Data Structures

A **Data Structure** is a specialized format for **organizing, processing, retrieving, and storing data**. Think of it as a container designed to hold data in a way that makes it efficient to work with. The choice of data structure depends on the problem you're trying to solve.

- **Analogy**: Imagine your clothes. You could just throw them all in a big pile (like an unorganized array). Or, you could use a closet with hangers, shelves, and drawers (like different data structures). Using the closet makes it much faster to find the specific shirt you want to wear.
- **Examples**:
    - **Array**: A list of items stored at contiguous memory locations. Great for fast access if you know the index.
    - **Linked List**: A series of connected nodes, where each node points to the next. Great for easy insertion and deletion.
    - **Stack**: A Last-In, First-Out (LIFO) structure. Think of a stack of plates.
    - **Queue**: A First-In, First-Out (FIFO) structure. Think of a line at a ticket counter.
    - **Tree**: A hierarchical structure with a root node and child nodes.
    - **Graph**: A collection of nodes (vertices) and edges that connect them.

Time and Space Complexity

Time Complexity and **Space Complexity** are ways to measure the performance of an algorithm. They don't measure the *exact* time or memory in seconds or bytes, but rather how the resource requirements **grow as the input size (n) increases**.

- **Time Complexity**: Measures how the execution time of an algorithm scales with the input size. The goal is to have an algorithm that remains fast even when the input is very large.
- **Space Complexity**: Measures how the amount of memory (or space) an algorithm requires scales with the input size. This includes both the space for the input and any auxiliary space the algorithm uses.

Asymptotic Notations: Big O, Big Theta (Θ), Big Omega (Ω)

These notations are used to describe the limiting behavior of a function when the argument tends towards a particular value or infinity. In computer science, we use them to describe the performance of algorithms.

Big O Notation (O)

- **What it is**: **Worst-case complexity** or an **upper bound**. It describes the maximum amount of time or space an algorithm will take for an input of size n.
- **Meaning**: The algorithm's performance will be *at most* this bad. It's the most commonly used notation because we are often concerned with the worst-case scenario.
- **Example**: Linear search in an array has a time complexity of O(n). In the worst case, we have to check every single one of the n elements to find our target.

Big Omega Notation (Ω)

- **What it is**: **Best-case complexity** or a **lower bound**. It describes the minimum amount of time or space an algorithm will take for an input of size n.
- **Meaning**: The algorithm's performance will be *at least* this good.
- **Example**: For linear search, the best-case complexity is Ω(1). This happens when the element we're looking for is the very first one in the array.

Big Theta Notation (Θ)

- **What it is**: **Tight bound** or **average-case complexity**. It's used when an algorithm's best-case and worst-case performance are the same.
- **Meaning**: The algorithm's performance is bounded both from above and below by the same function. It gives a precise description of the algorithm's performance.
- **Example**: If you have an algorithm that must iterate through every element of an array, regardless of the data, its time complexity is Θ(n). Its best case and worst case are identical.

Recursion and Iteration

These are two different ways to implement algorithms that involve repetition.

Recursion

Recursion is a process in which a function calls itself directly or indirectly. A recursive function solves a problem by breaking it down into smaller, simpler versions of the same problem until it reaches a point where the solution is known.

- **Key Components**:
    1. **Base Case**: The condition that stops the recursion. Without it, the function would call itself infinitely, leading to a "stack overflow" error.
    2. **Recursive Step**: The part of the function that calls itself with a modified input, bringing it closer to the base case.

Iteration

Iteration is the process of repeating a set of instructions a specific number of times or until a condition is met. This is typically done using loops like `for`, `while`, or `do-while`.

- **Key Components**:
    1. **Initialization**: Setting up a counter or control variable.
    2. **Condition**: The test performed before each iteration. The loop continues as long as the condition is true.
    3. **Update**: Modifying the control variable after each iteration.

Recursion vs. Iteration: Recursion can lead to more elegant and readable code for problems that are naturally recursive (like tree traversals), but it can be less efficient and risks stack overflow. Iteration is often more efficient in terms of memory and can be faster.

