MCA T11 - Design and Analysis of Algorithms [40L]

1. Introduction and Basic Concepts [11L]

- **Basics**:
    - Definition, properties, problems, and instances.
    - Elementary operations.
- **Time and Space Complexity**:
    - Worst-case, best-case, and average-case analysis.
- **Asymptotic Notations**:
    - Big-O, Big-Theta, Big-Omega.
- **Solving Recurrence Relations**:
    - Substitution method, iteration method, change of variables, recursion trees, and master theorem.
- **Review of Basic Data Structures**:
    - Stack, queue, linked list, tree.
- **Algorithm Design Techniques**:
    - Brute Force and Exhaustive Search.
    - Decrease and Conquer.
    - Divide and Conquer.
    - Transform and Conquer.
    - Dynamic Programming.
    - Iterative Improvement.
    - Greedy Technique.
    - Backtracking.
    - Branch and Bound.

2. Sorting and Searching [6L]

- **Comparison Sort with Analysis**:
    - Heap sort, radix sort, quicksort, merge sort.
    - Lower bound for comparison-based sorting.
- **Linear Time Sorting**:
    - Counting sort, radix sort, bucket sort.
- **Searching Techniques**:

- Linear search, modified linear search, binary search (with analysis).

3. Data Structures for Disjoint Sets [2L]

- **Set Representation**:
  - Tree representation of a set.
- **Union and Find Methods**:
  - Algorithms, improvement by rank, and path compression heuristics.
  - Analysis and applications.

4. Graph Problems [8L]

- **Graph Representation**:
  - Adjacency matrix, adjacency list.
- **Graph Traversal**:
  - Breadth-first search (BFS), depth-first search (DFS).
- **Graph Sorting**:
  - Topological sorting.
- **Graph Components**:
  - Strongly connected components.
- **Minimum Spanning Trees**:
  - Kruskal's algorithm, Prim's algorithm.
- **Shortest Path**:
  - Single-source shortest path, Dijkstra's algorithm, Bellman-Ford algorithm.

5. Backtracking [3L]

- **Basic Terminologies**.
- **Problems**:
  - N-Queens problem.
  - Hamiltonian circuit problem.
  - Subset sum problem.

6. Branch and Bound [2L]

- **Basic Terminologies**.
- **Problems**:
  - Knapsack problem.
  - Travelling salesman problem (TSP).

7. String Processing [4L]

- **String Searching and Pattern Matching**.
- **String Matching Algorithms (with Analysis)**:
    - Naïve method.
    - Finite automata method.
    - Knuth-Morris-Pratt (KMP) method.

8. Conceptual Introduction to NP-Completeness [4L]

- **Deterministic and Non-deterministic Algorithms**.
- **P and NP**:
    - Definitions and concepts.
- **NP-Completeness**:
    - Statement of Cook's theorem.
- **Standard NP-Complete Problems**.
- **Approximation Algorithms**.

1.1 Basics

- **Algorithm**:
  An algorithm is a finite sequence of well-defined instructions designed to solve a specific problem. Each step of the algorithm should be unambiguous and lead to a clear outcome. Algorithms are the core of computational problem-solving.

Key properties of algorithms:

1. **Input**: Zero or more inputs are externally supplied to the algorithm.
2. **Output**: An algorithm must produce at least one output.
3. **Finiteness**: An algorithm must terminate after a finite number of steps.
4. **Definiteness**: Each instruction must be clear and unambiguous.
5. **Effectiveness**: Each step of the algorithm must be basic enough to be carried out, in principle, by a human or machine using a pen and paper.

1.2 Algorithmic Problems

- **Problem**:
  A task that requires a solution, typically presented in terms of an input-output relationship.
- **Problem Instance**:
  A specific case of a problem with actual values for the input. For example, if the problem is sorting an array, the problem instance could be an array like [5, 2, 8, 1].
- **Examples of Algorithmic Problems**:
  1. **Sorting**: Organize a list of numbers in increasing or decreasing order.
  2. **Searching**: Find a particular value in a collection of items.
  3. **Graph Problems**: Find the shortest path between nodes, detect cycles, etc.

1.3 Elementary Operations

- **Elementary Operations**:
  These are the basic computational steps that an algorithm performs. These steps generally have a constant time complexity, meaning that the time to execute the operation does not depend on the size of the input.

Examples of elementary operations:

- Assigning a value to a variable (e.g., x = 10).
- Arithmetic operations (e.g., addition a + b, subtraction a - b, multiplication a * b).
- Comparisons (e.g., a > b, a == b).
- Accessing elements in an array (e.g., arr[i]).
- Incrementing or decrementing counters (e.g., i++ or i--).

Why elementary operations matter:
The performance of an algorithm is often analyzed based on how many times it performs elementary operations. These are key to understanding the time complexity of an algorithm, as more elementary operations often imply a higher running time.

1.4 Characteristics of Good Algorithms

A well-designed algorithm has the following characteristics:

1. **Correctness**:
   The algorithm should produce the correct output for all valid inputs.

2. **Efficiency**:
   The algorithm should run in the least possible time and use the least amount of memory. Efficiency is typically analyzed in terms of:
   - **Time Complexity**: The amount of time an algorithm takes to complete, usually measured as a function of the input size (n).
   - **Space Complexity**: The amount of memory the algorithm uses, also a function of the input size.
3. **Scalability**:
   The algorithm should scale well with increasing input sizes. Algorithms that perform well on small inputs may not be practical for large datasets.
4. **Clarity and Simplicity**:
   The algorithm should be easy to understand, implement, and maintain.

1.5 Classification of Algorithms

Algorithms can be classified based on various parameters:

1. **Design Paradigm**:
   - **Divide and Conquer**: Split the problem into subproblems, solve each subproblem, and combine their solutions.
   - **Dynamic Programming**: Break down a problem into simpler overlapping subproblems and solve them only once, storing their results.
   - **Greedy Algorithms**: Make locally optimal choices at each step with the hope of finding a global optimum.
   - **Backtracking**: Explore possible solutions and backtrack if a certain solution path is not viable.
2. **Complexity**:
   - **Time complexity** (e.g., O(n), O(n^2), O(log n), etc.).
   - **Space complexity** (e.g., O(1), O(n)).
3. **Type of Problems Solved**:
   - **Searching**: Binary Search, Linear Search.
   - **Sorting**: Merge Sort, Quick Sort, Bubble Sort.
   - **Graph Algorithms**: Dijkstra's Algorithm, Depth-First Search (DFS), Breadth-First Search (BFS).

Summary:

- An algorithm is a clear and finite set of instructions to solve a problem.
- Key properties include input, output, definiteness, finiteness, and effectiveness.
- Elementary operations form the building blocks of algorithms and affect time complexity.
- The design and classification of algorithms can help optimize solutions for different problems.

Time and Space Complexity

1.6 Time and Space Complexity

- **Time Complexity**:
  Time complexity refers to the amount of time an algorithm takes to run as a function of the size of the input. It's a measure of the efficiency of an algorithm in terms of time, and it helps us understand how an algorithm's runtime increases with input size.
- **Space Complexity**:
  Space complexity refers to the amount of memory space an algorithm uses during its execution, also as a function of the size of the input. This includes the memory for input data, auxiliary storage, and the call stack in recursive algorithms.
- Time and space complexities are typically expressed using **Big O notation** to describe the **upper bounds** on the growth of the algorithm's runtime or memory usage.

1.6.1 Time Complexity Analysis

There are three major cases considered when analyzing the time complexity of an algorithm:

1. **Worst-Case Complexity**:
   - Represents the maximum amount of time an algorithm can take for any input of size n.
   - It is the most commonly used measure because it provides an upper bound on the running time, ensuring the algorithm will not exceed this time limit.
   - Denoted by O(n) for an algorithm where the time grows linearly with input size n.

Example:
In **linear search**, where we search for an element in an unsorted list, the worst case occurs

when the element is not found, requiring all n elements to be checked.
Worst-case time complexity: **O(n)**.

2. **Best-Case Complexity**:
    - Represents the minimum amount of time an algorithm can take for an input of size n.
    - It gives an optimistic scenario and can sometimes be misleading, as the best-case scenario may rarely occur.

Example:
In **linear search**, the best case occurs when the element is the first one in the list.
Best-case time complexity: **O(1)**.

3. **Average-Case Complexity**:
    - Represents the expected time an algorithm will take for an average input of size n. It is often calculated using probability and reflects the typical scenario.
    - Useful when the input is random, but harder to compute than worst-case complexity.

Example:
In **linear search**, the average case assumes the element might be located somewhere in the middle of the list on average.
Average-case time complexity: **O(n/2) ≈ O(n)**.

1.6.2 Common Time Complexity Classes

Here are the most common time complexity classes, in order of increasing time complexity:

1. **Constant Time: O(1)**
    - The algorithm's running time does not change with input size.
    - Example: Accessing an element in an array by index.
2. **Logarithmic Time: O(log n)**
    - The time complexity grows logarithmically with input size. Algorithms that divide the problem in half at each step often have this complexity.
    - Example: Binary search.
3. **Linear Time: O(n)**
    - The running time increases directly in proportion to the input size.
    - Example: Linear search.
4. **Linearithmic Time: O(n log n)**

- The running time is a combination of linear and logarithmic growth. Many efficient sorting algorithms like Merge Sort and Quick Sort have this time complexity.
- Example: Merge Sort.

5. **Quadratic Time: $O(n^2)$**
   - The running time grows quadratically with the input size. Nested loops often lead to quadratic time complexity.
   - Example: Bubble sort, selection sort.
6. **Cubic Time: $O(n^3)$**
   - The running time grows cubically with the input size. Algorithms with three nested loops typically exhibit cubic complexity.
   - Example: Some matrix multiplication algorithms.
7. **Exponential Time: $O(2^n)$**
   - The running time doubles with each additional input element. Algorithms with exponential time complexity are impractical for large inputs.
   - Example: Recursive algorithms for solving problems like the traveling salesman problem (brute-force).
8. **Factorial Time: O(n!)**
   - The running time grows factorially with the input size. Algorithms with this complexity are infeasible for even moderately large inputs.
   - Example: Brute-force solution to the traveling salesman problem.

1.6.3 Space Complexity Analysis

Just like time complexity, space complexity is evaluated based on the input size, and we focus on the amount of extra memory used by an algorithm beyond the input data.

1. **Auxiliary Space**:
   This refers to the extra memory or temporary space used by an algorithm aside from the input data. For example, variables, recursive call stack, or data structures like arrays, stacks, or queues used by the algorithm.
2. **Input Space**:
   This is the memory occupied by the input itself. Space complexity often focuses on auxiliary space because the input space is necessary and cannot be avoided.

1.6.4 Space Complexity Examples

1. **Constant Space: O(1)**
    - The algorithm uses a fixed amount of space, regardless of the input size.
    - Example: Algorithms that simply use a few variables like counters or pointers.
2. **Linear Space: O(n)**
    - The algorithm's space requirement grows linearly with the input size.
    - Example: Storing an array of size n, or using recursion with n levels of depth.
3. **Quadratic Space: O(n^2)**
    - The space requirement grows quadratically with the input size. This often happens when a two-dimensional array or matrix is used.
    - Example: Dynamic programming algorithms like the one used in solving the longest common subsequence (LCS) problem.

1.6.5 Trade-offs Between Time and Space

Some algorithms trade-off time for space and vice versa. For example:

- **Memoization** in dynamic programming reduces time complexity by storing intermediate results but increases space complexity.
- **Iterative algorithms** often use less space (since they avoid the recursion stack), but sometimes increase time complexity.

Summary

- **Time complexity** measures the time an algorithm takes to complete, while **space complexity** measures the memory it uses.
- Time complexity can be evaluated based on the worst-case, best-case, and average-case scenarios.
- Different algorithms belong to different **complexity classes** (e.g., constant, logarithmic, linear, etc.), and the choice of algorithm depends on the trade-off between time and space efficiency.

Understanding both time and space complexity is crucial for selecting the right algorithm based on problem size and computational resource limits.

Asymptotic Notations

Asymptotic notations are used to describe the running time or space requirements of an algorithm in terms of input size, especially for large inputs. These notations provide a high-

level understanding of an algorithm's efficiency without worrying about machine-specific constants.

1.7 Asymptotic Notations

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

These notations describe the **upper**, **lower**, and **tight bounds** of an algorithm's performance as the input size (n) approaches infinity.

1.7.1 Big-O Notation (O)

- **Big-O Notation** is used to describe the **upper bound** or **worst-case time complexity** of an algorithm.
- It tells us the maximum time an algorithm can take relative to the input size as it grows.

Formal Definition:

- An algorithm is said to be **O(f(n))** if there exist positive constants c and n0 such that for all n ≥ n0, the running time of the algorithm is less than or equal to c * f(n).

Mathematical Representation:

- T(n) = O(f(n)) means:
  There exist constants c > 0 and n0 > 0 such that for all n ≥ n0,
  T(n) ≤ c * f(n).
- **Example**: Suppose the time complexity of an algorithm is T(n) = 2n + 3. The dominant term is n, and the constant coefficients (2 and 3) are ignored in Big-O notation. Therefore, we say T(n) = O(n).
- **Interpretation**: Big-O provides the **worst-case** scenario, giving an upper bound on the time complexity. It's helpful for ensuring that an algorithm will not take longer than O(f(n)) even in the worst case.

1.7.2 Big-Theta Notation (Θ)

- **Big-Theta Notation** gives a **tight bound** on the time complexity, meaning it represents both the **upper** and **lower bounds** of an algorithm's runtime. It describes the **exact** growth rate for large input sizes.

Formal Definition:

- An algorithm is said to be **Θ(f(n))** if there exist positive constants c1, c2, and n0 such that for all n ≥ n0, the running time is sandwiched between two functions:
  c1 * f(n) ≤ T(n) ≤ c2 * f(n).

Mathematical Representation:

- T(n) = Θ(f(n)) means:
  There exist constants c1 > 0, c2 > 0, and n0 > 0 such that for all n ≥ n0,
  c1 * f(n) ≤ T(n) ≤ c2 * f(n).
- **Example**: For the function T(n) = 5n + 10, the time complexity is exactly proportional to n, so T(n) = Θ(n).
- **Interpretation**: Big-Theta describes the **average-case** time complexity, meaning it provides a **tight bound** and represents the growth rate of an algorithm in both the worst and best scenarios.

1.7.3 Big-Omega Notation (Ω)

- **Big-Omega Notation** is used to describe the **lower bound** or **best-case time complexity** of an algorithm.
- It tells us the minimum time the algorithm can take relative to the input size.

Formal Definition:

- An algorithm is said to be **Ω(f(n))** if there exist positive constants c and n0 such that for all n ≥ n0, the running time of the algorithm is at least c * f(n).

Mathematical Representation:

- T(n) = Ω(f(n)) means:
  There exist constants c > 0 and n0 > 0 such that for all n ≥ n0,
  T(n) ≥ c * f(n).
- **Example**: In the case of **binary search**, the best case occurs when the element is found at the first step. Therefore, the time complexity for the best case is T(n) = Ω(1).
- **Interpretation**: Big-Omega provides the **best-case** scenario, offering a lower bound on the running time. It shows that the algorithm will not run faster than Ω(f(n)).

1.7.4 Relationship Between Notations

1. **Big-O** gives the upper bound:
   - Tells us the **maximum time** the algorithm will take in the worst case.
2. **Big-Theta** gives the tight bound:

- Represents the **exact time complexity** of the algorithm, considering both worst and best cases.

3. **Big-Omega** gives the lower bound:
    - Shows the **minimum time** the algorithm will take in the best case.

In some cases, an algorithm may have the same asymptotic bounds for best, average, and worst cases. For example, in merge sort, the time complexity is O(n log n), Θ(n log n), and Ω(n log n) because the algorithm performs the same in every scenario.

