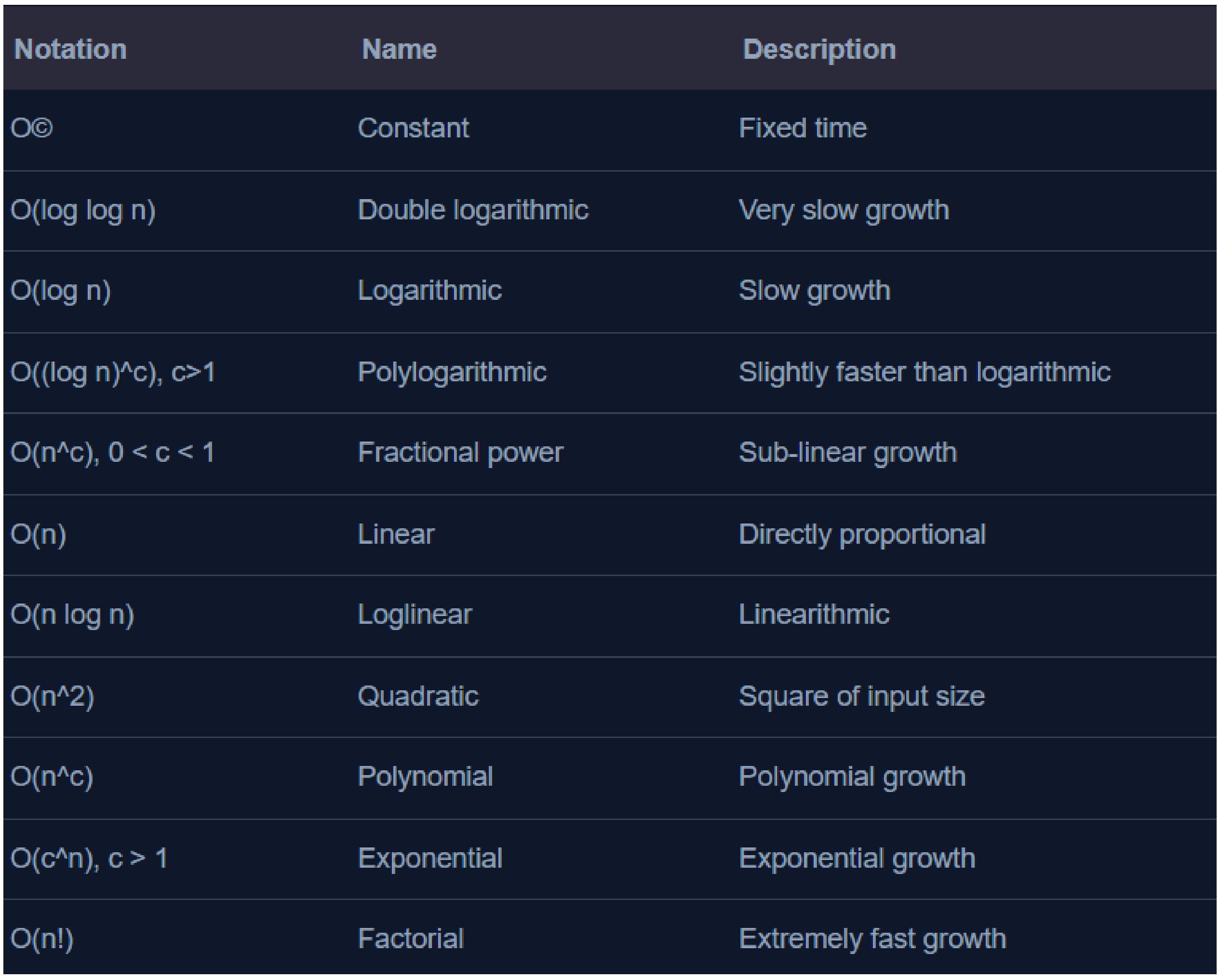
Programming, Data Structures and Algorithms using Python

This course, instructed by Prof. Madhavan Mukund covers fundamental concepts in programming, data structures, and algorithms using Python. The content spans performance measurement, algorithm analysis, complexity classes, classic searching and sorting algorithms, and Python-specific data structure implementations.

Week 1 - 3

Key Topics and Concepts

- **Timing Code Execution:**
  Use time.perf_counter() to measure elapsed time for performance evaluation.
- **Algorithm Analysis:**
  - Two main metrics: **time complexity** (running time) and **space complexity** (memory usage).
  - Running time, denoted as T(n), depends on input size n.
  - Worst-case upper bounds provide guarantees on performance.
- **Asymptotic Notations:**
  - **Big O (O):** Upper bound on growth rate (worst case).
  - **Omega (Ω):** Lower bound on growth rate (best case).
  - **Theta (Θ):** Tight bound indicating exact growth rate.
- **Orders of Magnitude (Common Complexity Classes):**



