure (a) represents a complete directed graph with 4 cities (nodes), labeled 1 through 4. The objective is to find the minimum-cost tour that starts at city 1, visits every other city exactly once, and returns to city 1.

Figure (b) is the cost matrix where the entry at row i, column j represents the cost c_{ij} of traveling from city i to city j. Diagonal entries are 0 since there is no cost to travel from a city to itself.
Let:

- g(i,S): Minimum cost to start from city i, visit all cities in set S, and end at city 1.
- S: Subset of cities to be visited.
- The goal is to compute g(1,{2,3,4})

Step-by-Step Calculation:
Base Case – Visiting no other city (empty set):
From each city i ≠1, returning directly to city 1:

- g(2,∅)= C_{21}=5
- g(3,∅)=C_{31}=6
- g(4,∅)=C_{41}=8

First Level: Subsets of size 1

- $g(2,\{3\}) = c_{23} + g(3,\emptyset) = 15 + 6 = 21$
- $g(2,\{4\}) = c_{24} + g(4,\emptyset) = 10 + 8 = 18$
- $g(3,\{2\}) = c_{32} + g(2,\emptyset) = 13 + 5 = 18$
- $g(3,\{4\}) = c_{34} + g(4,\emptyset) = 12 + 8 = 20$
- $g(4,\{2\}) = c_{42} + g(2,\emptyset) = 8 + 5 = 13$
- $g(4,\{3\}) = c_{43} + g(3,\emptyset) = 9 + 6 = 15$

Second Level: Subsets of size 2

- $g(2,\{3,4\}) = \min \begin{cases} c_{23} + g(3,\{4\}) = 15 + 20 = 35 \\ c_{24} + g(4,\{3\}) = 10 + 15 = 25 \end{cases} = 25$
- $g(3,\{2,4\}) = \min \begin{cases} c_{32} + g(2,\{4\}) = 13 + 18 = 31 \\ c_{34} + g(4,\{2\}) = 12 + 13 = 25 \end{cases} = 25$
- $g(4,\{2,3\}) = \min \begin{cases} c_{42} + g(2,\{3\}) = 8 + 21 = 29 \\ c_{43} + g(3,\{2\}) = 9 + 18 = 27 \end{cases} = 27$

Final Step – Full Tour from City 1:

$$g(1,\{2,3,4\}) = \min \begin{cases} c_{12} + g(2,\{3,4\}) = 10 + 25 = 35 \\ c_{13} + g(3,\{2,4\}) = 15 + 25 = 40 \\ c_{14} + g(4,\{2,3\}) = 20 + 27 = 47 \end{cases}$$

Minimum Tour Cost:

$$\boxed{g(1,\{2,3,4\}) = 35}$$

This is the minimum cost to start at city 1, visit cities 2, 3, 4 in the optimal order, and return to city 1.

The above steps illustrate how the Travelling Salesman Problem can be solved using dynamic programming by breaking it into smaller subproblems. The solution utilizes the principle of

optimality and computes the minimum tour cost by recursively evaluating all subsets of cities and tracking the minimum cost path.

13.5 FLOW SHOP SCHEDULING

Flow Shop Scheduling is a type of production scheduling problem where a set of jobs must be processed on a set of machines in the same order. Each job consists of a sequence of tasks that must be completed by passing through the machines in a fixed, linear order.

This scheduling problem is particularly relevant in manufacturing systems where the production process is organized as a flow line.

Basic Characteristics:

- **Jobs ($J_1, J_2, ..., J_n$):** A finite set of tasks or items that need processing.
- **Machines ($M_1, M_2, ..., M_m$):** A fixed set of machines or workstations.
- **Order of processing:** Each job must be processed on all machines in the same sequence, usually from M_1 to Mm.
- **Processing time:** Each job Ji requires a specific time Pij on machine Mj.

OBJECTIVE:

The goal is typically to optimize a performance criterion, such as:

- Minimizing the makespan (total completion time of all jobs)
- Minimizing the total flow time
- Minimizing job tardiness or lateness
- Balancing machine workloads

Assumptions:

- Each machine can process only one job at a time.
- Each job can be processed by only one machine at a time.
- No preemption is allowed; once a job starts on a machine, it must complete before the next one begins.
- The processing order of machines is the same for all jobs.

Variants of Flow Shop Scheduling:

1. **Two-Machine Flow Shop:** The simplest case involving only two machines; solvable optimally using Johnson's Rule.
2. **Multi-Machine Flow Shop:** Involves more than two machines; often solved using heuristic methods or approximation algorithms.
3. **Permutation Flow Shop:** A restricted version where the same job sequence is used on all machines.

Applications:
Flow Shop Scheduling is commonly applied in:

- Automotive assembly lines
- Food processing plants

- Electronics and semiconductor manufacturing
- Textile and garment industries

Flow Shop Scheduling models a structured and constrained production environment where job sequences and machine usage are strictly ordered. It is an important class of scheduling problems in operations research and industrial engineering, often requiring efficient algorithms and heuristics to derive optimal or near-optimal solutions due to its combinatorial complexity.

Example
A Flow Shop Scheduling problem involves two jobs that need to be processed on three machines (processors), with each job required to follow the same sequence of machines:
M1→M2→M3
The processing time matrix J is given as:

$$J = \begin{bmatrix} 2 & 0 \\ 3 & 3 \\ 5 & 2 \end{bmatrix}$$

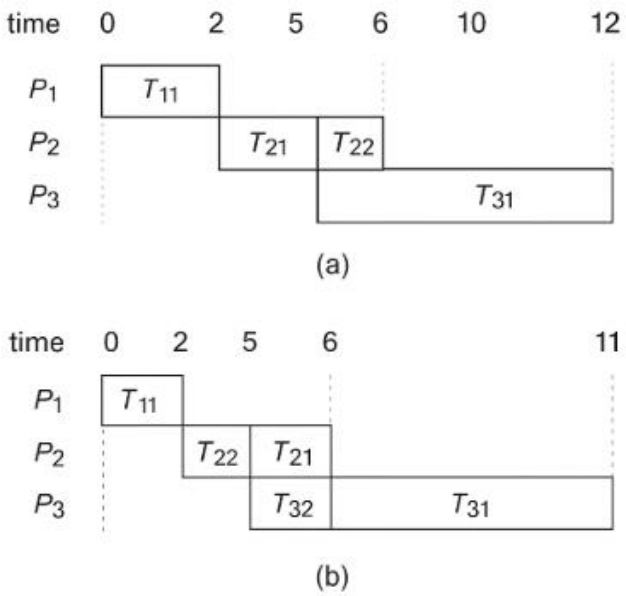


Figure13.2 Gantt Chart Representations for Two Job Sequences

The image illustrates two Gantt charts labeled (a) and (b), representing Flow Shop Scheduling for two jobs ($J_1$and J_2) across three processors or machines (P_1,P_2,P_3). Each task is denoted by Tij, where i is the machine number and j is the job number.
Schedule for Job Sequence $J_1 \rightarrow J_2$

Processor	Tasks (in order)
P_1	T_{11} (0–2)
P_2	T_{22} (0–3), T_{21} (3–5)
P_3	T_{31} (6–10), T_{32} (10–12)

- **Job 1** proceeds through:
  - P_1: T_{11} (0–2)
  - P_2: T_{21} (3–5)
  - P_3: T_{31} (6–10)
- **Job 2** proceeds through:
  - P_2: T_{22} (0–3)
  - P_1: waits until P_1 is free → executes T_{12} not shown (possibly 0 duration)
  - P_3: T_{32} (10–12)
- **Makespan** = 12 units

(b): Schedule for Job Sequence $J_2 \rightarrow J_1$

Processor	Tasks (in order)
P_1	T_{11} (0–2)
P_2	T_{22} (0–2), T_{21} (3–5)
P_3	T_{32} (2–5), T_{31} (6–11)

- **Job 2** proceeds through:
  - P_2: T_{22} (0–2)
  - P_3: T_{32} (2–5)
- **Job 1** proceeds through:
  - P_1: T_{11} (0–2)
  - P_2: T_{21} (3–5)
  - P_3: T_{31} (6–11)
- **Makespan** = 11 units

Chart (b) with job sequence $J_2 \rightarrow J_1$ produces a better schedule with a smaller makespan (11) compared to chart (a) with makespan 12. This example demonstrates how job ordering in flow shop scheduling can significantly impact total completion time.

13.6 SUMMARY

This lesson explored four major optimization problems:

- The0/1 Knapsack Problem, using backward state-space DP
- Reliability Design, optimizing component redundancy under cost constraints
- The Travelling Salesman Problem (TSP)**,** solved via recursive subset DP
- Flow Shop Scheduling, improving job sequences for reduced completion time

These examples demonstrate how dynamic programming handles combinatorially explosive problems by reusing subproblem solutions, applying constraints, and pruning infeasible paths.

13.7 KEY TERMS

Dynamic Programming, Knapsack Problem, State-Space, Reliability Design, Travelling Salesman Problem, Flow Shop Scheduling

13.8 REVIEW QUESTIONS

1. What is the 0/1 Knapsack Problem and how is it solved using dynamic programming?
2. Explain the difference between series and parallel systems in reliability design.
3. Describe the recursive formulation of the Travelling Salesman Problem.
4. What are the objectives and constraints of flow shop scheduling?
5. How does job sequence affect the makespan in flow shop scheduling?

13.9 SUGGESTED READINGS

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms(3rd ed.). MIT Press.
2. Ahuja, R. K., Magnanti, T. L., & Orlin, J. B. (1993). *Network Flows: Theory,* Algorithms, and Applications. Prentice-Hall.
3. Garey, M. R., & Johnson, D. S. (1979). Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman.
4. Pinedo, M. (2016). Scheduling: Theory, Algorithms, and Systems. Springer.

Mrs. Appikatla PushpaLatha

LESSON-14

TRAVERSAL AND SEARCH TECHNIQUES

OBJECTIVES

The objectives of this lesson are to:

- Understand the concept of tree and graph traversal as fundamental operations in data structures.
- Learn different methods for traversing binary trees, including inorder, preorder, and postorder techniques.
- Study graph traversal algorithms such as Breadth First Search (BFS) and Depth First Search (DFS).
- Explore how traversal methods are used to identify connected components and spanning trees in graphs.

STRUCTURE

14.1 INTRODUCTION

Traversal and search techniques are among the most essential operations in data structures. They provide systematic ways to access, visit, and process each element or node within hierarchical structures like trees and graphs. These operations form the foundation for many algorithms used in computer science, such as searching, sorting, pathfinding, and optimization. In a tree, traversal is used to visit every node in a structured order—whether starting from the root, exploring left or right subtrees, or processing children after their parents. Techniques such as inorder, preorder, and postorder traversals serve specific purposes, from expression evaluation to data sorting.

In a graph, traversal helps explore all vertices and edges efficiently. Two widely used techniques are Breadth First Search (BFS) and Depth First Search (DFS). These form the basis for advanced applications like shortest-path algorithms, connectivity checks, topological sorting, and network analysis. Understanding these traversal and search methods is fundamental to designing efficient algorithms, solving real-world problems, and building optimized systems in domains like artificial intelligence, networking, compiler design, and database management.

14.2 TRAVERSAL TECHNIQUES FOR BINARY TREES

A binary tree is a type of hierarchical data structure in which each node can have zero, one, or at most two children. These children are typically referred to as the left child and the right child. Binary trees are widely used in computer science for representing hierarchical relationships, organizing data for quick retrieval, and implementing expression parsers, decision-making structures, and more.

In binary trees, each node contains three components:

1. Data (the actual value stored in the node)
2. Pointer to the left child
3. Pointer to the right child

Traversal in a binary tree refers to the process of visiting each node in the tree exactly once, in a specific order, to perform some operation (such as printing data, searching for an element, or modifying values). This is a fundamental operation necessary for nearly every application involving binary trees.
There are three primary ways to traverse a binary tree. They are

- Inorder Traversal
- Preorder Traversal
- Postorder Traversal

14.2.1 Inorder Traversal (Left → Root → Right)

Inorder traversal is a type of depth-first traversal for binary trees. The traversal follows a specific sequence:

1. Visit the left subtree
2. Visit the root node
3. Visit the right subtree

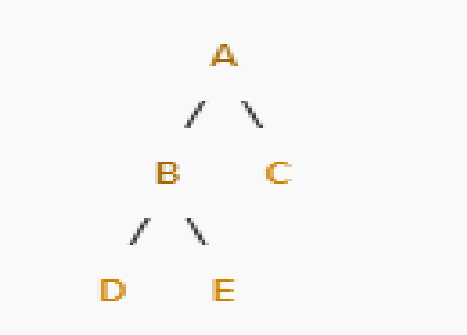
This systematic approach ensures that the nodes are accessed in a linear order, particularly useful when dealing with Binary Search Trees (BSTs). In a BST, Inorder traversal results in nodes being visited in ascending sorted order, as all values in the left subtree are less than the root and all values in the right subtree are greater.