1.7.5 Examples of Asymptotic Notations

1. **Example 1: Linear Search**
    - **Worst case**: The element is at the last position or not found at all.
      Time complexity: **O(n)**.
    - **Best case**: The element is at the first position.
      Time complexity: **Ω(1)**.
    - **Average case**: The element is somewhere in the middle.
      Time complexity: **Θ(n)**.
2. **Example 2: Binary Search**
    - **Worst case**: The element is not found, and the search reaches the smallest subarray.
      Time complexity: **O(log n)**.
    - **Best case**: The element is found at the first mid-point check.
      Time complexity: **Ω(1)**.
    - **Average case**: On average, binary search cuts down the problem size by half in each iteration.
      Time complexity: **Θ(log n)**.
3. **Example 3: Merge Sort**
    - Merge sort consistently divides the problem in half and merges subproblems.
    - **Worst, best, and average cases** all have the same time complexity since the algorithm always performs n log n work. Time complexity: **O(n log n)**, **Θ(n log n)**, and **Ω(n log n)**.

Summary

- **Big-O** notation (O) gives the upper bound, typically used to analyze the worst-case scenario of an algorithm.

- **Big-Theta** notation (Θ) gives the tight bound, describing the exact time complexity in the average case.
- **Big-Omega** notation (Ω) gives the lower bound, often used to describe the best-case time complexity.
- Asymptotic notations abstract away constants and lower-order terms, helping compare algorithms based on their growth rate.

Solving Recurrence Relations

Recurrence relations arise when the time complexity of a recursive algorithm depends on the time complexity of smaller subproblems. These relations express the overall time complexity in terms of the problem size, and solving them gives the exact or asymptotic time complexity of the algorithm.

2.1 Substitution Method

The **substitution method** is used to prove a solution by making an educated guess and then verifying that guess using mathematical induction.

Steps:

1. **Guess the form of the solution**.
2. **Prove the guess is correct** using mathematical induction.

Example: Consider the recurrence relation: $T(n) = 2T\left(\frac{n}{2}\right) + n$

- **Step 1 (Guess the solution)**: Let's guess the solution is of the form $T(n) = O(n \log n)$.
- **Step 2 (Induction base case)**: Prove that the guess holds for the base case, say $n = 1$. For small n, $T(1)$ is constant (e.g., $T(1) = O(1)$).
- **Step 3 (Induction hypothesis)**: Assume the guess holds for all n' such that $n' < n$.
- **Step 4 (Induction step)**: Substituting the hypothesis into the recurrence: $T(n) = 2T\left(\frac{n}{2}\right) + n$ By the inductive hypothesis: $T\left(\frac{n}{2}\right) = O\left(\frac{n}{2} \log \frac{n}{2}\right)$ Substituting this back: $T(n) = 2 \cdot \frac{n}{2} \log \frac{n}{2} + n$ Simplifying: $T(n) = n \log n - n + n = O(n \log n)$

Thus, the guess $T(n) = O(n \log n)$ holds, and the time complexity is $O(n \log n)$.

2.2 Iteration Method

The **iteration method** involves expanding the recurrence relation step by step and finding a pattern. This is often used for simpler recurrence relations.

Steps:

1. **Expand the recurrence** step by step.
2. **Find a pattern or closed form** for the number of steps.
3. **Solve the resulting summation**.

Example: Consider the recurrence: $T(n) = T(n-1) + 1$ Expanding the recurrence step by step:

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

Thus, after expanding for n steps, we get: $T(n) = 1 + 1 + ... + 1 = n$ Therefore, $T(n) = O(n)$.

2.3 Change of Variables Method (or Substitution)

In the **change of variables** method, we simplify the recurrence by transforming it into a more familiar form, often by changing the variable to reduce complexity.

Steps:

1. **Change the variable** to simplify the recurrence.
2. **Solve the transformed recurrence**.
3. **Substitute back** the original variables.

Example: Consider the recurrence: $T(n) = 2T\left(\sqrt{n}\right) + \log n$

Let $n = 2^m$. Then $\log n = m$, and the recurrence becomes: $T(2^m) = 2T(2^{m/2}) + m$

Now, let $S(m) = T(2^m)$, and the recurrence becomes: $S(m) = 2S(m/2) + m$

This is a simpler recurrence that can be solved using methods like recursion trees or the master theorem.

2.4 Recursion Tree Method

The **recursion tree method** visualizes the recurrence as a tree where each node represents the cost of a subproblem. By summing the costs at each level of the tree, we can estimate the total cost.

Steps:

1. **Draw the recursion tree**, starting with the original problem at the root.

2. **Label each node** with the cost at that level.
3. **Sum the costs** across all levels to get the total cost.

Example: Consider the recurrence: $T(n) = 2T\left(\frac{n}{2}\right) + n$

- At the root level, the cost is n.
- At the next level, there are two subproblems of size $\frac{n}{2}$, each with a cost of $\frac{n}{2}$, so the total cost is n.
- At the next level, there are four subproblems of size $\frac{n}{4}$, each with a cost of $\frac{n}{4}$, so the total cost is again n.

This process repeats for $\log n$ levels, with the cost at each level being n. Thus, the total cost is: $T(n) = n \log n$

Therefore, the time complexity is $O(n \log n)$.

2.5 Master Theorem

The **Master Theorem** provides a direct way to solve recurrences of the form: $T(n) = aT\left(\frac{n}{b}\right) + f(n)$ where $a \geq 1$ and $b > 1$ are constants, and $f(n)$ is an asymptotically positive function.

The Master Theorem categorizes the solution into three cases based on the relationship between $f(n)$ and $n^{\log_b a}$:

- **Case 1**: If $f(n) = O(n^{\log_b a - \epsilon})$ for some $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
- **Case 2**: If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$.
- **Case 3**: If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some $\epsilon > 0$ and if $af\left(\frac{n}{b}\right) \leq cf(n)$ for some constant $c < 1$, then $T(n) = \Theta(f(n))$.

Example: Consider the recurrence: $T(n) = 2T\left(\frac{n}{2}\right) + n$

Here, $a = 2$, $b = 2$, and $f(n) = n$.

- $n^{\log_b a} = n^{\log_2 2} = n$.
- Since $f(n) = \Theta(n)$, this matches **Case 2** of the Master Theorem.

Thus, the time complexity is $T(n) = \Theta(n \log n)$.

Summary

- **Substitution method**: Guess the solution and prove by induction.
- **Iteration method**: Expand the recurrence step by step and find a pattern.
- **Change of variables**: Simplify the recurrence by transforming variables.
- **Recursion trees**: Visualize the recurrence as a tree and sum the costs.
- **Master theorem**: Directly apply to solve common divide-and-conquer recurrences.

Review of Basic Data Structures

Data structures are fundamental in algorithm design as they enable efficient data storage, manipulation, and retrieval. Below is an overview of some essential data structures: **stack**, **queue**, **linked list**, and **tree**.

---

1. Stack

A **stack** is a linear data structure that follows the **LIFO** (Last In, First Out) principle. This means that the last element inserted into the stack is the first one to be removed.

1.1 Basic Operations:

1. **push(x)**: Add an element x to the top of the stack.
2. **pop()**: Remove and return the element from the top of the stack.
3. **peek() / top()**: Return the element on the top of the stack without removing it.
4. **isEmpty()**: Check whether the stack is empty.
5. **size()**: Return the number of elements in the stack.

1.2 Time Complexity:

- Push: $O(1)$
- Pop: $O(1)$
- Peek: $O(1)$

1.3 Applications:

- **Function call management** in recursion.
- **Undo mechanisms** in text editors.
- **Expression evaluation** (e.g., conversion of infix to postfix).

1.4 Example:

Consider a sequence of stack operations:

- **push(10)** → [10]
- **push(20)** → [10, 20]
- **pop()** → [10] (returns 20)
- **peek()** → [10] (top is 10)

---

2. Queue

A **queue** is a linear data structure that follows the **FIFO** (First In, First Out) principle. The first element inserted is the first one to be removed.

2.1 Basic Operations:

1. **enqueue(x)**: Add an element x to the rear of the queue.
2. **dequeue()**: Remove and return the element from the front of the queue.
3. **front()**: Return the front element without removing it.
4. **isEmpty()**: Check whether the queue is empty.
5. **size()**: Return the number of elements in the queue.

2.2 Time Complexity:

- Enqueue: $O(1)$
- Dequeue: $O(1)$
- Peek (front): $O(1)$

2.3 Applications:

- **Scheduling** processes in operating systems (e.g., Round Robin).
- **Breadth-First Search (BFS)** in graph traversal.
- **Print job scheduling** in printers.

2.4 Example:

Consider a sequence of queue operations:

- **enqueue(5)** → [5]
- **enqueue(10)** → [5, 10]
- **dequeue()** → [10] (returns 5)
- **front()** → [10] (front is 10)

---

3. Linked List

A **linked list** is a linear data structure where elements are stored in **nodes**, and each node points to the next one in the sequence. Unlike arrays, linked lists provide dynamic memory allocation and can efficiently handle insertions and deletions.

3.1 Types of Linked Lists:

1. **Singly Linked List**: Each node has a data field and a pointer (reference) to the next node.
2. **Doubly Linked List**: Each node has a data field, a pointer to the next node, and a pointer to the previous node.
3. **Circular Linked List**: The last node points back to the first node, creating a circular structure.

3.2 Basic Operations:

1. **insertAtBeginning(x)**: Insert x at the beginning of the list.
2. **insertAtEnd(x)**: Insert x at the end of the list.

3. **delete(x)**: Delete the first occurrence of x in the list.
4. **search(x)**: Search for an element x in the list.
5. **traverse()**: Traverse through all the elements of the list.

3.3 Time Complexity:

- Insertion at beginning: $O(1)$
- Insertion at end: $O(n)$ (or $O(1)$ if tail pointer is used)
- Deletion: $O(n)$
- Search: $O(n)$

3.4 Applications:

- **Dynamic memory allocation**.
- **Implementation of stacks and queues**.
- **Undo functionality** in software applications.

3.5 Example:

A singly linked list of three nodes:

```
Head -> [10 | *] -> [20 | *] -> [30 | NULL]
```

- Insert at the beginning:
  New list: `Head -> [5 | *] -> [10 | *] -> [20 | *] -> [30 | NULL]`

---

4. Tree

A **tree** is a hierarchical data structure made up of **nodes**, where each node contains a value and references to child nodes. A tree is a collection of nodes that is **acyclic** and connected.

4.1 Terminology:

- **Root**: The topmost node in a tree (no parent).
- **Child**: A node that is a descendant of another node.
- **Parent**: A node that has child nodes.
- **Leaf**: A node with no children.
- **Subtree**: A portion of a tree that is itself a tree.

- **Depth**: The length of the path from the root to a node.
- **Height**: The number of edges on the longest path from a node to a leaf.

4.2 Types of Trees:

1. **Binary Tree**: Each node has at most two children (left and right).
2. **Binary Search Tree (BST)**: A binary tree where the left child contains values smaller than the parent, and the right child contains values greater than the parent.
3. **AVL Tree**: A balanced binary search tree where the height difference between left and right subtrees is at most 1.
4. **Heap**: A complete binary tree where the parent node is either greater than (max-heap) or less than (min-heap) its children.

4.3 Basic Operations:

1. **insert(x)**: Insert a value x into the tree.
2. **delete(x)**: Delete a value x from the tree.
3. **search(x)**: Search for a value x in the tree.
4. **traverse()**: Visit all the nodes in a specific order (e.g., in-order, pre-order, post-order).

4.4 Traversal Methods:

1. **In-order Traversal**: Visit the left subtree, the node, and then the right subtree (left-root-right).
    - Used to get the nodes of a **binary search tree** in ascending order.
2. **Pre-order Traversal**: Visit the node, the left subtree, and then the right subtree (root-left-right).
    - Used to create a **copy** of the tree.
3. **Post-order Traversal**: Visit the left subtree, the right subtree, and then the node (left-right-root).
    - Used for **deleting** the tree.
4. **Level-order Traversal**: Visit nodes level by level using a queue (also known as breadth-first traversal).

4.5 Time Complexity:

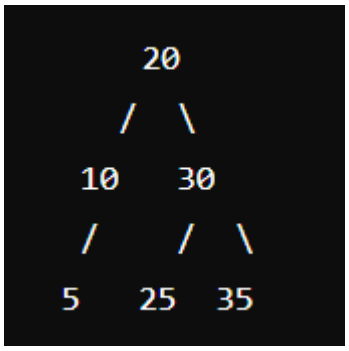
- Insertion: $O(\log n)$ in a balanced binary tree, $O(n)$ in the worst case for unbalanced trees.
- Deletion: $O(\log n)$ in a balanced binary tree, $O(n)$ in the worst case for unbalanced trees.
- Search: $O(\log n)$ in a balanced binary tree, $O(n)$ in the worst case for unbalanced trees.

4.6 Applications:

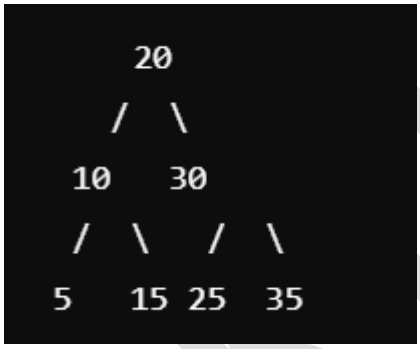
- **Hierarchical data representation** (e.g., file systems, organizational charts).
- **Binary search trees** for efficient search operations.
- **Heaps** for implementing priority queues.
- **Expression parsing** in compilers.

4.7 Example:

A binary search tree (BST):



Insert 15:
The new BST:



Summary

- **Stack**: LIFO structure used in scenarios like recursion and expression evaluation.
- **Queue**: FIFO structure used in scheduling and traversal algorithms.
- **Linked List**: Dynamic structure where nodes are linked; useful in dynamic memory allocation and implementing other structures.
- **Tree**: Hierarchical structure used in searches, hierarchical data representation, and maintaining sorted data.

5.Algorithm Design Techniques:

5.1. Brute Force and Exhaustive Search

Brute Force:

- **Definition**: Brute force algorithms solve problems by exploring all possible solutions to find the correct one. It involves checking all candidates and selecting the best solution.
- **Approach**: Systematic enumeration of all possible candidates.
- **Example**: Searching for an element in an unsorted array through linear search, where each element is compared sequentially.

Exhaustive Search:

- **Definition**: Similar to brute force but typically refers to solving combinatorial problems by generating all possible solutions and evaluating each.
- **Example**: Solving the Traveling Salesman Problem by evaluating every possible tour.

Advantages:

- Simple to implement.
- Provides correct solutions.

Disadvantages:

- Inefficient for large inputs (time complexity can be very high).

5.2. Decrease and Conquer

- **Definition**: In this technique, the problem size is reduced (usually by a constant factor) before solving the problem recursively or iteratively.

Types:

1. **Decrease by a constant**: Reduces the problem size by a constant value, typically by one.
    - **Example**: Insertion sort reduces the unsorted portion of the array by inserting one element at a time.
2. **Decrease by a constant factor**: Reduces the size by a constant factor.
    - **Example**: Binary search divides the problem size in half with each recursive call.
3. **Variable size decrease**: The problem size is reduced by a variable amount.
    - **Example**: Euclidean algorithm for finding the GCD.

Advantages:

- More efficient than brute force.

Disadvantages:

- Still may not be the most optimal solution for larger inputs.

5.3. Divide and Conquer

- **Definition**: This technique divides the problem into smaller subproblems, solves each subproblem independently, and combines the solutions.

Steps:

1. **Divide**: Break the problem into subproblems.
2. **Conquer**: Solve each subproblem recursively.
3. **Combine**: Combine the solutions to solve the original problem.

Examples:

- **Merge Sort**: Divides the array into two halves, sorts each half recursively, and merges the two sorted halves.
- **Quick Sort**: Partitions the array around a pivot and recursively sorts the partitions.

Advantages:

- Efficient for many problems (O(n log n) time complexity).

Disadvantages:

- Recursive overhead.