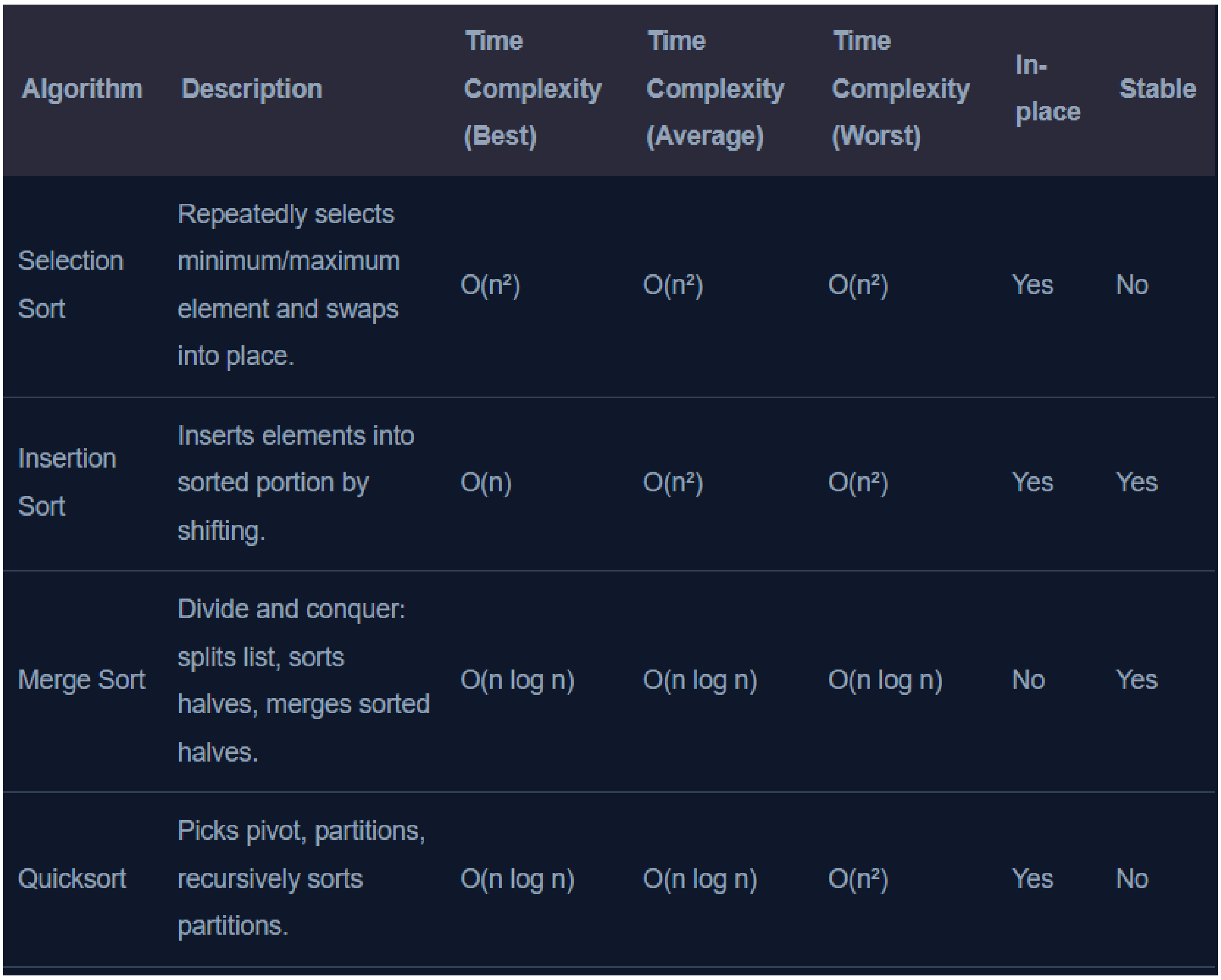
Notation	Name	Description
O©	Constant	Fixed time
O(log log n)	Double logarithmic	Very slow growth
O(log n)	Logarithmic	Slow growth
O((log n)^c), c>1	Polylogarithmic	Slightly faster than logarithmic
O(n^c), 0 < c < 1	Fractional power	Sub-linear growth
O(n)	Linear	Directly proportional
O(n log n)	Loglinear	Linearithmic
O(n^2)	Quadratic	Square of input size
O(n^c)	Polynomial	Polynomial growth
O(c^n), c > 1	Exponential	Exponential growth
O(n!)	Factorial	Extremely fast growth

- **Calculating Complexity:**
  - For **iterative algorithms**, focus on loop iterations.
  - For **recursive algorithms**, write and solve recurrence relations.

Searching Algorithms

- **Linear Search:**
  - Checks every element sequentially.
  - Worst-case complexity: O(n).
  - Best-case: O(1).
- **Binary Search:**
  - Requires sorted list.
  - Halves the search space each step.
  - Recurrence: $T(n) = T(n/2) + 1$.
  - Complexity: O(log n) for average and worst cases; O(1) best case.

