Data Structures and Algorithms

Theory Exam Notes

DSA serves as the foundation for crafting efficient and scalable software solutions. Choosing the right data structure and algorithm can significantly impact the performance, resource utilisation, and maintainability of your code. A deep understanding of DSA empowers developers to:

- **Optimise Performance:** By selecting appropriate data structures, developers can minimise the time and space required for operations like searching, sorting, and inserting data. This is crucial for applications that handle large datasets or require real-time responsiveness.
- **Tackle Complex Problems:** Algorithms provide a systematic approach to solving a wide array of problems,from finding the shortest path in a network to compressing data for storage or transmission.
- **Enhance Scalability:** DSA enables the design of algorithms and data structures that can gracefully handle increasing data volumes and user demands, ensuring that applications remain responsive and efficient as they grow.
- **Excel in Technical Interviews:** DSA is a cornerstone of technical assessments in the software industry. Proficiency in DSA demonstrates a strong grasp of fundamental computer science concepts and problem-solving skills.

Abstract Data Types (ADTs)

ADTs are high-level data structures that encapsulate data and provide a set of well-defined operations for manipulating that data. This abstraction simplifies the design and implementation of complex algorithms by allowing developers to focus on the operations' functionality rather than their low-level implementation details.

- **Stacks:** A LIFO (Last-In, First-Out) structure, ideal for tasks like function call management, expression evaluation, and undo/redo functionality.
- **Queues:** A FIFO (First-In, First-Out) structure, essential for scheduling, breadth-first search algorithms, and managing waiting lines or buffers.
- **Priority Queues:** Elements are prioritized based on a key or value, with higher-priority elements being dequeued first. Used in Dijkstra's algorithm, Huffman coding, and task scheduling.
- **Double-Ended Queues (Deque):** A versatile structure that combines the features of stacks and queues, allowing insertions and deletions at both ends. Useful for implementing undo/redo with multiple levels and certain types of simulations.

Arrays

- **Memory Representation:** Contiguous block of memory.
- **Multi-dimensional Arrays:** Arrays of arrays.
- **Sparse Arrays:** Arrays with mostly empty elements (consider hash tables or linked lists for efficiency).
- **Array Operations:**
    - **Traversal:** Accessing each element sequentially.
    - **Insertion:** Shifting elements to make space (expensive).
    - **Deletion:** Shifting elements to fill the gap (expensive).
    - **Searching:** Linear search (O(n)), binary search (O(log n) if sorted).

Linked Lists

- **Node Structure:** Data field and a pointer (reference) to the next node.
- **Types:**
    - **Singly Linked:** Each node points to the next.
    - **Doubly Linked:** Each node points to both the next and previous.
    - **Circular Linked:** Last node points back to the first.
- **Sentinel Nodes:** Dummy nodes at the beginning/end for simplified code.
- **Operations:**
    - **Insertion:** Updating pointers (O(1) if you have the node reference).
    - **Deletion:** Updating pointers (O(1) if you have the node reference).
    - **Traversal:** Iterating through the list (O(n)).

Stacks

- **Array Implementation:** Push/pop from the end of the array.
- **Linked List Implementation:** Push/pop at the head of the list.
- **Applications:**
    - **Expression Evaluation:** Infix, prefix, postfix notation.
    - **Parenthesis Matching.**
    - **Function Calls:** Maintaining the call stack.

Queues

- **Array Implementation:** Circular buffer to handle wraparound.
- **Linked List Implementation:** Enqueue at the tail, dequeue at the head.
- **Variations:**
    - **Deque (Double-Ended Queue):** Insertion/deletion at both ends.
    - **Priority Queue:** Elements are dequeued based on priority (implemented with heaps).

Trees

Trees are hierarchical data structures where nodes are connected by edges. They come in various forms, each with unique properties and applications:

- **Perfect:** All levels are completely filled.
- **Complete:** All levels except possibly the last are filled, and the last level has all keys as left as possible.
- **Full (Strict):** Every node has either 0 or 2 children.
- **Balanced:** The heights of the left and right subtrees of every node differ by at most 1 (AVL, Red-Black trees)
- **Binary Search Trees (BSTs):** Enable efficient searching, insertion, and deletion of data in sorted order.
    - Inorder traversal gives a sorted list.
    - Operations: Search, insertion, deletion (O(log n) on average for balanced trees, O(n) worst case).
- **Self-Balancing BSTs (AVL and Red-Black Trees):** Guarantee logarithmic time complexity for operations even in the worst case, ensuring consistent performance.
- **B-Trees and B+ Trees:** Designed for managing data on disk (e.g., in databases), optimizing disk access patterns.
- **Tries (Prefix Trees):** Specialized for storing and searching strings, making them useful for autocomplete, spell checking, and IP routing.
- **Heaps:**
    - Max-Heap: Parent is always greater than or equal to its children.
    - Min-Heap: Parent is always less than or equal to its children.
    - Applications: Priority queues, heapsort.

Hashing

Hashing is a technique that uses a hash function to map keys to indices in an array (the hash table). This enables incredibly fast (often constant-time) data retrieval, insertion, and deletion on average. Hash tables are widely used for dictionaries, sets, caches, and database indexing.

