UNIT – 4

1. **Matrix Algorithms**

Basics of Matrices

- **Matrix**: A matrix is a rectangular arrangement of numbers or symbols into rows and columns.
- Represented as:

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix}$$

- Each element is denoted by a_{ij}, where i is the row number and j is the column number.
- **Types of Matrices**:
  1. **Square Matrix**: Rows = Columns (e.g., 3x3 matrix).
  2. **Diagonal Matrix**: Only diagonal elements (from top-left to bottom-right) are non-zero.
  3. **Identity Matrix**: A diagonal matrix where diagonal elements are all 1.
  4. **Transpose**: Flipping rows and columns. (A^T) is the transpose of matrix A.
  5. **Symmetric Matrix**: ($A = A^T$).

Strassen's Matrix Multiplication Algorithm

Strassen's algorithm is an efficient method for multiplying matrices. It was developed by Volker Strassen in 1969 and is faster than the traditional $O(n^3)$ matrix multiplication algorithm, particularly for large matrices. The algorithm uses a **divide-and-conquer** approach to reduce the number of multiplications required, thereby reducing the time complexity.

- **Purpose**: An efficient algorithm for matrix multiplication. It reduces the time complexity compared to the traditional method.
- **Traditional Algorithm**: Takes $O(n^3)$ time for multiplying two matrices of size (n x n).

$$C[i][j] = \sum_{k=1}^{n} A[i][k] \times B[k][j]$$

where C[i][j] is the element at row i and column j of the resultant matrix C.

- **Strassen's Algorithm**: Reduces time complexity to $O(n^{\log_2 7})$ ~ $O(n^{2.81})$.

Strassen's Matrix Multiplication Overview

- **Objective**: Reduce the number of multiplications from 8 (in a naive 2×2 matrix multiplication) to 7, leading to a faster algorithm.
- **Time Complexity**: Strassen's algorithm achieves a time complexity of approximately $O(n^{2.81})$ which is better than the traditional $O(n^3)$.

Key Idea: Divide and Conquer

- Strassen's algorithm works by dividing the matrices into smaller submatrices and performing recursive multiplications.
- The two n × n matrices A and B are divided into four n/2 × n/2 submatrices as follows:

$$M_1 = (A_{11} + A_{22}) \times (B_{11} + B_{22})$$
$$M_2 = (A_{21} + A_{22}) \times B_{11}$$
$$M_3 = A_{11} \times (B_{12} - B_{22})$$
$$M_4 = A_{22} \times (B_{21} - B_{11})$$
$$M_5 = (A_{11} + A_{12}) \times B_{22}$$
$$M_6 = (A_{21} - A_{11}) \times (B_{11} + B_{12})$$
$$M_7 = (A_{12} - A_{22}) \times (B_{21} + B_{22})$$

Using these intermediate matrices, the resultant submatrices C_{11}, C_{12}, C_{21} and C_{22} of matrix C are computed as:

$$C_{11} = M_1 + M_4 - M_5 + M_7$$
$$C_{12} = M_3 + M_5$$
$$C_{21} = M_2 + M_4$$
$$C_{22} = M_1 - M_2 + M_3 + M_6$$

Final Result

- The final matrix CCC is obtained by combining the submatrices:

$$C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

Advantages of Strassen's Algorithm

1. **Faster Time Complexity**: Strassen's algorithm reduces the time complexity to approximately $O(n^{2.81})$ compared to the traditional $O(n^3)$ making it faster for large matrices.
2. **Efficient for Large Matrices**: It performs better when n (the matrix size) is large, as the divide-and-conquer approach reduces the number of multiplications needed.

Limitations

1. **Memory Overhead**: Strassen's algorithm introduces extra overhead in terms of memory due to the need for temporary storage of intermediate matrices.
2. **Not Practical for Small Matrices**: For small matrices, the traditional multiplication method can be more efficient due to the overhead of recursion and intermediate matrix storage.
3. **Numerical Stability**: Strassen's algorithm may be less numerically stable in some cases because of how it combines matrix elements during multiplication.