Steps Involved in Inorder Traversal:
Given a node N, the traversal is performed as follows:

- Step 1: Recursively traverse the left subtree of node N.
- Step 2: Visit node N (i.e., process or print the data of the node).
- Step 3: Recursively traverse the right subtree of node N.

```
Inorder(Node)
{
If Node ≠ Null Then
Inorder(Node.Left)
Visit(Node.Data)
        Inorder(Node.Right)
}
```

Example:
Consider the binary tree structure below:



To perform an Inorder Traversal:
1.Traverse the left subtree of A → (B → D, E)

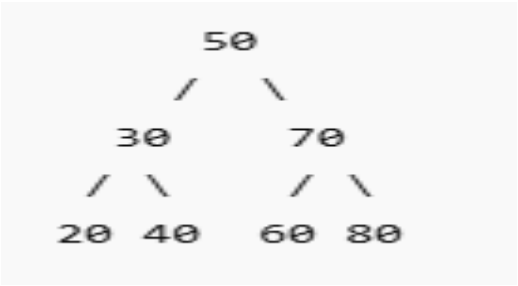
- Traverse left subtree of B → D → visit(D)
- Visit B
- Traverse right subtree of B → E → visit(E)

2.Visit A
3.Traverse right subtree of A → C → visit(C)
Output (Inorder Traversal): D → B → E → A → C

Application in Binary Search Trees:
For Binary Search Trees (BSTs), the Inorder traversal is highly significant because:

- It retrieves the elements in ascending order.
- It can be used for validating the structure of a BST.
- It assists in converting a BST to a sorted array or list.

For example, for the following BST:



Inorder Traversal: 20 → 30 → 40 → 50 → 60 → 70 → 80
Advantages:

- Inorder traversal maintains natural order of data in BSTs.
- Useful for report generation, data retrieval, and expression evaluation in expression trees.

Inorder traversal is one of the most fundamental and useful techniques for navigating a binary tree. It is especially powerful when applied to Binary Search Trees, where it guarantees a sorted sequence of the stored elements. Whether used recursively or iteratively, it provides a reliable method for systematically accessing each node in a structured and logical order.

14.2.2 Preorder Traversal (Root → Left → Right)

Preorder traversal is a type of depth-first traversal for binary trees in which each node is visited before its children. The traversal follows this specific sequence:

1. **Visit the root node**
2. **Traverse the left subtree**
3. **Traverse the right subtree**

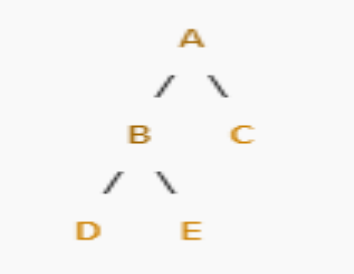
This traversal order is known as pre-order because the root node is processed before its subtrees. It provides a natural way to explore a tree structure starting from the top (root) and going deeper before moving laterally.

Steps Involved in Preorder Traversal:
Given a node N, the traversal steps are:

- **Step 1:** Process (visit) node N.
- **Step 2:** Recursively traverse the left subtree of node N.
- **Step 3:** Recursively traverse the right subtree of node N.

```
Preorder(Node)
{
If Node ≠ Null Then
Visit(Node.Data)
Preorder(Node.Left)
Preorder(Node.Right)
}
```

Example:
Consider the binary tree:



To perform a **Preorder Traversal**:
1.Visit A (root)
- Traverse left subtree of A:

2.Visit B
   - Traverse left subtree of B → Visit D
   - Traverse right subtree of B → Visit E

3.Traverse right subtree of A → Visit C

Output (Preorder Traversal): A → B → D → E → C

Applications of Preorder Traversal:
1. **Tree Copying:**
   - Preorder traversal is ideal when duplicating a tree, since the structure is preserved from the root downward.
2. **Serialization and Deserialization:**
   - Trees can be stored or transmitted using Preorder traversal, where a unique marker (e.g., NULL or #) is used for null pointers.
   - It is commonly used in saving tree data into files or converting trees into string formats for network transmission.
3. **Expression Trees:**
   - In the case of expression trees (used in compilers), Preorder traversal yields prefix expressions (also known as Polish notation).

Preorder traversal is a powerful technique in binary tree operations, especially when the structure of the tree is important. By visiting the root node first, it enables early processing and is suitable for scenarios where the hierarchy or order of construction is critical, such as in creating a replica, storing trees for future reconstruction, or evaluating prefix expressions.

14.2.3 Postorder Traversal (Left → Right → Root)

Postorder traversal is one of the three fundamental depth-first traversal techniques used to visit nodes in a binary tree. In this traversal method, the process follows the order:

1. Traverse the left subtree
2. Traverse the right subtree
3. Visit the root node

This traversal pattern delays the processing of the root node until after both its children have been processed. It reflects a bottom-up approach, making it particularly useful for operations where the processing of child nodes is required before the parent node.

Steps Involved in Postorder Traversal:

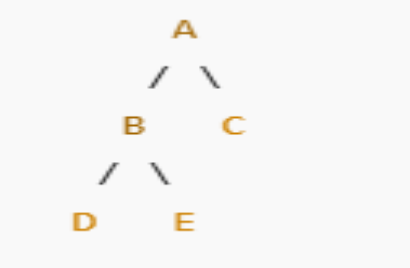
For a given node N, the traversal is carried out as follows:

- Step 1: Recursively traverse the left subtree of node N.
- Step 2: Recursively traverse the right subtree of node N.
- Step 3: Visit or process node N.

Algorithm (Recursive Form):

```
Postorder(Node)
{
If Node ≠ Null Then
  {
Postorder(Node.Left)
Postorder(Node.Right)
Visit(Node.Data)
  }
 }
```

Example:
Consider the binary tree:



To perform a **Postorder Traversal**:
1.Traverse left subtree of A (B)
   - Traverse left subtree of B → Visit D
   - Traverse right subtree of B → Visit E

2.Visit B
3.Traverse right subtree of A → Visit C
   - Visit A

Output (Postorder Traversal): D → E → B → C → A

Applications of Postorder Traversal:
Postorder traversal is widely used in real-world computing tasks. Key applications include:

1. **Deleting or Freeing a Tree:**
   - When deleting a tree, it is important to delete the children before deleting the parent. Postorder ensures that no node is deleted before its subtrees are cleared.
2. **Evaluating Postfix Expressions:**
   - In expression trees, where leaf nodes are operands and internal nodes are operators, Postorder traversal yields the postfix (Reverse Polish) notation which is essential for computation without parentheses.
3. **Computing the Height or Depth of a Tree:**
   - To calculate the height of a binary tree, the height of both subtrees must be known before determining the height of the root. Postorder supports this bottom-up calculation naturally.

Comparison of the tree traversals:

Traversal Type	**Order**	**Primary Use Case**
Inorder	Left → Root → Right	Sorted output from Binary Search Trees
Preorder	Root → Left → Right	Tree duplication or serialization
Postorder	Left → Right → Root	Tree deletion, postfix evaluation, height calculation

Each of the three traversal techniques has a unique role and is suited for different computational tasks:

- Inorder is optimal for reading values in sorted order from a Binary Search Tree.
- Preorder is ideal when the root needs to be processed early, such as in serialization or structural copying.
- Postorder is most useful when children need to be fully handled before their parent, such as in deletion, evaluation, or computing metrics like size or height.

All traversal techniques can be implemented either recursively, which is straightforward due to the tree's recursive nature, or iteratively using a stack, which simulates the recursive call stack for environments where recursion may not be practical.

14.3 TRAVERSAL AND SEARCH TECHNIQUES FOR GRAPHS

Graphs are data structures consisting of vertices (nodes) and edges (connections). Traversal ensures every vertex is visited systematically.Two fundamental techniques used to explore or traverse graphs are Breadth First Search (BFS) and Depth First Search (DFS). These methods are essential in solving various graph-related problems such as finding the shortest path, checking connectivity, and performing topological sorting.

14.3.1 BREADTH FIRST SEARCH (BFS) AND TRAVERSAL

Definition

Breadth First Search (BFS) is a graph traversal technique where one starts at a selected node (called the source node) and explores all its immediate neighbors first before moving on to the neighbors' neighbors. It proceeds level by level, which makes it ideal for finding the shortest path in an unweighted graph.

Algorithm

BFS uses a queue data structure to maintain the order of traversal.

1. Start from the source node and mark it as visited.
2. Enqueue the source node.
3. While the queue is not empty:
    - Dequeue a node u.
    - Visit all adjacent nodes of u that haven't been visited.
    - Mark each as visited and enqueue them.

Pseudocode

```
BFS(Graph G, Vertex start):
  Create a queue Q
  Mark start as visited and enqueue it into Q
  while Q is not empty:
    vertex = Q.dequeue()
    for each neighbor n of vertex:
      if n is not visited:
        mark n as visited
        enqueue n
```

Characteristics

- Traverses graph level by level.
- Uses a queue.
- Guarantees shortest path in unweighted graphs.
- Time Complexity: O(V + E)
  (V = number of vertices, E = number of edges)

Applications

- Finding shortest path in unweighted graphs.
- Peer-to-peer networking (e.g., BitTorrent).
- Web crawlers.
- Social networking analysis.
- GPS navigation systems.

14.3.2 Depth First Search (DFS) and Traversal

Definition

Depth First Search (DFS) is a graph traversal technique that starts at the source node and explores as far as possible along each branch before backtracking. It follows a depthward motion, making it ideal for solving puzzles and detecting cycles in graphs.

Algorithm Overview

DFS uses a **stack** (either explicitly or via recursion) to keep track of vertices.

1. Start from the source node and mark it as visited.
2. Push the node onto the stack.
3. While the stack is not empty:
   - Pop a node u.
   - For each unvisited neighbor of u, mark it as visited and push it onto the stack.

Pseudocode (Recursive Version):

```
DFS(Graph G, Vertex v):
  mark v as visited
  for each neighbor n of v:
    if n is not visited:
      DFS(G, n)
```

Characteristics:

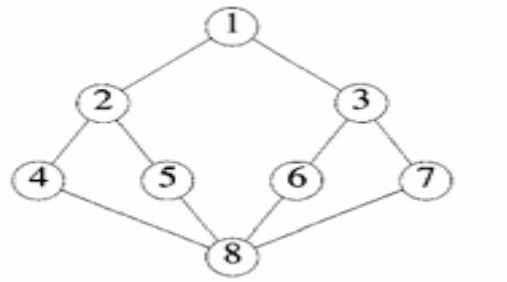
- Traverses graph deep before wide.
- Uses a stack or recursion.
- Useful for backtracking problems.
- Time Complexity: O(V + E)

Applications:

- Detecting cycles in a graph.
- Solving mazes or puzzles.
- Topological sorting in Directed Acyclic Graphs (DAGs).
- Pathfinding in game development.
- Checking for connected components.

14.3.3 Comparison of BFS and DFS

Example

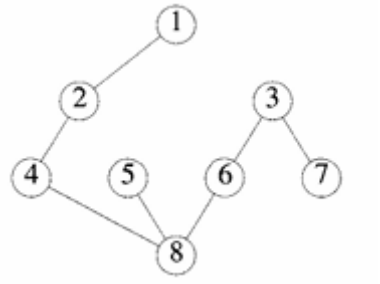


The above graph consists of 8 nodes labeled **1 to 8**, connected as follows:

- Node 1 connects to 2 and 3
- Node 2 connects to 4 and 5
- Node 3 connects to 6 and 7
- Nodes 5, 6, and 7 connect to 8

This forms an undirected graph with multiple levels and branching paths.

(a) DFS Spanning Tree



Depth First Search from Node 1

DFS Characteristics:

- Goes deep along a branch before backtracking.
- Typically uses a stack or recursion.
- Child nodes are visited in the order they appear (assumed left to right here).

DFS(1) Traversal Steps:

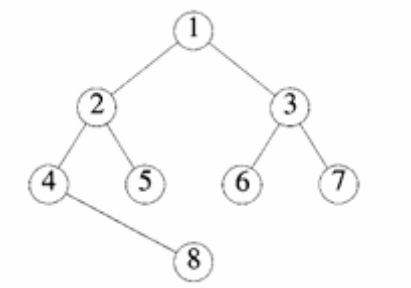
1. Start at node **1**
2. Visit node **2**
3. Visit node **4** (deepest left path)
4. Backtrack to **2**
5. Visit node **5**
6. Visit node **8** from node **5**
7. Backtrack all the way to **1**
8. Visit node **3**
9. Visit node **6**
10. Visit node **7**

DFS Spanning Tree Edges:

- (1,2), (2,4), (2,5), (5,8), (1,3), (3,6), (6,7)

This builds a tree where each edge represents the first visit to a new node during the DFS traversal.

(b) BFSSpanning Tree



Breadth First Search from Node 1

BFS Characteristics:

- Explores all neighbors at the current level before going deeper.
- Uses a **queue** to manage node processing.

BFSTraversal Steps:

1. Start at node 1
2. Enqueue neighbors → visit 2 and 3
3. Dequeue 2, visit its unvisited neighbors → 4, 5
4. Dequeue 3, visit → 6, 7
5. Next, visit 8 via any unvisited link (say from 5)

BFS Spanning Tree Edges:

- (1,2), (1,3), (2,4), (2,5), (3,6), (3,7), (5,8)

Here, BFS explores in levels:

Level 1 → {1}
Level 2 → {2, 3}
Level 3 → {4, 5, 6, 7}
Level 4 → {8}

Table 14.1 Comparison Between BFS and DFS

Feature	BFS	DFS
Data Structure Used	Queue	Stack / Recursion
Traversal Order	Level-wise	Depth-wise
Shortest Path (Unweighted Graph)	Yes	No
Suitable for	Finding shortest path, level order	Detecting cycles, topological sort
Time Complexity	O(V + E)	O(V + E)
Space Complexity	O(V)	O(V) (for recursion/visited stack)

Both BFS and DFS are cornerstone techniques for traversing and analyzing graphs. The choice between them depends on the problem being solved:

- Use BFS for shortest paths and problems involving levels.
- Use DFS for problems involving depth, connectivity, or cycle detection.