5.4. Transform and Conquer

- **Definition**: This technique transforms the problem into another problem, solves the transformed problem, and converts the solution back to the original problem.

Types:

1. **Instance simplification**: Simplify the problem instance.
    - **Example**: Balanced Search Trees (transform into a balanced tree for efficient searching).
2. **Representation change**: Change the problem representation.
    - **Example**: Converting a graph into an adjacency matrix for easier manipulation.

3. **Problem reduction**: Reduce one problem to another.
    - **Example**: Reducing a general sorting problem to an easier subproblem like sorting integers.

Advantages:

- Simplifies problem-solving by converting complex problems into simpler ones.

5.5. Dynamic Programming

- **Definition**: A technique that solves problems by breaking them down into simpler overlapping subproblems and solving each subproblem only once, storing its result for future reference (memoization or tabulation).

Characteristics:

1. **Overlapping subproblems**.
2. **Optimal substructure**: The solution to the main problem can be composed of solutions to subproblems.

Examples:

- **Fibonacci Sequence**: Each number is the sum of the two preceding ones.
- **Knapsack Problem**: Maximize the value of items that can fit into a knapsack with limited capacity.

Advantages:

- Avoids recomputation (more efficient than brute force).

Disadvantages:

- Requires significant memory for storing intermediate results.

5.6. Iterative Improvement

- **Definition**: Iteratively improve a solution by making small, local changes until an optimal or near-optimal solution is found.

Types:

1. **Local Search**: Continuously explore the neighborhood of a current solution to find a better one.
    - **Example**: Hill Climbing.
2. **Greedy Iteration**: In each step, choose the best immediate option.
    - **Example**: Simplex method in linear programming.

Advantages:

- Can often lead to fast, approximate solutions.

Disadvantages:

- May converge to local optima instead of global optima.

5.7. Greedy Technique

- **Definition**: The greedy technique builds a solution by making a sequence of choices, each of which is locally optimal at that point, with the hope that these choices lead to a globally optimal solution.

Steps:

1. **Select**: Choose the best option available at the moment.
2. **Feasibility**: Ensure that the choice is feasible.
3. **Solution**: Build the solution iteratively.

Examples:

- **Prim's Algorithm**: For finding the Minimum Spanning Tree.
- **Huffman Coding**: For data compression.

Advantages:

- Simple to implement.
- Often faster than dynamic programming or divide-and-conquer.

Disadvantages:

- May not always produce the optimal solution.

5.8. Backtracking

- **Definition**: A trial-and-error approach where the algorithm incrementally builds candidates for solutions and abandons candidates ("backtracks") as soon as it determines that the candidate cannot lead to a valid solution.

Steps:

1. **Construct**: Incrementally build a solution.
2. **Feasibility check**: Test whether the current solution can still lead to a valid solution.
3. **Backtrack**: If not feasible, abandon the current path and backtrack to explore other possibilities.

Examples:

- **N-Queens Problem**: Placing N queens on an N×N chessboard without them attacking each other.
- **Sudoku Solver**: Incrementally place numbers on the grid and backtrack if conflicts arise.

Advantages:

- Useful for constraint satisfaction problems.

Disadvantages:

- May explore unnecessary paths (inefficient without pruning).

5.9. Branch and Bound

- **Definition**: A technique used to solve optimization problems. It systematically explores branches of the solution space and uses bounds to eliminate regions that cannot contain the optimal solution.

Steps:

1. **Branching**: Divide the problem into subproblems (branches).
2. **Bounding**: Calculate bounds for the subproblems.
3. **Pruning**: Eliminate branches that cannot lead to the optimal solution.

Examples:

- **Knapsack Problem**: Use bounds to eliminate non-promising solutions.
- **TSP (Traveling Salesman Problem)**: Use bounds to prune suboptimal paths.

Advantages:

- More efficient than exhaustive search.

Disadvantages:

- Still computationally expensive for very large problem spaces.

Sorting and Searching

---

1. Comparison Sort with Analysis

In comparison-based sorting algorithms, elements are compared to each other to determine their relative order. The performance of these algorithms is often analyzed based on the number of comparisons made between elements. Below are the main comparison sorting algorithms, along with a non-comparison sort (Radix Sort), and their respective analyses.

Heap Sort

- **Concept**: Heap sort uses a binary heap data structure to sort an array.
- **Steps**:
    1. Build a max-heap from the input data.
    2. The largest element (root of the heap) is swapped with the last element.
    3. Reduce the heap size and heapify the root again.
    4. Repeat until all elements are sorted.
- **Time Complexity**:
    - **Building Heap**: O(n)
    - **Heapify Operation**: O(log n) (called n times)
    - **Total Time Complexity**: O(n log n)
- **Space Complexity**: O(1) (in-place sorting)
- **Advantages**:
    - Efficient O(n log n) time complexity.
    - In-place sorting (does not require extra memory for recursion).
- **Disadvantages**:
    - Not stable (relative order of equal elements may not be preserved).

Radix Sort

- **Concept**: Radix sort is a non-comparison sorting algorithm that sorts numbers digit by digit (or character by character for strings). It is useful when the range of possible input values is limited.
- **Steps**:
    1. Sort elements by their least significant digit (using a stable sorting algorithm like counting sort).
    2. Move to the next significant digit and repeat.
    3. Continue until all digits have been processed.

- **Time Complexity**: O(d * (n + k)), where:
    - **d** = Number of digits.
    - **n** = Number of elements.
    - **k** = Range of the digits (radix).
- **Space Complexity**: O(n + k)
- **Advantages**:
    - Efficient for sorting integers or strings with a fixed number of digits.
    - Linear time complexity when k is small compared to n.
- **Disadvantages**:
    - Limited to sorting integers or strings.
    - Requires extra memory for storing auxiliary arrays.

Quicksort

- **Concept**: Quicksort is a divide-and-conquer algorithm that selects a pivot element, partitions the array around the pivot, and recursively sorts the subarrays.
- **Steps**:
    1. Select a pivot element (typically, the first, last, or a random element).
    2. Partition the array such that all elements smaller than the pivot are on its left, and all elements larger than the pivot are on its right.
    3. Recursively apply the same process to the left and right subarrays.
- **Time Complexity**:
    - **Best Case**: O(n log n) (when the pivot divides the array into two equal halves).
    - **Average Case**: O(n log n)
    - **Worst Case**: $O(n^2)$ (when the pivot is the smallest or largest element, leading to highly unbalanced partitions).
- **Space Complexity**: O(log n) (due to recursive calls).
- **Advantages**:
    - Typically faster than other O(n log n) algorithms due to its low hidden constants.
    - In-place sorting (requires only a small amount of extra memory).
- **Disadvantages**:

- Worst-case time complexity is $O(n^2)$, though this can be mitigated by using randomized or median-of-three pivot selection.
- Not stable.

Merge Sort

- **Concept**: Merge sort is a stable divide-and-conquer algorithm that divides the array into halves, recursively sorts each half, and then merges the sorted halves.
- **Steps**:
  1. Divide the array into two halves.
  2. Recursively sort each half.
  3. Merge the two sorted halves back into one sorted array.
- **Time Complexity**: O(n log n) for all cases (best, worst, and average).
- **Space Complexity**: O(n) (requires extra memory for the merge process).
- **Advantages**:
  - Guarantees O(n log n) time complexity regardless of input order.
  - Stable (preserves the relative order of equal elements).
- **Disadvantages**:
  - Requires additional space for merging.
  - Not in-place (due to the extra memory used during merging).

---

2. Lower Bound for Comparison-Based Sorting

- **Theoretical Bound**: Any comparison-based sorting algorithm that works by comparing elements to each other has a lower bound of **O(n log n)** for the worst-case time complexity.
- **Reasoning**:
  - Sorting can be thought of as constructing a decision tree where each internal node represents a comparison.
  - For n elements, there are n! possible permutations (arrangements).
  - To differentiate between all possible arrangements, the decision tree must have at least n! leaves.
  - The height of a binary decision tree with n! leaves is $\log_2(n!)$, which is O(n log n) (using Stirling's approximation).

- **Implication**:
    - No comparison-based sorting algorithm can have a worst-case time complexity better than O(n log n).
    - Sorting algorithms like Merge Sort, Heap Sort, and Quick Sort (average case) are optimal with respect to this bound.

Proof Sketch:

- To sort n elements, there are n! possible ways to arrange them.
- Any comparison-based sort must identify the correct order by distinguishing between these n! possibilities.
- The decision tree used for sorting has n! leaves, and the minimum height of this tree is $\log_2(n!)$.
- The height of the decision tree represents the number of comparisons made.
- Therefore, the minimum number of comparisons needed is O(n log n).

Practical Implications:

- Algorithms like Radix Sort, which are not comparison-based, can sometimes outperform comparison-based sorts when input constraints are favorable.
- In general cases, comparison-based sorts cannot break the O(n log n) barrier.

Linear Time Sorting Algorithms

While comparison-based sorting algorithms have a theoretical lower bound of O(nlogn), certain sorting algorithms can achieve linear time complexity O(n) under specific conditions. These algorithms do not rely on comparisons but instead use the properties of the input data, such as the range of values or the distribution of numbers.

---

1. Counting Sort

- **Concept**: Counting Sort is a non-comparison, integer-based algorithm that sorts an array by counting the occurrences of each unique value and using this count to determine the correct positions of elements in the sorted array.
- **Steps**:
    1. **Find the Range**: Determine the range of the input data (minimum and maximum values).
    2. **Count the Occurrences**: Create a count array where each index corresponds to a possible input value, and store the count of each element from the input array.

3. **Cumulative Count**: Modify the count array such that each element at index i contains the cumulative sum of elements up to i.
4. **Place the Elements**: Iterate through the input array in reverse order, placing elements into their correct positions based on the cumulative count.

- **Time Complexity**:
  - **Best Case**: O(n + k) (where n is the number of elements and k is the range of the input).
- **Space Complexity**: O(n + k) (for the count array).
- **Advantages**:
  - **Linear time** sorting when the range of values k is small relative to n.
  - Stable sort (preserves the relative order of elements with equal values).
- **Disadvantages**:
  - Limited to integers or items that can be mapped to a small, fixed range of integers.
  - Inefficient for large ranges (i.e., when k is much larger than n).

---

2. Radix Sort

- **Concept**: Radix Sort processes the digits of numbers (or characters of strings) starting from the least significant digit (LSD) or the most significant digit (MSD) and sorts based on each digit using a stable sorting algorithm like Counting Sort.

Types:

1. **Least Significant Digit (LSD) Radix Sort**: Sorts numbers starting from the least significant digit to the most significant.
2. **Most Significant Digit (MSD) Radix Sort**: Sorts numbers starting from the most significant digit to the least significant.

- **Steps (LSD)**:
  1. **Initialize**: Determine the maximum number of digits in the input.
  2. **Sort by Digit**: Starting from the least significant digit, apply Counting Sort (or another stable sort) to sort the numbers based on that digit.
  3. **Repeat**: Move to the next significant digit and repeat the sorting process.
- **Time Complexity**: O(d * (n + k)), where:
  - d = Number of digits (or significant bits) in the input numbers.

  - n = Number of elements.
  - k = Range of digits or bits (radix).
- **Space Complexity**: O(n + k) (for the auxiliary space used in Counting Sort).
- **Advantages**:
  - Linear time sorting for fixed-size integer or string inputs.
  - Can be more efficient than comparison-based algorithms for large datasets with small digit ranges.
- **Disadvantages**:
  - Requires a stable sorting algorithm (like Counting Sort) at each step.
  - Only efficient for numbers, strings, or objects that can be decomposed into digits or fixed-length representations.

---

3. Bucket Sort

- **Concept**: Bucket Sort distributes the elements of an array into several "buckets" (subarrays), each containing a range of values. Each bucket is then sorted individually, often using another sorting algorithm, and the sorted buckets are concatenated to produce the final sorted array.
- **Steps**:
  1. **Initialize Buckets**: Create n empty buckets.
  2. **Distribute Elements**: Scatter the elements of the array into these buckets based on their values (e.g., by mapping values to specific buckets).
  3. **Sort Buckets**: Sort each bucket individually (using another sorting algorithm such as Insertion Sort or Quick Sort).
  4. **Concatenate Buckets**: Merge the sorted buckets to form the final sorted array.
- **Time Complexity**:
  - **Average Case**: O(n + k), where:
    - n = Number of elements.
    - k = Number of buckets.
  - **Worst Case**: $O(n^2)$ (if all elements land in the same bucket and that bucket is sorted using a quadratic algorithm).
- **Space Complexity**: O(n + k) (for the buckets).

- **Advantages**:
    - Linear time complexity when the input is uniformly distributed across buckets.
    - Efficient for sorting data that is distributed over a known range.
- **Disadvantages**:
    - Performance depends on the choice of bucket distribution (poor distribution leads to degraded performance).
    - Requires extra space for the buckets.
    - May require a secondary sorting algorithm for sorting within each bucket.

Comparison of Linear Time Sorting Algorithms

Algorithm	Time Complexity	Space Complexity	Stable	When to Use
Counting Sort	O(n + k)	O(n + k)	Yes	When range of input is small and fixed.
Radix Sort	O(d * (n + k))	O(n + k)	Yes	When sorting integers or fixed-length strings.
Bucket Sort	O(n + k) (average)	O(n + k)	Yes (with stable sub-sorting)	When input data is uniformly distributed.

Key Insights

- **Counting Sort** is efficient when sorting integers with a limited range. It requires additional space but guarantees linear time complexity when the range of values is not too large.
- **Radix Sort** can achieve linear time complexity for large datasets, provided the number of digits is small or fixed. It works particularly well for fixed-length keys like integers or strings.
- **Bucket Sort** is useful for data that is uniformly distributed across a known range. It is often combined with other algorithms to sort the individual buckets, and its performance is closely tied to how well the data is distributed across buckets.

Searching Techniques

---

1. Linear Search

- **Concept**: Linear Search is the simplest searching algorithm. It works by sequentially checking each element of the array until a match is found or all elements are checked.
- **Algorithm**:
    1. Start from the first element of the array.
    2. Compare each element with the target value.
    3. If a match is found, return the index of the element.
    4. If no match is found after checking all elements, return that the element is not present.
- **Time Complexity**:
    - **Best Case**: O(1) (if the target is the first element).
    - **Average Case**: O(n/2) ≈ O(n) (on average, half the elements need to be checked).
    - **Worst Case**: O(n) (if the target is the last element or not present).
- **Space Complexity**: O(1) (no extra memory required).
- **Advantages**:
    - Simple to implement and works well for small datasets.
    - Does not require the data to be sorted.
- **Disadvantages**:
    - Inefficient for large datasets due to its linear time complexity.
    - No improvement for partially sorted data.

---

2. Modified Linear Search

- **Concept**: Modified Linear Search includes optimizations over the basic linear search to reduce the number of comparisons. Some optimizations are:

a. Sentinel Search:

- **Concept**: The idea is to reduce the number of comparisons by placing the search element (sentinel) at the end of the array.
- **Algorithm**:

1. Append the target element as a sentinel at the end of the array.
2. Perform linear search without checking the bounds in every iteration (because the sentinel guarantees termination).
3. If the target is found before reaching the sentinel, return the index. Otherwise, return "not found."

- **Time Complexity**: Same as linear search but slightly optimized (O(n)) due to fewer checks.

b. Bidirectional Search:

- **Concept**: Instead of searching in one direction (from the start), bidirectional search performs two simultaneous searches from both ends of the array.
- **Algorithm**:
  1. Initialize two pointers, one at the start and one at the end.
  2. Compare the target with both ends in each step.
  3. If a match is found on either end, return the index.
  4. Continue until the two pointers meet.
- **Time Complexity**: O(n/2) ≈ O(n), but with better practical performance compared to unidirectional linear search.
- **Advantages**:
  - Reduces the number of comparisons slightly compared to simple linear search.
  - Efficient for small or unsorted datasets.
- **Disadvantages**:
  - Still O(n) time complexity, which is inefficient for large datasets.