Divide and Conquer

Divide and Conquer is a powerful algorithmic paradigm for solving complex problems. It involves three steps:

1. **Divide**: Break the given problem into several smaller sub-problems of the same type.
2. **Conquer**: Solve the sub-problems recursively. If a sub-problem is small enough, solve it directly (this is the base case).
3. **Combine**: Combine the solutions of the sub-problems to get the solution for the original problem.

- **Classic Examples**:
    - **Merge Sort**: Divides the array in half, recursively sorts each half (Conquer), and then merges the two sorted halves (Combine).
    - **Quick Sort**: Picks a 'pivot' element and partitions the array into two sub-arrays, those less than the pivot and those greater than the pivot (Divide). It then recursively sorts the sub-arrays (Conquer). The combining step is trivial as the sorting happens in place.
    - **Binary Search**: Divides the search space in half at each step by checking the middle element.

Here are the explanations for the next set of topics. Let's dive in!

Greedy Algorithms

A **Greedy Algorithm** is an approach for solving optimization problems by making the **locally optimal choice** at each stage with the hope of finding a global optimum. In simple terms, you make the best possible choice at the current moment without worrying about the future consequences.

- **How it works**: It builds up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit.
- **Analogy**: Imagine you're giving change for a certain amount. A greedy approach would be to always give the largest possible coin first until you reach the target amount. For standard currency (like 1, 5, 10, 25 cents), this strategy works perfectly. However, it doesn't work for all coin systems.
- **Classic Problems**: Fractional Knapsack, Dijkstra's Shortest Path Algorithm, Kruskal's Minimum Spanning Tree Algorithm.

Dynamic Programming (DP)

Dynamic Programming is a powerful technique for solving complex problems by breaking them down into a collection of simpler, **overlapping subproblems**. It solves each subproblem only once and stores its solution, typically in a table (an array or hash map). When the same subproblem occurs again, it looks up the previously computed solution instead of re-calculating it.

- **Key Properties**:
    1. **Overlapping Subproblems**: The problem can be broken down into subproblems that are reused several times.
    2. **Optimal Substructure**: The optimal solution to the overall problem can be constructed from the optimal solutions of its subproblems.
- **Analogy**: Imagine you're calculating the 10th Fibonacci number. You need the 9th and 8th. To get the 9th, you need the 8th and 7th. You can see that the 8th Fibonacci number is needed twice. DP calculates it once, saves the result, and reuses it.
- **Classic Problems**: 0/1 Knapsack, Fibonacci Sequence, Longest Common Subsequence.

Backtracking

Backtracking is an algorithmic technique for solving problems recursively by trying to build a solution incrementally. It abandons a path ("backtracks") as soon as it determines that the current path cannot possibly lead to a valid solution. It's essentially a refined brute-force approach that systematically explores the set of all possible solutions.

- **How it works**: It explores a path. If the path leads to a solution, great. If it doesn't, or leads to a dead end, it reverses its last step and tries a different option. This process continues until all possible combinations are explored.
- **Analogy**: Solving a maze. You go down one path. If you hit a dead end, you retrace your steps to the last junction and try a different path.

- **Classic Problems**: N-Queens problem, Sudoku solver, generating all permutations of a set.

Sliding Window

The **Sliding Window** technique is used on data structures like arrays or strings to solve problems that involve finding a contiguous sub-part (a sub-array or sub-string) that satisfies a given condition. A "window" of a certain size slides over the data, and we only operate on the elements within that window.

- **How it works**: You maintain a window (defined by a start and end pointer). You expand the window by moving the end pointer and shrink it by moving the start pointer. This avoids re-computing values for overlapping parts of the sub-arrays.
- **Analogy**: Imagine looking at a long train through a small window. Instead of moving your head to see the next part of the train, you just let the train move.
- **Classic Problems**: Maximum sum sub-array of size K, longest substring with K distinct characters.