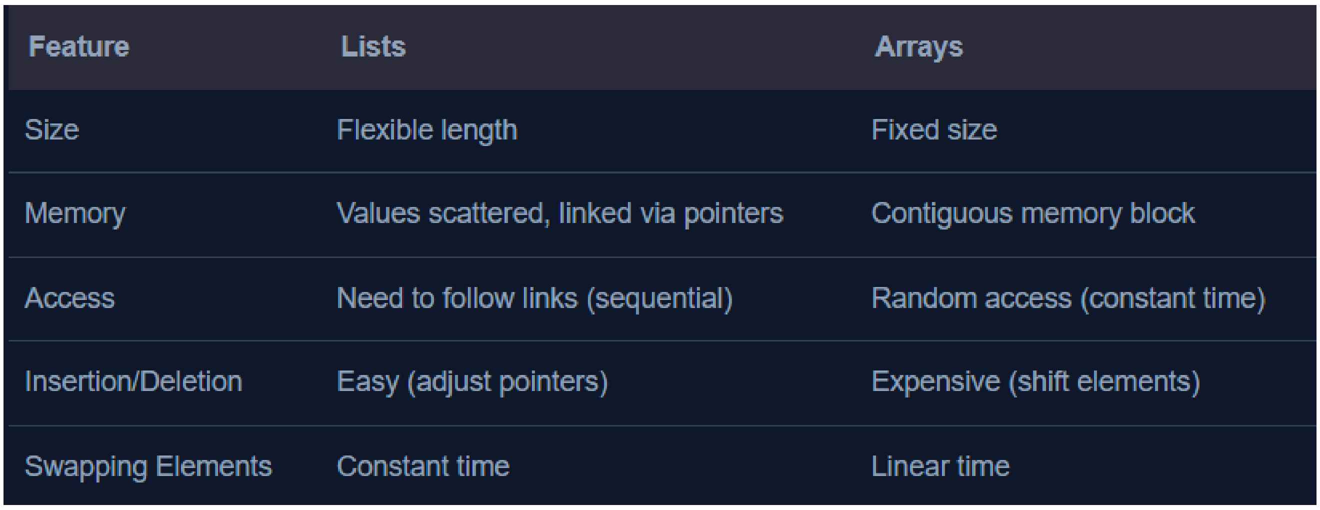
Sorting Algorithms



Algorithm	Description	Time Complexity (Best)	Time Complexity (Average)	Time Complexity (Worst)	In-place	Stable
Selection Sort	Repeatedly selects minimum/maximum element and swaps into place.	$O(n^2)$	$O(n^2)$	$O(n^2)$	Yes	No
Insertion Sort	Inserts elements into sorted portion by shifting.	O(n)	$O(n^2)$	$O(n^2)$	Yes	Yes
Merge Sort	Divide and conquer: splits list, sorts halves, merges sorted halves.	O(n log n)	O(n log n)	O(n log n)	No	Yes
Quicksort	Picks pivot, partitions, recursively sorts partitions.	O(n log n)	O(n log n)	$O(n^2)$	Yes	No

- **Comparison Highlights:**
  - Merge sort is stable but not in-place.
  - Quicksort is in-place but not stable and has worst-case $O(n^2)$.
  - Selection sort is simple but inefficient.
  - Insertion sort performs well on nearly sorted data due to best-case O(n).

Data Structures: Lists vs. Arrays



Feature	Lists	Arrays
Size	Flexible length	Fixed size
Memory	Values scattered, linked via pointers	Contiguous memory block
Access	Need to follow links (sequential)	Random access (constant time)
Insertion/Deletion	Easy (adjust pointers)	Expensive (shift elements)
Swapping Elements	Constant time	Linear time

Python-Specific Implementations

- **Python Lists:**
  - Internally implemented as arrays, not linked lists.
  - Allocate fixed-size arrays which are doubled when capacity is exceeded.
  - append() and pop() operate in amortized O(1) time.
  - Insertion and deletion generally O(n).
  - Behave more like dynamic arrays than traditional lists.
- **Python Dictionaries:**
  - Implemented as **hash tables** combining arrays with hash functions.
  - Collision handling strategies include:
    - Open addressing (probing for free space)
    - Open hashing (each slot points to a list of key-value pairs)
  - Design of a good hash function is crucial and challenging.

Core Insights

- Performance measurement and algorithmic complexity are foundational for understanding efficiency.
- Sorting and searching algorithms have distinct performance trade-offs and use-cases.
- Python’s built-in data structures are optimized with underlying implementations that differ from classical theoretical models.
- Asymptotic notation provides a common language to describe algorithm efficiency abstractly.

Week 4 - 6

Core Topics Covered

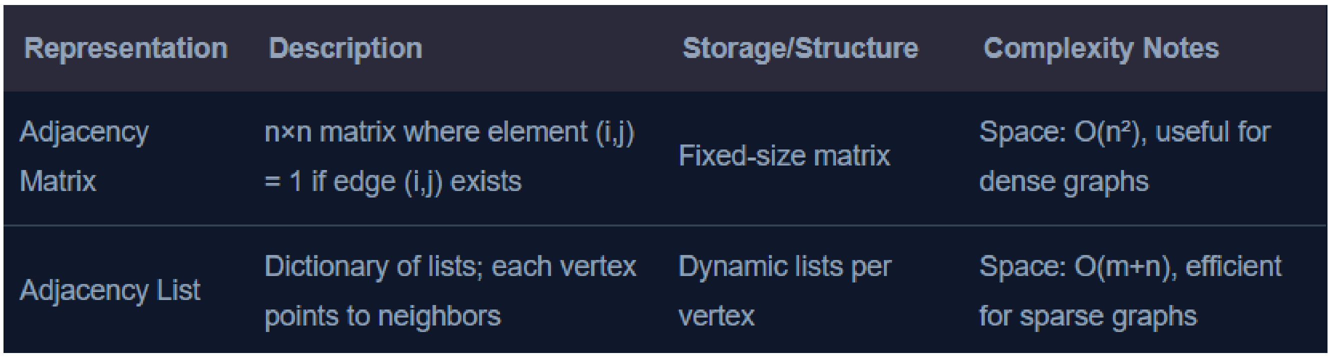
The content provides a comprehensive overview of fundamental concepts and algorithms related to **graphs, shortest paths, trees, spanning trees, and efficient data structures** in the context of programming with Python. The material is structured across multiple weeks, focusing on theory, algorithms, complexity, and practical data structures useful in algorithm design and implementation.

Detailed Outline and Key Concepts

1. Graphs and Graph Representation

- **Graphs** represent relationships between entities:
    - **Vertices (nodes):** Entities in the graph.
    - **Edges:** Connections or relationships between vertices.
- Graphs can be:
    - **Directed:** Edges have a direction (e.g., parent-child relationship).
    - **Undirected:** Edges have no direction (e.g., friendship).
- **Paths:** Sequences of connected edges.
- **Reachability:** Whether a path exists between two vertices.

Graph Representations:



Representation	Description	Storage/Structure	Complexity Notes
Adjacency Matrix	n×n matrix where element (i,j) = 1 if edge (i,j) exists	Fixed-size matrix	Space: $O(n^2)$, useful for dense graphs
Adjacency List	Dictionary of lists; each vertex points to neighbors	Dynamic lists per vertex	Space: $O(m+n)$, efficient for sparse graphs

---

2. Graph Traversal Algorithms

- **Breadth First Search (BFS):**
    - Explores graph level-by-level using a queue.
    - Tracks visited vertices and parent info to reconstruct shortest paths.
    - Computes shortest path length in terms of edges.
    - Complexity: $O(n^2)$ with adjacency matrix, $O(m+n)$ with adjacency list.
- **Depth First Search (DFS):**
    - Explores graph by diving deep using a stack (explicit or implicit via recursion).
    - Records visitation order to gain insights.
    - Complexity: $O(n^2)$ with adjacency matrix, $O(m+n)$ with adjacency list.