---

3. Binary Search

- **Concept**: Binary Search is an efficient algorithm for finding an element in a sorted array by repeatedly dividing the search interval in half. If the target element is less than the middle element, the search continues in the left half; otherwise, it continues in the right half.

- **Algorithm**:
  1. Let the left pointer L be at index 0, and the right pointer R be at the last index.
  2. While $L \leq R$:
     - Calculate the middle index: $M = \frac{L+R}{2}$.
     - If the element at M matches the target, return M.
     - If the target is less than the element at M, update $R = M - 1$ (search the left half).
     - If the target is greater than the element at M, update $L = M + 1$ (search the right half).
  3. If $L > R$, return "element not found."

  Time Complexity:
  - **Best Case**: O(1) (if the target is the middle element).
  - **Average Case**: O(log n).
  - **Worst Case**: O(log n).
- **Space Complexity**:
  - **Iterative Version**: O(1) (no additional memory required).
  - **Recursive Version**: O(log n) (due to the recursive call stack).
- **Advantages**:
  - Significantly faster than linear search for large datasets, with O(log n) time complexity.
  - Well-suited for searching in large sorted arrays.
- **Disadvantages**:
  - Requires the array to be sorted beforehand.
  - If the data is unsorted, the overhead of sorting the array can negate the benefits of binary search.
  - Less effective on small datasets due to the overhead of calculating midpoints.

Binary Search: Recursive vs Iterative Analysis

Recursive Binary Search:

- Calls itself with a reduced search range until the target is found or the search space is exhausted.
- **Advantages**:

    - Code is often more elegant and easier to understand.

- **Disadvantages**:
    - Requires more memory due to recursive function calls (stack space).
    - May hit stack overflow for large arrays.

Iterative Binary Search:

- Uses a loop to reduce the search range without the overhead of recursive calls.
- **Advantages**:
    - Uses constant space (O(1)).
    - Avoids the risk of stack overflow.
- **Disadvantages**:
    - Slightly more complex to implement than recursive, but more efficient for large arrays.

Binary Search Variants

a. Exponential Search:

- **Concept**: Used when the size of the array is unknown (e.g., in an infinite or unbounded array). It finds the range where the target might be present using exponential jumps and then applies binary search within that range.
- **Time Complexity**: O(log n) (same as binary search after locating the range).

b. Interpolation Search:

- **Concept**: Similar to binary search but instead of dividing the search space into equal halves, it estimates the position of the target based on the distribution of the array's values (used for uniformly distributed sorted arrays).
- **Time Complexity**: O(log log n) in the best case (when elements are uniformly distributed) but can degrade to O(n) in the worst case.

Comparison of Searching Techniques

Algorithm	Time Complexity	Space Complexity	When to Use
Linear Search	O(n)	O(1)	Small datasets or unsorted arrays.

Algorithm	Time Complexity	Space Complexity	When to Use
Modified Linear Search	O(n)	O(1)	Small datasets with slight optimization.
Binary Search	O(log n)	O(1) (iterative)	Large, sorted arrays.
Exponential Search	O(log n)	O(log n)	When array size is unknown.
Interpolation Search	O(log log n) (best)	O(1)	Large sorted arrays with uniformly distributed values.

3.Data Structures for Disjoint Sets

Disjoint Set Data Structures, also known as Union-Find Data Structures, are used to efficiently manage a collection of non-overlapping sets. The main operations for disjoint sets are:

- **Find**: Determine which set a particular element belongs to.
- **Union**: Merge two sets into a single set.

Tree Representation of a Set

In the tree representation of a set, each set is represented by a tree where each node points to its parent, and the root of the tree is the representative of the set. The root element is used to identify the set.

To optimize the performance, two main techniques are used:

1. **Union by Rank**: Always attach the smaller tree under the root of the larger tree.
2. **Path Compression**: Flatten the structure of the tree whenever Find is called, by making all nodes directly point to the root.

C++ Code Implementation: Tree Representation of Disjoint Sets

```
#include <iostream>
```

```
using namespace std;

class DisjointSet {
    int *parent, *rank;
    int n;

public:
    // Constructor to initialize disjoint sets
    DisjointSet(int n) {
        this->n = n;
        parent = new int[n];
        rank = new int[n];

        // Initially, every element is its own parent
(representative)
        for (int i = 0; i < n; i++) {
            parent[i] = i;
            rank[i] = 0;
        }
    }

    // Find function with path compression
    int find(int u) {
        // If u is not the root, set its parent to the root of its
set
        if (u != parent[u])
            parent[u] = find(parent[u]);
        return parent[u];
    }

    // Union by rank
    void unionSets(int u, int v) {
        int rootU = find(u);
        int rootV = find(v);

        // If they are already in the same set, do nothing
        if (rootU == rootV)
            return;

        // Union by rank
        if (rank[rootU] < rank[rootV]) {
            parent[rootU] = rootV;  // Attach smaller tree under the
larger one
        }
        else if (rank[rootU] > rank[rootV]) {
            parent[rootV] = rootU;
        }
        else {
            parent[rootV] = rootU;
            rank[rootU]++;  // Increment rank if they were of the
same rank
```

```
        }
    }

    // Function to print the parent array (for debugging)
    void printParentArray() {
        for (int i = 0; i < n; i++) {
            cout << "Element: " << i << ", Parent: " << parent[i] <<
endl;
        }
    }
};

int main() {
    int n = 7;  // Number of elements in the set
    DisjointSet ds(n);

    ds.unionSets(0, 1);
    ds.unionSets(1, 2);
    ds.unionSets(3, 4);
    ds.unionSets(4, 5);
    ds.unionSets(6, 5);

    // Find representatives of some elements
    cout << "Find(2): " << ds.find(2) << endl;  // Output: 0
(representative of set containing 0, 1, 2)
    cout << "Find(5): " << ds.find(5) << endl;  // Output: 3
(representative of set containing 3, 4, 5, 6)

    // Print the parent array after union operations
    ds.printParentArray();

    return 0;
}
```

Explanation of the Code:

1. **Class DisjointSet**:
    - **parent[]**: Each element stores its parent in the disjoint set tree.
    - **rank[]**: This array keeps track of the rank (depth) of each tree to keep the tree flat.
    - **find(u)**: Recursively finds the root of the set containing u and performs path compression so that each node points directly to the root.
    - **unionSets(u, v)**: Combines the sets containing u and v using union by rank, ensuring that the smaller tree gets attached under the larger tree.
2. **Main Program**:

- Performs union operations on some elements and checks which set they belong to by calling find().
- Prints the parent array for debugging purposes.

Key Concepts:

- **Path Compression**: Every time find() is called, the tree is flattened by pointing each node directly to the root. This helps reduce the height of the tree, leading to very efficient subsequent operations.
- **Union by Rank**: This ensures that the smaller tree (by rank) is always attached under the root of the larger tree, minimizing the height of the tree.

The **time complexity** of both `find()` and `unionSets()` is nearly constant, $O(\alpha(n))$, where α is the inverse Ackermann function, which grows very slowly and is practically constant for all reasonable input sizes.

Union and Find Methods (Disjoint Set Operations)

The **Union-Find** data structure is used to efficiently manage and merge disjoint sets. The two primary operations are:

- **Union**: Merge two sets into one.
- **Find**: Determine which set an element belongs to, typically returning the representative (or root) of the set.

Two key optimizations are commonly applied to improve the efficiency of these operations:

1. **Union by Rank (or Size)**: Ensures that the smaller tree is attached under the larger tree to keep the tree shallow.
2. **Path Compression**: Flattens the tree structure during Find operations, so that subsequent queries take constant time.

---

1. Basic Union-Find Algorithm

Basic Union Algorithm (without optimizations):

- **Union(u, v)**: Merge the sets containing u and v.
  - Find the roots of u and v.
  - Make one root the parent of the other.

Basic Find Algorithm (without optimizations):

- **Find(u)**: Traverse up the tree from node u to find the root of the set containing u.

---

2. Union by Rank

Union by Rank is an optimization technique that keeps the tree shallow, reducing the time required for Find operations. Instead of arbitrarily making one set the parent of another, we always attach the shorter tree (or smaller set) under the root of the taller tree (or larger set).

Rank:

- Each set (tree) is associated with a rank, which is a rough measure of the depth of the tree.
- When two sets are merged, the root of the set with a smaller rank becomes the child of the root of the set with a larger rank.
- If both sets have the same rank, the rank of the resulting set is incremented by 1.

Union by Rank Algorithm:

```
void unionByRank(int u, int v) {
    int rootU = find(u);
    int rootV = find(v);

    if (rootU == rootV) return; // Already in the same set

    // Attach smaller rank tree under larger rank tree
    if (rank[rootU] < rank[rootV]) {
        parent[rootU] = rootV;
    } else if (rank[rootU] > rank[rootV]) {
        parent[rootV] = rootU;
    } else {
        // If both ranks are equal, attach one to the other and
increase rank
        parent[rootV] = rootU;
        rank[rootU]++;
    }
}
```

Advantages of Union by Rank:

- By attaching the smaller tree to the larger one, the height of the tree is minimized.
- This ensures that the tree structure remains shallow, leading to faster Find operations.

Time Complexity: O(1) for each Union operation, due to the constant-time merging of trees.

---

3. Path Compression

Path Compression is a heuristic that optimizes the Find operation. When we perform a Find on a node, the algorithm not only finds the root of the set but also flattens the tree by making all nodes on the path from the node to the root point directly to the root.

Path Compression Algorithm:

```
int find(int u) {
    // Path compression: set parent[u] directly to the root
    if (u != parent[u]) {
        parent[u] = find(parent[u]);  // Recursively find root and
compress the path
    }
    return parent[u];
}
```

Advantages of Path Compression:

- Every time Find is called, the structure of the tree is flattened, ensuring that all nodes on the path from u to the root point directly to the root.
- This greatly reduces the depth of the tree, making subsequent Find operations faster.

Time Complexity: O(1) on average for each Find operation. Path compression ensures that the depth of the trees remains logarithmic or even smaller.

---

4. Union-Find with Both Optimizations

By combining **Union by Rank** and **Path Compression**, we can create a highly efficient implementation of the Union-Find data structure. Both Find and Union operations are reduced to nearly constant time, making them very efficient for large datasets.

Complete Union-Find Algorithm in C++:

```
#include <iostream>
using namespace std;

class DisjointSet {
    int *parent, *rank;
    int n;

public:
    // Constructor to initialize disjoint sets
    DisjointSet(int n) {
        this->n = n;
        parent = new int[n];
        rank = new int[n];

        // Initially, each element is its own parent
        for (int i = 0; i < n; i++) {
            parent[i] = i;
            rank[i] = 0;
        }
```

```
    }

    // Find function with path compression
    int find(int u) {
        if (u != parent[u])
            parent[u] = find(parent[u]);  // Path compression
        return parent[u];
    }

    // Union by rank
    void unionByRank(int u, int v) {
        int rootU = find(u);
        int rootV = find(v);

        if (rootU == rootV) return;  // Same set, no union needed

        // Union by rank
        if (rank[rootU] < rank[rootV]) {
            parent[rootU] = rootV;
        } else if (rank[rootU] > rank[rootV]) {
            parent[rootV] = rootU;
        } else {
            parent[rootV] = rootU;
            rank[rootU]++;
        }
    }

    // Print the parent array for debugging
    void printParentArray() {
        for (int i = 0; i < n; i++) {
            cout << "Element: " << i << ", Parent: " << parent[i] <<
endl;
        }
    }
};

int main() {
    int n = 7;  // Number of elements
    DisjointSet ds(n);

    ds.unionByRank(0, 1);
    ds.unionByRank(1, 2);
    ds.unionByRank(3, 4);
    ds.unionByRank(4, 5);
    ds.unionByRank(6, 5);

    // Find representatives
    cout << "Find(2): " << ds.find(2) << endl;  // Output: 0
    cout << "Find(5): " << ds.find(5) << endl;  // Output: 3

    // Print the parent array
```

```
    ds.printParentArray();

    return 0;
}
```

---

5. Analysis of Union-Find

- **Time Complexity**:
    - Both Find and Union operations are nearly constant time, with an amortized time complexity of **O(α(n))**, where α(n)\alpha(n)α(n) is the inverse Ackermann function, which grows extremely slowly and is considered constant for all practical purposes.
    - Path Compression ensures that trees remain shallow, while Union by Rank ensures that tree heights do not grow unnecessarily.
- **Space Complexity**: O(n) for the parent and rank arrays, where n is the number of elements.

---

6. Applications of Union-Find

Union-Find is commonly used in scenarios involving connectivity between components. Some key applications include:

a. Kruskal's Minimum Spanning Tree (MST) Algorithm:

- Union-Find is used to detect cycles when adding edges to a spanning tree.
- The algorithm works by considering edges in increasing order of weight and using Union-Find to check if adding an edge creates a cycle.

b. Connected Components in Graphs:

- Union-Find helps to determine the number of connected components in an undirected graph.

c. Dynamic Connectivity Problem:

- Union-Find allows efficient handling of problems where the goal is to determine whether two elements are connected, and connections can be added dynamically.

d. Image Processing:

- In image processing, Union-Find can be used for tasks like labeling connected components in a binary image.

e. Network Connectivity:

- In network theory, it is used to determine if different nodes (computers or devices) are connected, even as connections change dynamically.

Conclusion

The **Union-Find** data structure, when optimized with **Union by Rank** and **Path Compression**, is a powerful tool for efficiently managing disjoint sets. It has widespread applications in graph algorithms, network theory, and other computational problems where connectivity is essential. The amortized time complexity of nearly constant time makes it ideal for large datasets and dynamic scenarios.

Graph Representation

Graphs are a fundamental data structure used to model relationships between objects. A graph G=(V,E) consists of:

- **V**: A set of vertices (nodes).
- **E**: A set of edges (links between pairs of vertices).

Depending on the problem or application, there are two main ways to represent graphs:

1. **Adjacency Matrix**
2. **Adjacency List**

---

1. Adjacency Matrix

An adjacency matrix is a 2D array (or matrix) used to represent a graph. Each element in the matrix represents whether there is an edge between two vertices.

Structure:

- For a graph with n vertices, the adjacency matrix is an n×nn \times nn×n matrix A, where:
    - A[i][j]=1 if there is an edge from vertex i to vertex j.
    - A[i][j]=0 if there is no edge.

For weighted graphs, the matrix can store the weight of the edge instead of a binary value.

Example:

Consider a graph with 4 vertices and the following edges: $(0 \to 1), (1 \to 2), (2 \to 0), (3 \to 2)$.

The adjacency matrix is:

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Code Implementation (Adjacency Matrix):

```
#include <iostream>
using namespace std;

#define V 4  // Number of vertices

// Function to print the adjacency matrix
void printAdjMatrix(int adjMatrix[V][V]) {
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            cout << adjMatrix[i][j] << " ";
        }
        cout << endl;
    }
}

int main() {
    // Initialize adjacency matrix
    int adjMatrix[V][V] = { {0, 1, 0, 0},
                            {0, 0, 1, 0},
                            {1, 0, 0, 0},
                            {0, 0, 1, 0} };

    // Print adjacency matrix
    cout << "Adjacency Matrix:" << endl;
    printAdjMatrix(adjMatrix);

    return 0;
}
```

Advantages:

- **Fast Access**: Checking if there is an edge between two vertices is O(1).
- **Good for Dense Graphs**: If the number of edges is close to n^2, adjacency matrices are efficient.

Disadvantages:

- **Space Complexity**: The matrix requires O(n^2) space, which is inefficient for sparse graphs (where the number of edges is much smaller than n^2).

- **Edge Iteration**: Iterating over all edges takes O(n^2) time, even if the graph is sparse.

---

2. Adjacency List

An adjacency list is a collection of lists or arrays used to represent a graph. Each vertex has a list that contains all the vertices it is connected to.