Two Pointer Technique

The **Two Pointer Technique** is a simple and effective approach where two pointers iterate through a data structure, typically a sorted array, until they meet or cross. This technique helps reduce the time complexity from O(n2) to O(n).

- **Common Patterns**:
    1. **Opposite Direction**: One pointer starts at the beginning, and the other starts at the end. They move towards each other. (e.g., finding a pair that sums to a target).
    2. **Same Direction**: Both pointers start at the beginning but move at different speeds. This is often called the "fast and slow pointer" method (e.g., detecting a cycle in a linked list).

Linear Data Structures

A **linear data structure** is one where elements are arranged in a sequential or linear order. Each element is connected to its previous and next element, like links in a chain. This simple, sequential arrangement makes them easy to implement.

Arrays

An **array** is a collection of items stored at **contiguous memory locations**. This means all the elements are stored side-by-side in memory. It's one of the simplest data structures.

- **Key Features**:
    - **Fixed Size**: The size of an array is fixed when it's created.
    - **Indexed Access**: Elements are accessed using an index (starting from 0).
    - **Fast Access**: Accessing any element by its index is very fast, taking constant time, or O(1).
    - **Slow Insertion/Deletion**: Adding or removing an element in the middle requires shifting all subsequent elements, which is slow (O(n)).

Strings

A **string** is a data structure that represents a sequence of characters. In most programming languages, strings are implemented as an array of characters, often terminated by a special null character (`\0`).

- **Key Features**:
    - They are used to store and manipulate text.
    - Common operations include finding the length, concatenating strings, finding substrings, and replacing characters.
    - In some languages (like Java), strings are **immutable**, meaning they cannot be changed after they are created. In others (like C++), they are mutable.

Linked Lists

A **linked list** is a linear data structure where elements are not stored at contiguous memory locations. Instead, the elements are stored in **nodes**, where each node contains the data and a pointer (or link) to the next node in the sequence.

- **Key Features**:
    - **Dynamic Size**: Can easily grow or shrink at runtime.
    - **Efficient Insertion/Deletion**: Adding or removing a node is very efficient (O(1)) if you have a pointer to the previous node.
    - **No Wasted Memory**: Memory is allocated as needed.
    - **Slow Access**: To access an element, you must traverse the list from the beginning, which takes linear time, or O(n).

Singly Linked List

This is the standard linked list. Each node contains the data and a single pointer that points to the **next** node in the list. The last node's pointer is null, indicating the end of the list.

Doubly Linked List

In a doubly linked list, each node has three parts: the data, a pointer to the **next** node, and a pointer to the **previous** node.

- **Advantage**: It can be traversed in both forward and backward directions, which makes some operations, like deleting a node, more efficient.

Circular Linked List

In a circular linked list, the `next` pointer of the last node points back to the **first node** instead of being null. This forms a circle. A circular list can be either singly or doubly linked.

- **Use Case**: Useful for applications where you need to cycle through a list repeatedly, like round-robin scheduling in operating systems.

Stacks

A **stack** is a linear data structure that follows the **Last-In, First-Out (LIFO)** principle. The last element added to the stack will be the first one to be removed.

- **Analogy**: Think of a stack of plates. You place a new plate on top, and you can only remove the topmost plate. 🍽️
- **Core Operations**:
  - `push`: Adds an element to the top of the stack.
  - `pop`: Removes the element from the top of the stack.
  - `peek` or `top`: Returns the top element without removing it.
- **Applications**: Function call management (the "call stack"), undo/redo features, expression evaluation.

Queues

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

- **Analogy**: Think of people lining up for a ticket. The first person to get in line is the first person to be served. 🚶 🚶 🚶
- **Core Operations**:

- `enqueue`: Adds an element to the rear (back) of the queue.
- `dequeue`: Removes an element from the front of the queue.
- `front`: Returns the front element without removing it.

Simple Queue

This is the basic queue, often implemented using an array. It has two pointers, `front` and `rear`. The issue with a simple array implementation is that after several enqueue and dequeue operations, the queue might appear "full" (when `rear` reaches the end of the array) even if there are empty spots at the beginning.