2. **Data Structures for Set Manipulation Problems**

Fundamental Operations on Sets

- **Sets**: A collection of distinct objects (elements). Sets are used in algorithms to manage groups of items.
- **Basic Operations**:

1. **Union**: Combine two sets into a single set containing all elements.
2. **Find**: Find the "representative" or "leader" of the set to which a specific element belongs.
3. **Intersection**: Find common elements between two sets.
4. **Difference**: Elements that are in one set but not in the other.

Disjoint-Set Union Algorithm

Disjoint Sets: A collection of sets where no element is shared between any two sets. The **Union-Find** algorithm, also known as **Disjoint Set Union (DSU)**, is a data structure that helps manage a collection of disjoint (non-overlapping) sets. It supports two primary operations:

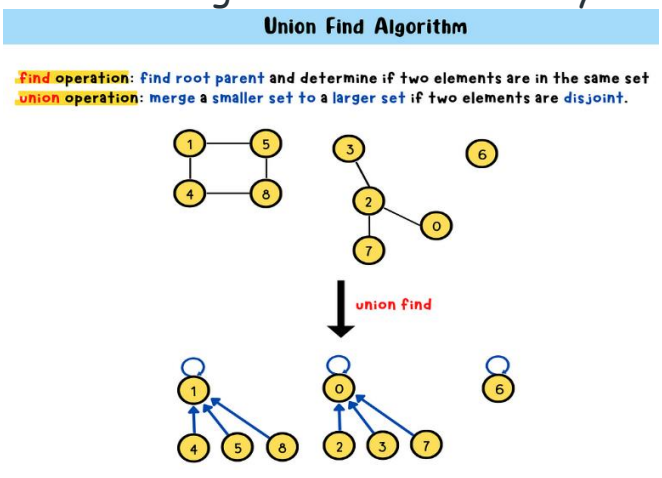
1. **Find:** Determines which set a particular element belongs to.
2. **Union:** Merges two sets into one.

The Union-Find algorithm is commonly used in graph-related problems, especially in **Kruskal's Minimum Spanning Tree (MST)** algorithm and in detecting **cycle** in an undirected graph. **Key Concepts:**

- **Find Operation:** It determines the root or representative of the set containing an element. To optimize this operation, **path compression** is used, which flattens the structure of the tree, making future queries faster.
- **Union Operation:** This merges two sets into one. To maintain efficiency, **union by rank** or **union by size** is used, which ensures the smaller set (or tree) is merged under the larger set.

Applications of the UNION-FIND Algorithm

- **Kruskal's Algorithm for Minimum Spanning Tree**: The Union-Find data structure is used to check if two vertices are in the same set and to union them, preventing cycles in the spanning tree.
- **Connected Components in a Graph**: Union-Find helps identify and manage connected components by determining which vertices are part of the same set.
- **Cycle Detection in Graphs**: By checking if two vertices are already in the same set before merging, the Union-Find algorithm can detect cycles in a graph.



UNION FIND ALGORITHM

```
function find(x):
    if parent[x] != x:
        parent[x] = find(parent[x])  # Recursively find the root and compress the path
    return parent[x]

function union(x, y):
    rootX = find(x)
    rootY = find(y)

    if rootX != rootY:  # If x and y are in different sets
        if rank[rootX] > rank[rootY]:
            parent[rootY] = rootX  # Make rootX the parent
        elif rank[rootX] < rank[rootY]:
            parent[rootX] = rootY  # Make rootY the parent
        else:
            parent[rootY] = rootX  # Arbitrarily make rootX the parent
            rank[rootX] += 1  # Increase the rank of rootX
```

UNIT - 5

1. Finite Automata and Regular Expressions

Finite Automata (FA): A Finite Automata is a mathematical model that consists of a finite number of states. It processes a string of input symbols, one symbol at a time and changes states according to a transition function.

Components of Finite Automata:

1. **Q:** A finite set of states.
2. **Σ:** A finite set of input symbols (alphabet).
3. **q_0:** The initial state (one of the states in Q).
4. **F:** A set of accepting/final states (subset of Q).
5. **δ (delta):** A transition function that maps state and input symbol pairs to a next state, i.e., $\delta: Q \times \Sigma \rightarrow Q$.