Structure:

- For each vertex i, store a list of all vertices adjacent to i. In a directed graph, this list contains vertices that have an edge from iii. In an undirected graph, this list contains all neighbors of i.
- Each vertex's adjacency list only stores the neighbors of that vertex, making this representation more space-efficient.

Example:

For the same graph as the previous example, the adjacency list would look like:

- Vertex 0: 1
- Vertex 1: 2
- Vertex 2: 0
- Vertex 3: 2

Code Implementation (Adjacency List):

```
#include <iostream>
#include <list>
using namespace std;

class Graph {
    int V;  // Number of vertices
    list<int> *adj;  // Array of adjacency lists

public:
    // Constructor
    Graph(int V) {
        this->V = V;
        adj = new list<int>[V];
    }

    // Add an edge to the graph
    void addEdge(int u, int v) {
        adj[u].push_back(v);  // Add v to u's adjacency list
    }

    // Print the adjacency list
    void printAdjList() {
```

```
        for (int i = 0; i < V; i++) {
            cout << "Vertex " << i << ":";
            for (auto it : adj[i]) {
                cout << " -> " << it;
            }
            cout << endl;
        }
    }
};

int main() {
    Graph g(4);  // Create a graph with 4 vertices

    g.addEdge(0, 1);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(3, 2);

    // Print adjacency list
    cout << "Adjacency List:" << endl;
    g.printAdjList();

    return 0;
}
```

Advantages:

- **Space Efficiency**: The adjacency list only uses O(V+E) space, where V is the number of vertices and E is the number of edges. This is more efficient for sparse graphs.
- **Efficient Edge Iteration**: Iterating over all edges takes O(V+E) time, which is efficient for sparse graphs.

Disadvantages:

- **Slower Access**: Checking if there is an edge between two vertices takes O(degree(u)), which can be slower than the O(1) time of adjacency matrices.

Comparison: Adjacency Matrix vs. Adjacency List

Feature	Adjacency Matrix	Adjacency List
Space Complexity	O(V^2)	O(V+E)
Edge Existence Check	O(1)	O(degree(u))
Edge Iteration	O(V^2)	O(V+E)

Feature	Adjacency Matrix	Adjacency List
Best For	Dense graphs	Sparse graphs
Storage	Stores all possible connections	Only stores actual connections

---

Applications:

- **Adjacency Matrix**:
  - Useful for dense graphs where the number of edges is close to V^2.
  - Suitable for algorithms where fast edge existence checking is necessary, like in **Floyd-Warshall** for all-pairs shortest path.
- **Adjacency List**:
  - More efficient for sparse graphs where the number of edges is much smaller than V^2.
  - Suitable for algorithms like **Dijkstra** and **Kruskal's MST**, where edge iteration is crucial.

Conclusion:

- **Adjacency Matrix** is preferred for dense graphs and when quick edge existence checks are needed.
- **Adjacency List** is preferred for sparse graphs due to its space efficiency and ability to quickly iterate over all edges.

Both representations have their merits and are chosen based on the specific problem and graph properties.

Graph Traversal: Breadth-First Search (BFS) and Depth-First Search (DFS)

Graph traversal is the process of visiting all the vertices in a graph systematically. Two common traversal techniques are **Breadth-First Search (BFS)** and **Depth-First Search (DFS)**. These algorithms help in exploring a graph, processing its vertices, and discovering edges.

---

1. Breadth-First Search (BFS)

Breadth-First Search (BFS) explores a graph level by level, starting from a given vertex. It visits all vertices at the current depth level before moving on to the vertices at the next depth level. BFS is particularly useful for finding the shortest path in an unweighted graph.

Steps in BFS:

1. Start from a source vertex.
2. Visit all its adjacent vertices (neighbors).
3. Move to the next set of vertices that are at distance two from the source, and so on.
4. Use a queue to keep track of the vertices that need to be explored next.

Algorithm:

1. **Initialization**: Use a queue to store the current vertex and a boolean array to mark visited vertices.
2. **Process**: Dequeue a vertex, visit its neighbors, mark them as visited, and enqueue them if they haven't been visited before.
3. Repeat the process until the queue is empty.

Code Implementation (BFS in C++):

```
#include <iostream>
#include <list>
#include <queue>
using namespace std;

class Graph {
    int V;  // Number of vertices
    list<int> *adj;  // Adjacency list

public:
    // Constructor
    Graph(int V) {
        this->V = V;
        adj = new list<int>[V];
    }

    // Add an edge to the graph
    void addEdge(int u, int v) {
        adj[u].push_back(v);  // Add v to u's adjacency list
    }

    // BFS function
    void BFS(int start) {
        // Create a visited array and initialize all vertices as not
visited
        bool *visited = new bool[V];
        for (int i = 0; i < V; i++) {
```

```
            visited[i] = false;
        }

        // Create a queue for BFS
        queue<int> q;

        // Mark the current node as visited and enqueue it
        visited[start] = true;
        q.push(start);

        while (!q.empty()) {
            // Dequeue a vertex from the queue and print it
            int v = q.front();
            cout << v << " ";
            q.pop();

            // Get all adjacent vertices of the dequeued vertex v
            // If an adjacent has not been visited, mark it visited
and enqueue it
            for (auto it = adj[v].begin(); it != adj[v].end(); ++it)
{
                if (!visited[*it]) {
                    visited[*it] = true;
                    q.push(*it);
                }
            }
        }
    }
};

int main() {
    Graph g(5);  // Create a graph with 5 vertices
    g.addEdge(0, 1);
    g.addEdge(0, 4);
    g.addEdge(1, 2);
    g.addEdge(1, 3);
    g.addEdge(2, 4);

    // Perform BFS starting from vertex 0
    cout << "Breadth-First Search starting from vertex 0: ";
    g.BFS(0);

    return 0;
}
```

Output:

Breadth-First Search starting from vertex 0: 0 1 4 2 3

Time Complexity:

- **O(V + E)**, where V is the number of vertices and E is the number of edges.

- Each vertex is visited once, and each edge is processed once.

Space Complexity:

- **O(V)** for the queue and visited array.

Applications of BFS:

- **Shortest Path**: BFS can be used to find the shortest path in an unweighted graph.
- **Level Order Traversal**: In trees, BFS corresponds to level-order traversal.
- **Connected Components**: It can be used to find connected components in an undirected graph.
- **Cycle Detection**: BFS can detect cycles in an undirected graph.

---

2. Depth-First Search (DFS)

Depth-First Search (DFS) explores a graph by going as deep as possible along each branch before backtracking. DFS is particularly useful for solving problems involving connectivity, paths, and cycles.

Steps in DFS:

1. Start from a source vertex.
2. Visit an unvisited neighbor and explore its deepest neighbors first (recursively).
3. Backtrack once there are no unvisited neighbors, and repeat until all vertices are explored.
4. Use a stack (either explicitly or implicitly with recursion) to manage the exploration.

Algorithm:

1. **Initialization**: Use a boolean array to mark visited vertices.
2. **Process**: Recursively visit an unvisited vertex, mark it as visited, and explore its neighbors.
3. Backtrack when there are no more unvisited neighbors.

Code Implementation (DFS in C++):

```
#include <iostream>
#include <list>
using namespace std;

class Graph {
    int V;  // Number of vertices
    list<int> *adj;  // Adjacency list
```

```
public:
    // Constructor
    Graph(int V) {
        this->V = V;
        adj = new list<int>[V];
    }

    // Add an edge to the graph
    void addEdge(int u, int v) {
        adj[u].push_back(v);  // Add v to u's adjacency list
    }

    // DFS function
    void DFSUtil(int v, bool visited[]) {
        // Mark the current node as visited and print it
        visited[v] = true;
        cout << v << " ";

        // Recur for all the vertices adjacent to this vertex
        for (auto it = adj[v].begin(); it != adj[v].end(); ++it) {
            if (!visited[*it]) {
                DFSUtil(*it, visited);
            }
        }
    }

    // DFS traversal of the vertices reachable from v
    void DFS(int start) {
        // Mark all the vertices as not visited
        bool *visited = new bool[V];
        for (int i = 0; i < V; i++) {
            visited[i] = false;
        }

        // Call the recursive helper function to print DFS traversal
        DFSUtil(start, visited);
    }
};

int main() {
    Graph g(5);  // Create a graph with 5 vertices
    g.addEdge(0, 1);
    g.addEdge(0, 4);
    g.addEdge(1, 2);
    g.addEdge(1, 3);
    g.addEdge(2, 4);

    // Perform DFS starting from vertex 0
    cout << "Depth-First Search starting from vertex 0: ";
    g.DFS(0);
```

```
    return 0;
}
```

Output:

Depth-First Search starting from vertex 0: 0 1 2 4 3

Time Complexity:

- **O(V + E)**, where V is the number of vertices and E is the number of edges.

Space Complexity:

- **O(V)** for the recursion stack (in case of recursive DFS) and the visited array.

Applications of DFS:

- **Topological Sorting**: DFS can be used to perform topological sorting in Directed Acyclic Graphs (DAGs).
- **Path Finding**: It can be used to find paths in a maze or a game.
- **Cycle Detection**: DFS is useful for detecting cycles in both directed and undirected graphs.
- **Connected Components**: It can be used to find all the connected components in a graph.
- **Strongly Connected Components (SCCs)**: In directed graphs, DFS is used in algorithms like **Kosaraju's** and **Tarjan's** algorithms for finding SCCs.

Comparison: BFS vs. DFS

Criterion	BFS	DFS
Data Structure	Queue	Stack (explicit or implicit via recursion)
Traversal Pattern	Level by level (breadth-wise)	Depth-wise exploration
Shortest Path	Finds the shortest path in unweighted graphs	Does not necessarily find the shortest path
Space Complexity	O(V)	O(V)

Criterion	BFS	DFS
Time Complexity	O(V + E)	O(V + E)
Applications	Shortest path, connected components	Path finding, topological sorting, SCC detection
Best for	Shallow traversal	Deep traversal

Conclusion:

Both BFS and DFS are fundamental graph traversal algorithms. BFS is preferred when you need to explore the graph level by level, such as when finding the shortest path in unweighted graphs. DFS, on the other hand, is useful for deep exploration, such as in pathfinding, cycle detection, and topological sorting. Choosing between BFS and DFS depends on the problem at hand and the structure of the graph.

Topological Sorting

Topological Sorting is a linear ordering of vertices in a **Directed Acyclic Graph (DAG)** such that for every directed edge u→v, vertex u appears before vertex v in the ordering. This type of sorting is useful for problems that involve dependency resolution, where certain tasks must precede others.

For example, consider a set of tasks with dependencies between them, such as prerequisites in a course schedule. Topological sorting provides a valid order to perform all tasks while respecting their dependencies.

Key Points:

- Topological sorting applies only to **DAGs**. If the graph contains cycles, it's impossible to produce a valid topological order.
- The graph must be directed and acyclic, as cyclic dependencies would prevent a valid ordering.

Applications of Topological Sorting:

- **Task Scheduling**: Where some tasks must be completed before others.
- **Course Scheduling**: To determine the order in which courses should be taken based on prerequisites.

- **Dependency Management**: Such as resolving software package dependencies.

Methods to Perform Topological Sorting:

There are two primary algorithms to perform topological sorting:

1. **Kahn's Algorithm** (BFS-based)
2. **DFS-based Approach**

---

1. Kahn's Algorithm (BFS-Based Approach)

This algorithm uses the concept of **in-degree** of vertices (the number of incoming edges to a vertex). It repeatedly removes vertices with zero in-degrees (i.e., vertices that have no dependencies) and reduces the in-degree of their neighbors.

Steps:

1. Compute the in-degree of each vertex.
2. Initialize a queue with all vertices that have an in-degree of 0 (no dependencies).
3. Dequeue a vertex, add it to the topological order, and reduce the in-degree of all its adjacent vertices by 1.
4. If an adjacent vertex's in-degree becomes 0, enqueue it.
5. Repeat until the queue is empty.

Code Implementation (Kahn's Algorithm in C++):

```
#include <iostream>
#include <list>
#include <queue>
using namespace std;

class Graph {
    int V;  // Number of vertices
    list<int> *adj;  // Adjacency list

public:
    // Constructor
    Graph(int V) {
        this->V = V;
        adj = new list<int>[V];
    }

    // Add an edge to the graph
    void addEdge(int u, int v) {
        adj[u].push_back(v);  // Add v to u's adjacency list (u ->
v)
```

```
    }

    // Kahn's Algorithm for Topological Sorting (BFS-based)
    void topologicalSort() {
        vector<int> inDegree(V, 0);  // Initialize in-degree of all
vertices to 0

        // Calculate in-degree (number of incoming edges) for each
vertex
        for (int i = 0; i < V; i++) {
            for (int neighbor : adj[i]) {
                inDegree[neighbor]++;
            }
        }

        // Create a queue and enqueue all vertices with in-degree 0
        queue<int> q;
        for (int i = 0; i < V; i++) {
            if (inDegree[i] == 0) {
                q.push(i);
            }
        }

        // Initialize a count of visited vertices and a vector to
store the result (topological order)
        int count = 0;
        vector<int> topOrder;

        // One by one, dequeue vertices with in-degree 0 and reduce
the in-degree of their neighbors
        while (!q.empty()) {
            int u = q.front();
            q.pop();
            topOrder.push_back(u);

            // Reduce in-degree for all adjacent vertices
            for (int neighbor : adj[u]) {
                inDegree[neighbor]--;
                if (inDegree[neighbor] == 0) {
                    q.push(neighbor);
                }
            }
            count++;
        }

        // Check if there was a cycle
        if (count != V) {
            cout << "There exists a cycle in the graph, topological
sort not possible!" << endl;
            return;
        }
```

```
        // Print topological order
        cout << "Topological Sorting (BFS-based): ";
        for (int i : topOrder) {
            cout << i << " ";
        }
        cout << endl;
    }
};

int main() {
    Graph g(6);  // Create a graph with 6 vertices
    g.addEdge(5, 2);
    g.addEdge(5, 0);
    g.addEdge(4, 0);
    g.addEdge(4, 1);
    g.addEdge(2, 3);
    g.addEdge(3, 1);

    // Perform Topological Sorting
    g.topologicalSort();

    return 0;
}
```

Output:

Topological Sorting (BFS-based): 5 4 2 3 1 0

Time Complexity:

- **O(V + E)**, where V is the number of vertices and E is the number of edges. Each vertex and edge is processed once.

Space Complexity:

- **O(V + E)**, due to the adjacency list and in-degree array.

---

2. DFS-Based Topological Sorting

This approach is based on **Depth-First Search (DFS)**. The idea is to explore each vertex and, once a vertex has no more outgoing edges to explore, it is added to the topological order. The vertices are stored in reverse order of completion.

Steps:

1. Perform a DFS traversal on the graph.
2. During the DFS, as soon as a vertex finishes (i.e., all its neighbors are explored), push it onto a stack.

3. Once all vertices are processed, the stack contains the topological order (in reverse order).

Code Implementation (DFS-Based Topological Sort in C++):

```
#include <iostream>
#include <list>
#include <stack>
using namespace std;

class Graph {
    int V;  // Number of vertices
    list<int> *adj;  // Adjacency list

public:
    // Constructor
    Graph(int V) {
        this->V = V;
        adj = new list<int>[V];
    }

    // Add an edge to the graph
    void addEdge(int u, int v) {
        adj[u].push_back(v);  // Add v to u's adjacency list (u ->
v)
    }

    // DFS utility function to visit vertices
    void DFSUtil(int v, bool visited[], stack<int> &Stack) {
        // Mark the current vertex as visited
        visited[v] = true;

        // Recur for all the vertices adjacent to this vertex
        for (int neighbor : adj[v]) {
            if (!visited[neighbor]) {
                DFSUtil(neighbor, visited, Stack);
            }
        }

        // Push current vertex to the stack once all its neighbors
are visited
        Stack.push(v);
    }

    // Topological Sorting using DFS
    void topologicalSort() {
        stack<int> Stack;  // Stack to store topological order
        bool *visited = new bool[V];  // Mark all vertices as not
visited

        // Initialize visited array
        for (int i = 0; i < V; i++) {
```

```
            visited[i] = false;
        }

        // Perform DFS for all unvisited vertices
        for (int i = 0; i < V; i++) {
            if (!visited[i]) {
                DFSUtil(i, visited, Stack);
            }
        }

        // Print the topological order
        cout << "Topological Sorting (DFS-based): ";
        while (!Stack.empty()) {
            cout << Stack.top() << " ";
            Stack.pop();
        }
        cout << endl;
    }
};

int main() {
    Graph g(6);  // Create a graph with 6 vertices
    g.addEdge(5, 2);
    g.addEdge(5, 0);
    g.addEdge(4, 0);
    g.addEdge(4, 1);
    g.addEdge(2, 3);
    g.addEdge(3, 1);

    // Perform Topological Sorting
    g.topologicalSort();

    return 0;
}
```

Output:

Topological Sorting (DFS-based): 5 4 2 3 1 0

Time Complexity:

- **O(V + E)**, where V is the number of vertices and E is the number of edges.