Graphs

Graphs consist of vertices (nodes) connected by edges, representing relationships between entities. Graph algorithms like breadth-first search (BFS), depth-first search (DFS), Dijkstra's algorithm, and others find applications in routing,network analysis, social network analysis, and more.

Terminology: Vertex (node), edge, adjacency, path, cycle, connected components.

Representations:

- Adjacency Matrix: 2D array, space O(V^2).
- Adjacency List: Array of linked lists, space O(V+E).

Graph Traversal:

- **BFS:** Level-order traversal, uses a queue.
- **DFS:** Explores as far as possible along each branch before backtracking, uses a stack.

Shortest Path Algorithms:

- Dijkstra's: Single-source shortest path for non-negative weights.
- Bellman-Ford: Single-source shortest path, can handle negative weights.
- Floyd-Warshall: All-pairs shortest paths.

Algorithm Design Paradigms

- **Divide and Conquer:** Break the problem into smaller subproblems, solve them recursively, and combine the solutions (merge sort, quicksort).
- **Dynamic Programming:** Solve overlapping subproblems and store solutions for future use (Fibonacci sequence,knapsack problem).
- **Greedy Algorithms:** Make locally optimal choices at each step (Dijkstra's algorithm, Huffman coding).
- **Backtracking:** Explore all possible solutions by incrementally building a solution, abandoning paths that don't lead to a viable answer (N-Queens problem, Sudoku solver).
- **Branch and Bound:** Systematically explore the solution space, pruning branches that cannot lead to an optimal solution (traveling salesman problem).

Choosing the Right Tools

Selecting the most suitable data structure and algorithm requires careful consideration of the problem's characteristics,the desired time and space complexity, and potential trade-offs. By mastering DSA, you unlock the ability to design elegant and efficient solutions that meet the challenges of modern software development.

Sorting and Searching

1. Comparison-Based Sorting
    - Merge Sort:
        - Divide and conquer: Split the array in half, sort each half recursively, then merge the sorted halves.
        - Time complexity: O(n log n) in all cases (best, average, worst).
        - Space complexity: O(n) due to the auxiliary array for merging.

- Stable sort: Maintains the relative order of equal elements.* Implementation: Recursive approach is common.
- Use cases: External sorting (large datasets that don't fit in memory), linked lists.
- Advantages: Consistent performance, good for linked lists.
- Disadvantages: Not in-place, requires extra memory.
- Key steps in merge:
    1. Divide the array until you have single-element subarrays.
    2. Compare corresponding elements of subarrays, merging them in sorted order.
    3. Repeat the merging process until you have a single sorted array.

- Selection Sort:
    - Repeatedly select the smallest element from the unsorted part and place it at the beginning.
    - Time complexity: O(n^2) in all cases.
    -
    - Implementation: Simple loops for selection and swapping.
    - Use cases: Small datasets, when memory is limited.
    - Advantages: Simple to understand and implement, in-place.
    - Disadvantages: Inefficient for large datasets.
- Insertion Sort:
    - Build the sorted array one item at a time, inserting each element into its correct position in the already sorted part.
    - Time complexity: O(n^2) in the average and worst cases, O(n) in the best case (already sorted).
    - Space complexity: O(1), in-place
    - Implementation: Two loops - outer for iterating through elements, inner for finding the insertion point.
    - Use cases: Small datasets, partially sorted arrays, online algorithms.
    - Advantages: Simple, efficient for small or nearly sorted data, online (can sort as data arrives).
    - Disadvantages: Inefficient for large datasets.
- Quick Sort:
    - Divide and conquer: Choose a pivot, partition the array such that elements smaller than the pivot are before it, and elements larger are after it. Recursively sort the subarrays.
    - Time complexity: Average and best case O(n log n), worst case O(n^2) (rare, can be mitigated with randomized pivot selection).
    - Space complexity: Average O(log n), worst case O(n) for the recursion stack.
    - Unstable sort.
    - Implementation: Recursive with partitioning logic.
    - Use cases: General-purpose sorting, efficient on average.

        - Advantages: Highly efficient on average, in-place (with optimizations).
        - Disadvantages: Worst-case scenario can be problematic, not stable.
        - Key steps in partitioning:
            1. Select a pivot (e.g., last element).
            2. Rearrange elements: smaller than pivot to the left, larger to the right.
            3. Place the pivot in its correct position.
            4. Recursively sort the subarrays on either side of the pivot.
    - Heap Sort:
        - Build a max-heap from the array.
        - Repeatedly extract the maximum element (root of the heap) and place it at the end of the array.
        - Time complexity: O(n log n) in all cases.
        - Space complexity: O(1), in-place sorting.
        - Unstable sort.
        - Implementation: Heapify the array, then repeatedly extract the max and rebuild the heap.
        - Use cases: When you need a guaranteed O(n log n) time complexity even in the worst case.
        - Advantages: Guaranteed O(n log n) performance, in-place.
        - Disadvantages: Not as cache-friendly as quicksort, not stable.

Sorting Algorithm Summary

Algorithm	**Best Case**	**Average Case**	**Worst Case**	**Space Complexity**	**Stable**
Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n)	Yes
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)	No
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)	Yes
Quick Sort	O(n log n)	O(n log n)	O(n^2)	O(log n)	No
Heap Sort	O(n log n)	O(n log n)	O(n log n)	O(1)	No

2. Non-Comparison-Based Sorting (Linear Sorting Algorithms)

- Counting Sort:
    - Suitable for sorting integers within a limited range.
    - Counts the occurrences of each element, then reconstructs the sorted array.