---

3. Applications of BFS & DFS

- BFS finds shortest paths by edge count.
- Both BFS and DFS:
    - Identify connected components in undirected graphs.
    - Produce underlying spanning trees.
    - DFS numbering techniques help detect:
        - Strongly connected components.
        - Articulation points (cut vertices).
        - Bridges (cut edges).

---

4. Directed Acyclic Graphs (DAGs)

- DAGs model dependencies (e.g., task scheduling).
- **Topological Sorting:**

- Produces a linear ordering respecting dependencies.
- Complexity: O(n^2) adjacency matrix, O(m+n) adjacency list.

- **Longest Paths in DAGs:**
  - Computed efficiently due to acyclic nature.
  - Complexity: O(m+n).

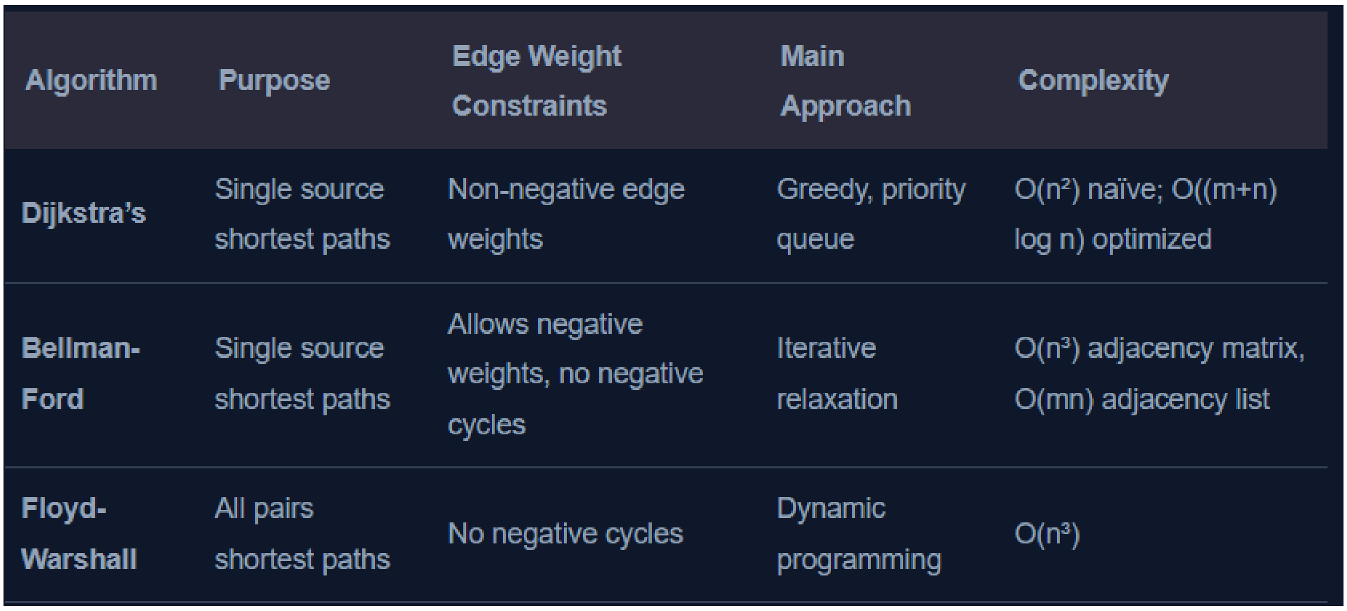
---

5. Shortest Paths in Weighted Graphs

- **Single Source Shortest Paths:**
  - **Dijkstra's Algorithm:** For graphs with non-negative edge weights.
  - **Bellman-Ford Algorithm:** Handles graphs with negative edge weights but no negative cycles.
- **All Pairs Shortest Paths:**
  - **Floyd-Warshall Algorithm:** Computes shortest paths between all vertex pairs; complexity O(n^3).
- **Negative Cycles:**
  - Presence invalidates shortest path definitions.

---

6. Algorithms for Shortest Paths



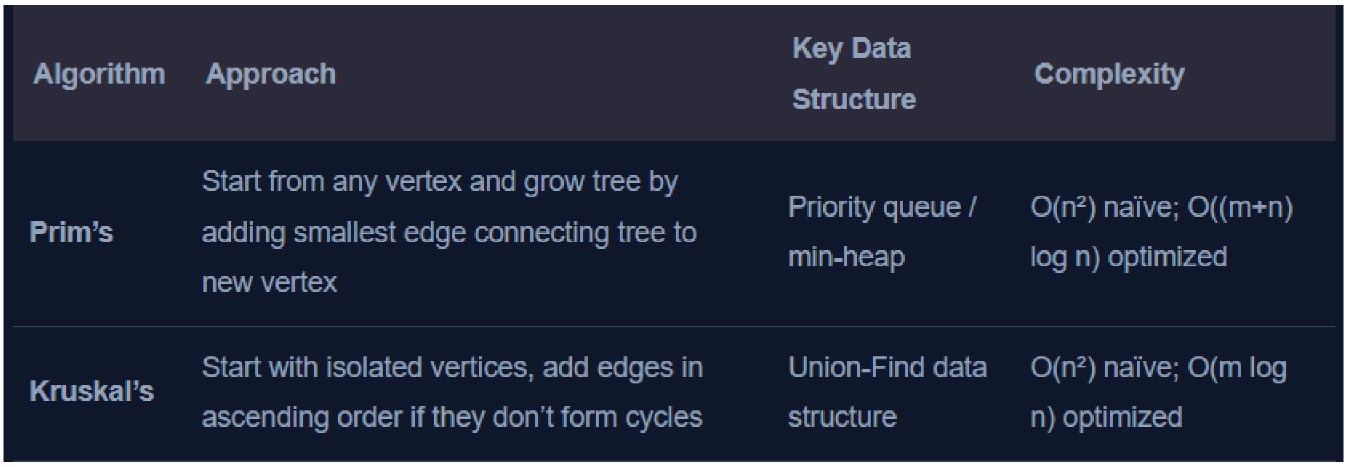
Algorithm	Purpose	Edge Weight Constraints	Main Approach	Complexity
Dijkstra's	Single source shortest paths	Non-negative edge weights	Greedy, priority queue	O(n^2) naïve; O((m+n) log n) optimized
Bellman-Ford	Single source shortest paths	Allows negative weights, no negative cycles	Iterative relaxation	O(n^3) adjacency matrix, O(mn) adjacency list
Floyd-Warshall	All pairs shortest paths	No negative cycles	Dynamic programming	O(n^3)