Types of Finite Automata:

- **Deterministic Finite Automaton (DFA):**
  - For each state and input symbol, there is exactly one possible next state.
  - Example:
  - DFA to accept binary strings that end in '01':

- States: $Q = \{q_0, q_1, q_2\}$
- Alphabet: $\Sigma = \{0, 1\}$
- Transition function:
- $\delta(q_0, 0) = q_0$, $\delta(q_0, 1) = q_1$
- $\delta(q_1, 0) = q_2$, $\delta(q_1, 1) = q_1$
- $\delta(q_2, 0) = q_0$, $\delta(q_2, 1) = q_1$
- Final state: q_2

- **Non-deterministic Finite Automaton (NFA):**
  - A state can have multiple transitions for the same input symbol, or no transitions.
  - Example: Same language as DFA but with nondeterministic transitions.

Key Differences:

- In a DFA, the next state is uniquely determined, whereas in an NFA, multiple transitions are possible.

b. Regular Expressions

- **Definition:** Regular expressions are patterns used to describe regular languages. They are used to search for strings that match specific patterns.
- **Components:**
- **Literals:** Basic characters, e.g., a, b, 0, 1.
- **Operators:**
  - **Concatenation:** Combining symbols (e.g., ab matches "ab").
  - **Union (|):** Matches either symbol (e.g., a|b matches "a" or "b").
  - **Kleene Star (*):** Matches zero or more repetitions of the preceding symbol (e.g., a* matches "", "a", "aa", etc.).

Example Regular Expression:

- (a|b)*abb: Describes all strings that start with any combination of "a" or "b" and end with "abb".

Equivalence to Finite Automata:

- Any regular expression can be represented by a finite automaton, and any language that can be described by a finite automaton can also be described by a regular expression.

c. Recognition of Regular Expressions and Patterns

- **Pattern Recognition:**
- Regular expressions are widely used in programming for text pattern matching.
- Example: Finding all email addresses in a document.
  - Regular Expression: [a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}.
- **Recognition of Substrings:**

- FA can be designed to check if a particular substring exists within a string. For example, a DFA can be created to recognize whether the substring "101" is present in a binary string.

d. **Conversion from NFA to DFA (Subset Construction Algorithm)**

- **Algorithm Steps:**

1. **Start** with the initial state of the NFA.
2. **Create new DFA states** by combining NFA states into sets.
3. **Add transitions** based on the possible states the NFA can move to from each set.
4. **Repeat** until all states and transitions are processed.

Example:

- Convert an NFA that accepts the language of strings ending with "01" into a DFA. You will have a DFA with sets of NFA states representing each possible situation in the string processing.

2. Complexity Theory Overview

a. **Turing Machine**

- **Definition:** A Turing Machine (TM) is a theoretical device that manipulates symbols on a strip of tape according to a set of rules. It serves as the foundation of modern computational theory.

Components of a Turing Machine:

- **Tape:** Infinite memory divided into cells.
- **Head:** Reads and writes symbols on the tape and moves left or right.
- **States:** The machine has a finite set of states.
- **Transition Function:** Specifies the machine's actions based on the current state and symbol under the head.

Significance:

- Turing Machines define what is computationally possible. If a problem can be solved by a Turing Machine, it is called **decidable**.

b. **Polynomial and Non-Polynomial Problems**

- **Polynomial Time (P):**
- A problem is said to be in **P** if it can be solved by an algorithm in polynomial time, i.e., the running time is a polynomial function of the input size (e.g., n^2, n^3).
- Example: Sorting algorithms like Merge Sort and searching algorithms like Binary Search.
- **Non-Polynomial Time (NP):**
- A problem is in **NP** if a solution can be verified in polynomial time, but finding the solution may take exponential time.

- Example: The Traveling Salesman Problem (TSP) where you must find the shortest path visiting all cities once.

c. **Deterministic and Non-Deterministic Algorithms**

- **Deterministic Algorithms:**
- These algorithms have a predefined behavior and always produce the same output for the same input.
- Example: Depth-First Search (DFS).
- **Non-Deterministic Algorithms:**
- These algorithms can make several possible moves at each step. They are often theoretical and are used in discussions of complexity classes like NP.
- Example: Guessing the correct solution in NP problems and verifying it in polynomial time.