        - Time complexity: O(n + k) where n is the number of elements and k is the range of the input.
        - Space complexity: O(n + k).
    - Radix Sort:
        - Sorts integers digit by digit, starting from the least significant digit.
        - Can use counting sort as a subroutine for each digit.
        - Time complexity: O(d * (n + k)) where d is the number of digits and k is the range of digits.
        - Space complexity: O(n + k).

3. Searching Algorithms

- **Linear Search:**
    - **Concept:** Sequentially checks each element in the list until a match is found or the end is reached.
    - **Time Complexity:**
        1. Best Case: O(1) (target found at the beginning)
        2. Average Case: O(n/2)
        3. Worst Case: O(n) (target found at the end or not present)
    - **Space Complexity:** O(1)
    - **Advantages:**
        1. Simplicity: Easy to understand and implement.
        2. Works on unsorted lists.
    - **Disadvantages:**
        1. Inefficient for large datasets.
        2. Not suitable for frequent searches.
    - **Use Cases:**
        1. Small datasets or lists where the order is unknown.
        2. Situations where simplicity is prioritized over speed.

2. Binary Search:

- **Concept:** Works on sorted lists. Repeatedly divides the search interval in half. If the middle element matches the target, it's found. If the middle element is greater, search the left half; otherwise, search the right half.
- **Time Complexity:**
    - Best Case: O(1) (target found at the middle)
    - Average Case: O(log n)
    - Worst Case: O(log n)
- **Space Complexity:**
    - Iterative: O(1)
    - Recursive: O(log n) (due to call stack)
- **Advantages:**
    - Highly efficient for sorted data.
    - Suitable for frequent searches.
- **Disadvantages:**

    - Requires the list to be sorted beforehand.
    - Not ideal for lists with frequent insertions/deletions (since sorting would be required after each change).
- **Use Cases:**
    - Searching in large, sorted datasets (e.g., dictionaries, phonebooks).
    - Applications where search speed is critical.

3. Jump Search:

- **Concept:** An improvement over linear search for sorted arrays. Instead of searching one by one, it jumps ahead by fixed steps. If the target is greater than the value at the jump index, it performs a linear search between the previous jump and the current one.
- **Time Complexity:**
    - Average Case: O(√n)
    - Worst Case: O(√n)
- **Space Complexity:** O(1)
- **Advantages:**
    - Faster than linear search in most cases (especially for very large arrays).
- **Disadvantages:**
    - Still less efficient than binary search.
    - Requires the array to be sorted.
- **Use Cases:**
    - Searching in large, sorted arrays where jumping can save significant time compared to linear search.
    - Situations where a moderate improvement over linear search is sufficient.

4. Interpolation Search:

- **Concept:** An improvement over binary search for certain types of data. It assumes a uniform distribution of elements and estimates the position of the target based on its value. The search interval is then narrowed down accordingly.
- **Time Complexity:**
    - Average Case: O(log log n) (if elements are uniformly distributed)
    - Worst Case: O(n) (if elements are not uniformly distributed)
- **Space Complexity:** O(1)
- **Advantages:**
    - Can be faster than binary search for specific distributions.
- **Disadvantages:**
    - Requires knowledge about the distribution of data.
    - Can be slower than binary search in the worst case.
- **Use Cases:**
    - Searching in sorted arrays where the values are uniformly distributed (e.g., searching for a phone number in a directory).

5. Hashing:

- **Concept:** Stores key-value pairs in a hash table using a hash function. The hash function calculates an index (hash code) for each key, determining its position in the table.
- **Time Complexity:**
    - Average Case: O(1) (for search, insert, and delete)
    - Worst Case: O(n) (if many collisions occur)
- **Space Complexity:** O(n)
- **Advantages:**
    - Extremely efficient for search, insert, and delete operations on average.
- **Disadvantages:**
    - Requires a good hash function to minimize collisions.
    - Not suitable for range searches (e.g., finding all values between two keys).
- **Use Cases:**
    - Databases for fast record retrieval.
    - Symbol tables in compilers.
    - Caching for storing frequently accessed data.
    - Set implementations (to check if an element exists).
    -
    - **Binary Search:** Works on sorted arrays, repeatedly halves the search interval (O(log n)).
    - **Jump Search:** An improvement over linear search for sorted arrays.
    - **Interpolation Search:** An improvement over binary search for certain types of data.
    - **Hashing:** Stores elements in a hash table for constant-time average lookup (potential collisions need to be handled).

Abstract Data Types (ADTs)

1. Stack:
    - LIFO (Last-In, First-Out) structure.
    - Operations: `push(item)`, `pop()`, `peek()`, `isEmpty()`.
    - Implementation: Array or linked list.
    - Applications: Function call stack, undo/redo, expression evaluation, backtracking, browser history.
2. Queue:
    - FIFO (First-In, First-Out) structure.
    - Operations: `enqueue(item)`, `dequeue()`, `front()`, `isEmpty()`.
    - Implementation: Array (with circular buffer for efficiency) or linked list.

- Applications: Scheduling, breadth-first search, message queues, print queues.