---

7. Trees and Spanning Trees

- **Trees:**
  - Connected acyclic graphs.
  - Properties:
    - n vertices → n–1 edges.
    - Adding an edge creates a cycle; removing an edge disconnects the graph.
    - Unique path between every pair of vertices.
- **Spanning Trees:**
  - Subgraphs connecting all vertices with minimal edges.
  - Multiple spanning trees possible for a graph.
  - **Minimum Cost Spanning Tree (MCST):** Spanning tree with the lowest sum of edge weights.
  - Unique MCST cost but possibly multiple MCSTs.

8. Algorithms for Minimum Cost Spanning Trees



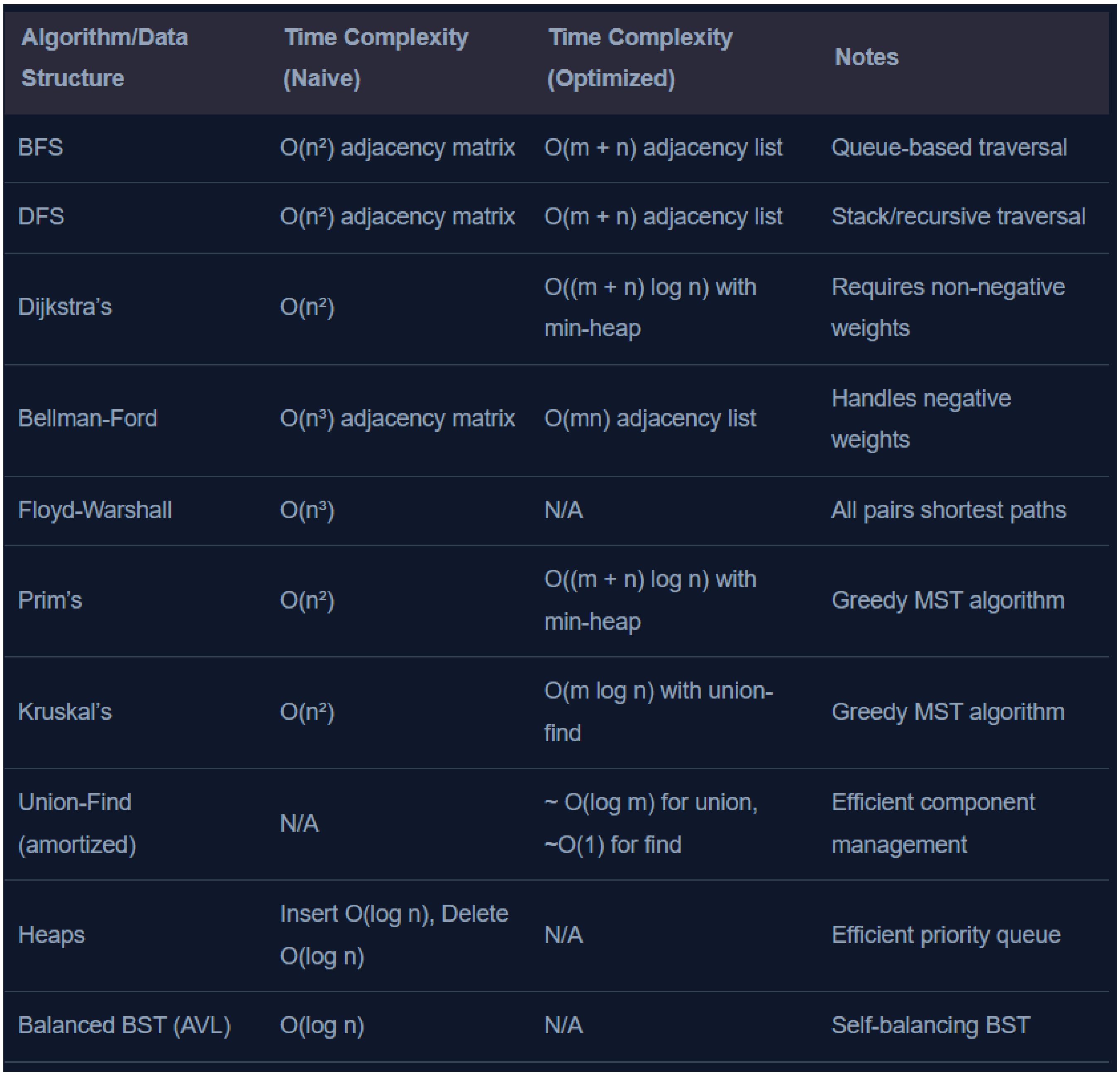
Algorithm	Approach	Key Data Structure	Complexity
Prim's	Start from any vertex and grow tree by adding smallest edge connecting tree to new vertex	Priority queue / min-heap	O(n^2) naïve; O((m+n) log n) optimized
Kruskal's	Start with isolated vertices, add edges in ascending order if they don't form cycles	Union-Find data structure	O(n^2) naïve; O(m log n) optimized