Circular Queue

A **circular queue** is an improvement on the simple queue that solves the wasted space problem. When the `rear` pointer reaches the end of the array, it wraps around to the beginning, provided there is empty space there. This is achieved using the modulo operator for calculating the indices.

Deque (Double-Ended Queue)

A **deque** (pronounced "deck") is a generalized version of a queue. It allows insertion and deletion of elements from both the **front** and the **rear**. It can be used as both a stack and a queue.

Priority Queue

A **priority queue** is a special type of queue where each element has an associated "priority." Elements with higher priority are dequeued before elements with lower priority. If two elements have the same priority, they are served according to their order in the queue (FIFO).

- **Implementation**: They are commonly implemented using a data structure called a **Heap**.

Of course! Let's wrap up your data structures preparation by covering the non-linear ones.

Non-Linear Data Structures

Unlike linear data structures, **non-linear data structures** don't have elements arranged in a sequence. Instead, elements are organized in a **hierarchical** or **network** manner. This allows for more complex relationships between data items.

Trees

A **tree** is a hierarchical data structure consisting of **nodes** connected by **edges**. It's used to represent hierarchical relationships.

- **Key Terminology**:
  - **Root**: The topmost node of the tree.
  - **Parent**: A node that has child nodes.
  - **Child**: A node that extends from a parent node.
  - **Leaf**: A node with no children.

Binary Tree

A **binary tree** is a type of tree where each node can have at most **two children**: a **left child** and a **right child**.

Binary Search Tree (BST)

A **Binary Search Tree** is a special type of binary tree with a specific ordering property:

1. All nodes in the **left subtree** have values less than the parent node's value.
2. All nodes in the **right subtree** have values greater than the parent node's value.
3. Both the left and right subtrees must also be binary search trees.

- **Advantage**: This property makes searching, insertion, and deletion very efficient, with an average time complexity of O(logn). However, in the worst case (a skewed tree), it can become O(n).

AVL Tree & Red-Black Tree (Self-Balancing BSTs)

These are types of BSTs that automatically keep themselves balanced to avoid the worst-case scenario.

- **AVL Tree**: A height-balancing tree. It ensures that the difference in height between the left and right subtrees of any node (the "balance factor") is at most 1. It maintains this balance using "rotations." It's very strictly balanced.
- **Red-Black Tree**: Uses "color" properties (red or black) for each node to ensure that no path from the root to a leaf is significantly longer than any other. This balancing is less strict than AVL trees, resulting in faster insertions and deletions but slightly slower lookups. They are widely used in practice (e.g., C++ `std::map` and `std::set`).

Heaps

A **heap** is a specialized tree-based data structure that is a **complete binary tree** and satisfies the **heap property**. A complete binary tree is a binary tree that is completely filled, with the possible exception of the bottom level, which is filled from left to right.

Min Heap

In a **Min Heap**, the value of each parent node is **less than or equal to** the value of its children. This means the **smallest element** is always at the root of the tree.

Max Heap

In a **Max Heap**, the value of each parent node is **greater than or equal to** the value of its children. This means the **largest element** is always at the root of the tree.

- **Primary Use**: Heaps are the perfect data structure for implementing **Priority Queues**.

Graphs

A **graph** is a non-linear data structure consisting of a set of **vertices** (or nodes) and a set of **edges** that connect these vertices. Graphs are used to model networks and relationships, like social networks, road maps, or the internet.

Directed vs. Undirected Graph

- **Directed Graph (Digraph)**: The edges have a direction. An edge from A to B doesn't necessarily mean there's an edge from B to A. Think of one-way streets.
- **Undirected Graph**: The edges are bidirectional. An edge between A and B means you can travel in both directions. Think of two-way streets.