Problem Classes: P, NP, NP-Complete, and NP-Hard

1. **P-Class (Polynomial Time):**

- These are problems that can be **solved in polynomial time**, meaning the time required to solve the problem can be expressed as a polynomial function of the input size (e.g., $O(n^k)$ where (n) is the input size and (k) is a constant).
- In simple terms, problems in P can be efficiently solved by an algorithm in a reasonable amount of time.
- **Example**: Finding the shortest path in a graph using **Dijkstra's algorithm** (which runs in $O(V^2)$ for V vertices).

2. **NP-Class (Non-deterministic Polynomial Time):**

- These are problems for which a solution, once found, can be **verified in polynomial time**, even though finding the solution itself might not be possible in polynomial time.
- NP problems do not have a known efficient solution algorithm, but if you are given a solution, you can verify its correctness quickly (in polynomial time).
- **Example**: Solving a **Sudoku puzzle**. It might take a long time to find the correct solution, but once you have one, checking whether it is valid is quick.

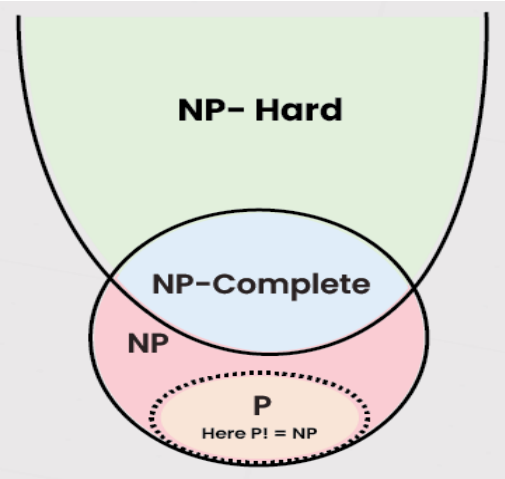
3. **NP-Complete Problems:**

- These are the **hardest problems** within the NP class. An NP-complete problem is a problem in NP for which every other problem in NP can be reduced to it in polynomial time.
- If **any NP-complete problem** can be solved in polynomial time, then **all NP problems can be solved in polynomial time**.
- NP-complete problems are both in NP and as hard as any other problem in NP.

- **Example**: The **Boolean satisfiability problem (SAT)** is an NP-complete problem, meaning if you could solve SAT in polynomial time, you could solve all NP problems efficiently.

4. **NP-Hard Problems:**

- NP-Hard problems are at least as hard as the hardest problems in NP. However, unlike NP-complete problems, they are **not necessarily in NP**, meaning they don't have to be verifiable in polynomial time.
- These problems are considered harder than NP-complete problems because they are not limited to decision problems.
- **Example**: The **Travelling Salesman Problem (TSP)** is an NP-hard problem. Finding the shortest possible route that visits each city exactly once is a computationally challenging task.
- Importantly, NP-Hard problems might not be decision problems and can be **optimization** or other types of problems.



DIFFERENCE BETWEEN DFA AND NFA

Basis	DFA (Deterministic Finite Automata)	NFA (Nondeterministic Finite Automata)
Full Form	DFA stands for Deterministic Finite Automata.	NFA stands for Nondeterministic Finite Automata.
State Transition	For each symbolic representation of the alphabet, there is only one state transition.	No need to specify how the NFA reacts according to a symbol; it can have multiple transitions.
Empty String Transition (Epsilon Move)	DFA cannot use Empty String (Epsilon) transition.	NFA can use Empty String (Epsilon) transition.
Ease of Construction	DFA is more difficult to construct.	NFA is easier to construct.