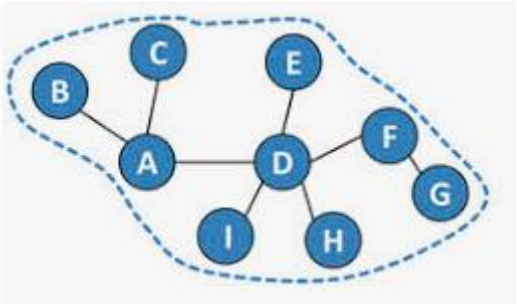
Each technique plays a crucial role in computer science, ranging from algorithm design to real-world applications in networking, AI, and software systems.

14.4 CONNECTED COMPONENTS AND SPANNING TREES

A connected component in an undirected graph is a maximal set of vertices such that each pair of vertices is connected by a path.

- If the graph is fully connected, it has only one connected component.
- In a disconnected graph, there will be multiple connected components.

Example



In the above graph, the nodes are:

A, B, C, D, E, F, G, H, I

Upon observation:

- All nodes are reachable from one another through some path.
- There are no isolated nodes or groups.

A spanning tree of a connected, undirected graph is a subgraph that:

- Includes all the vertices of the original graph.
- Is connected.
- Contains no cycles.
- Has exactly V - 1 edges, where V is the number of vertices.

Properties:

- For a graph with 9 vertices, a spanning tree will contain 8 edges.
- A graph can have multiple valid spanning trees.

Spanning Tree Diagram (One Possible Example)
Below is one of the possible spanning trees formed from the original graph:

Explanation:

- All 9 vertices are included: A, B, C, D, E, F, G, H, I
- There are no cycles.
- There are exactly 8 edges:
- A–B, A–C, A–D, D–E, D–F, F–G, G–H, D–I

This forms a valid spanning tree.

Concept	Description
Connected Component	The graph has one connected component consisting of all nodes A to I
Spanning Tree	A tree structure that connects all nodes without cycles and with exactly 8 edges

14.5 BICONNECTED COMPONENTS AND DFS

This section explores the concept of biconnected components in graph theory and how the Depth-First Search (DFS) algorithm is leveraged to identify them efficiently. Understanding these components is crucial for analyzing the reliability and structure of networks.

14.5.1Definition of Biconnected Components

A biconnected component (also called a 2-connected component) of an undirected graph is a maximal subgraph such that:

- It is connected.
- It cannot be disconnected by removing any single vertex (also known as a non-articulation point).

In simpler terms, in a biconnected component:

- There are at least two distinct paths between any two vertices.
- Removing one vertex does not disconnect the component.

Key Terms

- Articulation Point: A vertex whose removal increases the number of connected components in the graph.
- Bridge: An edge whose removal increases the number of connected components. In biconnected components, bridges are not present.

14.5.2 Importance of Biconnected Components

- Used in network reliability analysis: ensures that the network remains connected even if a node fails.
- Important in circuit design, computer networks, and compiler construction.
- Helps identify critical points (articulation points) in the graph.

14.5.3 Role of DFS in Finding Biconnected Components

Depth First Search (DFS) is the primary technique used to find biconnected components efficiently.

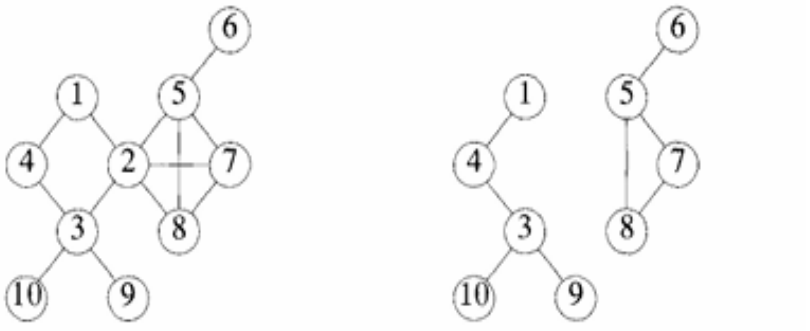
How DFS Helps

- DFS traverses the graph and numbers vertices in discovery order (DFS number).
- It keeps track of:
  - Discovery time: When a vertex is first visited.
  - Low value: The lowest discovery time reachable from the vertex through its subtree or back edge.

DFS-Based Algorithm Overview

1. Perform DFS traversal.
2. For each vertex, compute the low value.
3. If low[v] ≥ disc[u] for a child v of u, then u is an articulation point.
4. The set of nodes/edges visited from the root to the articulation point forms a biconnected component.

14.5.4 Example: Identifying an Articulation Point



(a) Graph *G* (b) Result of deleting vertex 2

The above figure contains two subgraphs:

(a) Graph G – The original undirected graph
(b) Result of deleting vertex 2 – Shows the structure of the graph after removing vertex 2

(a) Graph G – Original Graph

Nodes:1 through 10

Edges (selectively inferred from the diagram):

- Node 2 connects to 1, 3, 5, and 8
- Node 5 connects to 6, 7, 8
- Node 3 connects to 9 and 10
- Other nodes are connected as per adjacency

This graph is connected, meaning there is at least one path between every pair of nodes.

(b) Result of Deleting Vertex 2

In the second diagram, vertex 2 has been removed from the graph. As a result:

- The graph splits into two disconnected components:
  - Component 1: Nodes {1, 3, 4, 9, 10}
  - Component 2: Nodes {5, 6, 7, 8}

This shows that removing vertex 2 disconnects the graph.

1. Articulation Point

Based on the observed result of the vertex removal, the critical nature of node 2 can be identified.

- A vertex is an articulation point if removing it increases the number of connected components.
- In this case, vertex 2 is an articulation point because:
  - Original Graph G is connected
  - After removing vertex 2, the graph becomes disconnected

2. Biconnected Components

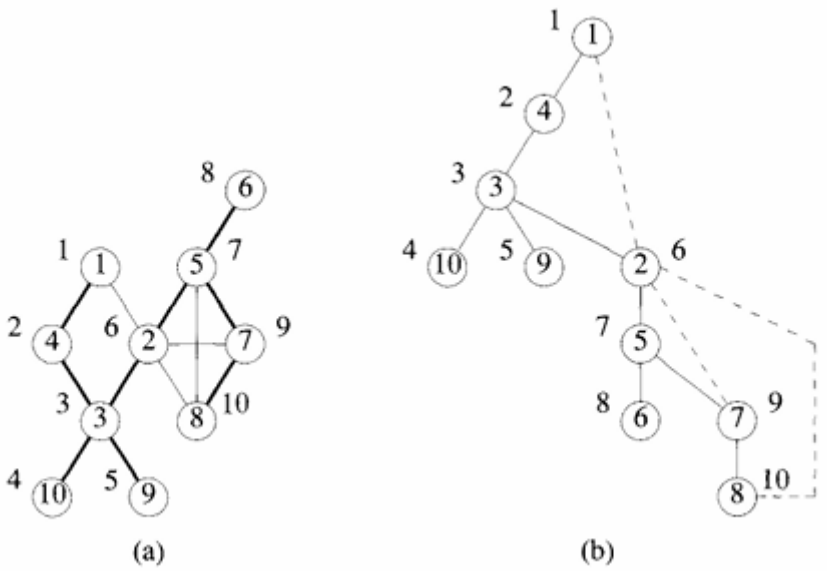
The identification of an articulation point allows us to define the distinct biconnected components that were previously joined by that point.

- A biconnected component is a maximal subgraph where no single vertex is an articulation point.
- In Graph G:
  - Vertices and edges around vertex 2 (like 1–2–3 or 2–5–8) belong to different biconnected components
  - Vertex 2 connects multiple biconnected components together.

14.5.5 DFS Tree and Algorithm Working

The DFS traversal provides the necessary structure to mathematically prove and find these components.

Example



(a) Original Graph with Biconnected Components Highlighted

- The graph has 10 nodes, labeled from 1 to 10.
- Some edges are shown in thick lines.
- Thick lines represent DFS tree edges – these are the edges followed during the Depth First Search.
- Other edges (thin ones) are either:
  - Back edges, or
  - Belong to other biconnected components.
- This helps us see how DFS is used to divide the graph into biconnected parts.

(b) DFS Tree Representation

- The structure shows how DFS visits each node one by one like a tree.
- Dashed edges are back edges:
  - These are edges that go back to an earlier (ancestor) node in the DFS path.
- Back edges help in:
  - Finding cycles,
  - Calculating low values,
  - And identifying articulation points.

(c) Algorithm Working

- DFS starts from a node (like node 1) and visits all other nodes step by step.
- For every visited node:
  - A discovery time is given (order in which the node is visited).
  - A low value is calculated (the earliest node it can reach).

If a node has an adjacent node that is already visited and is not its parent, it is a back edge.

- After visiting all child nodes, the algorithm checks:
  - If a child node cannot reach any ancestor of its parent, then the parent is an articulation point.
- Articulation points are nodes that, if removed, split the graph into separate parts.
- These points are used to divide the graph into biconnected components.

(d) Why This Is Important

- Helps find critical nodes in a network.
- Used in network design, computer systems, and reliable connections.
- Makes sure the system does not break if one node fails.

14.6 SUMMARY

This lesson introduced the essential traversal and search techniques for both trees and graphs. Traversal is a fundamental operation that provides systematic ways to visit every node in a structure exactly once. Binary Trees utilize three depth-first methods: Inorder for sorted output from a BST; Preorder for cloning and serialization; and Postorderfor deletion and postfix evaluation. Graph search relies on Breadth First Search (BFS), which uses a queue for level-by-level exploration and guaranteed shortest paths in unweighted graphs, and Depth First Search (DFS), which uses a stack or recursion for deep exploration, cycle detection, and topological sorting. Furthermore, these techniques are applied to analyze graph connectivity, specifically to find Connected Components, generate Spanning Trees, and identify Biconnected Components. DFS is the primary method for finding Articulation Points, which are critical nodes separating these biconnected components, thus ensuring network reliability analysis.

14.7 KEY TERMS

Traversal, Inorder Traversal, Preorder Traversal, Postorder Traversal, Breadth First Search (BFS), Depth First Search (DFS), Connected Component.

14.8 REVIEW QUESTIONS

1. Explain the difference in traversal order and primary application between BFS and DFS.
2. Describe the three fundamental binary tree traversal methods. Which method is most useful for retrieving data from a Binary Search Tree in sorted order?
3. What is a Spanning Tree? List the three properties a subgraph must satisfy to be considered a spanning tree of a connected graph.
4. Define an Articulation Point and a Bridge in the context of graph connectivity.
5. Explain the concept of a Biconnected Component and its significance in network reliability.
6. Provide one real-world application for both BFS and DFS.

14.9 SUGGESTED READINGS

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press.
2. Horowitz, E., Sahni, S., & Anderson-Freed, S. (2008). Fundamentals of Data Structures in C. University Press.
3. Sedgewick, R., & Wayne, K. (2011). Algorithms (4th ed.). Addison-Wesley.
4. Aho, A. V., Hopcroft, J. E., & Ullman, J. D. (1983). Data Structures and Algorithms. Addison-Wesley.
5. Kleinberg, J., & Tardos, E. (2005). Algorithm Design. Pearson Education.

Mrs. Appikatla PushpaLatha

LESSON-15

BACKTRACKING TECHNIQUE

OBJECTIVES

The objectives of this lesson are to:

- Understand the general method of the backtracking algorithm.
- Explore the 8-Queens problem and how backtracking finds solutions.
- Analyze the Sum of Subsets problem using different tree representations.
- Learn the application of backtracking in the Graph Coloring problem.
- Investigate how Hamiltonian Cycles are generated with backtracking.
- Apply backtracking techniques to the 0/1 Knapsack problem using bounding functions.

STRUCTURE

15.1 INTRODUCTION

Backtracking is a general algorithmic technique used to solve constraint satisfaction and combinatorial problems by building solutions incrementally. At each decision point, the algorithm explores a partial solution and abandons it ("backtracks") as soon as it determines the solution cannot be completed. This systematic search method is widely applied in puzzles, games, optimization, and real-world scheduling problems.
Backtracking avoids brute-force by eliminating infeasible paths early using a constraint (bounding) function. It is most effective when the solution space can be structured as a tree or a graph.

15.2 THE GENERAL METHOD

Backtracking is a general algorithmic technique that involves searching through all possible configurations or solutions to a problem in a systematic manner. It is most effective for

problems involving combinatorial search, where the solution space is large and structured in a tree or graph format.

Definition and Principle

Backtracking is used to solve problems incrementally, building candidates to the solution step by step, and abandoning a candidate ("backtrack") as soon as it is determined that the candidate cannot possibly lead to a valid solution. This technique avoids unnecessary computation by eliminating infeasible solutions early.

The general structure of a backtracking algorithm is based on the following:

1. **State Space Tree**:
   The solution process is modeled as a tree, where:
   - Each node represents a partial solution.
   - The root node corresponds to the initial state (usually an empty solution).
   - Leaf nodes represent complete solutions.
   - Paths from root to leaves represent choices made at each step.