3. Priority Queue:
   - Elements have priority, dequeued in order of priority.
   - Operations: `enqueue(item, priority)`, `dequeue()`, `front()`, `isEmpty()`.
   - Implementation: Heap (usually a max-heap).
   - Applications: Dijkstra's algorithm, Huffman coding, event-driven simulations, task scheduling.
4. Double-Ended Queue (Deque):
   - Operations allowed at both ends: `push_front(item)`, `push_back(item)`, `pop_front()`, `pop_back()`, `front()`, `back()`, `isEmpty()`.
   - Implementation: Doubly linked list or dynamic array.
   - Applications: Supporting both stack and queue operations, text editors, undo/redo with multiple levels.

Trees

1. Binary Trees:
   - Each node has at most two children.
   - Types:
     - Full Binary Tree: Every node has either 0 or 2 children.
     - Complete Binary Tree: All levels are filled except possibly the last, which is filled from left to right.
     - Perfect Binary Tree: All levels are completely filled.
   - Properties:
     - Maximum nodes at level 'l': 2^l
     - Maximum nodes in a tree of height 'h': 2^(h+1) - 1
     - For a tree with 'n' nodes, minimum possible height (or depth) is log2(n+1) - 1
2. Binary Search Trees (BSTs):
   - Left subtree nodes are smaller than the root.
   - Right subtree nodes are greater than the root.
   - Inorder traversal gives sorted order.
   - Operations: Search (O(h)), Insertion (O(h)), Deletion (O(h)).
   - Average time complexity for operations is O(log n) if the tree is balanced, O(n) in the worst case (skewed tree).
3. Self-Balancing BSTs (AVL and Red-Black Trees):
   - AVL Trees:
     - Maintain a balance factor (difference in height between left and right subtrees) of -1, 0, or 1 for every node.
     - Use rotations (single or double) to rebalance after insertions/deletions.

    - Red-Black Trees:
        - Each node is colored red or black.
        - Rules maintain the tree's balance (e.g., the root is always black, no two consecutive red nodes).
        - Use rotations and recoloring to maintain balance.
4. B-Trees and B+ Trees:
    - Used for storing data in disk/secondary storage (e.g., databases).
    - Each node can have multiple keys and children.
        - B-Trees: Store keys and data in internal nodes.
        - B+ Trees: Store data only in leaf nodes, linked sequentially for efficient range queries.
5. Tries (Prefix Trees):
    - Efficient for storing and searching strings.
    - Each node represents a prefix.
    - Children of a node represent the next character in the prefix.
    - Used for autocomplete, spell checking, IP routing.

Hash Tables (Hash Maps)

- Core Idea:
    - A hash table uses a hash function to compute an index (a hash code) into an array of buckets or slots, from which the desired value can be found.
    - This provides fast O(1) lookup, insert, and delete operations on average.
- Hash Functions:
    - A hash function maps keys to integer indices.
    - A good hash function should distribute keys uniformly across the hash table to minimize collisions.
    - Common hash functions:
        - Division method: `hash(key) = key % table_size`
        - Multiplication method: `hash(key) = floor(table_size * (key * A - floor(key * A)))` where A is a constant between 0 and 1.
- Collision Resolution:
    - Separate Chaining:
        - Each bucket contains a linked list of key-value pairs that hash to the same index.
        - Lookup involves traversing the linked list.
    - Open Addressing:
        - When a collision occurs, probe other slots in the table until an empty slot is found or the entire table has been searched.
        - Probing methods: linear probing, quadratic probing, double hashing.

- Hash Table Operations:
  - `insert(key, value)`: Hash the key, insert into the bucket (chaining) or find an empty slot (open addressing).
  - `get(key)`: Hash the key and search the corresponding bucket or slots.
  - `remove(key)`: Hash the key, find the element, and remove it.
- Load Factor:
  - `load_factor = number_of_items / table_size`
  - A high load factor (close to 1) increases the likelihood of collisions and degrades performance.
  - Rehashing (increasing the table size and redistributing elements) is typically performed when the load factor exceeds a certain threshold.
- Applications:
  - Dictionaries/Maps
  - Sets
  - Caches
  - Databases (indexing)
  - Cryptography (password storage)

Graphs

1. Graph Representation:
   - Adjacency List

     An array of lists, where each index represents a vertex and the list at that index stores its neighbors. Efficient for sparse graphs (few edges).

   - Adjacency Matrix

     A 2D array, where `matrix[i][j]` is 1 if there's an edge from vertex i to j, and 0 otherwise. Efficient for dense graphs (many edges) and quick edge lookups.

2. Graph Traversal:
   - Breadth-First Search (BFS):
     - Explores the graph level by level.
     - Uses a queue to keep track of the next vertices to visit.
     - Applications: Shortest paths in unweighted graphs, social network analysis.
   - Depth-First Search (DFS):
     - Explores as far as possible along each branch before backtracking.
     - Uses a stack (implicit in recursion) or explicit stack.
     - Applications: Topological sorting, cycle detection, finding connected components.