---

9. Efficient Data Structures Covered

- **Union-Find (Disjoint Set):**
  - Supports union and find operations efficiently.
  - Amortized complexity near O(log m) for union, nearly O(1) for find with path compression.
- **Priority Queues:**
  - Insert: O(n)
  - Delete max/min: O(n)
  - Processing n items: O(n^2)
- **Heaps:**
  - Complete binary trees with max-heap or min-heap property.
  - Operations:
    - Insert: O(log n)
    - Delete max/min: O(log n)
    - Heapify: O(n)
  - Used for efficient sorting (heap sort) in O(n log n).
- **Search Trees:**
  - Binary Search Tree (BST) properties:
    - Left subtree < node < right subtree.
    - No duplicates.
  - Traversals: In-order, Pre-order, Post-order.
  - Worst case unbalanced height: O(n).
  - Operations find, insert, delete can degrade to O(n).
- **Balanced Search Trees (e.g., AVL Trees):**
  - Maintain height balance: difference between left and right subtree heights ≤ 1.
  - Use rotations to maintain balance.
  - Height O(log n).
  - Operations find, insert, delete run in O(log n).

Summary Table: Algorithm Complexities



Algorithm/Data Structure	Time Complexity (Naive)	Time Complexity (Optimized)	Notes
BFS	O(n^2) adjacency matrix	O(m + n) adjacency list	Queue-based traversal
DFS	O(n^2) adjacency matrix	O(m + n) adjacency list	Stack/recursive traversal
Dijkstra's	O(n^2)	O((m + n) log n) with min-heap	Requires non-negative weights
Bellman-Ford	O(n^3) adjacency matrix	O(mn) adjacency list	Handles negative weights
Floyd-Warshall	O(n^3)	N/A	All pairs shortest paths
Prim's	O(n^2)	O((m + n) log n) with min-heap	Greedy MST algorithm
Kruskal's	O(n^2)	O(m log n) with union-find	Greedy MST algorithm
Union-Find (amortized)	N/A	~ O(log m) for union, ~O(1) for find	Efficient component management
Heaps	Insert O(log n), Delete O(log n)	N/A	Efficient priority queue
Balanced BST (AVL)	O(log n)	N/A	Self-balancing BST

Key Insights and Conclusions

- Graph traversal algorithms (BFS and DFS) form the foundation for exploring and analyzing graph structures, identifying connected components, shortest paths, and critical nodes.
- Weighted graph algorithms like Dijkstra's and Bellman-Ford allow efficient computation of shortest paths under different edge weight conditions, with Bellman-Ford uniquely handling negative weights.
- Spanning trees and MCST algorithms (Prim's and Kruskal's) are crucial for network design and optimization, relying heavily on efficient data structures like union-find and priority queues.
- Efficient data structures (heaps, union-find, balanced trees) underpin the performance improvements in classical algorithms, highlighting the importance of implementation details in algorithm efficiency.
- Balanced search trees maintain logarithmic operation times, essential for scalable data handling in dynamic sets.

Not specified/Uncertain

- Details on implementation code or Python-specific syntax are *not specified*.
- Specific use cases or examples beyond algorithm descriptions are *not specified*.

- Memory complexity beyond brief mentions is *not specified* in detail.
- Empirical performance or practical considerations in real-world datasets are *not specified*.

Week 7 - 9

Overview

This course covers fundamental algorithmic paradigms and data structures, focusing on practical and theoretical aspects of programming using Python. The content is organized into three major algorithmic strategies—**Greedy algorithms**, **Divide and Conquer**, and **Dynamic Programming**—along with detailed examples and applications.

Key Topics and Concepts

1. Greedy Algorithms

- **Definition:** A method that builds up a solution piece by piece, selecting the locally optimal choice at each step without revisiting previous decisions.
- **Advantages:** Significantly reduces the search space compared to exhaustive methods.
- **Requirement:** Requires a formal proof to guarantee global optimality.
- **Examples:**
    - Dijkstra's algorithm (shortest path)
    - Prim's and Kruskal's algorithms (minimum spanning tree)

Applications Covered:

- Interval scheduling
- Minimizing lateness
- Huffman coding

Interval Scheduling

- Problem: Select a maximum number of non-overlapping time slots (e.g., teachers booking a special video classroom).
- Strategies explored for booking selection:
    - Earliest start time
    - Shortest interval
    - Minimum overlap with other bookings
    - Earliest finish time (optimal greedy choice)

Minimizing Lateness

- Scenario: Scheduling jobs (e.g., 3D printer usage) to minimize the maximum lateness of any job.
- Strategies considered:
    - Sort by increasing job length
    - Prioritize smaller slack times
    - Sort by increasing deadlines (optimal approach)

Huffman Coding

- Goal: Efficient prefix-free encoding to compress data based on symbol frequencies.
- Approach:
    - Recursively combine the two least frequent symbols into a composite symbol.
    - Naïve implementation using linear scans has complexity **O(k^2)** for k symbols.
    - Using a heap reduces complexity to **O(k log k)**.

2. Divide and Conquer

- **Definition:** Break a problem into smaller, disjoint subproblems, solve each recursively, and combine solutions efficiently.
- **Common examples:** Merge sort, Quicksort.
- **Applications covered:**
    - Counting inversions
    - Closest pair of points
    - Integer multiplication
    - Quick select

Counting Inversions

- Measures similarity between two rankings by counting pairs out of order.
- Range: 0 (identical) to n(n-1)/2 (maximally dissimilar).
- Recurrence relation: T(n) = 2T(n/2) + n
- Time complexity: **O(n log n)**

Closest Pair of Points

- Divide points by a vertical line and find closest pairs in each half and across the dividing line.
- Recurrence: T(n) = 2T(n/2) + O(n)
- Overall time complexity: **O(n log n)**

Integer Multiplication

- Traditional method complexity: **$O(n^2)$**
- Naive divide and conquer: T(n) = 4T(n/2) + n = **$O(n^2)$**
- Karatsuba's algorithm improves to: T(n) = 3T(n/2) + n = **O(n^1.59)** (approx.)