2. **Recursive Construction**:
   The algorithm explores the state space using recursion, progressing through the tree depth-first.

3. **Bounding Function (Constraint Test)**:
   At each node, a constraint function checks whether the partial solution satisfies the problem constraints.
   - If it does, the algorithm continues to explore further by extending the current solution.
   - If not, the algorithm backtracks and tries a different option.

General Backtracking Algorithm

```
procedure Backtrack(v)
{
        if v is a solution:
                process_solution(v)
        else:
        for each child u of v:
         if u is promising:
         Backtrack(u)
}
```

- v is a node in the state space tree.
- The function checks if v is a solution, and if so, processes it.
- Otherwise, it generates all children (possible next steps).
- For each child u, the algorithm checks if u is promising (i.e., satisfies constraints so far).
- If promising, the algorithm recursively calls Backtrack(u).

Applications

The general method of backtracking can be applied to:

- N-Queens problem – placing queens on a chessboard so that no two queens attack each other.
- Graph coloring – assigning colors to graph vertices so adjacent vertices have different colors.
- Subset and permutation generation.
- Knapsack problem, sudoku solvers, maze navigation, and many other combinatorial problems.

Efficiency Considerations

Backtracking is more efficient than brute-force because:

- It prunes the search space using constraints.
- It avoids full traversal of all possibilities by cutting off branches early.

However, in the worst case, it can still be exponential in time complexity, especially if pruning is ineffective.

15.3 THE 8-QUEENS PROBLEM

- The 8-Queens problem is a classical example of a constraint satisfaction problem.
- The goal is to place 8 queens on an 8×8 chessboard such that:
- No two queens share the same row, same column, or same diagonal.
- This problem can be extended to n-Queens, where the board is n × n and n queens must be placed with the same constraints.

Representation of the Problem

- Since each queen must be in a different row, the position of queen i can be uniquely identified by specifying the column number where she is placed in row i.
- Therefore, a solution can be represented as an n-tuple (x_1, x_2, ..., x_n) where each x_i indicates the column position of the queen in row i.

Backtracking Approach

Backtracking is used to systematically explore all possible arrangements while pruning those that violate constraints:

Steps Involved:

1. Start with an empty board.
2. Place the first queen in the first row, trying each column one by one.
3. For each valid placement, move to the next row and repeat the process.
4. If a conflict arises (same column or diagonal), backtrack and try the next column in the previous row.
5. Continue until either:
   - All queens are placed successfully (a solution is found).
   - All possibilities are exhausted (no solution from the current path).

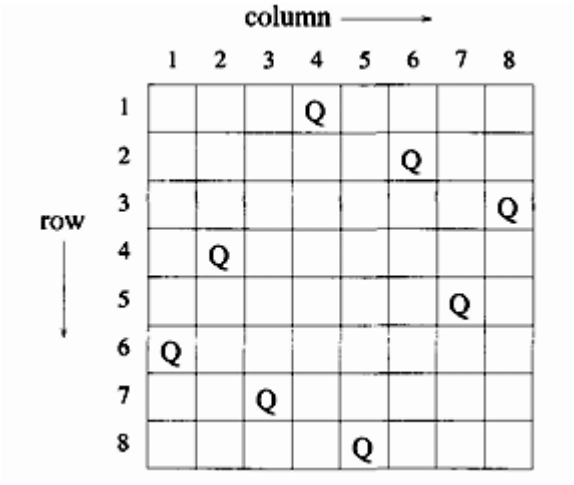


Figure 15.1One solution to the 8-queens problem

- The above diagram shows a correct solution to the 8-Queens problem.
- Each queen (Q) is placed such that no two queens attack each other.
- This final board configuration corresponds to one valid tuple solution.

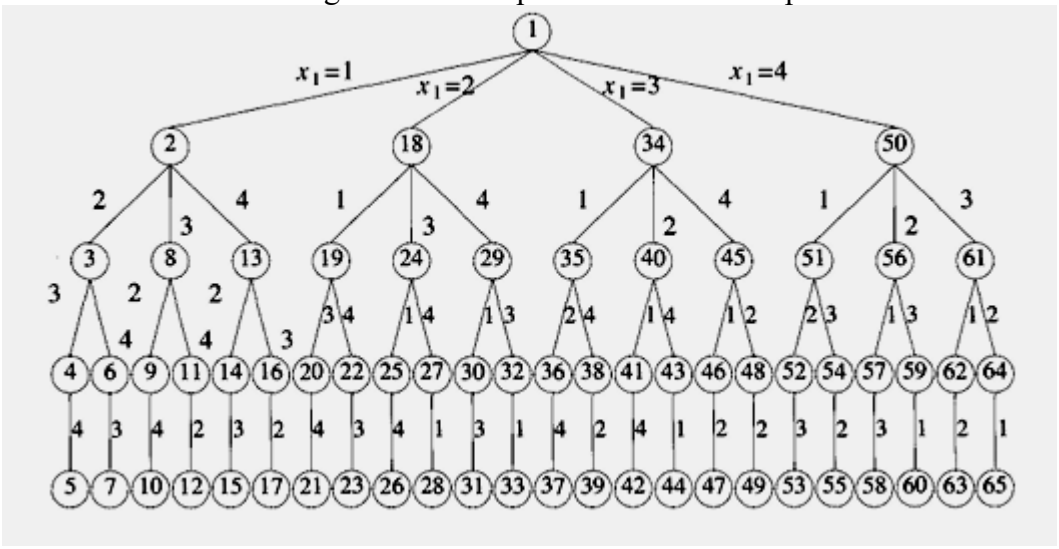


Figure 15.2 Tree Representation for 4-Queens

Figure 15.2 provides a visual representation of the solution space for the 4-Queens problem using a state space tree.

- The figure illustrates a tree structure where each node corresponds to a partial placement of queens.
- Each level of the tree represents a row on the chessboard, indicating the current queen being placed.
- Each branch from a node shows a possible column position for the queen in that row.
- The nodes are numbered according to a depth-first search (DFS) traversal, indicating the order in which configurations are explored during backtracking.
- This tree is a permutation tree – representing all permutations of column values.

Working of Backtracking with 4-Queens as per Figure 15.2

- In this process, the algorithm:
  - Starts with placing the first queen.
  - Tries different placements for the next queen.
  - Uses a bounding function to reject invalid placements early.

- Dots in the board diagrams represent invalid tries.
- Backtracking occurs when the algorithm fails to place a queen in a valid column in the current row, and it returns to the previous row to try the next option.
- This continues until a complete solution is found.

Bounding Function (Constraint Function)

The algorithm uses a bounding function to eliminate invalid partial solutions:

- A placement ($x_1, x_2, ..., x_i$) is rejected if:
  - $x_j = x_k$ (same column) for some $j \neq k$
  - $|x_j - x_k| = |j - k|$ (same diagonal)
- These checks are performed at every step to ensure early pruning.

Advantages of Backtracking in This Problem

- Avoids unnecessary computations by eliminating invalid configurations early.
- Efficient in memory usage due to depth-first search.
- Capable of finding all possible solutions, not just one.

15.4 SUM OF SUBSETS

In the context of the Sum of Subsets problem, we aim to find subsets from a given set of positive integers $w = \{w_1, w_2, ..., w_n\}$ such that the sum of the elements in each subset is equal to a given target value m.

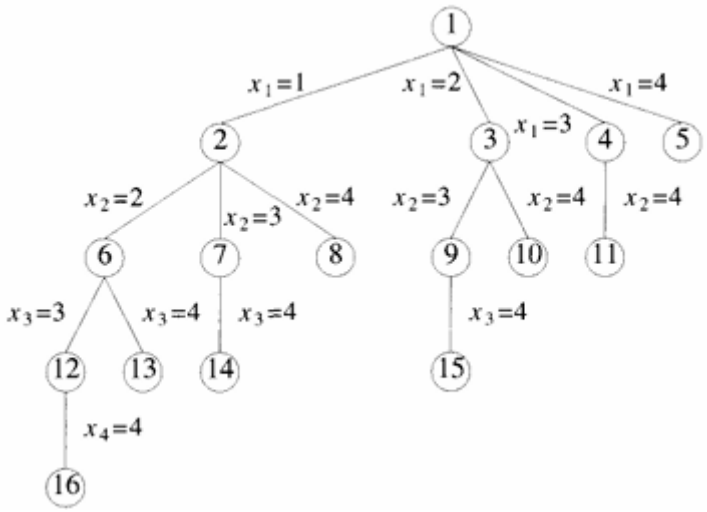


Figure 15.3 A possible solution space organization for the sum of subsets problem (Variable Tuple Size Tree)

The backtracking approach can be visualized through two different types of state space trees, as shown in Figure 15.3 and Figure 15.4. These two figures represent different tree organizations for the same problem but differ in how the solution space is structured and explored.

- Figure 15.3 shows a tree structure where each node represents a partial subset built from the set of input weights.
- The decision at each level is whether to include a specific item from the set into the current subset.
- Each level corresponds to the inclusion decision for one weight element.
- The branching is not uniform; some branches may terminate earlier if they are found to be infeasible (i.e., the current sum exceeds the target sum).
- This representation is depth-first in nature and mimics how subsets are constructed step-by-step by making inclusion choices.
- It is called variable tuple size because the subset represented at each node varies in length depending on how many weights have been included so far.
- A successful path from root to a leaf that meets the target sum m indicates a valid solution.

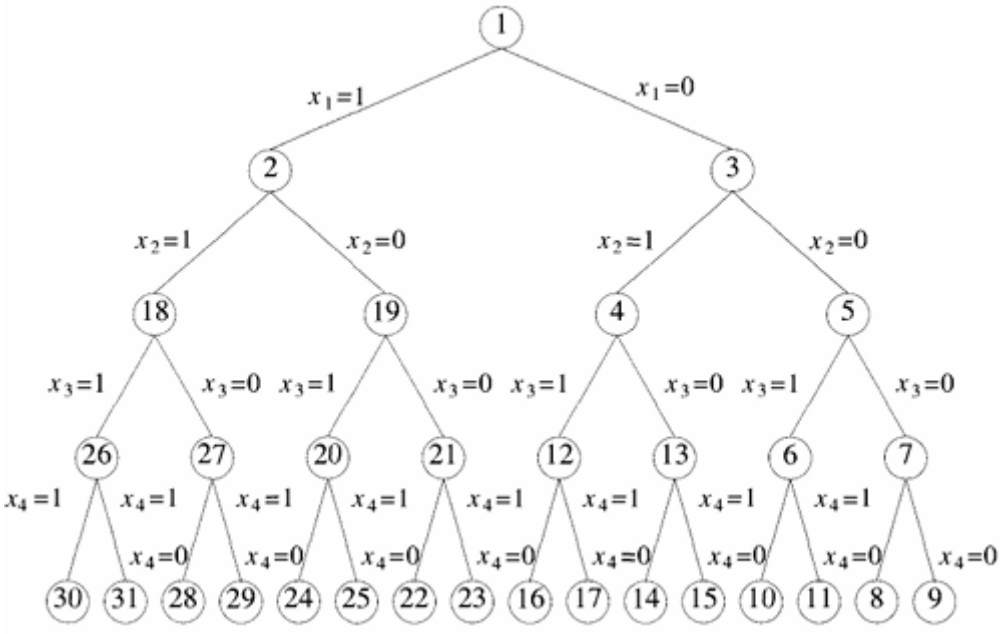


Figure 15.4 – Fixed Tuple Size Tree (Binary Tree of Subsets)

- Figure 7.4 illustrates a binary tree where each node at level i represents a decision for the i-th weight: whether to include it ($x_i = 1$) or exclude it ($x_i = 0$).
- The tree is complete and balanced, with all paths from the root to the leaf containing exactly n decisions.
- Each leaf node corresponds to one of the 2^n possible subsets of the input set.
- This representation is referred to as fixed tuple size, where each subset is shown as an n-tuple ($x_1, x_2, ..., x_n$) of binary values indicating inclusion (1) or exclusion (0).
- The left child of each node corresponds to $x_i = 1$ (include the item), and the right child to $x_i = 0$ (exclude the item).
- Although it explores all possible subsets, only the leaf nodes are tested to verify if the subset sum equals m.

15.5 GRAPH COLORING

The Graph Coloring Problem is defined as assigning colors to the vertices of a graph such that:

- No two adjacent vertices share the same color.
- The total number of colors used does not exceed a given number m.

This is called the m-Coloring Problem.

Applications

- Scheduling tasks without conflicts
- Register allocation in compilers
- Map coloring
- Frequency assignment in cellular networks

Input and Constraints

Given:

- A graph G = (V, E) with n vertices
- An integer m representing the maximum number of colors
- The goal is to assign each vertex a color (from 1 to m) such that:
  - If (u, v) is an edge in E, then color[u] ≠ color[v]

Backtracking Approach

The problem is solved using backtracking, where we attempt to assign colors one vertex at a time, verifying at each step if the assignment is valid (no adjacent vertex has the same color).

Steps.

1. Start from the first vertex.
2. Assign a color from 1 to m that is not used by any adjacent vertex.
3. Move to the next vertex and repeat the process.
4. If no color is valid for a vertex, backtrack to the previous vertex and try a different color.