3. Minimum Spanning Trees (MST):
    - Prim's Algorithm:
        - Greedy algorithm.
        - Starts from an arbitrary vertex and at each step adds the cheapest edge that connects the growing tree to a new vertex.
    - Kruskal's Algorithm:
        - Greedy algorithm.
        - Sorts edges by weight and adds them to the MST as long as they don't create a cycle (uses Union-Find data structure).
4. Shortest Paths:
    - Dijkstra's Algorithm:
        - Finds shortest paths from a single source vertex to all other vertices in a weighted graph (non-negative edge weights).
        - Greedy algorithm using a priority queue.
    - Bellman-Ford Algorithm:
        - Works with negative edge weights (but detects negative cycles).
        - Dynamic programming approach with relaxation of edges.
    - Floyd-Warshall Algorithm**:**
5. Advanced Graph Algorithms:
    - Topological Sorting:
        - Orders vertices in a directed acyclic graph (DAG) such that for every edge (u, v), vertex u comes before vertex v in the ordering.
        - Used in scheduling, dependency resolution.
    - Strongly Connected Components (SCC):
        - Identifies groups of vertices in a directed graph where every vertex in a group is reachable from every other vertex in the same group.

Advanced Graph Algorithms

1. Topological Sorting
    - **Definition:** A linear ordering of vertices in a directed acyclic graph (DAG) such that for every directed edge (u, v), vertex u comes before vertex v in the ordering.
    - **Algorithms:**
        - Kahn's Algorithm:
            - Start with vertices having no incoming edges (indegree 0).
            - Add them to the sorted list and remove their outgoing edges.
            - Repeat until all vertices are processed.
            - Time complexity: O(V + E) where V is the number of vertices and E is the number of edges.
        - DFS-Based Algorithm:
            - Perform DFS on each unvisited node.

        - Upon finishing a node's exploration (no more neighbors), push it onto a stack.
        - Reverse the order of elements in the stack to get the topological order.
        - Time complexity: O(V + E).
    - **Applications:**
        - **Scheduling Tasks:** When tasks have dependencies, topological sorting gives a valid order of execution.
        - **Course Prerequisites:** Determining a valid order for taking courses with prerequisites.
        - **Build Systems:** Determining the order to compile files based on their dependencies.
        - **Deadlock Detection:** Identifying cycles in resource allocation graphs.

2. Strongly Connected Components (SCCs)
    - **Definition:** A strongly connected component (SCC) in a directed graph is a maximal set of vertices where there's a path between any two vertices in the set. In other words, you can reach any vertex in the SCC from any other vertex within the same SCC.
    - Kosaraju's Algorithm:
        - Perform DFS on the original graph, keeping track of finishing times of vertices.
        - Transpose the graph (reverse the direction of all edges).
        - Perform DFS on the transposed graph, starting with vertices in decreasing order of finishing times from step 1.
        - Each DFS tree in step 3 forms an SCC.
        - Time complexity: O(V + E).
    - Tarjan's Algorithm:
        - Based on a single DFS traversal.
        - Uses a stack to keep track of potential SCC members.
        - Assigns a unique "low-link" value to each vertex, which helps identify SCCs during backtracking.
        - Time complexity: O(V + E).
    - **Applications:**
        - **Social Network Analysis:** Finding communities in social networks.
        - **Web Page Ranking:** Identifying clusters of related web pages.
        - **Compiler Optimization:** Analyzing dependencies in code.
        - **Graph Condensation:** Simplifying a graph by collapsing each SCC into a single supernode.

3. Eulerian Path and Circuit:
    - A path (or circuit) that visits every edge of a graph exactly once.
    - Existence conditions and algorithms to find them (Hierholzer's algorithm).

4. Bipartite Graphs and Matching:
    - **Bipartite graphs:** Graphs whose vertices can be divided into two disjoint sets such that every edge connects a vertex from one set to the other.
    - **Matching:** A set of edges that do not share any common vertices.
    - **Applications**: Assignment problems, job scheduling.
5. Network Flow Algorithms:
    - Maximum flow: Finding the maximum amount of flow that can be sent from a source to a sink in a flow network.
    - Minimum cut: Finding a cut (partition of vertices) that minimizes the total capacity of edges crossing the cut.

Advanced Data Structures

1. Disjoint Set Union (Union-Find)

- **Purpose:** Efficiently maintains a collection of disjoint sets (non-overlapping groups of elements).
- **Operations:**
    - `MakeSet(x)`: Creates a new set containing the single element x.
    - `Find(x)`: Returns the representative (parent) of the set containing x.
    - `Union(x, y)`: Merges the sets containing x and y.
- **Optimization:**
    - **Path Compression:** During `Find`, make all nodes on the path directly point to the root.
    - **Union by Rank/Size:** Attach the smaller set to the root of the larger set during `Union`.
- **Applications:**
    - Kruskal's Algorithm for MSTs.
    - Detecting cycles in graphs.
    - Connected component analysis.

2. Segment Tree:

- **Purpose:** Efficiently answer range queries (e.g., sum, min, max) on an array.
- **Structure:** A binary tree where each node stores the result of a query on a segment of the array.
- **Operations:**
    - `Build(array)`: Constructs the segment tree from the given array.
    - `Query(range)`: Returns the result of the query for the specified range.
    - `Update(index, value)`: Updates the value at the given index and propagates the changes up the tree.
- **Applications:**
    - Range sum queries (e.g., find the sum of elements in a subarray).
    - Range minimum/maximum queries.

        - Finding the kth smallest element in a range.