Weighted vs. Unweighted Graph

- **Weighted Graph**: Each edge has a numerical "weight" or "cost" associated with it. This can represent distance, time, or cost. (e.g., Google Maps finding the fastest route).
- **Unweighted Graph**: Edges do not have a weight. The goal is often to find the shortest path in terms of the number of edges.

Graph Representations

There are two common ways to represent a graph in memory:

Adjacency Matrix

An **adjacency matrix** is a 2D array of size V×V (where V is the number of vertices). A value `matrix[i][j] = 1` (or the edge weight) indicates there is an edge from vertex `i` to vertex `j`.

- **Pros**: Fast to check if an edge exists between two vertices (O(1)).
- **Cons**: Requires a lot of space (O(V2)), which is inefficient for graphs with few edges ("sparse graphs").

Adjacency List

An **adjacency list** is an array of linked lists. The entry `array[i]` contains a linked list of all the vertices that are adjacent to vertex `i`.

- **Pros**: Space-efficient for sparse graphs (O(V+E), where E is the number of edges).
- **Cons**: Slower to check if an edge exists between two vertices (O(k) where k is the number of neighbors).

Hashing

Hashing is the process of converting an input of arbitrary size (like a string, an object, etc.) into a fixed-size value, typically an integer. This is done using a **hash function**. The output of the hash function is called a **hash code** or simply a **hash**.

- **Key Idea**: A good hash function should be fast to compute and should distribute keys uniformly across the output range. For the same input, it must always produce the same output.
- **Analogy**: Think of a librarian assigning a specific shelf and spot number (a hash code) to every book (the key) based on its title and author. This makes finding the book later much faster than searching the entire library. 📚

Hash Tables, Hash Maps, and Hash Sets

These are data structures that use the hashing principle.

Hash Tables

A **hash table** is the underlying data structure that implements an associative array abstract data type. It uses a hash function to compute an index (a "slot" or "bucket") into an array, from which the desired value can be found.

- **Performance**: On average, hash tables provide constant time complexity, or O(1), for insertion, deletion, and search operations. In the worst case (due to collisions), the complexity can degrade to O(n).

Hash Maps

A **Hash Map** is a practical implementation of a hash table that stores data as **key-value pairs**. For each unique key, you can store a corresponding value. You use the key to look up the value.

- **Example**: Storing user profiles. The key could be the `userID` (e.g., "user123"), and the value could be an object containing the user's name, email, and age.

Hash Sets

A **Hash Set** is another implementation that stores only **unique keys**. It doesn't store values. Its primary purpose is to efficiently check for the presence of an element in a collection.

- **Example**: Keeping track of unique visitors to a website. You can add each visitor's IP address to a hash set. Adding a duplicate IP will have no effect, and you can quickly check if an IP has already been seen.

Collision Handling

A **collision** occurs when two different keys are hashed to the **same index** in the array. Since two elements cannot be stored in the same slot, we need strategies to handle this.

1. Chaining (or Separate Chaining)

In this method, each slot in the hash table array does not hold the element itself but points to a **linked list** of elements. When a collision occurs, the new element is simply added to the linked list at that index.

- **How it works**: To find an element, you first hash the key to find the correct slot (bucket), and then you traverse the linked list in that bucket to find the element.
- **Analogy**: Imagine an apartment building where multiple people (keys) can live at the same address (the hash index). The linked list is like the list of residents for that address.

2. Open Addressing

In this method, all elements are stored directly within the hash table array itself. When a collision occurs, the algorithm **probes** for the next available slot in the table according to a specific rule.

There are three common probing strategies:

- **Linear Probing**: If slot `i` is taken, try `i+1`, then `i+2`, `i+3`, and so on, wrapping around if necessary. This can lead to a problem called "primary clustering," where long chains of occupied slots form, degrading performance.
- **Quadratic Probing**: If slot `i` is taken, try `i+1^2`, then `i+2^2`, `i+3^2`, etc. This helps to break up the clusters formed by linear probing.
- **Double Hashing**: Use a second hash function to determine the step size for probing. If slot `i` is taken, the next slot to check is `i + step`, then `i + 2*step`, etc., where `step` is determined by the second hash function. This is one of the most effective methods for reducing clustering.