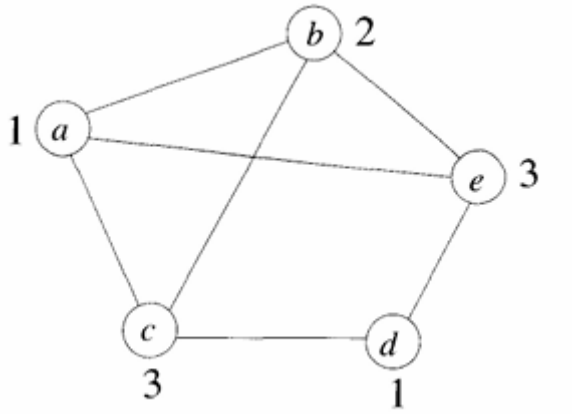


Figure 15.5 An example graph and its coloring

The above figure represents a Graph with Vertices a–e and Color Assignments

- This graph contains 5 vertices: a, b, c, d, e
- Each vertex has a number label indicating the color assigned:
  - a = 1, b = 2, c = 3, d = 1, e = 3
- The figure shows a valid 3-coloring, where adjacent vertices have different colors.

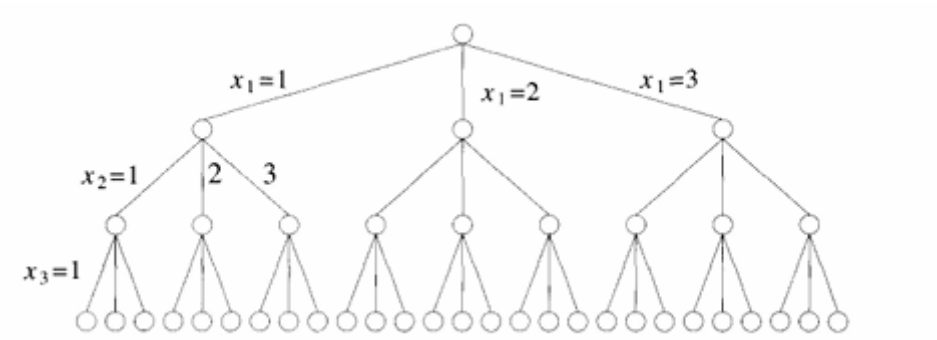


Figure 15.6 State space tree from Coloring when n =3and m = 3

In the above figure

- Each level of the tree corresponds to a vertex.
- Each branch represents an attempt to assign a color to that vertex.
- Nodes represent partial solutions (color assignments so far).
- The tree is traversed depth-first, using backtracking to reject invalid paths early.
- For example, at level 1, three choices are tried: $x_1 = 1$, $x_1 = 2$, and $x_1 = 3$
- If a color is invalid due to adjacency, the node is pruned.

Bounding Function (Promising Function)
Before assigning a color to a vertex, the algorithm checks:

- If the current color does not conflict with already assigned colors of adjacent vertices.

This check allows the algorithm to avoid exploring infeasible branches, improving efficiency.

Backtracking Algorithm

```
mColoring(k):
   for color = 1 to m:
      if isSafe(k, color):
         assign color to vertex k
         if k == n:
            print solution
         else:
mColoring(k + 1)
```

15.6 HAMILTONIAN CYCLES

A Hamiltonian cycle in a graph is a cycle that visits every vertex exactly once and returns to the starting vertex. This cycle must form a closed loop that includes all the vertices of the graph.

Problem Statement
Given a graph G = (V, E) with n vertices, the objective is to determine whether there exists a Hamiltonian cycle and, if so, generate all such cycles.
Backtracking Solution

The backtracking approach constructs the Hamiltonian cycle incrementally using the following recursive strategy:

Maintain a vector x[1...n] such that:

- x[k] represents the kth vertex visited in the cycle.
- The cycle always starts at vertex 1 to avoid repeated cycles in different rotations.

At each stage k, select a vertex that:

- Has not already been included in the path.
- Is connected to the previously selected vertex x[k-1].

If k = n, check whether the last vertex x[n] is connected to x[1] to complete the cycle.

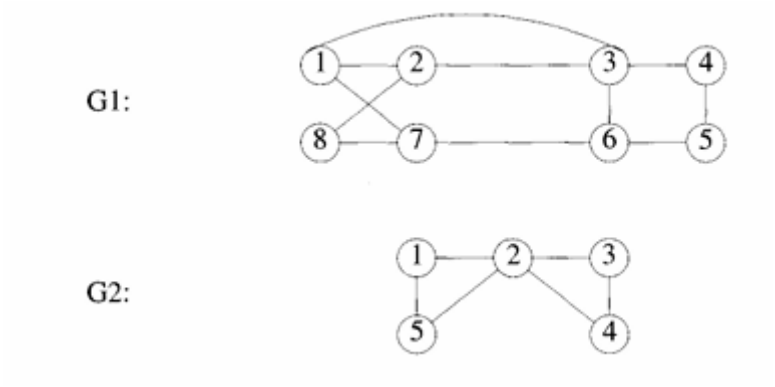


Figure 15.7 Graphs G1 and G2 for Hamiltonian Cycle Analysis

- Graph G1 has 8 vertices connected in such a way that the cycle:

$1 \rightarrow 2 \rightarrow 8 \rightarrow 7 \rightarrow 6 \rightarrow 5 \rightarrow 4 \rightarrow 3 \rightarrow 1$

forms a Hamiltonian cycle, visiting all nodes exactly once and returning to the start.

- Graph G2, on the other hand, lacks such a cycle.
  Despite being connected, it does not allow traversal of all vertices in a single cycle without repetition or missing a vertex.
  Hence, G2 does not contain a Hamiltonian cycle.

Algorithm for Hamiltonian Cycle

```
// This algorithm uses the recursive backtracking method
// to find all Hamiltonian cycles in a graph.
// The graph is stored as an adjacency matrix G[1...n][1...n].
// All cycles begin at node 1.
Algorithm Hamiltonian(k)
{
   repeat
      // Generate values for x[k]
      NextValue(k)              // Assign a legal next value to x[k]
      if x[k] = 0 then
         return                 // No more values to try
      if k = n then
         write(x[1...n])        // A complete Hamiltonian cycle is found
      else
         Hamiltonian(k + 1)     // Try next position
   until false
 }
```

Applications

- The Hamiltonian cycle problem has applications in:
  - Routing problems
  - Traveling Salesperson Problem (TSP)
  - Network topology design
  - DNA sequencing and optimization

15.7 KNAPSACK PROBLEM

The 0/1 Knapsack Problem is a classic optimization problem. You are given:

- n items, each with:
  - a weight w[i]
  - a profit p[i]
- A knapsack with a maximum capacity m

The goal is to choose a subset of the items such that:

- The total weight does not exceed m
- The total profit is maximized
- Each item can be either taken (x[i] = 1) or not taken (x[i] = 0)

Solution Space

- The number of possible combinations (subsets) is 2^n, because each item has two choices: include or exclude.
- This makes it similar to the sum of subsets problem, and the solution space can be explored using backtracking.

Backtracking Approach

To avoid checking all 2^n combinations, backtracking is used to explore only promising paths.
Key Ideas:

1. Represent choices as a tree (called a state space tree).
2. At each level of the tree:
   - Include the item (go to left child)
   - Exclude the item (go to right child)
3. Continue building the solution until:
   - The weight exceeds the knapsack's capacity → backtrack
   - All items are considered → check if it's the best solution

Bounding Function (Bound (cp, cw, k))

To improve efficiency, a bounding function is used:

- cp: current profit
- cw: current weight
- k: index of the last item added

The Bound function estimates the maximum possible profit from the current node by:

- Including full items until the capacity is reached
- Adding a fraction of the next item if needed (like in fractional knapsack)

If the bound is less than or equal to the best profit found so far, that path is pruned (discarded).
We are solving the 0/1 Knapsack Problem using backtracking. Given:

- Profits p = {11, 21, 31, 33, 43, 53, 55, 65}

- Weights w = {1, 11, 21, 23, 33, 43, 45, 55}
- Knapsack Capacity m = 110
- Number of items n = 8

Goal:

Select a subset of items such that:

- Total weight ≤ 110
- Total profit ismaximized
- Each item is either included (x[i] = 1) or excluded (x[i] = 0)

Tree Representations for Backtracking

To explore the solution space, binary trees are used where each level represents a decision for one item.

15. Fixed Tuple Size Tree

- Each node represents a binary decision (include or exclude) for a specific item.
- Left child of a node → item is included (x[i] = 1)
- Right child → item is excluded (x[i] = 0)
- The tree has depth = n, and each leaf represents a full 0/1 assignment.
- This structure is complete and uniform.

15. Variable Tuple Size Tree

- This tree builds only valid and promising paths.
- It dynamically adds decisions (0 or 1) as it goes deeper.
- Some branches terminate early based on pruning or bounding.
- The tree is irregular, and its size depends on decisions made during traversal.

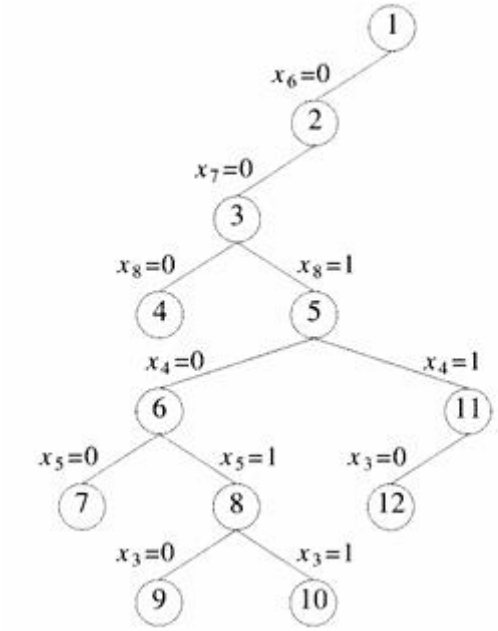


Figure 15.8 State Space Tree of KnapSack Problem

The above figure shows a variable tuple size decision tree for the given 8-item knapsack problem.

Key Observations:

- The root node (1) starts the decision process.
- Each branch indicates a decision:
- For example, $x_6 = 0$ means item 6 is excluded, and the algorithm proceeds.
- Nodes are numbered based on the DFS traversal order.

At each level:

- A left branch means setting $x_i = 0$
- A right branch means setting $x_i = 1$

Example Path:

Let's follow this path from the tree:
$1 \rightarrow 2$ ($x_6 = 0$)
$2 \rightarrow 3$ ($x_7 = 0$)
$3 \rightarrow 4$ ($x_8 = 0$)
$4 \rightarrow 6$ ($x_4 = 0$)
$6 \rightarrow 7$ ($x_5 = 0$)

This path represents a partial assignment:
$x_6 = 0, x_7 = 0, x_8 = 0, x_4 = 0, x_5 = 0$
The rest of the variables will be assigned later in the recursion or through backtracking.

The algorithm continues to:

- Check if the total weight is within the capacity
- Compute the current profit
- Use a bounding function (as discussed in the Bound(cp, cw, k) algorithm) to decide whether to proceed or prune

Importance of Variable Tuple Size Tree

- Saves time and memory by avoiding generation of non-promising nodes.
- Makes use of bounding to decide whether a node should be expanded.
- Allows the use of greedy upper bounds to prune paths early.

15.8 SUMMARY

Backtracking is a versatile algorithmic technique designed to solve combinatorial and constraint satisfaction problems by exploring solution spaces systematically. It builds potential solutions incrementally and abandons paths as soon as they violate the defined constraints, thereby optimizing the search process. The method is particularly effective for problems where the solution space can be structured as a tree or graph, such as the 8-Queens problem, Sum of Subsets, Graph Coloring, Hamiltonian Cycles, and the 0/1 Knapsack Problem. Each of these problems demonstrates how backtracking leverages state space trees and bounding functions to prune non-promising paths and efficiently reach valid solutions. Despite its potential for exponential time complexity in the worst case, backtracking significantly reduces computation when effective pruning is applied.

15.9 KEY TERMS

Backtracking, State Space Tree, Bounding Function, Constraint Satisfaction, Hamiltonian Cycle, 0/1 Knapsack Problem

15.10 REVIEW QUESTIONS

1. What is the role of a bounding function in the backtracking algorithm?
2. How does a state space tree represent the solution space in backtracking?
3. Explain how backtracking is applied to the 8-Queens problem.

4. What are the differences between fixed and variable tuple size trees in subset-related problems?
5. How does backtracking help in solving the m-coloring problem in graph theory?
6. What conditions must be met to form a valid Hamiltonian cycle using backtracking?

15.11 SUGGESTED READINGS

1. Horowitz, Ellis, Sartaj Sahni, and Sanguthevar Rajasekaran. *Fundamentals of Computer Algorithms*, Universities Press, 2008.
2. Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, MIT Press, 3rd Edition, 2009.
3. Levitin, Anany. *Introduction to the Design and Analysis of Algorithms*, Pearson Education, 3rd Edition, 2011.
4. Aho, Alfred V., John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.
5. Kleinberg, Jon, and Éva Tardos. *Algorithm Design*, Pearson Education, 2005.

Mrs. Appikatla PushpaLatha

LESSON-16

BRANCH AND BOUND TECHNIQUE

OBJECTIVES

The objectives of this lesson are to:

- Understand the general method of the Branch and Bound algorithm.
- Explore the use of bounding and pruning in large solution spaces.
- Study Least Cost (LC) Search and its control abstractions.
- Apply Branch and Bound to classic optimization problems like the 15-Puzzle and Traveling Salesman Problem (TSP).
- Analyze the FIFO-based Branch and Bound strategy.
- Compare variable and fixed tuple size trees in job scheduling problems.