Space Complexity:

- **O(V + E)** due to the adjacency list and recursion stack.

Comparison: Kahn's Algorithm vs. DFS-Based Approach

Feature	Kahn's Algorithm (BFS)	DFS-Based Approach
Data Structure Used	Queue	Stack (explicit or implicit)
Cycle Detection	Easy to detect if cycle exists	Cycle detection is possible, but not direct
Order of Processing	Processes nodes with zero in-degrees first	Processes nodes based on recursive depth
When to Use	Prefer when dealing with in-degrees	Prefer when recursive solutions are easier to implement

---

Conclusion:

Topological sorting is a crucial concept in many practical applications involving dependencies. Kahn's algorithm (BFS-based) is intuitive and uses in-degree counting, while the DFS-based approach uses recursion and backtracking. Both methods have the same time complexity and can be chosen based on the nature of the problem and preferred traversal strategy.

Strongly Connected Components (SCCs)

In graph theory, a **Strongly Connected Component (SCC)** of a **Directed Graph** is a maximal subgraph where every vertex is reachable from every other vertex in that subgraph. In simpler terms, if you can travel between any two vertices within a subgraph, both ways, the subgraph is strongly connected.

SCCs are crucial in solving many problems related to **directed graphs**, including circuit design, web crawling, and solving game reachability problems.

Definitions:

- A directed graph is **strongly connected** if there is a **path** from every vertex to every other vertex.
- An SCC is a **maximal strongly connected subgraph**, meaning it is not possible to add any more vertices from the graph to this subgraph while maintaining the strong connectivity.

Applications of Strongly Connected Components:

- **Network Analysis**: Finding clusters of strongly connected nodes, such as in social networks or citation graphs.
- **Dependency Resolution**: Detecting cyclic dependencies in software packages.
- **Game Theory**: Analyzing the reachability in games with directed moves.

Algorithms to Find SCCs:

There are two main algorithms used to find SCCs:

1. **Kosaraju's Algorithm**
2. **Tarjan's Algorithm**

---

1. Kosaraju's Algorithm

Kosaraju's Algorithm is an efficient method to find all SCCs in a directed graph. It uses **Depth-First Search (DFS)** and has a time complexity of **O(V + E)**, where V is the number of vertices and E is the number of edges.

Steps of Kosaraju's Algorithm:

1. **Perform DFS on the original graph**: Push vertices onto a stack in the order they finish.
2. **Transpose the graph**: Reverse the direction of all edges in the graph.
3. **Perform DFS on the transposed graph**: Process the vertices in the order they were finished in the stack from the first DFS. Each DFS tree forms an SCC.

Algorithm Explanation:

1. **First DFS (Original Graph)**: This step ensures that vertices are processed in the correct finishing order (i.e., vertices that finish later are processed first in the second DFS).
2. **Graph Transpose**: Reversing all edges allows the second DFS to discover SCCs in the reversed order of processing.
3. **Second DFS (On Transposed Graph)**: This DFS discovers SCCs by following the reversed edges.

Code Implementation (Kosaraju's Algorithm in C++):

```
#include <iostream>
#include <list>
#include <stack>
using namespace std;
```

```
class Graph {
    int V;  // Number of vertices
    list<int> *adj;  // Adjacency list for original graph
    list<int> *transposeAdj;  // Adjacency list for transposed graph

    // Helper function to perform DFS and fill the stack with finishing
order
    void DFSFillOrder(int v, bool visited[], stack<int> &Stack) {
        visited[v] = true;  // Mark the current node as visited

        // Recur for all vertices adjacent to this vertex
        for (auto neighbor : adj[v]) {
            if (!visited[neighbor]) {
                DFSFillOrder(neighbor, visited, Stack);
            }
        }

        // All vertices reachable from v are processed, push v to the stack
        Stack.push(v);
    }

    // Helper function to perform DFS on the transposed graph
    void DFSOnTranspose(int v, bool visited[]) {
        visited[v] = true;  // Mark the current node as visited
        cout << v << " ";  // Print the current vertex as part of the SCC

        // Recur for all vertices adjacent to this vertex in the transposed
graph
        for (auto neighbor : transposeAdj[v]) {
            if (!visited[neighbor]) {
                DFSOnTranspose(neighbor, visited);
            }
        }
    }

public:
    // Constructor
    Graph(int V) {
        this->V = V;
        adj = new list<int>[V];
        transposeAdj = new list<int>[V];
    }

    // Add an edge to the graph
    void addEdge(int u, int v) {
        adj[u].push_back(v);  // Add v to u's adjacency list
    }

    // Create the transpose of the graph
    void transpose() {
        for (int u = 0; u < V; u++) {
            for (auto v : adj[u]) {
                transposeAdj[v].push_back(u);  // Reverse the direction of
the edge u -> v to v -> u
            }
        }
```

```
    }

    // Kosaraju's algorithm to find all SCCs
    void findSCCs() {
        stack<int> Stack;  // Stack to store the order of finishing times

        // Step 1: Perform DFS on the original graph and store finishing
times of vertices
        bool *visited = new bool[V];
        for (int i = 0; i < V; i++) {
            visited[i] = false;
        }

        for (int i = 0; i < V; i++) {
            if (!visited[i]) {
                DFSFillOrder(i, visited, Stack);
            }
        }

        // Step 2: Transpose the graph
        transpose();

        // Step 3: Perform DFS on the transposed graph
        for (int i = 0; i < V; i++) {
            visited[i] = false;  // Reset the visited array for the second
DFS
        }

        cout << "Strongly Connected Components are: \n";
        while (!Stack.empty()) {
            int v = Stack.top();
            Stack.pop();

            // Perform DFS on the transposed graph starting from vertex v
            if (!visited[v]) {
                DFSOnTranspose(v, visited);
                cout << endl;
            }
        }
    }
};

int main() {
    Graph g(5);  // Create a graph with 5 vertices
    g.addEdge(1, 0);
    g.addEdge(0, 2);
    g.addEdge(2, 1);
    g.addEdge(0, 3);
    g.addEdge(3, 4);

    // Find and print all SCCs
    g.findSCCs();

    return 0;
}
```

Output:

Strongly Connected Components are:

0 2 1

3

4

Time Complexity:

- **O(V + E)**, where V is the number of vertices and E is the number of edges.
- Each vertex and edge is processed twice: once in the original graph and once in the transposed graph.

---

2. Tarjan's Algorithm

Tarjan's Algorithm is another efficient algorithm to find all SCCs in a directed graph. Unlike Kosaraju's algorithm, it doesn't require transposing the graph. It uses a single DFS traversal and a **low-link** value to track the smallest vertex reachable from each vertex.

Steps of Tarjan's Algorithm:

1. Perform a DFS and assign each vertex an index based on the order it is visited.
2. Track the smallest index reachable from each vertex using a **low-link** value.
3. When a vertex's low-link value is equal to its index, it is the **root** of an SCC. All vertices on the stack up to this root form the SCC.
4. Recursively pop the vertices from the stack to form the SCC.

Code Implementation (Tarjan's Algorithm in C++):

```
#include <iostream>
#include <list>
#include <stack>
#include <vector>
using namespace std;

class Graph {
    int V;  // Number of vertices
    list<int> *adj;  // Adjacency list

    // Helper function for Tarjan's algorithm
    void SCCUtil(int u, vector<int> &disc, vector<int> &low,
stack<int> &st, vector<bool> &inStack) {
        static int time = 0;
        disc[u] = low[u] = ++time;  // Initialize discovery and low-
link values
        st.push(u);
        inStack[u] = true;
```

```
        // Recur for all vertices adjacent to this vertex
        for (int v : adj[u]) {
            if (disc[v] == -1) {  // If v is not visited, recur for
v
                SCCUtil(v, disc, low, st, inStack);
                low[u] = min(low[u], low[v]);
            } else if (inStack[v]) {  // If v is in the stack,
update low-link value
                low[u] = min(low[u], disc[v]);
            }
        }

        // If u is a root of an SCC, pop the stack and print the SCC
        if (low[u] == disc[u]) {
            cout << "SCC: ";
            while (st.top() != u) {
                int v = st.top();
                cout << v << " ";
                inStack[v] = false;
                st.pop();
            }
            cout << u << endl;
            inStack[u] = false;
            st.pop();
        }
    }

public:
    // Constructor
    Graph(int V) {
        this->V = V;
        adj = new list<int>[V];
    }

    // Add an edge to the graph
    void addEdge(int u, int v) {
        adj[u].push_back(v);  // Add v to u's adjacency list
    }

    // Tarjan's algorithm to find all SCCs
    void findSCCs() {
        vector<int> disc(V, -1);  // Stores discovery times of
visited vertices
        vector<int> low(V, -1);  // Stores the lowest vertex
reachable from a vertex
        stack<int> st;  // Stack to store visited vertices
        vector<bool> inStack(V, false);  // Keeps track of vertices
in the stack

        for (int i = 0; i < V; i++) {
```

```
            if (disc[i] == -1) {
                SCCUtil(i, disc, low, st, inStack);
            }
        }
    }
};

int main() {
    Graph g(5);  // Create a graph with 5 vertices
    g.addEdge(1, 0);
    g.addEdge(0, 2);
    g.addEdge(2, 1);
    g.addEdge(0, 3);
    g.addEdge(3, 4);

    // Find and print all SCCs
    g.findSCCs();

    return 0;
}
```

Output:

SCC: 1 2 0

SCC: 3

SCC: 4

Time Complexity:

- **O(V + E)**, where V is the number of vertices and E is the number of edges.
- Each vertex is processed exactly once during DFS.

Conclusion:

- **Kosaraju's Algorithm** is conceptually simple but requires two passes of DFS (on the original graph and the transposed graph).
- **Tarjan's Algorithm** performs only one pass of DFS and uses the concept of low-link values to find SCCs directly. Both algorithms have the same time complexity but differ in approach.

Minimum Spanning Trees (MST)

A **Minimum Spanning Tree (MST)** of a connected, undirected graph is a subgraph that includes all the vertices of the original graph and is a tree (i.e., it has no cycles) with the minimum possible total edge weight.

Key Concepts:

- **Spanning Tree**: A subgraph that includes all vertices and is a tree.
- **Minimum Spanning Tree (MST)**: A spanning tree with the smallest possible sum of edge weights.

There are several algorithms to find the MST, with **Kruskal's** and **Prim's** algorithms being the most common.

---

1. Kruskal's Algorithm

Kruskal's Algorithm is a **greedy algorithm** that builds the MST by sorting all edges and adding the smallest available edge to the tree, provided it does not form a cycle.

Steps of Kruskal's Algorithm:

1. Sort all the edges of the graph by their weights in non-decreasing order.
2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If it doesn't, add it to the MST.
3. Repeat step 2 until there are V−1 edges in the MST, where V is the number of vertices in the graph.

Union-Find Data Structure:

To check whether adding an edge forms a cycle, Kruskal's algorithm uses the **Union-Find** data structure to keep track of connected components.

- **Find**: Determines which subset a particular element is in.
- **Union**: Joins two subsets into a single subset.

The **path compression** and **union by rank** heuristics improve the efficiency of these operations.

Code Implementation (Kruskal's Algorithm in C++):

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

class Edge {
public:
    int src, dest, weight;
};

// Comparator to sort edges by their weights
bool compare(Edge a, Edge b) {
    return a.weight < b.weight;
}
```

```
// Union-Find helper functions
int find(int parent[], int i) {
    if (parent[i] == i) {
        return i;
    }
    return parent[i] = find(parent, parent[i]);  // Path compression
}

void unionSets(int parent[], int rank[], int x, int y) {
    int rootX = find(parent, x);
    int rootY = find(parent, y);

    if (rootX != rootY) {
        if (rank[rootX] > rank[rootY]) {
            parent[rootY] = rootX;
        } else if (rank[rootX] < rank[rootY]) {
            parent[rootX] = rootY;
        } else {
            parent[rootY] = rootX;
            rank[rootX]++;
        }
    }
}

void kruskalMST(vector<Edge> &edges, int V) {
    // Sort edges by weight
    sort(edges.begin(), edges.end(), compare);

    // Initialize parent and rank arrays for Union-Find
    int parent[V], rank[V];
    for (int i = 0; i < V; i++) {
        parent[i] = i;
        rank[i] = 0;
    }

    vector<Edge> mst;  // Store the MST

    for (Edge edge : edges) {
        int u = edge.src;
        int v = edge.dest;
        int weight = edge.weight;

        // Find root of src and dest vertices
        int rootU = find(parent, u);
        int rootV = find(parent, v);

        // If they belong to different sets, add the edge to the MST
        if (rootU != rootV) {
            mst.push_back(edge);
            unionSets(parent, rank, rootU, rootV);
        }
```

```
        // Stop if we have enough edges for the MST
        if (mst.size() == V - 1) {
            break;
        }
    }

    // Print the MST
    cout << "Edges in the Minimum Spanning Tree:\n";
    for (Edge e : mst) {
        cout << e.src << " -- " << e.dest << " == " << e.weight <<
endl;
    }
}

int main() {
    int V = 4;  // Number of vertices
    vector<Edge> edges = {
        {0, 1, 10}, {0, 2, 6}, {0, 3, 5}, {1, 3, 15}, {2, 3, 4}
    };

    kruskalMST(edges, V);

    return 0;
}
```

Output:

Edges in the Minimum Spanning Tree:

2 -- 3 == 4

0 -- 3 == 5

0 -- 1 == 10

Time Complexity:

- Sorting the edges takes **O(E log E)**, where E is the number of edges.
- Union-Find operations take nearly constant time with path compression and union by rank.
- Overall time complexity: **O(E log E)**.

---

2. Prim's Algorithm

Prim's Algorithm is another **greedy algorithm** for finding the MST. It starts with a single vertex and grows the MST one edge at a time by adding the smallest edge that connects a vertex in the MST to a vertex outside the MST.

Steps of Prim's Algorithm:

1. Initialize a **min-heap** or priority queue to store the edges. Initially, start with any vertex (usually vertex 0).
2. Add the smallest edge from the current tree to a vertex outside the tree.
3. Repeat the process until all vertices are included in the MST.

Code Implementation (Prim's Algorithm in C++):

```
#include <iostream>
#include <vector>
#include <climits>
using namespace std;

// Function to find the vertex with the minimum key value that is
not included in MST
int minKey(int key[], bool mstSet[], int V) {
    int min = INT_MAX, minIndex;

    for (int v = 0; v < V; v++) {
        if (mstSet[v] == false && key[v] < min) {
            min = key[v];
            minIndex = v;
        }
    }

    return minIndex;
}

// Function to print the constructed MST
void printMST(int parent[], int graph[5][5], int V) {
    cout << "Edge \tWeight\n";
    for (int i = 1; i < V; i++) {
        cout << parent[i] << " - " << i << " \t" <<
graph[i][parent[i]] << " \n";
    }
}