3. Fenwick Tree (Binary Indexed Tree):

- **Purpose:** Similar to segment trees, used for efficient range queries and updates on an array.
- **Structure:** A tree-like structure based on binary representation of indices.
- **Operations:**
    - `Update(index, value)`: Adds the given value to the element at the given index and propagates the changes.
    - `Query(index)`: Returns the cumulative sum of elements from index 1 to the given index.
    - **Range Query Optimization:**
        - Use the prefix sums calculated by `Query` to calculate the sum of a given range.
        - Sum(left, right) = Query(right) - Query(left - 1)
- **Advantages over Segment Trees:**
    - Simpler implementation.
    - Less memory overhead.

Advanced Algorithms

1. Dynamic Programming (DP)

- **Core Idea:** Break down a complex problem into smaller overlapping subproblems and store their solutions to avoid redundant calculations.
- **Key Concepts:**
    - **Optimal Substructure:** The optimal solution to a problem can be constructed from optimal solutions to its subproblems.
    - **Overlapping Subproblems:** The same subproblems are solved multiple times.
- **Steps:**
    - Define subproblems.
    - Write a recurrence relation for the solution of each subproblem in terms of its smaller subproblems.
    - Build a DP table to store the solutions of subproblems.
    - Construct the final solution from the DP table.
- **Applications:**
    - Longest Common Subsequence (LCS)
    - Knapsack Problem
    - Edit Distance
    - Matrix Chain Multiplication
    - Coin Change

2. Greedy Algorithms

- **Core Idea:** Make locally optimal choices at each step with the hope of finding a global optimum.
- **Key Characteristics:**
  - **Greedy Choice Property:** The globally optimal solution can be reached by making locally optimal choices.
  - **Optimal Substructure:** Similar to DP, the optimal solution contains optimal solutions to subproblems.
- **Applications:**
  - Dijkstra's Algorithm
  - Kruskal's Algorithm
  - Prim's Algorithm
  - Huffman Coding
  - Fractional Knapsack Problem
  - Activity Selection Problem

IMPORTANT TOPICS:

1. Merge Sort

Merge Sort is a highly efficient comparison-based sorting algorithm that follows the divide-and-conquer paradigm.

Key Points:

- **Divide:** Recursively divide the unsorted list into smaller sublists until you have single-element lists.
- **Conquer:** Repeatedly merge these single-element lists back into sorted pairs, then sorted quads, and so on, until you have a single sorted list.
- **Time Complexity:** O(n log n) in all cases (best, average, worst). This makes it a very efficient algorithm for large datasets.
- **Space Complexity:** O(n), as it requires additional memory for merging.
- **Stable:** Maintains the relative order of equal elements.
- **Applications:** External sorting (large datasets that don't fit in memory), linked lists.

2. Selection Sort

Selection sort is a simple in-place comparison sorting algorithm that works by repeatedly finding the minimum element from the unsorted part of the list and putting it at the beginning.

Key Points:

- **Find the Minimum:** In each iteration, find the index of the minimum element in the unsorted portion of the array.
- **Swap:** Swap the found minimum element with the first element of the unsorted part.
- **Time Complexity:** O(n^2), which makes it inefficient for large lists.
- **Space Complexity:** O(1), as it works in place.
- **Unstable:** May change the relative order of equal elements.

3. Binary Search Trees (BSTs)

A binary search tree (BST) is a binary tree where the left child of a node contains only nodes with keys lesser than the node's key, and the right child of a node contains only nodes with keys greater than the node's key.

Key Points:

- **Efficient Search:** Allows for efficient searching, insertion, and deletion of items.
- **Time Complexity:** O(log n) for search, insertion, and deletion operations in a balanced BST. However, it can degrade to O(n) in the worst case (skewed tree).
- **Inorder Traversal:** Traversing a BST in inorder gives you the elements in sorted order.
- **Applications:** Symbol tables in compilers, dictionaries, priority queues, and many other applications.
- **Drawbacks:** Performance can degrade significantly if the tree becomes unbalanced.

4. Sorting

Sorting is the process of arranging a collection of items in a specific order (ascending, descending, etc.). It's a fundamental operation in computer science with numerous applications.

Key Points:

- **Importance:** Sorting is crucial for searching, data analysis, efficient algorithm design, and many real-world tasks.
- **Comparison-Based Sorting:** These algorithms compare elements to determine their order. Examples include Merge Sort, Selection Sort, Insertion Sort, Quick Sort, and Heap Sort. Their lower bound time complexity is O(n log n).
- **Non-Comparison-Based Sorting:** These algorithms exploit specific properties of the data to sort more efficiently than comparison-based methods. Examples include Counting Sort, Radix Sort, and Bucket Sort.
- **Choosing the Right Algorithm:** The best sorting algorithm depends on the characteristics of the data, the size of the input, and the desired balance between time and space complexity.

5. Searching

Searching is the process of finding a specific item (target) within a collection of data.

Key Points:

- **Linear Search (Sequential Search):** Iterates through the list until the target is found or the end is reached. Time complexity: O(n).
- **Binary Search:** Works on sorted lists. Repeatedly divides the search interval in half, eliminating half of the remaining elements with each comparison. Time complexity: O(log n).
- **Hash Tables:** Allow for efficient average-case O(1) lookup, insertion, and deletion based on hashing the key.
- **Choosing the Right Method:** The best searching method depends on whether the data is sorted, the frequency of searches, and the need for dynamic updates (insertions and deletions).