STRUCTURE

16.1 INTRODUCTION

Branch and Bound is a general-purpose algorithm design technique used to solve combinatorial optimization problems where exhaustive enumeration is computationally impractical. The technique constructs a solution space tree where each node represents a partial solution, and applies bounding functions to eliminate subtrees that cannot yield better

results than those already found. The method uses systematic exploration strategies such as FIFO, LIFO, or Least-Cost (LC) queues to prioritize nodes for expansion. Efficient pruning using bounding functions dramatically reduces the number of configurations to explore.

16.2 THE BRANCH AND BOUND METHOD

The Branch and Bound method is a powerful general-purpose algorithm design technique used to solve combinatorial optimization problems where the solution space is large and exponential in nature. It is especially effective in problems where a brute-force approach would be computationally infeasible. The method constructs a solution space tree where each node represents a partial solution to the problem, and systematically explores this tree in a way that avoids unnecessary computation.

The process begins by identifying the root of the solution tree, which represents an empty or initial solution. This root node is inserted into a list of live nodes—nodes that have been generated but not yet explored. The node currently being explored is called the E-node (Expansion node). When an E-node is chosen, all its children (which represent possible choices or extensions of the current solution) are generated. Before expanding these nodes further, the algorithm uses a bounding function to evaluate whether any of them could lead to an optimal solution. If the bounding function reveals that a particular node or its descendants cannot yield a better solution than one already known, that node is pruned from the tree.

The structure of the live node list determines the search strategy. When a FIFO (First In, First Out) queue is used, the approach resembles breadth-first search. When a LIFO (Last In, First Out) stack is used, it behaves like depth-first search. A more sophisticated version uses a least-cost (LC) priority queue, where nodes with the smallest bound values are given higher priority for exploration. This strategy ensures that the most promising solutions are considered earlier.

The bounding function plays a central role in the efficiency of the Branch and Bound technique. It estimates the best possible outcome from a given node by relaxing some constraints. If the bound from a node is worse than the current best-known solution, that node and all its descendants can be safely ignored. This avoids exploring large parts of the solution space and significantly improves performance over naive approaches.

16.3 LEAST COST (LC)SEARCH

Traditional search strategies such as FIFO (First-In-First-Out) and LIFO (Last-In-First-Out) used in branch-and-bound algorithms adopt a fixed and rigid selection rule for choosing the next node to expand (known as the *E-node*). These methods do not account for the likelihood of a node leading to a solution quickly. As a result, even nodes that are close to a solution may be delayed in expansion due to their order of generation.

To address this limitation, the Least Cost (LC) Search strategy introduces an intelligent method of ranking live nodes based on estimated cost, and expands the node with the least cost. This improves the efficiency of the search, especially in optimization problems.

16.3.1 RANKING AND COST ESTIMATION

In LC Search, each live node is assigned a cost function c(x) that estimates the total effort required to reach a solution through that node. The next E-node is selected as the one with the minimum cost value.

The cost function is defined as:

$$c(x) = f(h(x)) + g(x)$$

Where:

- h(x) denotes the cost incurred to reach node x from the root.
- g(x) is an estimate of the remaining cost to reach an answer node from x.
- f(h(x)) is a non-decreasing function that determines the weight of the already expended effort.

In many applications, f(h(x)) is set to zero, thereby focusing only on minimizing the future cost g(x).

16.3.2 IMPACT OF COST FUNCTION

Using $f(h(x)) = 0$ often biases the search towards deeper nodes since g(x) for children is expected to be lower than that of their parent. This can lead to deep and possibly unfruitful searches. To counteract this, setting $f(h(x)) \neq 0$ allows the algorithm to give preference to shallower nodes that are also promising, thus making the search more balanced and reducing the risk of excessive deep searches.

Special Cases of LC Search

LC Search is a general strategy that includes both BFS and DFS as specific cases:

- **Breadth-First Search (BFS):** Achieved when $g(x) = 0$ and $f(h(x))$ = level of x.
- **Depth-First Search (DFS):** Achieved when $f(h(x)) = 0$ and g(x) increases with node depth.

16.3.3 LC BRANCH AND BOUND

When LC Search is combined with a bounding function to eliminate nodes that cannot lead to a better solution than the current best, the method is referred to as LC Branch and Bound. This is particularly effective in optimization problems such as:

- 0/1 Knapsack Problem
- Traveling Salesman Problem (TSP)
- Job Scheduling and Resource Allocation

16.3.4 IDEAL VS. PRACTICAL COST FUNCTIONS

In an ideal setting, the cost c(x) of a node x would be the actual cost of the least-cost solution in its subtree. However, calculating this exact cost would require a full search of the subtree, which defeats the purpose of estimating. Therefore, in practice, a heuristic estimate g(x) is used instead.

The choice of an effective g(x) function is crucial for the efficiency of LC search. A poorly chosen estimate may mislead the search and cause unnecessary exploration.

LC Search can be summarized as

- LC Search improves node selection by estimating future cost.
- It generalizes BFS and DFS using a flexible cost function.
- A cost function c(x) = f(h(x)) + g(x) guides the expansion.
- Combining LC Search with bounding yields the LC Branch and Bound strategy.
- It is particularly useful in optimization problems requiring efficient pruning.

16.4 THE15-PUZZLE: AN EXAMPLE

The 15-puzzle is a classic sliding puzzle invented by Sam Loyd in 18716. It consists of 15 numbered square tiles arranged in a 4×4 grid (16 positions total), with one position left blank to allow movement. The goal is to reach a particular arrangement of tiles through a sequence of legal moves that slide adjacent tiles into the blank spot.

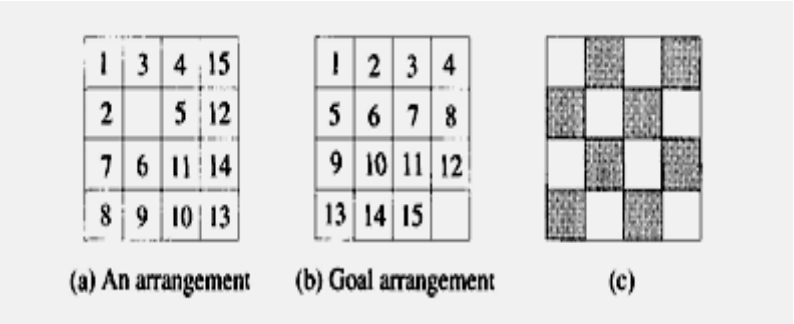


Figure16.1: 15-puzzle arrangements

In the above figure

- shows an initial arrangement (start state).
- is the goal arrangement.
- highlights the shaded positions used in determining reachability.

A move is legal if a tile adjacent to the blank space moves into that empty spot. Each move produces a new state of the puzzle. Hence, solving the puzzle is equivalent to finding a path from the initial state to the goal state in the state space tree.

16.4.1 STATE SPACE TREE

To model the problem as a search task, all reachable states from the initial configuration are structured as a tree, where:

- Each node is a valid puzzle state.
- Edges represent legal moves (up, down, left, right).
- Children of a node are the states that can be reached by moving the empty space.

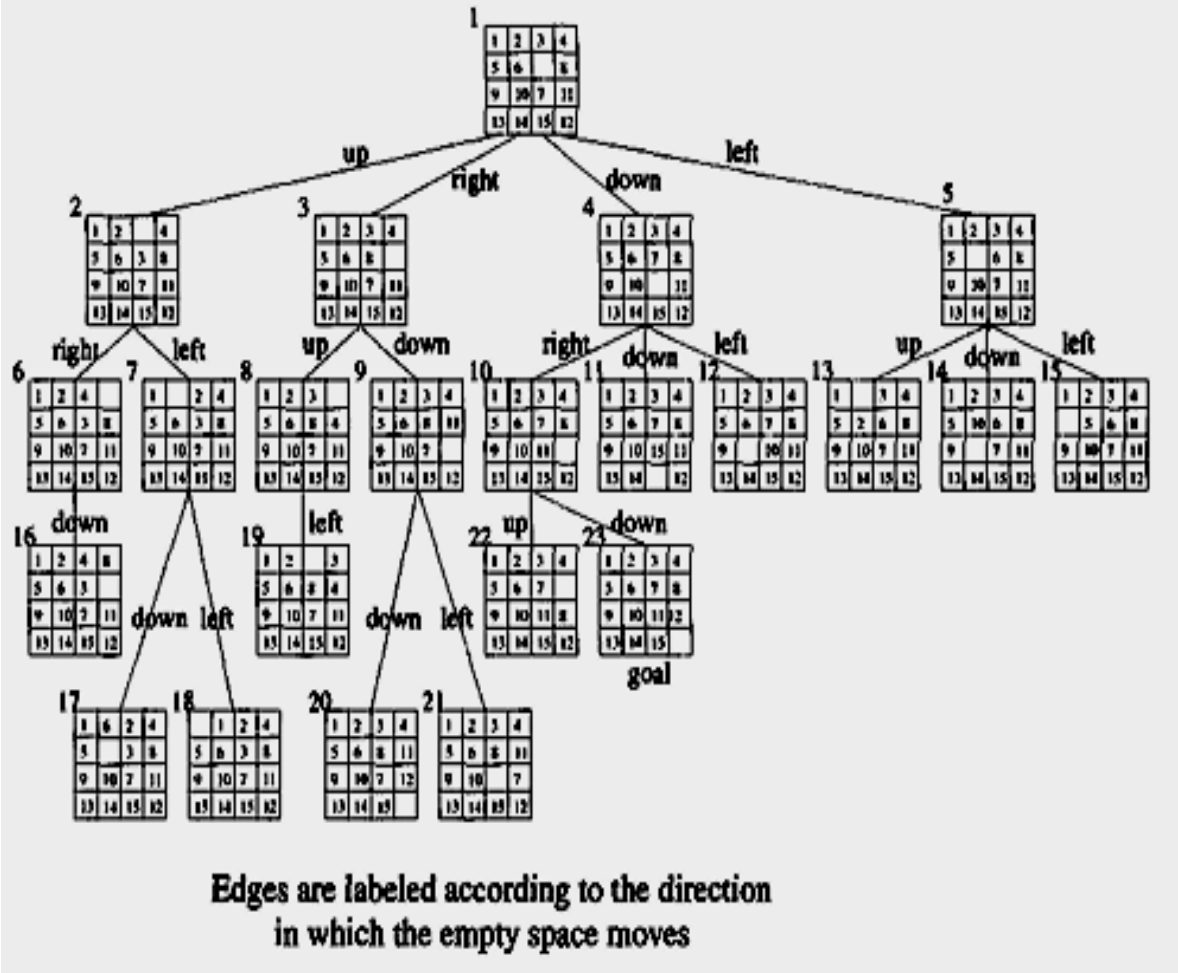


Figure16.2 Part of the state space tree for the 15-puzzle

The above figure represents a portion of the state space tree beginning from the initial state. Each edge is labeled by the direction in which the blank (empty) space moves.

16.4.2 REACHABILITY CHECK

Before exploring the state space, it is crucial to determine whether the goal configuration is reachable from the initial configuration. This is done using Theorem 16.1 from the textbook:
Theorem 16.1:Thegoalstateof Figure16.2(b) is reachable from the initial state.

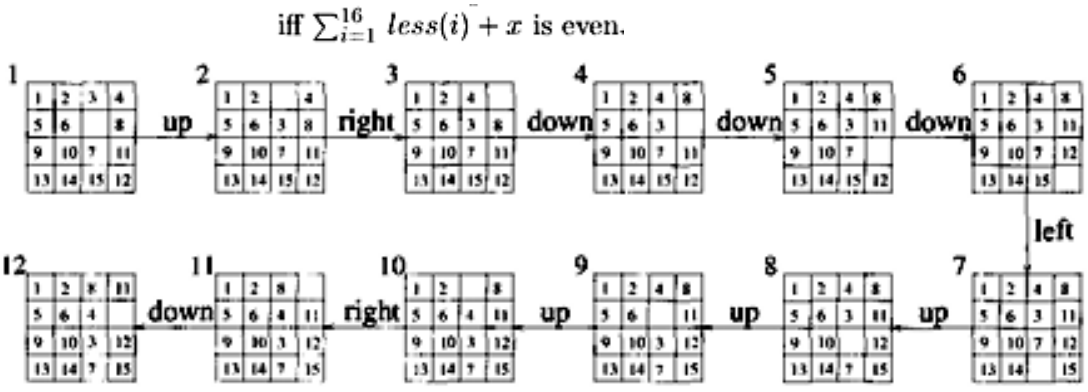


Figure16.3 First ten steps in a depth first search

The above diagram shows a sequence of valid moves in the 15-puzzle, where the blank tile is shifted step-by-step to transform the initial state into the goal configuration. Each move (up, down, left, right) results in a new puzzle state, forming a path toward the solution.

16.4.3 SEARCH STRATEGIES

Various search methods can be used to explore the state space:
1. Depth-First Search (DFS)

- Explores one path fully before backtracking.
- May go deep without reaching the solution.
- Figure 16.4 shows how DFS explores the leftmost path.
- This approach is blind, ignoring how close a state is to the goal.