// Function to construct and print the MST using Prim's algorithm
void primMST(int graph[5][5], int V) {
    int parent[V];  // Array to store constructed MST
    int key[V];     // Key values used to pick the minimum weight
edge
    bool mstSet[V];  // To represent the set of vertices included in
MST

    // Initialize all keys as INFINITE
    for (int i = 0; i < V; i++) {
        key[i] = INT_MAX;
        mstSet[i] = false;
    }
```

```
    // Include the first vertex in MST by setting its key value to 0
    key[0] = 0;
    parent[0] = -1;  // First node is always the root of MST

    for (int count = 0; count < V - 1; count++) {
        // Pick the minimum key vertex from the set of vertices not
yet included in MST
        int u = minKey(key, mstSet, V);

        // Add the picked vertex to the MST set
        mstSet[u] = true;

        // Update key values and parent index of the adjacent
vertices
        for (int v = 0; v < V; v++) {
            // graph[u][v] is non-zero for adjacent vertices
            // mstSet[v] is false for vertices not yet included in
MST
            // Update the key only if graph[u][v] is smaller than
key[v]
            if (graph[u][v] && mstSet[v] == false && graph[u][v] <
key[v]) {
                parent[v] = u;
                key[v] = graph[u][v];
            }
        }
    }

    // Print the constructed MST
    printMST(parent, graph, V);
}

int main() {
    // Define the graph as an adjacency matrix
    int graph[5][5] = {
        {0, 2, 0, 6, 0},
        {2, 0, 3, 8, 5},
        {0, 3, 0, 0, 7},
        {6, 8, 0, 0, 9},
        {0, 5, 7, 9, 0}
    };

    primMST(graph, 5);

    return 0;
}
```

Output:

Edge Weight

0 - 1 2

1 - 2 3

0 - 3 6

1 - 4 5

Time Complexity:

- Using a priority queue or min-heap, the time complexity is **O(V^2)** for an adjacency matrix. If we use an adjacency list with a binary heap, the complexity can be reduced to **O(E log V)**.

Comparison: Kruskal vs Prim:

- **Kruskal's Algorithm**:
  - Better suited for **sparse graphs**.
  - Uses a **Union-Find** structure.
  - Time complexity: **O(E log E)**.
- **Prim's Algorithm**:
  - Better for **dense graphs**.
  - Builds the MST starting from any node and grows.
  - Time complexity: **O(V^2)** (or **O(E log V)** with adjacency list and min-heap).

Both algorithms produce the same result (the MST) but use different approaches.

Shortest Path Algorithms

In graph theory, the **shortest path** is the path between two vertices in a graph such that the sum of the weights of its edges is minimized. Shortest path algorithms are important in network routing, navigation, and optimization problems.

1. Single-Source Shortest Path

The **Single-Source Shortest Path (SSSP)** problem finds the shortest paths from a single source vertex to all other vertices in the graph. There are two primary algorithms to solve this problem: **Dijkstra's Algorithm** and **Bellman-Ford Algorithm**.

---

2. Dijkstra's Algorithm

Dijkstra's Algorithm is a **greedy algorithm** used to find the shortest path from a source vertex to all other vertices in a weighted graph. It works well for graphs with **non-negative edge weights**.

Steps of Dijkstra's Algorithm:

1. Initialize distances from the source to all vertices as infinity and distance to the source itself as 0.
2. Use a **priority queue** (min-heap) to pick the vertex with the smallest distance that hasn't been processed yet.
3. For the selected vertex, update the distances of its adjacent vertices.
4. Repeat until all vertices are processed.

Code Implementation (Dijkstra's Algorithm in C++):

```
#include <iostream>
#include <vector>
#include <queue>
#include <climits>

using namespace std;

// A structure to represent an edge in the graph
class Edge {
public:
    int dest, weight;
};

// A structure to represent the graph as an adjacency list
class Graph {
public:
    int V;  // Number of vertices
    vector<vector<Edge>> adjList;  // Adjacency list

    Graph(int V) {
        this->V = V;
        adjList.resize(V);
    }

    // Function to add an edge to the graph
    void addEdge(int u, int v, int w) {
        adjList[u].push_back({v, w});
        adjList[v].push_back({u, w});  // For undirected graph
    }

    // Dijkstra's Algorithm to find the shortest path from a source
vertex
    void dijkstra(int src) {
        // Initialize distance array with infinity and priority
queue
        vector<int> dist(V, INT_MAX);
        priority_queue<pair<int, int>, vector<pair<int, int>>,
greater<pair<int, int>>> pq;
```

```
        // Start with the source vertex
        dist[src] = 0;
        pq.push({0, src});

        while (!pq.empty()) {
            int u = pq.top().second;
            pq.pop();

            // Traverse adjacent vertices
            for (Edge e : adjList[u]) {
                int v = e.dest;
                int weight = e.weight;

                // Relax the edge if a shorter path is found
                if (dist[u] + weight < dist[v]) {
                    dist[v] = dist[u] + weight;
                    pq.push({dist[v], v});
                }
            }
        }

        // Print the shortest distances from the source
        cout << "Vertex   Distance from Source\n";
        for (int i = 0; i < V; i++) {
            cout << i << " \t\t " << dist[i] << endl;
        }
    }
};

int main() {
    int V = 5;  // Number of vertices
    Graph g(V);

    g.addEdge(0, 1, 10);
    g.addEdge(0, 4, 5);
    g.addEdge(1, 2, 1);
    g.addEdge(4, 1, 3);
    g.addEdge(4, 3, 9);
    g.addEdge(1, 3, 2);
    g.addEdge(2, 3, 4);

    g.dijkstra(0);  // Starting from vertex 0

    return 0;
}
```

Output:

Vertex Distance from Source

0 0

1	8
2	9
3	10
4	5

Time Complexity:

- **O((V + E) log V)**, where V is the number of vertices and E is the number of edges. The time complexity depends on the priority queue (min-heap) operations.

---

3. Bellman-Ford Algorithm

The **Bellman-Ford Algorithm** is used to find the shortest paths from a single source vertex to all other vertices in a graph. Unlike Dijkstra's algorithm, Bellman-Ford can handle **negative edge weights**. It is slower but more versatile than Dijkstra's algorithm.

Steps of Bellman-Ford Algorithm:

1. Initialize distances from the source to all vertices as infinity and the distance to the source itself as 0.
2. Relax all edges V−1 times, where V is the number of vertices.
3. After V−1 iterations, check for negative-weight cycles. If a shorter path is found, it means the graph contains a negative-weight cycle.

Code Implementation (Bellman-Ford Algorithm in C++):

```
#include <iostream>
#include <vector>
#include <climits>

using namespace std;

// A structure to represent an edge in the graph
class Edge {
public:
    int src, dest, weight;
};

// Bellman-Ford Algorithm to find the shortest path from a source
vertex
void bellmanFord(vector<Edge>& edges, int V, int E, int src) {
    vector<int> dist(V, INT_MAX);
    dist[src] = 0;

    // Relax all edges V-1 times
    for (int i = 1; i <= V - 1; i++) {
        for (int j = 0; j < E; j++) {
```

```
            int u = edges[j].src;
            int v = edges[j].dest;
            int weight = edges[j].weight;
            if (dist[u] != INT_MAX && dist[u] + weight < dist[v]) {
                dist[v] = dist[u] + weight;
            }
        }
    }

    // Check for negative-weight cycles
    for (int i = 0; i < E; i++) {
        int u = edges[i].src;
        int v = edges[i].dest;
        int weight = edges[i].weight;
        if (dist[u] != INT_MAX && dist[u] + weight < dist[v]) {
            cout << "Graph contains negative weight cycle\n";
            return;
        }
    }

    // Print the shortest distances
    cout << "Vertex   Distance from Source\n";
    for (int i = 0; i < V; i++) {
        cout << i << " \t\t " << dist[i] << endl;
    }
}

int main() {
    int V = 5;  // Number of vertices
    int E = 8;  // Number of edges

    vector<Edge> edges = {
        {0, 1, -1}, {0, 2, 4}, {1, 2, 3}, {1, 3, 2},
        {1, 4, 2}, {3, 2, 5}, {3, 1, 1}, {4, 3, -3}
    };

    bellmanFord(edges, V, E, 0);  // Starting from vertex 0

    return 0;
}
```

Output:

Vertex Distance from Source

0 0

1 -1

2 2

3 -2

4 1

Time Complexity:

- **O(V * E)**, where V is the number of vertices and E is the number of edges.

Comparison: Dijkstra vs Bellman-Ford

Feature	Dijkstra's Algorithm	Bellman-Ford Algorithm
Edge Weights	Non-negative only	Handles negative weights
Time Complexity	O((V + E) log V)	O(V * E)
Graph Type	Weighted	Weighted
Cycle Detection	No cycle detection	Can detect negative cycles
Use Case	Faster for non-negative graphs	Handles more complex graphs

Both algorithms solve the single-source shortest path problem but in different scenarios. Dijkstra's is efficient for non-negative graphs, while Bellman-Ford is more flexible and can detect negative cycles.

Backtracking

Backtracking is an algorithmic paradigm that solves problems by building a solution incrementally and abandoning a path ("backtracks") as soon as it determines that this path cannot lead to a valid solution. It explores all possible options but eliminates many by pruning paths that are not feasible, optimizing search time.

Basic Terminologies

1. **Recursive Search**: Backtracking typically uses recursion to explore possible solutions.
2. **Pruning**: It avoids exploring paths that don't lead to a solution.
3. **State Space Tree**: It represents all possible solutions of a problem. Each node in this tree represents a partial solution.

In backtracking, a solution is built step by step, trying one option at each step. If the current option leads to an invalid state, the algorithm backtracks to the previous step and tries a different option.

Backtracking Problems

1. N-Queens Problem

The **N-Queens Problem** is a classic example of backtracking, where the task is to place N queens on an N×N chessboard such that no two queens threaten each other (i.e., no two queens share the same row, column, or diagonal).

Steps:

1. Place the first queen in the first column of the first row.
2. Place the second queen in the next row such that it doesn't conflict with the previously placed queens.
3. Continue placing queens row by row. If a conflict arises, backtrack to the previous row and try a different column.

Code Implementation (N-Queens in C++):

```
#include <iostream>
#include <vector>
using namespace std;

#define N 8

void printSolution(vector<vector<int>>& board) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            cout << board[i][j] << " ";
        }
        cout << endl;
    }
}

bool isSafe(vector<vector<int>>& board, int row, int col) {
    // Check this column on the upper side
    for (int i = 0; i < row; i++) {
        if (board[i][col] == 1) {
            return false;
        }
    }

    // Check upper diagonal on the left side
    for (int i = row, j = col; i >= 0 && j >= 0; i--, j--) {
        if (board[i][j] == 1) {
            return false;
        }
    }

    // Check upper diagonal on the right side
```

```
    for (int i = row, j = col; i >= 0 && j < N; i--, j++) {
        if (board[i][j] == 1) {
            return false;
        }
    }

    return true;
}

bool solveNQueensUtil(vector<vector<int>>& board, int row) {
    if (row >= N) {
        return true;  // All queens are placed
    }

    for (int col = 0; col < N; col++) {
        if (isSafe(board, row, col)) {
            board[row][col] = 1;  // Place the queen

            // Recursively place the rest of the queens
            if (solveNQueensUtil(board, row + 1)) {
                return true;
            }

            board[row][col] = 0;  // Backtrack
        }
    }

    return false;  // No valid position found
}

bool solveNQueens() {
    vector<vector<int>> board(N, vector<int>(N, 0));

    if (!solveNQueensUtil(board, 0)) {
        cout << "Solution does not exist\n";
        return false;
    }

    printSolution(board);
    return true;
}

int main() {
    solveNQueens();
    return 0;
}
```

Output:

```
1 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 1
0 0 1 0 0 0 0 0
```

```
0 0 0 0 0 1 0 0
0 1 0 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 0 0 1 0
```

Time Complexity:

- The worst-case time complexity is **O(N!)** as there are N! ways to arrange N queens on an N×N board.

---

2. Hamiltonian Circuit Problem

The **Hamiltonian Circuit** problem involves finding a cycle in a graph that visits each vertex exactly once and returns to the starting vertex.

Steps:

1. Start from an arbitrary vertex.
2. Recursively visit unvisited vertices.
3. If you visit all vertices and return to the start, you have a Hamiltonian cycle.
4. If no such cycle is found, backtrack to the previous vertex and try a different path.

Code Implementation (Hamiltonian Circuit in C++):

```
#include <iostream>
#include <vector>
using namespace std;

#define V 5

bool isSafe(int v, vector<vector<int>>& graph, vector<int>& path,
int pos) {
    if (graph[path[pos - 1]][v] == 0) {
        return false;  // No edge between previous vertex and
current vertex
    }

    for (int i = 0; i < pos; i++) {
        if (path[i] == v) {
            return false;  // Vertex already included in the path
        }
    }

    return true;
}

bool hamiltonianCycleUtil(vector<vector<int>>& graph, vector<int>&
path, int pos) {
    if (pos == V) {
```

```
        // Check if there is an edge from the last vertex to the
first
        return (graph[path[pos - 1]][path[0]] == 1);
    }

    for (int v = 1; v < V; v++) {
        if (isSafe(v, graph, path, pos)) {
            path[pos] = v;

            if (hamiltonianCycleUtil(graph, path, pos + 1)) {
                return true;
            }

            path[pos] = -1;  // Backtrack
        }
    }

    return false;
}

bool hamiltonianCycle(vector<vector<int>>& graph) {
    vector<int> path(V, -1);
    path[0] = 0;

    if (!hamiltonianCycleUtil(graph, path, 1)) {
        cout << "No Hamiltonian Cycle exists\n";
        return false;
    }

    cout << "Hamiltonian Cycle: ";
    for (int i = 0; i < V; i++) {
        cout << path[i] << " ";
    }
    cout << path[0] << endl;  // Returning to the starting vertex

    return true;
}

int main() {
    vector<vector<int>> graph = {
        {0, 1, 0, 1, 0},
        {1, 0, 1, 1, 1},
        {0, 1, 0, 0, 1},
        {1, 1, 0, 0, 1},
        {0, 1, 1, 1, 0}
    };

    hamiltonianCycle(graph);

    return 0;
}
```

Output:

Hamiltonian Cycle: 0 1 2 4 3 0

Time Complexity:

- The worst-case time complexity is **O(V!)**, where V is the number of vertices in the graph.

---

3. Subset Sum Problem

The **Subset Sum Problem** asks whether a subset of a given set of integers sums to a particular value. Backtracking can be used to explore all possible subsets.

Steps:

1. Start with an empty subset.
2. At each step, include the next element or exclude it.
3. If the current subset sum equals the target sum, return true.
4. If all subsets are explored without finding the target sum, return false.

Code Implementation (Subset Sum in C++):

```
#include <iostream>
#include <vector>
using namespace std;

bool subsetSumUtil(vector<int>& set, int n, int sum) {
    if (sum == 0) {
        return true;
    }

    if (n == 0 && sum != 0) {
        return false;
    }

    if (set[n - 1] > sum) {
        return subsetSumUtil(set, n - 1, sum);
    }

    return subsetSumUtil(set, n - 1, sum) || subsetSumUtil(set, n -
1, sum - set[n - 1]);
}

bool subsetSum(vector<int>& set, int sum) {
    int n = set.size();
    return subsetSumUtil(set, n, sum);
}
```