Quick Select

- Find the k-th largest element in a sequence.
- Naive approach: Sorting and indexing, **O(n log n)**.
- For median (k = n/2), naive selection is **$O(n^2)$**.
- "Median of medians" algorithm achieves **O(n)** worst-case time.
- Quicksort average complexity: **O(n log n)**.

Recursion Trees

- Visual tool to analyze recursive algorithms' time complexity.
- Root node represents the total work at the original problem size.
- Each node branches into recursive calls on smaller subproblems.
- Recurrence: T(n) = rT(n/c) + f(n), where r = number of recursive calls, c = factor by which input size reduces, f(n) = non-recursive work at each node.

---

3. Dynamic Programming

- **Definition:** Solves problems by combining solutions of overlapping subproblems, exploiting optimal substructure.
- **Approach:** Solve subproblems in topological order to avoid redundant computation.
- **Examples given:** Factorial, Fibonacci series, insertion sort (imbedded in inductive solutions).

Memoization

- Technique to store previously computed results to avoid repeated work in recursive solutions.
- Implemented via a memory table (lookup table).

Grid Paths

- Calculate number of paths from origin (0,0) to (m,n) on a grid with movement restricted to right and up.
- Blocked intersections discard paths passing through them.
- Uses DAG structure and dynamic programming to fill the grid.
- Complexity: **O(mn)** with dynamic programming, **O(m + n)** with memoization.

Common Subwords and Subsequences

- **Longest common subword (LCW):** Find the longest contiguous matching substring between two strings.
- **Longest common subsequence (LCS):** Find the longest sequence of characters appearing in order but not necessarily contiguously.
- Both solved via dynamic programming with an $(m+1) \times (n+1)$ table.
- Time complexity: **O(mn)**.

Edit Distance

- Measures minimum number of edits (insertions, deletions, substitutions) to transform one string into another.
- Uses a dynamic programming table similar to LCS and LCW.
- Time complexity: **O(mn)**.

Matrix Multiplication

- Matrix multiplication is associative but the order of multiplying matrices affects computational cost.
- Task: Find optimal parenthesization to minimize multiplication operations.
- Uses dynamic programming to fill a matrix C with costs of multiplying subchains of matrices.
- Time complexity: **O(n^3)**.

Summary Table of Algorithmic Techniques and Complexities

Algorithm/Problem	Approach	Recurrence/Complexity	Key Insight
Interval Scheduling	Greedy	*Not specified* (greedy choice)	Earliest finish time yields optimal set
Minimizing Lateness	Greedy	*Not specified*	Sort by increasing deadlines
Huffman Coding	Greedy + Heap	O(k^2) naive, O(k log k) heap	Heap reduces complexity
Counting Inversions	Divide & Conquer	T(n)=2T(n/2)+n = O(n log n)	Counts dissimilarity between rankings
Closest Pair of Points	Divide & Conquer	T(n)=2T(n/2)+O(n) = O(n log n)	Divide points, combine solutions
Integer Multiplication	Divide & Conquer	O(n^2) naive, O(n^1.59) Karatsuba	Karatsuba reduces complexity
Quick Select	Divide & Conquer	O(n) with median of medians	Select k-th largest efficiently
Recursion Trees	Analysis method	T(n)=rT(n/c)+f(n)	Visualize recursive work
Grid Paths	Dynamic Prog.	O(mn)	Paths in grid with obstacles

Algorithm/Problem	Approach	Recurrence/Complexity	Key Insight
Longest Common Subword	Dynamic Prog.	O(mn)	Contiguous matching substring
Longest Common Subsequence	Dynamic Prog.	O(mn)	Non-contiguous matching sequence
Edit Distance	Dynamic Prog.	O(mn)	String transformation cost
Matrix Multiplication	Dynamic Prog.	$O(n^3)$	Optimal parenthesization of multiplications

Core Concepts and Insights

- **Greedy algorithms** work by making locally optimal choices, but require proof of correctness for global optimality.
- **Divide and conquer** algorithms solve large problems by breaking them into smaller subproblems, achieving efficiency by combining solutions.
- **Dynamic programming** handles overlapping subproblems with optimal substructure, avoiding redundant computations via memoization or tabulation.
- Recursion trees provide a uniform framework to analyze time complexity of recursive algorithms.
- Efficient data structures such as **heaps** can significantly improve algorithmic performance (e.g., Huffman coding).
- Understanding problem structure (e.g., DAG dependencies) is critical to designing efficient dynamic programming solutions.

Conclusion

This course offers a comprehensive introduction to algorithmic problem-solving with Python, emphasizing three fundamental paradigms: greedy algorithms, divide and conquer, and dynamic programming. By exploring classical problems and their efficient solutions, learners gain practical skills in algorithm design, complexity analysis, and implementation strategies critical for software development and computational research. The material balances theoretical foundations with applications, ensuring a robust understanding of core algorithmic techniques.

Week 10 – 12

1. String Matching

Problem Definition:

- Given a **text string** (t) of length (n) and a **pattern string** (p) of length (m), both drawn from an alphabet (\Sigma), the goal is to find every position (i) in (t) where the substring (t[i : i+m] = p).
- The naive string matching algorithm has a time complexity of (O(nm))

---

2. Key String Matching Algorithms

Algorithm	Key Idea	Complexity	Notes
Boyer-Moore	Compares pattern to text from right to left; uses a dictionary last to shift pattern based on mismatch	Worst case (O(nm))	Efficient in practice; bottleneck is computing last dictionary (O(
Rabin-Karp	Treats strings as numbers; compares hash of pattern and substring blocks	Worst case (O(nm)); Average (O(n + m))	Uses hashing to reduce comparisons; may have spurious hits requiring verification
Automata-based	Constructs an automaton tracking longest prefix matched; string matching done by running automaton on text	(O(n)) for matching; preprocessing (O(m^3 \cdot	\Sigma
Knuth-Morris-Pratt (KMP)	Uses fail() function to precompute prefix matches; automaton-like approach	(O(m + n))	Preprocessing (O(m)); efficient linear-time matching; often faster than Boyer-Moore

---

3. Data Structures: Tries

- A **trie** is a rooted tree structure where:
    - Each node (except root) is labeled by a letter from the alphabet (\Sigma).
    - Children of a node have distinct labels.
    - Maximal paths correspond to words; a special symbol $ denotes word termination.
- Tries provide efficient operations such as:
    - Checking if a word (w) is in a set (S).
    - Checking if (w) is a substring of another string (s).
    - Counting occurrences of (w) as a substring.
    - Finding the longest repeated substring in (s).