2. Breadth-First Search (BFS)

- Expands nodes level by level.
- Guaranteed to find the shortest solution path if one exists.
- However, this method may generate a large number of states due to lack of direction.

3. LC Branch and Bound (Least-Cost Search)

- Assigns a cost function c(x) = f(x) + g(x) to each node:
  - f(x) = cost to reach the node (depth level).
  - g(x) = heuristic estimate of moves to reach the goal.
- A common heuristic g(x) is:

Number of misplaced tiles (not in their goal positions).

- The node with the lowest total estimated cost is chosen for expansion.
- Efficient in guiding the search toward the goal.

The 15-puzzle exemplifies the importance of efficient state space exploration in solving combinatorial problems. Methods like LC Branch and Bound, when enhanced with heuristics, outperform blind searches (DFS and BFS) by pruning unnecessary states and prioritizing promising paths.

16.5 CONTROL ABSTRACTIONS FOR LC-SEARCH

The LC (Least Cost) Search is a general state space search method designed to find a solution node efficiently by always expanding the live node with the lowest estimated cost. This strategy improves the performance over blind methods like FIFO (BFS) and LIFO (DFS) by using a heuristic cost function c(x) that predicts how close a node is to a solution.

The control abstraction of LC Search refers to the high-level framework or structure used to implement this search strategy. It defines how the algorithm initializes, selects the next node, generates children, manages live nodes, and terminates.

```
listnode = record {
listnode *next, *parent;
   float cost;}
Algorithm LCSearch(t)
// Search t for an answer node.
{
   if *t is an answer node then output *t and return;
E: = t; // E-node.
   Initialize the list of live nodes to be empty;
   repeat {
      for each child x of E do {
         if x is an answer node then {
            output the path from x to t and return;}
         Add(x); // x is a new live node.
         (x → parent):= E; // Pointer for path to root.
}
      if there are no more live nodes then {
         write ("No answer node");
         return;}
E: = Least ();
   } until (false);}
```

16.5.1 ALGORITHM EXPLANATION : LCSEARCH(T)

The input to the algorithm is a node t, representing the root of the state space tree. The goal is to search this tree for a solution (answer node).

1. **Check if the Root Node is a Solution**:
The algorithm first checks if the root node t is an answer node. If so, it is immediately reported as the solution.
2. **Initialization**:
The E-node E is initialized to t. The list of live nodes is set to empty.
3. **Repeat Until Solution is Found or No More Live Nodes Exist**:
   - For each child x of the current E-node:
   - If x is an answer node, print the path from x to t and terminate.
   - If not, x is added to the list of live nodes and its parent pointer is set to E.
   - If there are no more live nodes, print "No answer node" and stop.
   - Otherwise, the next E-node is chosen using the Least() function, which selects the node from the live list with the minimum cost c(x).

16.5.2 KEY COMPONENTS

- **Live Node List**:

A collection of all nodes that have been generated but not yet expanded. This list is typically implemented as a min-heap, where the node with the lowest c(x) can be efficiently retrieved.

- **Add(x)**:
  A function to insert a new live node x into the live node list.
- **Least()**:
  A function to remove and return the live node with the smallest cost value c(x).
- **Cost Function c(x)**:

An estimated cost function used to guide the search toward promising nodes. If a node is a solution, the cost is actual; otherwise, c(x) is a heuristic estimate.

16.5.3 PARENT POINTER AND PATH TRACING

Each node x that is added to the list of live nodes is associated with a parent pointer, pointing to the node from which it was generated. This allows easy backtracking from a solution node to the root to construct the full path.

The algorithm stops under two conditions:

1. A solution node is found during the expansion of the current E-node.
2. The live node list is empty, meaning all reachable states have been explored without finding a solution.

Termination is guaranteed for finite state spaces. For infinite spaces, termination depends on the design of the cost function c(x)—it must be designed to prefer shallower (or bounded) paths to avoid infinite exploration.

LC Search generalizes other search strategies:

- If the live node list is managed as a queue and c(x) is determined only by depth level, the algorithm behaves like Breadth-First Search (FIFO).
- If the list is a stack, and deeper nodes are preferred, it behaves like Depth-First Search (LIFO).
- When a heuristic is used in c(x), the algorithm becomes an informed search and can significantly reduce the number of nodes explored.

16.6 BOUNDING

The bounding strategy in branch-and-bound methods is used to limit exploration of the solution space. A node in the state space tree is not expanded if it can be determined that no solution in its subtree could improve upon the best solution found so far. This decision is made by comparing a bound on the best solution within the subtree with the current best solution.

Consider the following set of four jobs, each represented as a triple (p_i,d_i,t_i)where:

- p_i: penalty for not executing job i
- d_i: deadline by which job iii must be completed
- t_i: time required to complete the job

Given data:

Job	**p_i**	**d_i**	**t_i**
1	5	1	1
2	10	3	2

3	6	2	1
4	3	1	1

The objective is to select a subset of these jobs such that their execution fits within the available time slots (subject to deadlines), and the total penalty for unexecuted jobs is minimized. Each job requires uninterrupted processing and the processor can handle only one job at a time.

The solution space can be explored using a state space tree. Two distinct tree formulations are presented: variable tuple size and fixed tuple size.

(i) Variable Tuple Size Formulation

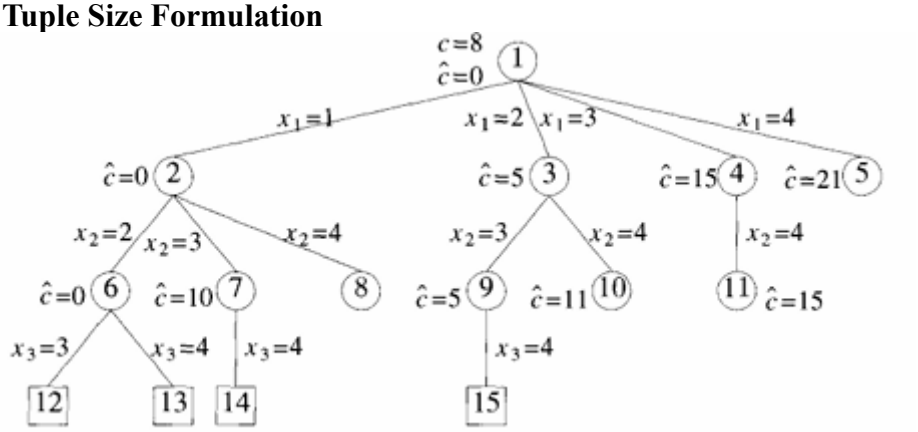


Figure 16.4 State space tree corresponding to variable tuple size formulation

This formulation generates a tree in which each node represents a subset of jobs selected so far. The root node represents the empty set. Each child is created by adding one new job to the parent's subset, provided that constraints on time and deadlines are not violated.

- Circular nodes indicate feasible schedules.
- Square nodes represent infeasible selections due to deadline violations or time overflows.
- Each node is labeled with the set of selected jobs and the penalty incurred by the remaining unselected jobs.
- The tree is grown dynamically, including only those nodes that maintain feasibility.

The optimal solution in this tree appears at Node 9, corresponding to the job subset {2,3}, yielding a minimum total penalty of 16.

(ii) Fixed Tuple Size Formulation

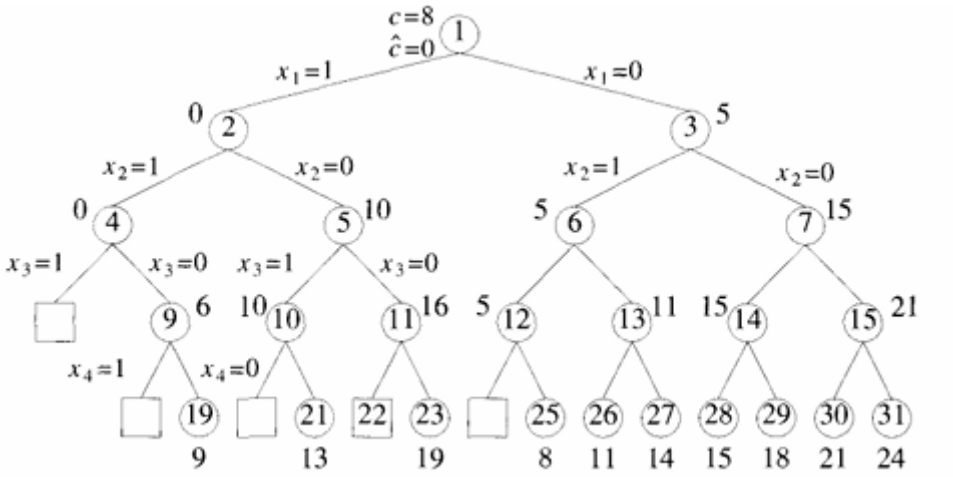


Figure 16.5 State space tree corresponding to fixed tuple size formulation

In this formulation, the tree is constructed by making binary decisions at each level — whether to include or exclude a job.

- Each path from the root to a leaf represents a binary vector of length 4 (one bit per job).
- Every leaf node represents a complete decision about all jobs.
- As in the previous tree, circular nodes denote feasible schedules, and square nodes represent infeasible ones.
- Feasibility is determined by checking both the cumulative execution time and job deadlines.
- Bounding Function and Pruning
- To reduce computation, a bounding function c^(x) is used at each node x. This function provides a lower bound on the cost (penalty) of any solution obtainable from the subtree rooted at x.
- Let:
- c(x): cost (penalty) of the complete solution from node x.
- c^(x): lower bound estimate on c(x).

A node is pruned if c^(x) is greater than or equal to the best cost found so far, since it cannot lead to a better solution.

For instance, in Figure 16.6:

- Node 3 has c=8
- Node 2 has c=9
- The bounding function helps avoid expanding Node 2 and other worse nodes.

The bounding strategy is essential in branch-and-bound algorithms to avoid exhaustive enumeration of all feasible solutions. Both tree formulations successfully identify the optimal subset of jobs with the lowest penalty. The variable tuple size formulation tends to explore only feasible nodes and thus offers early pruning, while the fixed tuple size formulation ensures a complete exploration but may involve additional nodes. The bounding mechanism enhances efficiency by systematically eliminating unpromising paths from consideration.

16.7 FIFOBRANCH-AND-BOUND SOLUTION

The FIFO (First In First Out) Branch-and-Bound method is a systematic strategy for exploring the state space of optimization problems such as the 0/1 Knapsack problem. The

algorithm explores nodes in the order they are generated, using a queue structure. This guarantees level-wise traversal of the state space tree.

FIFO State Space Tree

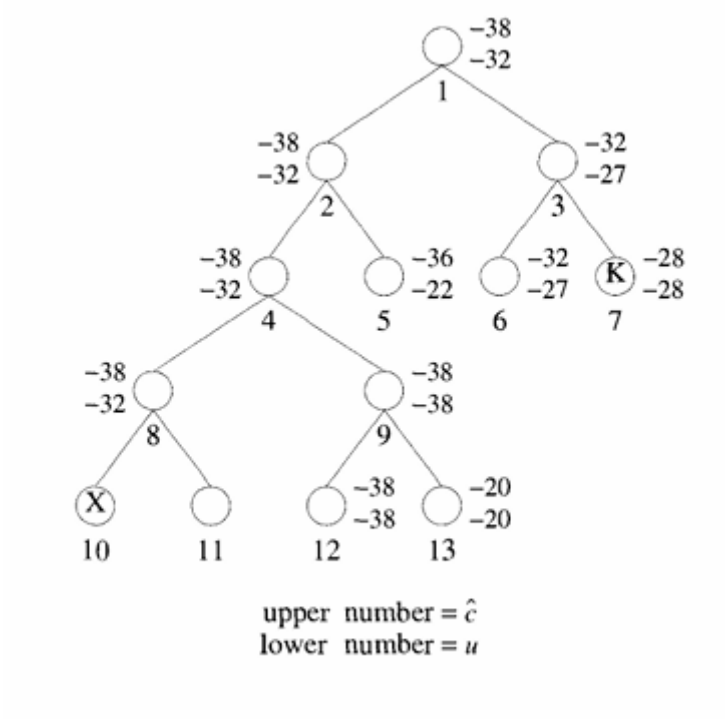


Figure16.6 FIFO branch-and-bound tree

Each node in the tree represents a partial solution. The tree in Figure 16.9 uses two bounding functions:

- c^(x): an upper bound on the cost achievable in the subtree rooted at node x
- u(x): a lower bound (or actual cost) of the solution represented by node x

The search starts at the root node (Node 1) with:

- c^(1)=−38 (upper bound)
- u(1)=−32 (current profit for this node, negated due to minimization)

Tree Traversal Steps:

1. Node 1 is expanded, generating Nodes 2 and 3.
2. Node 2 is dequeued and expanded to Nodes 4 and 5.
3. Node 4 is further expanded into Nodes 8 and 9.
4. The process continues with FIFO order, always expanding the oldest unprocessed node.

Node annotations in the tree:

- **Upper number**: c^(bounding estimate using fractional knapsack logic)
- **Lower number**: u (accumulated cost of selected items)

At Node 7, a feasible solution node is found with a cost of -28, updating the global upper bound. Other nodes are compared against this and pruned accordingly.

BOUNDING STRATEGY

During expansion:

- If a node's lower bound u(x) exceeds the current best solution cost (stored in a global variable), the node is pruned.
- The function c^(x) is calculated using the fractional knapsack heuristic, allowing for a quick but optimistic estimate of best achievable profit.
- Nodes with c^(x)<upper bound are promising and retained for expansion.