```
int main() {
    vector<int> set = {3, 34, 4, 12, 5, 2};
    int sum = 9;

    if (subsetSum(set, sum)) {
        cout << "Found a subset with given sum\n";
    } else {
        cout << "No subset with given sum\n";
    }

    return 0;
}
```

Output:

Found a subset with given sum

Time Complexity:

- The worst-case time complexity is **O(2^n)**, where n is the number of elements in the set.

Backtracking is a versatile method for solving combinatorial problems, as seen in the examples above. It systematically explores all possibilities while avoiding invalid paths through pruning.

Branch and Bound

Branch and Bound is an algorithmic paradigm for solving optimization problems. It is similar to backtracking but with a key difference: it uses **bounding functions** to prune parts of the search space that cannot contain the optimal solution. It is commonly used to solve combinatorial and discrete optimization problems.

Basic Terminologies

1. **Branching**: The process of dividing the problem into smaller subproblems or branches. Each branch represents a possible decision or choice in the problem.
2. **Bounding**: The use of bounds to eliminate branches of the solution space that cannot possibly lead to a better solution than the current best one. Bounding functions calculate upper and lower bounds for the optimal solution in a given branch.
3. **Pruning**: The process of discarding a branch if its bound indicates that it cannot possibly contain the optimal solution. This reduces the search space and speeds up the process.
4. **Feasible Solution**: A solution that satisfies all the constraints of the problem.

5. **Optimal Solution**: The best possible solution that maximizes or minimizes the objective function.
6. **State Space Tree**: A tree where each node represents a subproblem and its branches represent the decisions that lead to solving the subproblem.

Branch and Bound Problems

1. Knapsack Problem

The **0/1 Knapsack Problem** is a classic optimization problem where you have a set of items, each with a weight and a value, and a knapsack with a maximum weight capacity. The goal is to maximize the total value of the items placed in the knapsack without exceeding the capacity.

Steps for Branch and Bound in Knapsack:

1. **Branching**: At each step, decide whether to include or exclude the current item in the knapsack.
2. **Bounding**: Use an upper bound to estimate the maximum possible value you can get by including or excluding the item. If the bound is worse than the current best solution, prune the branch.
3. **Pruning**: Eliminate any subproblem that cannot lead to a better solution.

Code Implementation (0/1 Knapsack Problem using Branch and Bound in C++):

```
#include <iostream>
#include <queue>
#include <vector>
using namespace std;

// Structure to represent a node in the decision tree
struct Node {
    int level;
    int profit;
    int weight;
    float bound;
};

// Function to calculate upper bound on profit
float bound(Node u, int n, int W, vector<int>& wt, vector<int>& val)
{
    if (u.weight >= W) {
        return 0;
    }

    float profitBound = u.profit;
    int j = u.level + 1;
    int totalWeight = u.weight;
```

```
    while ((j < n) && (totalWeight + wt[j] <= W)) {
        totalWeight += wt[j];
        profitBound += val[j];
        j++;
    }

    if (j < n) {
        profitBound += (W - totalWeight) * (float)val[j] / wt[j];
    }

    return profitBound;
}

// Function to solve the 0/1 Knapsack Problem using Branch and Bound
int knapsack(int W, vector<int>& wt, vector<int>& val, int n) {
    queue<Node> Q;
    Node u, v;

    u.level = -1;
    u.profit = u.weight = 0;
    Q.push(u);

    int maxProfit = 0;

    while (!Q.empty()) {
        u = Q.front();
        Q.pop();

        if (u.level == -1) {
            v.level = 0;
        } else if (u.level == n - 1) {
            continue;
        } else {
            v.level = u.level + 1;
        }

        v.weight = u.weight + wt[v.level];
        v.profit = u.profit + val[v.level];

        if (v.weight <= W && v.profit > maxProfit) {
            maxProfit = v.profit;
        }

        v.bound = bound(v, n, W, wt, val);

        if (v.bound > maxProfit) {
            Q.push(v);
        }

        v.weight = u.weight;
```

```
        v.profit = u.profit;
        v.bound = bound(v, n, W, wt, val);

        if (v.bound > maxProfit) {
            Q.push(v);
        }
    }

    return maxProfit;
}

int main() {
    int W = 50;  // Knapsack capacity
    vector<int> wt = {10, 20, 30};  // Weights
    vector<int> val = {60, 100, 120};  // Values
    int n = wt.size();

    cout << "Maximum profit: " << knapsack(W, wt, val, n) << endl;

    return 0;
}
```

Output:

Maximum profit: 220

Time Complexity:

- The time complexity of Branch and Bound for the knapsack problem depends on the number of nodes in the state space tree, which can be up to **O(2^n)** in the worst case.

---

2. Travelling Salesman Problem (TSP)

The **Travelling Salesman Problem (TSP)** is a well-known combinatorial optimization problem where a salesman must visit a set of cities exactly once and return to the starting city, minimizing the total distance traveled. The TSP is an NP-hard problem, meaning that exact solutions are computationally expensive for large numbers of cities.

Steps for Branch and Bound in TSP:

1. **Branching**: The solution space is represented as a tree where each level represents visiting a new city. At each node, the algorithm decides the next city to visit.
2. **Bounding**: At each node, a lower bound on the cost of completing the tour is calculated. If the bound is higher than the current best solution, the node is pruned.
3. **Pruning**: If a branch's bound is worse than the best known solution, it is discarded.

Code Implementation (TSP using Branch and Bound in C++):

```
#include <iostream>
```

```
#include <vector>
#include <climits>
using namespace std;

#define N 4  // Number of cities

int tspBound(vector<vector<int>>& graph, int n, int currBound, int
currWeight, vector<bool>& visited, int level) {
    if (level == n) {
        if (graph[currWeight][0] != 0) {
            return currBound + graph[currWeight][0];
        } else {
            return INT_MAX;
        }
    }

    int minBound = INT_MAX;

    for (int i = 0; i < n; i++) {
        if (!visited[i] && graph[currWeight][i] != 0) {
            visited[i] = true;
            int tempBound = currBound + graph[currWeight][i];
            minBound = min(minBound, tspBound(graph, n, tempBound,
i, visited, level + 1));
            visited[i] = false;
        }
    }

    return minBound;
}

int tsp(vector<vector<int>>& graph) {
    vector<bool> visited(N, false);
    visited[0] = true;

    return tspBound(graph, N, 0, 0, visited, 1);
}

int main() {
    vector<vector<int>> graph = {
        {0, 10, 15, 20},
        {10, 0, 35, 25},
        {15, 35, 0, 30},
        {20, 25, 30, 0}
    };

    cout << "Minimum cost: " << tsp(graph) << endl;

    return 0;
}
```

Output:

Minimum cost: 80

Time Complexity:

- The worst-case time complexity is **O(n!)**, where n is the number of cities.

Comparison: Branch and Bound vs Backtracking

Feature	Backtracking	Branch and Bound
Pruning Criteria	Invalid or infeasible solutions	Use of bounding functions to prune search
Optimality	Not guaranteed to find the optimal solution	Always finds the optimal solution
Use Case	Combinatorial problems without focus on optimization	Optimization problems (minimization/maximization)

Branch and Bound is generally used for optimization problems where we need the best solution, whereas backtracking is used to explore all feasible solutions. Both techniques systematically explore the solution space, but Branch and Bound uses bounds to reduce computation.

String Processing

String processing involves operations and algorithms related to strings (sequences of characters). Key areas in string processing include string searching, pattern matching, and various algorithms designed for efficient manipulation and analysis of strings.

String Searching and Pattern Matching

String Searching is the process of finding a substring (pattern) within a larger string (text). **Pattern Matching** refers to the methods and algorithms used to find occurrences of a pattern in a text efficiently.

1. Naïve Method

The Naïve method for string searching is the simplest approach. It checks for the presence of the pattern in the text by sliding the pattern over the text one character at a time and checking for matches.

Algorithm:

1. Start at the beginning of the text.
2. Compare the first character of the pattern with the current character of the text.
3. If all characters match, a match is found.
4. If not, move the pattern one position to the right and repeat the process until the end of the text is reached.

Code Implementation (Naïve Method in C++):

```
#include <iostream>
#include <string>
using namespace std;

void naiveSearch(const string& text, const string& pattern) {
    int M = pattern.length();
    int N = text.length();

    for (int i = 0; i <= N - M; i++) {
        int j;
        for (j = 0; j < M; j++) {
            if (text[i + j] != pattern[j]) {
                break;  // Mismatch found
            }
        }
        if (j == M) {
            cout << "Pattern found at index " << i << endl;
        }
    }
}

int main() {
    string text = "ABABDABACDABABCABAB";
    string pattern = "ABABCABAB";
    naiveSearch(text, pattern);
    return 0;
}
```

Time Complexity:

- **O(N*M)** in the worst case, where N is the length of the text and M is the length of the pattern.

---

2. Finite Automata Method

The Finite Automata method is a more efficient way of string matching that uses a finite state machine to determine if the pattern occurs in the text. It preprocesses the pattern to construct a state transition table, allowing for quicker searches.

Algorithm:

1. Construct a finite automaton based on the pattern.
2. Use the automaton to scan the text.
3. Transition through states based on the current character of the text until the pattern is found.

Code Implementation (Finite Automata Method in C++):

```
#include <iostream>
#include <vector>
#include <string>
using namespace std;

#define ALPHABET_SIZE 256

void computeTF(const string& pattern, vector<vector<int>>& TF) {
    int M = pattern.length();
    int state = 0;

    for (int i = 0; i < M; i++) {
        TF[state][pattern[i]] = i + 1;
        for (int j = 0; j < ALPHABET_SIZE; j++) {
            TF[state][j] = TF[0][j];  // Default transition
        }
        state = TF[state][pattern[i]];
    }
}

void finiteAutomataSearch(const string& text, const string& pattern)
{
    int M = pattern.length();
    int N = text.length();
    vector<vector<int>> TF(M + 1, vector<int>(ALPHABET_SIZE, 0));

    computeTF(pattern, TF);

    int state = 0;
    for (int i = 0; i < N; i++) {
        state = TF[state][text[i]];
        if (state == M) {
            cout << "Pattern found at index " << i - M + 1 << endl;
        }
    }
}

int main() {
    string text = "ABABDABACDABABCABAB";
    string pattern = "ABABCABAB";
    finiteAutomataSearch(text, pattern);
```

```
    return 0;
}
```

Time Complexity:

- **O(N + M + |Σ|)**, where |Σ| is the size of the alphabet.

---

3. Knuth-Morris-Pratt (KMP) Method

The KMP algorithm improves the efficiency of string matching by preprocessing the pattern to create a longest prefix-suffix (LPS) array. This array helps to skip unnecessary comparisons in the text when a mismatch occurs.

Algorithm:

1. Construct the LPS array for the pattern.
2. Use the LPS array to skip characters in the text when a mismatch occurs, allowing for faster searching.

Code Implementation (KMP Method in C++):

```
#include <iostream>
#include <vector>
#include <string>
using namespace std;

void computeLPS(const string& pattern, vector<int>& lps) {
    int M = pattern.length();
    int len = 0;  // Length of previous longest prefix suffix
    lps[0] = 0;   // lps[0] is always 0
    int i = 1;

    while (i < M) {
        if (pattern[i] == pattern[len]) {
            len++;
            lps[i] = len;
            i++;
        } else {
            if (len != 0) {
                len = lps[len - 1];
            } else {
                lps[i] = 0;
                i++;
            }
        }
    }
}

void KMPsearch(const string& text, const string& pattern) {
    int N = text.length();
    int M = pattern.length();
```

```
    vector<int> lps(M, 0);

    computeLPS(pattern, lps);

    int i = 0;  // Index for text
    int j = 0;  // Index for pattern
    while (i < N) {
        if (pattern[j] == text[i]) {
            i++;
            j++;
        }
        if (j == M) {
            cout << "Pattern found at index " << i - j << endl;
            j = lps[j - 1];
        } else if (i < N && pattern[j] != text[i]) {
            if (j != 0) {
                j = lps[j - 1];
            } else {
                i++;
            }
        }
    }
}

int main() {
    string text = "ABABDABACDABABCABAB";
    string pattern = "ABABCABAB";
    KMPsearch(text, pattern);
    return 0;
}
```

Time Complexity:

- **O(N + M)**, where N is the length of the text and M is the length of the pattern.

Comparison of String Matching Algorithms

Algorithm	Time Complexity	Space Complexity	Best Use Case
Naïve Method	O(N*M)	O(1)	Simple implementations with small data
Finite Automata Method	O(N + M + Σ)		
Knuth-Morris-Pratt (KMP)	O(N + M)	O(M)	Efficient single pattern matching

---

Conceptual Introduction to NP-Completeness

NP-completeness is a fundamental concept in computer science and computational theory that deals with the classification of problems based on their computational difficulty and the resources required to solve them.

---

1. Deterministic and Non-deterministic Algorithms

- **Deterministic Algorithms**:
    - An algorithm is said to be deterministic if, for a given input, it produces the same output every time. The execution of the algorithm follows a predictable sequence of operations.
    - **Example**: A sorting algorithm like QuickSort, where the same array will yield the same sorted order each time it is executed.
- **Non-deterministic Algorithms**:
    - An algorithm is non-deterministic if it can produce different outputs on the same input or can explore multiple paths simultaneously. Non-deterministic algorithms often involve guessing and checking.
    - **Example**: A non-deterministic Turing machine that can take multiple paths to find a solution, effectively trying all possibilities at once.

---

2. P and NP: Definitions and Concepts

- **Class P**:
    - **Definition**: The class of decision problems (problems with a yes/no answer) that can be solved by a deterministic algorithm in polynomial time. That is, the time taken to solve the problem grows polynomially with the input size.
    - **Example**: Sorting a list, finding the greatest common divisor (GCD), etc.
- **Class NP**:
    - **Definition**: The class of decision problems for which a given solution can be verified in polynomial time by a deterministic algorithm. In other words, if we are given a candidate solution, we can check if it's correct in polynomial time.

  - **Example**: The Boolean satisfiability problem (SAT), where given a boolean expression, we can verify if a specific assignment of variables satisfies it.

- **P vs NP**:
  - The major question in computer science is whether P=NP or P≠NP. This translates to whether every problem for which a solution can be verified quickly (in polynomial time) can also be solved quickly.

---

3. NP-Completeness: Statement of Cook's Theorem

- **NP-Complete**:
  - A decision problem is NP-complete if:
    1. It is in NP.
    2. Every problem in NP can be reduced to it in polynomial time.
- **Cook's Theorem**:
  - **Statement**: The Boolean satisfiability problem (SAT) is NP-complete. This was the first problem proven to be NP-complete, establishing the foundation for the theory of NP-completeness.
  - The theorem implies that if any NP-complete problem can be solved in polynomial time, then every problem in NP can also be solved in polynomial time.

---

4. Standard NP-Complete Problems

Some well-known NP-complete problems include:

1. **Satisfiability Problem (SAT)**:
   - Given a boolean expression, determine if there is an assignment of variables that makes the expression true.
2. **Vertex Cover**:
   - Given a graph, find the smallest subset of vertices such that every edge in the graph is incident to at least one vertex in the subset.
3. **Hamiltonian Cycle**:
   - Determine whether there exists a cycle in a graph that visits each vertex exactly once.
4. **Traveling Salesman Problem (Decision Version)**:

   - Given a set of cities and distances between them, is there a tour that visits each city exactly once with a total distance less than or equal to a given number?

5. **Subset Sum Problem**:
   - Given a set of integers, is there a non-empty subset whose sum equals a given target sum?

---

5. Approximation Algorithms

- **Definition**:
  - Approximation algorithms are algorithms designed to find approximate solutions to optimization problems where finding the exact solution is computationally infeasible (often NP-hard).
- **Characteristics**:
  - These algorithms provide solutions that are close to the optimal solution within some factor or guarantee.
  - They are particularly useful in real-world applications where exact solutions are less critical than obtaining a solution quickly.
- **Example**:
  - The **Traveling Salesman Problem** can be solved using approximation algorithms such as the **Nearest Neighbor** heuristic, which quickly finds a tour but does not guarantee the shortest possible route.

Summary

Understanding NP-completeness is crucial for recognizing the limitations of algorithmic problem-solving. It helps in identifying which problems can be solved efficiently and which require alternative approaches, such as approximation algorithms, to yield acceptable solutions in reasonable time frames. The ongoing debate around PPP versus NPNPNP remains one of the most significant open questions in theoretical computer science.