---

4. Regular Expressions and Automata

- Regular expressions describe **regular sets** of words and correspond exactly to the languages accepted by finite automata.
- Automata for regular expressions may have multiple paths and accept multiple words.
- There is a **one-to-one correspondence**:
    - For every automaton, a matching regular expression exists.
    - For every regular expression, a corresponding automaton can be constructed.
- String matching can be extended to pattern matching by building an automaton for the pattern.
- Python provides libraries for matching regular expressions, facilitating practical implementation.

---

5. Linear Programming

- **Problem Setup Example:**
  - Producing two products: barfis and halwa.
  - Profit per box: Barfis = Rs. 100, Halwa = Rs. 600.
  - Constraints:
    - Maximum demand for barfis: 200 boxes.
    - Maximum demand for halwa: 300 boxes.
    - Total production capacity: 400 boxes per day.
    - Non-negativity constraints: (b \geq 0, h \geq 0).
- **Objective:**
  - Maximize profit: (100b + 600h).
- **Solution Approach:**
  - Start at any vertex of feasible region.
  - Move to adjacent vertex if it improves objective.
  - Stop when no adjacent vertex improves the objective.
- **Properties:**
  - Feasible region is convex.
  - Problem can be infeasible (empty feasible region) or unbounded.
  - Constraints and objective are linear: (a_1x_1 + a_2x_2 + \dots \leq K), etc.

---

6. Network Flows and Ford-Fulkerson Algorithm

- **Network Flow Setup:**
  - Directed graph (G = (V, E)) with capacities (c_e) on edges.
  - Special nodes: source (s) and sink (t).
  - Flow (f_e) on each edge satisfies (f_e \leq c_e).
  - Conservation: sum of incoming flows equals sum of outgoing flows at all nodes except (s) and (t).
- **Ford-Fulkerson Algorithm:**
  - Initialize flow to zero.
  - Repeatedly find unsaturated path from (s) to (t) and augment flow.
  - Use residual graph to keep track of remaining capacities and reverse edges.
  - BFS can be used to find augmenting paths with fewest edges.
- **Example:**
  - Network with max flow 2 demonstrated.
- **Key insight:**
  - If an initial flow choice is poor, reverse edges allow corrections.

---

7. Reductions

- **Concept:**
  - Problem (A) reduces to problem (B) if input for (A) can be transformed into input for (B), and output of (B) interpreted as output for (A).
- **Implications:**
  - Efficient solutions for (B) imply efficient solutions for (A).
  - Preprocessing and postprocessing must be efficient (typically polynomial time).
- **Examples:**
  - Bipartite matching reduces to max flow.
  - Max flow reduces to linear programming.
- **Intractability:**

    - If (A) is known intractable and reduces to (B), then (B) is also intractable.

---

8. Checking Algorithms

- **Definition:**
    - Given a problem (P), a checking algorithm (C) takes an instance (I) and a solution certificate (S).
    - Outputs “yes” if (S) is a valid solution for (I), “no” otherwise.
- **Example:**
    - Factorization:
        - Input: (N) (a large number).
        - Certificate: ({p, q}) such that (p \times q = N).
        - Checking involves verifying multiplication.
- **Use cases:**
    - Satisfiability, Traveling Salesman, Vertex Cover, Independent Set, etc.

---

9. Computational Complexity: P, NP, and NP-Complete

Class	Definition	Key Characteristics
P	Problems solvable by polynomial-time algorithms (worst-case)	Efficiently solvable
NP	Problems where solutions can be *checked* in polynomial time via a checking algorithm	Solution verification is efficient
NP-Complete	Problems in NP to which every other NP problem can be polynomial-time reduced	Hardest problems in NP; polynomial-time solution for one implies solution for all NP

- **NP-Complete criteria:**
    - Problem (A) is in NP.
    - Every problem in NP reduces to (A) in polynomial time.
- **Significance:**
    - NP-Complete problems are central to understanding computational hardness.

Key Insights

- **String matching** is fundamental with multiple algorithmic approaches balancing preprocessing time and matching efficiency.
- **Tries and automata** are powerful tools for efficient pattern storage and matching, with automata underpinning regular expressions.
- **Linear programming** handles optimization under linear constraints with geometric interpretations.
- **Network flow algorithms** like Ford-Fulkerson enable solving capacity-constrained routing problems.

- **Reductions** unify problems by transforming complex problems into known problems.
- **Checking algorithms and complexity classes** provide a framework for understanding the feasibility and hardness of computational problems.

Summary Table of Algorithms and Their Complexities

Algorithm/Concept	Preprocessing Time	Matching/Execution Time	Notes
Naive String Matching	None	(O(nm))	Simple but inefficient for large inputs
Boyer-Moore	(O(	\Sigma	))
Rabin-Karp	(O(m))	Average (O(n + m))	Uses hashing; possible false positives
Automata-based	(O(m^3 \cdot	\Sigma	))
Knuth-Morris-Pratt	(O(m))	(O(n))	Efficient, linear-time matching
Ford-Fulkerson (Max Flow)	(O(1))	Depends on augmenting paths	Uses residual graphs and BFS
Linear Programming	*Not specified*	*Not specified*	Convex feasible region, vertex search

---

Important