B-Tree & B+ Tree

These are self-balancing search trees specifically designed for storage systems where data is read in large blocks (or pages), like hard drives and SSDs.

- **Purpose**: To minimize the number of disk reads, which are very slow compared to memory access. This is the core data structure used in most **databases** and **file systems**.
- **Key Feature**: Nodes in a B-Tree can have a large number of children (a high "fanout"). This makes the tree very wide and shallow, ensuring that finding any element requires traversing only a few levels (and thus, very few disk reads).
- **B+ Tree vs. B-Tree**: In a **B+ Tree**, all the actual data is stored only in the leaf nodes, and these leaf nodes are linked together in a list. This makes range queries and full table scans much more efficient than in a standard B-Tree.

Sorting Algorithms

Sorting algorithms are used to rearrange elements of a list in a specific order (like ascending or descending).

Bubble Sort

- **How it works**: Repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The larger elements "bubble" up to the end of the list.
- **Complexity**: O(n2). It's very slow and mostly used for educational purposes.

Selection Sort

- **How it works**: It divides the list into a sorted and an unsorted part. It then repeatedly finds the smallest element from the unsorted part and swaps it to the end of the sorted part.
- **Complexity**: O(n2).

Insertion Sort

- **How it works**: It builds the final sorted array one item at a time. It iterates through the input elements and inserts each element into its correct position in the sorted part of the array.
- **Analogy**: Like sorting a hand of playing cards. You pick up one card at a time and insert it into its correct place in your hand. 🃏
- **Complexity**: O(n2), but it's very efficient for small or nearly sorted datasets.

Merge Sort

- **How it works**: A "Divide and Conquer" algorithm. It divides the array into two halves, recursively sorts them, and then **merges** the two sorted halves back into one sorted array.
- **Key Features**: It's a **stable** sort (doesn't change the relative order of equal elements) and has a guaranteed time complexity.
- **Complexity**: O(nlogn) in all cases. It requires extra space (O(n)) for the merging process.

Quick Sort

- **How it works**: Another "Divide and Conquer" algorithm. It picks an element as a **pivot** and **partitions** the array around the pivot, placing all smaller elements before it and all larger elements after it. It then recursively sorts the two sub-arrays.
- **Key Features**: It's an **in-place** algorithm (requires minimal extra space). It's often faster in practice than Merge Sort.
- **Complexity**: Average case is O(nlogn), but the worst case is O(n2) (if the pivots are chosen poorly).

Heap Sort

- **How it works**: Uses a **Max Heap** data structure. First, it builds a Max Heap from the input array. Then, it repeatedly swaps the root element (the maximum) with the last element, reduces the heap size, and "heapifies" the root to maintain the heap property.
- **Key Features**: It's an in-place sort.
- **Complexity**: O(nlogn) in all cases.

Counting Sort

- **How it works**: A non-comparison sort. It works by counting the number of occurrences of each distinct element in the input array. This count information is then used to place the elements directly into their correct sorted positions.
- **Constraint**: Only works for integers within a specific, limited range.
- **Complexity**: O(n+k), where n is the number of elements and k is the range of the input.

Radix Sort

- **How it works**: Another non-comparison sort that works on integers. It sorts the numbers digit by digit, starting from the least significant digit to the most significant digit. It uses a stable sort like Counting Sort as a subroutine for sorting each digit.
- **Complexity**: O(d⋅(n+b)), where d is the number of digits, n is the number of elements, and b is the base (e.g., 10 for decimal numbers).

Bucket Sort

- **How it works**: Distributes elements into a number of "buckets." Each bucket is then sorted individually, either using a different sorting algorithm or by recursively applying the bucket sort algorithm. Finally, the sorted buckets are concatenated.
- **Constraint**: Works best when the input data is uniformly distributed.
- **Complexity**: Average case is O(n+k), where k is the number of buckets. Worst case is O(n2).