Basis	DFA (Deterministic Finite Automata)	NFA (Nondeterministic Finite Automata)
Computation Model	DFA can be understood as one machine.	NFA can be understood as multiple machines computing at the same time.
Next State	In DFA, the next state is uniquely determined for each input symbol.	In NFA, each pair of state and input symbol can have multiple possible next states.
String Rejection	DFA rejects the string if it terminates in a state different from the accepting state.	NFA rejects the string if all branches die or fail to accept the string.
Execution Time	DFA generally takes less time to execute as there is only one transition for each input.	NFA can take more time due to multiple possible transitions and branches.
Relation	All DFAs are NFAs.	Not all NFAs are DFAs.
Space Complexity	DFA requires more space to store transitions.	NFA requires less space than DFA due to fewer distinct state transitions.
Dead Configuration	Dead configuration is not allowed. Example: Input "0" on q_0 must lead to a valid transition.	Dead configuration is allowed. Example: Input "0" on q_0 may lead to multiple states or even no state.
Transition Function	$\delta: Q \times \Sigma \rightarrow Q$ (next state belongs to Q).	$\delta: Q \times (\Sigma \cup \varepsilon) \rightarrow 2^Q$ (next state belongs to the power set of Q).
Conversion from Regular Expression	Conversion of a regular expression to DFA is difficult.	Conversion of a regular expression to NFA is simpler compared to DFA.
Epsilon Move	Epsilon move is not allowed in DFA.	Epsilon move is allowed in NFA.
Transition per Input Symbol	In a DFA, there is only one possible transition for each input symbol from any state.	In an NFA, there can be multiple transitions for a single input symbol from a given state.

DIFFERENCE BETWEEN ALL THE METHODS

Basis	Divide and Conquer	Greedy Method	Backtracking	Dynamic Programming	Branch and Bound
Approach	Breaks the problem into smaller subproblems, solves independently, and combines.	Builds the solution step by step, choosing locally optimal choices at each step.	Explores all possible solutions, backtracking if a solution fails.	Solves subproblems and stores results to avoid redundant work.	Explores all possible solutions but uses bounds to eliminate some paths.
Problem Type	Suitable for problems that can be divided into independent subproblems.	Suitable for optimization problems that exhibit a greedy choice property.	Suitable for problems with multiple solutions, where pruning helps reduce the search space.	Works for problems with overlapping subproblems and optimal substructure.	Suitable for optimization problems, especially where bounding can reduce complexity.
Optimal Solution	May or may not give an optimal solution depending on the problem.	Provides optimal solutions only for problems with the greedy choice property.	Provides all possible solutions, optimal or not.	Guarantees an optimal solution for most problems.	Guarantees an optimal solution, but may not explore all paths.
Efficiency	More efficient when problems can be broken down into independent subproblems.	Highly efficient when the greedy choice property holds.	Inefficient as it tries all possibilities, making it slower.	More efficient than backtracking due to memorization or tabulation.	More efficient than backtracking but slower than dynamic programming.
Overlapping Subproblems	Does not reuse previously solved subproblems.	Does not solve subproblems, works step by step.	Does not reuse previously solved subproblems.	Reuses solutions of overlapping subproblems to save computation.	Does not reuse subproblems directly but uses bounds to prune them.

Basis	Divide and Conquer	Greedy Method	Backtracking	Dynamic Programming	Branch and Bound
Solution Strategy	Combines solutions of independent subproblems to solve the main problem.	Chooses the best option at each step based on a heuristic.	Tries all possibilities and undoes choices when a dead-end is reached.	Builds solutions from previously solved subproblems and reuses them.	Explores the solution space using bounds to limit the number of solutions explored.
Common Examples	Merge Sort, Quick Sort, Binary Search.	Kruskal's Algorithm, Prim's Algorithm, Huffman Coding.	N-Queens Problem, Sudoku Solver.	Fibonacci Series, Matrix Chain Multiplication, Longest Common Subsequence.	Traveling Salesman Problem, Knapsack Problem, Job Scheduling.
Recursive / Iterative	Typically uses a recursive approach to solve subproblems.	Can be both recursive or iterative, depends on the problem.	Generally uses a recursive approach to explore solutions.	Typically uses recursion or iteration with memoization.	Often uses recursion but with bounding conditions to prune.
Storage Requirement	No need to store previous results as subproblems are independent.	No need to store previous results, as only current decisions matter.	Requires storage for all possible partial solutions explored.	Requires storage to save results of subproblems.	Storage needed for bounds and partial solutions.
When to Use	When the problem can be divided into independent smaller subproblems.	When a locally optimal choice leads to a globally optimal solution.	When the problem involves exploring all possibilities with constraints.	When subproblems overlap and solutions can be reused.	When optimization with bounding can reduce the problem space.