The FIFO Branch-and-Bound technique explores the solution tree level-by-level using bounding functions to eliminate unpromising paths. This makes the method systematic and easy to implement, though it can generate more nodes than other strategies such as Least-Cost (LCBB) or LIFO due to its lack of priority-based expansion.

16.8 TRAVELING SALESMAN PROBLEM (*)

The Traveling Salesman Problem (TSP) is a classic optimization problem where a salesman must visit every city once and return to the starting point, such that the total travel cost (or distance) is minimized.

- Given a complete directed graph G=(V,E) where:
  - Vis the set of cities (nodes),
  - E is the set of edges with associated travel costs Cij from city i to city j,
- The goal is to find the minimum-cost tour that starts and ends at city 1 and visits each of the other cities exactly once.

1. State Space Tree for TSP

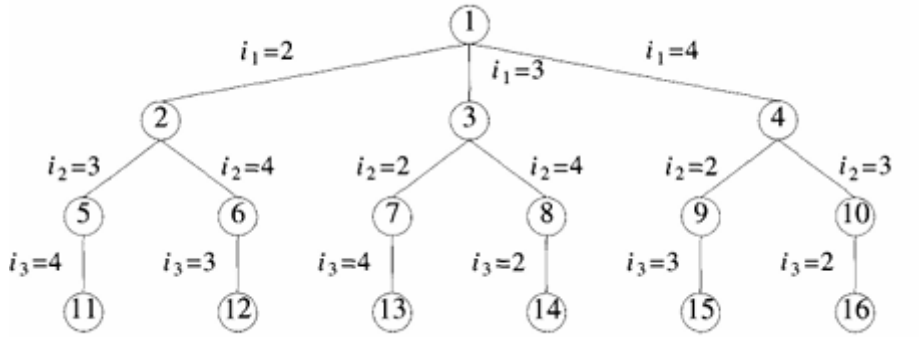


Figure16.7 State space tree for the traveling salesperson problem with n = 4and $i_0 = i_4 = 1$
The above figure shows a state space tree, where:

- The root node represents the starting city (city 1).
- Each level of the tree adds one more city to the current path.
- Each leaf node represents a complete tour returning to city 1.

The tree grows by branching into all unvisited cities, representing different possible tours.

2. Cost Matrix and Reduced Cost Matrix

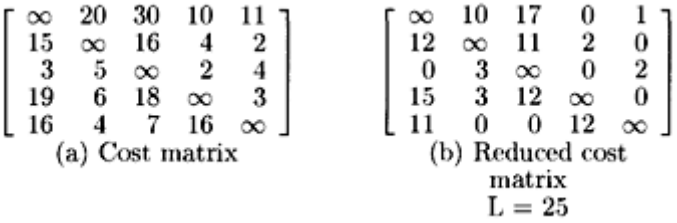


Figure16.8 An example

- Figure 16.11 (a) explainsthe original cost matrix, where:
  - Entry C[i] represents the cost of traveling from city i to city j.
  - ∞ means no direct path or self-loop.
- Figure 16.11(b) represents the reduced cost matrix:
  - Obtained by reducing rows and columns to introduce zeros.
  - This reduction helps estimate a lower bound (L) of the tour cost.
  - In this example, the lower bound is L = 25.

Why reduce?

- Every tour must select one entry per row and one per column.
- Row and column reductions help normalize costs and give a quick lower-bound estimate.

1. LCBB Procedure and State Tree

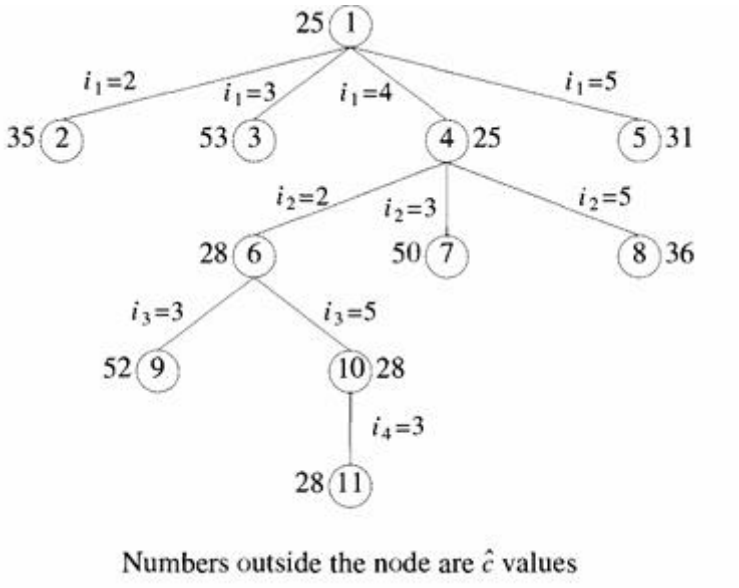


Figure16.9 State space tree generated by procedure LCBB

- LCBB (Least Cost Branch and Bound) is used to efficiently search for the optimal tour.
- Figure 16.12: A state space tree generated using LCBB.
  - Each node has a number (e.g., 25, 35) outside the circle, which is the lower bound cost (ĉ) of the partial path.
  - The LCBB algorithm always expands the node with the smallest c^\hat{c}c^ value (called the _E-node).

Benefit: It avoids expanding unpromising paths, reducing computation time.

2. Reduced Matrices for Nodes

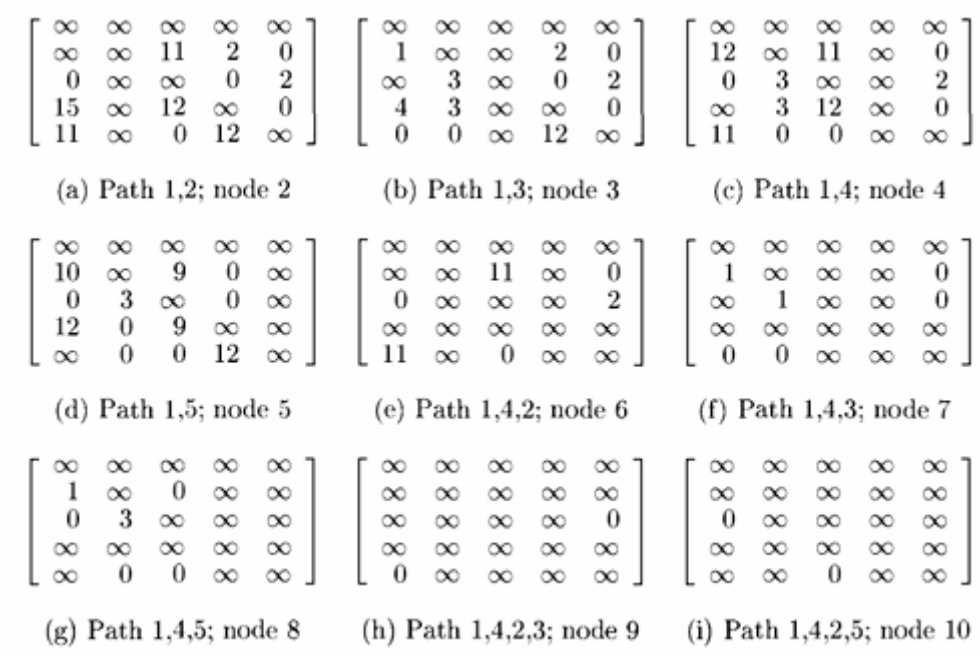


Figure16.10 Reduced cost matrices corresponding to nodes in Figure 16.9

- Each node (partial path) in Figure 16.12 has an associated reduced cost matrix shown in Figure 16.13.
- These matrices are generated by:
  - Setting rows and columns to ∞∞∞∞ to block already-visited cities.
  - Further reduction to find new zeros.
  - The total reduction amount adds to the lower bound.

Example:

- Node 3 (path 1 → 3):
  - Cost so far = 25,
  - Cost of edge (1,3) = 17 (from reduced matrix),
  - Additional reduction = 11,
  - So, c^=25+17+11=53.

3. Dynamic LCBB Tree (Figures 16.15 and 16.16)

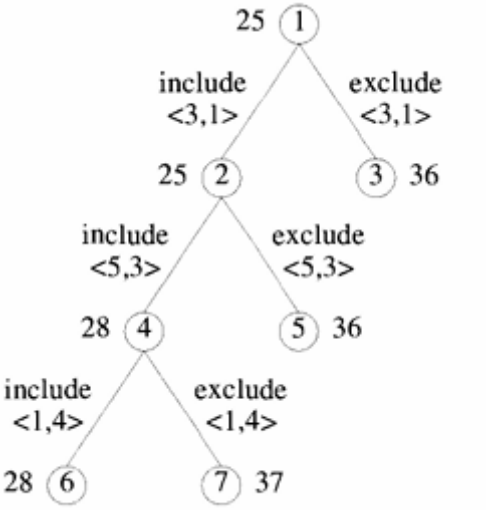


Figure16.11 State space tree for Figure.16.8a

- Instead of a static tree, we can also build a dynamic binary state space tree:
  - Left branch → Include edge (i, j),
  - Right branch → Exclude edge (i, j).

The Figure 16.15 shows a dynamic tree where branching decisions are made based on edge inclusion/exclusion.

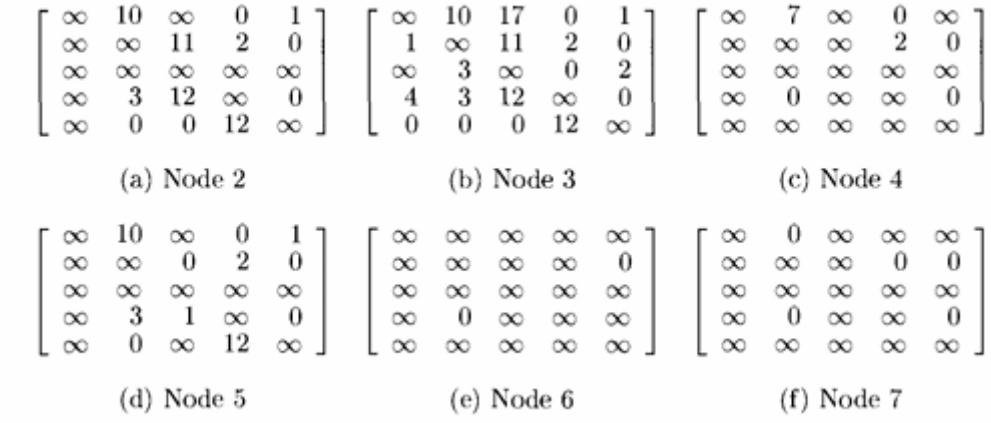


Figure16.12 Reduced cost matrices for Figure 16.11

The above figure shows reduced cost matrices after branching.For each node, edge selections and matrix updates ensure no cycles or repeated visits.

6. LCBB Algorithm for TSP

Step-by-Step:

1. Start with the initial cost matrix and reduce it.
2. Compute a lower bound on the cost (sum of reductions).
3. Create a state space tree, where each node:
   - Represents a partial path,
   - Has a reduced matrix and a c^ value.

1. Select the node with the smallest cost (E-node).
2. Expand the E-node by adding child nodes with updated paths and costs.
3. Repeat until a complete tour is found.
4. Prune nodes with c^ higher than known best tour (upper bound).

Traveling Salesman Problem can be summarized as

- TSP is NP-hard but can be tackled efficiently for small/medium-sized instances using Branch and Bound techniques like LCBB.
- The state space trees (static and dynamic) and cost matrix reductions are powerful tools to prune the search space.
- Using these figures (16.10–16.16), we clearly see how costs are calculated, paths are expanded, and decisions are made to find the optimal tour efficiently.

16.9 SUMMARY

Branch and Bound is a general algorithmic framework for solving combinatorial optimization problems by systematically exploring and pruning a solution space tree. The technique uses bounding functions to discard subtrees that cannot yield better results, greatly improving efficiency over brute-force enumeration. The Least-Cost (LC) Search strategy enhances performance by ranking nodes based on heuristic cost, while variants like FIFO Branch and Bound offer structured exploration. Applications such as the 15-Puzzle, Job Scheduling, and Traveling Salesman Problem (TSP) demonstrate how bounding and search control together enable optimal solutions in large search spaces.

16.10 KEY TERMS

Branch and Bound E-node Bounding Function LC Search Heuristic FIFO Queue LCBB

16.11 REVIEW QUESTIONS

1. Explain the principle of the Branch and Bound technique.
2. What is the role of the bounding function in pruning?
3. Define the cost function used in LC Search.
4. Compare BFS, DFS, and LC Branch and Bound strategies.
5. Describe how the 15-Puzzle can be modeled as a state-space search problem.
6. Explain the difference between variable and fixed tuple size formulations in job scheduling.
7. What are the advantages of using LCBB for solving the Traveling Salesman Problem?

16.12 SUGGESTED READINGS

1. Ellis Horowitz, Sartaj Sahni, Fundamentals of Computer Algorithms.
2. Cormen, Leiserson, Rivest, and Stein, Introduction to Algorithms.
3. Aho, Hopcroft, Ullman, Design and Analysis of Computer Algorithms.
4. T. H. Cormen, Algorithmic Problem-Solving Approaches.

Mrs. Appikatla PushpaLatha