Shell Sort

- **How it works**: An improvement over Insertion Sort. It allows the comparison and exchange of elements that are far apart. It starts by sorting pairs of elements far apart from each other, then progressively reduces the gap between elements to be compared.
- **Complexity**: Depends on the gap sequence, but it's better than O(n2).

Tim Sort

- **How it works**: A hybrid, stable sorting algorithm derived from Merge Sort and Insertion Sort. It's designed to perform very well on many kinds of real-world data. It finds natural sorted "runs" in the data and merges them.
- **Fun Fact**: This is the standard sorting algorithm used in **Python** and **Java** (for objects).
- **Complexity**: O(nlogn).

Searching Algorithms 🔎

Searching algorithms are used to find a specific element (the "key") within a collection of items.

Linear Search

- **How it works**: The simplest method. It sequentially checks each element of the list until a match is found or the whole list has been searched.
- **Constraint**: Works on **unsorted** arrays.
- **Complexity**: O(n).

Binary Search

- **How it works**: The classic efficient search algorithm. It repeatedly divides the search interval in half. It compares the key with the middle element. If they don't match, the half in which the key cannot lie is eliminated, and the search continues on the remaining half.
- **Analogy**: Like looking up a word in a physical dictionary. You open to the middle, see if your word comes before or after, and then repeat the process on the relevant half. 📖
- **Constraint**: The array **must be sorted**.
- **Complexity**: O(logn).

Graph Representation

First, a quick recap of how graphs are stored.

- **Adjacency Matrix**: A 2D array where `matrix[i][j] = 1` if there's an edge from vertex `i` to `j`. It's fast for checking if an edge exists (O(1)) but uses a lot of space (O(V2)).
- **Adjacency List**: An array of lists, where `list[i]` stores all the vertices adjacent to vertex `i`. It's space-efficient for sparse graphs (O(V+E)) and is the most common representation.

Graph Traversal

DFS (Depth-First Search)

- **How it works**: DFS explores as far as possible along each branch before backtracking. It uses a **stack** (often the implicit function call stack in recursion).
- **Analogy**: Exploring a maze by taking a path to its very end. If it's a dead end, you backtrack to the last junction and try a different path. 🗺️
- **Use Cases**: Cycle detection, topological sorting, finding connected components.

BFS (Breadth-First Search)

- **How it works**: BFS explores all the neighbor nodes at the present "depth" or "level" before moving on to the nodes at the next level. It uses a **queue**.
- **Analogy**: The ripple effect when you drop a stone in water. It explores layer by layer.
- **Use Cases**: Finding the shortest path in an **unweighted** graph, checking for bipartiteness.

Topological Sort

- **What it is**: A linear ordering of vertices in a **Directed Acyclic Graph (DAG)**. For every directed edge from vertex `u` to `v`, vertex `u` must come before `v` in the ordering.
- **Analogy**: Task dependencies. You must put on your socks before your shoes. "Socks" comes before "Shoes" in the topological sort. 🧦 -> 👟
- **Algorithm**: Commonly done using **Kahn's Algorithm**, which uses a queue and tracks the "in-degree" (number of incoming edges) of each node.

Shortest Path Algorithms

Dijkstra’s Algorithm

- **Purpose**: Finds the shortest path from a single source vertex to all other vertices in a **weighted graph with non-negative edge weights**.
- **How it works**: It's a greedy algorithm that maintains a set of visited nodes. In each step, it picks the unvisited node with the smallest known distance from the source and explores its neighbors. It uses a **priority queue** for efficiency.

Bellman-Ford Algorithm

- **Purpose**: Also finds the shortest path from a single source, but it can handle graphs with **negative edge weights**.
- **How it works**: It's slower than Dijkstra's. It relaxes all the edges `V-1` times. A final `V`-th relaxation can be used to detect if the graph contains a **negative-weight cycle**.

Floyd-Warshall Algorithm

- **Purpose**: Finds the shortest paths between **all pairs** of vertices in a weighted graph.
- **How it works**: A dynamic programming algorithm that iteratively considers each vertex `k` and checks if the path from `i` to `j` can be shortened by going through `k`.