6. Height Balanced AVL Trees

AVL trees are self-balancing binary search trees (BSTs) that maintain their balance by ensuring that the height difference between the left and right subtrees of any node is at most 1.

Key Points:

- **Balance Factor:** The balance factor of a node is the height of its right subtree minus the height of its left subtree. It must be -1, 0, or 1 for every node in an AVL tree.
- **Rotations:** AVL trees use single or double rotations to restore balance after insertions or deletions. These rotations are local operations that rearrange nodes while preserving the binary search tree property.
- **Time Complexity:** O(log n) for search, insertion, and deletion operations due to the balanced nature of the tree.This ensures consistent performance even in the worst case.
- **Applications:** AVL trees are used in scenarios where frequent insertions and deletions are required while maintaining efficient search operations.

7. Red-Black Trees

Red-black trees are another type of self-balancing binary search tree. They use a coloring scheme (nodes are either red or black) and a set of rules to ensure that the tree remains roughly balanced.

Key Points:

- **Coloring Rules:**
    - The root is always black.
    - A red node cannot have a red child.
    - All paths from a node to any of its descendant NIL nodes contain the same number of black nodes.
- **Rotations and Recoloring:** Similar to AVL trees, red-black trees use rotations and recoloring to maintain balance.

- **Time Complexity:** O(log n) for search, insertion, and deletion operations due to the balanced nature of the tree.
- **Applications:** Red-black trees are widely used in the implementation of various data structures like associative arrays (maps), sets, and in certain scheduling algorithms.

8. B-Trees

B-trees are self-balancing search trees designed to work efficiently with block-oriented storage, such as disks. They are commonly used in databases and file systems.

Key Points:

- **Multi-key Nodes:** Each node in a B-tree can store multiple keys and children, allowing for a lower tree height and fewer disk accesses.
- **Balanced Structure:** B-trees maintain balance by ensuring that all leaf nodes are at the same level and that each internal node (except the root) has at least a minimum number of keys and children.
- **Search, Insertion, Deletion:** B-trees support efficient search, insertion, and deletion operations with logarithmic time complexity in the average and worst cases (O(log n)).
- **Applications:** Primarily used in databases and file systems to index data for fast retrieval.
- **Advantages:** Efficient disk I/O operations, suitable for large datasets.

9. B+ Trees

B+ trees are a variation of B-trees optimised for sequential access. They differ from B-trees in that data is stored only in leaf nodes, and the leaf nodes are linked together in a sequential list.

Key Points:

- **Data in Leaf Nodes:** All data (key-value pairs) are stored in the leaf nodes of a B+ tree. Internal nodes only store keys used for searching.
- **Linked Leaf Nodes:** The leaf nodes are linked together in a doubly linked list, making sequential access and range queries efficient.
- **Search, Insertion, Deletion:** Similar to B-trees, B+ trees support efficient operations with logarithmic time complexity.
- **Applications:** Commonly used in databases for indexing and maintaining sorted data.
- **Advantages:** Efficient range queries, better space utilization compared to B-trees.

10. Prim's Algorithm (for Minimum Spanning Trees)

A Minumum Spanning Tree is a subset of the edges of the graph that connects all the vertices without any cycles and with the minimum possible total edge weight.

Prim's algorithm is a greedy algorithm used to find a minimum spanning tree (MST) in a connected, weighted, undirected graph.

Key Points:

- **Greedy Choice:** At each step, Prim's algorithm selects the edge with the minimum weight that connects a vertex in the MST to a vertex outside the MST.
- **Implementation:** Often implemented using a **priority queue** to efficiently select the minimum-weight edge.
- **Time Complexity:** O(E log V) using a binary heap or Fibonacci heap as the priority queue, where E is the number of edges and V is the number of vertices.
- **Applications:** Network design, clustering, approximation algorithms for NP-hard problems.

11. Kruskal's Algorithm (for Minimum Spanning Trees)

Kruskal's algorithm is another greedy algorithm for finding the minimum spanning tree (MST) in a connected,weighted, undirected graph.

Key Points:

- **Sorting:** Sorts all the edges in non-decreasing order of their weights.
- **Union-Find:** Uses the **Union-Find** data structure to track connected components.
- **Adding Edges:** Iterates through the sorted edges and adds an edge to the MST if it doesn't create a cycle (i.e., if it connects two different components).
- **Time Complexity:** O(E log E) due to the sorting of edges, which dominates the time complexity of the Union-Find operations.

12. Floyd-Warshall Algorithm

The Floyd-Warshall algorithm is a dynamic programming algorithm used to find the shortest paths between all pairs of vertices in a weighted graph. It can handle both positive and negative edge weights, but it cannot handle negative cycles.

Key Points:

- **Dynamic Programming:** The algorithm uses a table to store the distances between vertices. It gradually builds up the solution by considering intermediate vertices.
- **Intermediate Vertices:** The core idea is to find the shortest path from vertex `i` to vertex `j` by considering all possible intermediate vertices `k`.
- **Recurrence Relation:** The distance from `i` to `j` through intermediate vertex `k` is the minimum of the direct path from `i` to `j` and the path from `i` to `k` followed by the path from `k` to `j`.
- **Time Complexity:** O(V^3), where V is the number of vertices.

- **Applications:**
    - Finding the shortest routes in networks.
    - Finding the transitive closure of a relation.
    - All-pairs shortest paths in graphs with negative weights (but no negative cycles).

13. Huffman Coding

Huffman coding is a lossless data compression algorithm that assigns variable-length codes to different characters based on their frequencies in the input data. More frequent characters get shorter codes, while less frequent characters get longer codes.

Key Points:

- **Prefix Codes:** Huffman codes are prefix codes, meaning that no code is a prefix of another code. This ensures that decoding is unambiguous.
- **Huffman Tree:** The algorithm constructs a binary tree (Huffman tree) from the input characters and their frequencies. The leaves of the tree represent the characters, and the path from the root to a leaf defines the character's code.
- **Greedy Algorithm:** The algorithm builds the tree greedily by repeatedly combining the two nodes with the lowest frequencies.
- **Time Complexity:** O(n log n), where n is the number of unique characters in the input.
- **Applications:** Widely used for text and image compression.

14. Dijkstra's Algorithm

Dijkstra's algorithm is a greedy algorithm that finds the shortest path from a single source vertex to all other vertices in a weighted graph with non-negative edge weights.

Key Points:

- **Priority Queue:** The algorithm maintains a priority queue to store vertices and their tentative distances from the source.
- **Relaxation:** It iteratively selects the vertex with the shortest tentative distance from the priority queue and "relaxes" its neighbours by updating their tentative distances if a shorter path is found.
- **Time Complexity:** O((E + V) log V) using a binary heap implementation for the priority queue, where E is the number of edges and V is the number of vertices.
- **Applications:** Routing protocols, network traffic optimization, finding the shortest path in GPS navigation systems.

15. Knapsack Problem

The knapsack problem is a classic optimization problem where you have a knapsack with a weight capacity and a set of items, each with a weight and a value. The goal is to determine the most valuable combination of items to include in the knapsack without exceeding its weight limit.

Key Points:

- **Types:**
  - **0/1 Knapsack:** Each item can be included either fully or not at all.
  - **Fractional Knapsack:** You can take fractions of items.
- **Approaches:**
  - **Greedy Algorithm:** For the fractional knapsack problem, you can sort items by value-to-weight ratio and greedily take items until the knapsack is full. This doesn't work for the 0/1 knapsack.
  - **Dynamic Programming:** This is the most common approach for the 0/1 knapsack problem. You build a table to store the maximum value achievable for each weight capacity and each subset of items.
- **Time Complexity:**
  - **Greedy:** O(n log n) for sorting.
  - **Dynamic Programming:** O(nW), where n is the number of items and W is the knapsack capacity.
- **Applications:** Resource allocation, investment portfolio optimization, cutting stock problems.

16. Branch and Bound (for Traveling Salesman Problem)

The Traveling Salesman Problem (TSP) is a famous NP-hard problem where you need to find the shortest possible route that visits a set of cities and returns to the origin city. Branch and Bound is a technique used to solve optimization problems, including TSP.

Key Points:

- **Divide and Conquer:** The algorithm systematically explores the solution space by branching (splitting the problem into smaller subproblems) and bounding (calculating lower bounds on the cost of solutions in each subproblem).
- **Bounding Function:** A good bounding function is crucial for the algorithm's efficiency. It helps prune the search space by discarding subproblems that cannot lead to an optimal solution.
- **Time Complexity:** Exponential in the worst case, but often performs much better in practice due to pruning.
- **Applications:** Logistics, planning, and various optimization problems.

17. Graph Coloring

Graph coloring is the assignment of colors to the vertices of a graph so that no two adjacent vertices share the same color.

Key Points:

- **Chromatic Number:** The minimum number of colors needed to color a graph is called its chromatic number.
- **Algorithms:** Greedy coloring, backtracking algorithms.
- **Applications:** Scheduling, register allocation in compilers, Sudoku solving, map coloring.

18. Hamiltonian Cycle

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

Key Points:

- **NP-Complete:** Finding a Hamiltonian cycle is an NP-complete problem, meaning there is no known efficient algorithm for solving it in the general case.
- **Algorithms:** Backtracking algorithms, dynamic programming (Held-Karp algorithm).
- **Applications:** Circuit design, DNA sequencing, planning and scheduling problems.

19. Longest Common Subsequence (LCS)

The LCS problem involves finding the longest subsequence common to two sequences. It is not necessarily a contiguous substring.

Key Points:

- **Dynamic Programming:** The LCS problem is typically solved using dynamic programming. You build a table to store the lengths of the LCS for different prefixes of the input sequences.
- **Time Complexity:** O(mn), where m and n are the lengths of the two sequences.
- **Applications:** DNA sequence alignment, version control systems (diffing), plagiarism detection.

20. Matrix Chain Multiplication

The matrix chain multiplication problem aims to find the most efficient way to multiply a sequence of matrices. The order in which you multiply matrices significantly affects the number of operations required.

Key Points:

- **Dynamic Programming:** The problem is solved using dynamic programming. You create a table to store the minimum number of scalar multiplications needed to compute each subchain of matrices.
- **Time Complexity:** O(n^3), where n is the number of matrices in the chain.
- **Applications:** Compiler optimization, computer graphics, scientific computing.