Minimum Spanning Tree (MST)

An MST is a subset of the edges of a connected, weighted, undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight.

Prim's Algorithm

- **How it works**: A greedy algorithm that "grows" the MST from an arbitrary starting vertex. At each step, it adds the cheapest possible edge that connects a vertex in the growing MST to a vertex outside of it.

Kruskal's Algorithm

- **How it works**: Another greedy algorithm. It sorts all edges in non-decreasing order of their weight. It then picks the smallest edge and adds it to the MST as long as adding it does not form a cycle. It uses the **Union-Find** data structure to efficiently detect cycles.

Other Key Graph Algorithms

- **Graph Coloring**: Assigning a "color" to each vertex such that no two adjacent vertices have the same color, using the minimum number of colors. This is an NP-hard problem.
- **Bipartite Check**: A graph is bipartite if its vertices can be divided into two disjoint sets such that every edge connects a vertex in one set to a vertex in the other. You can check this with BFS or DFS by trying to "two-color" the graph.
- **Network Flow**: Models problems where you need to find the maximum "flow" of something (like data or goods) from a source to a sink through a network with capacities. The **Ford-Fulkerson** method (often implemented with **Edmonds-Karp** using BFS) is a classic way to solve this.
- **Eulerian Path/Circuit**: An **Eulerian path** visits every **edge** of a graph exactly once. A circuit is a path that starts and ends at the same vertex. Its existence depends on the degrees of the vertices.
- **Hamiltonian Path**: A path that visits every **vertex** of a graph exactly once. Unlike the Eulerian path, there is no known efficient algorithm for finding a Hamiltonian path; it's an NP-complete problem.

Greedy Algorithms

- **Fractional Knapsack**: Given items with weights and values, pack a knapsack with a fixed capacity to maximize the total value. You can take fractions of items.
    - **Greedy Strategy**: Calculate the **value-to-weight ratio** (value/weight) for each item. Greedily pick the items with the highest ratio first until the knapsack is full.

- **Huffman Encoding**: A famous lossless data compression algorithm.
    - **Greedy Strategy**: It builds an optimal prefix-free binary tree (Huffman Tree). The algorithm repeatedly finds the two nodes (characters) with the lowest frequencies and merges them into a new parent node whose frequency is the sum of its children's frequencies. This process continues until only one node (the root) remains.

- **Minimum Number of Coins**: The problem of making change for a specific amount using the fewest possible coins.
    - **Greedy Strategy**: For standard coin systems (like USD, EUR, INR), the strategy is simple: always take the largest denomination coin that is less than or equal to the remaining amount.

- **Job Scheduling**: Given jobs with deadlines and profits, schedule them (each takes one unit of time) to maximize total profit.
    - **Greedy Strategy**: Sort jobs in descending order of their **profit**. For each job, place it in the latest possible available time slot that is still before its deadline.

- **Kruskal's & Prim’s Algorithms**: These algorithms for finding a Minimum Spanning Tree (MST) in a graph are classic examples of the greedy paradigm.
    - **Kruskal's** greedily picks the next cheapest edge that doesn't form a cycle.
    - **Prim's** greedily picks the next cheapest edge that connects the growing MST to a vertex not yet in it.

Divide and Conquer Algorithms

This paradigm solves a problem by breaking it down into smaller, more manageable sub-problems, solving them recursively, and then combining their solutions to get the final answer. It has three steps: **Divide, Conquer, Combine**.

- **Merge Sort, Quick Sort, Binary Search**: These are quintessential Divide and Conquer algorithms that you've already covered under Sorting and Searching. They perfectly illustrate the divide-conquer-combine pattern.

- **Closest Pair of Points**: The problem is to find the two points in a set that are closest to each other.
    - **D&C Approach**:
        1. **Divide**: Divide the set of points by a vertical line into two equal halves.
        2. **Conquer**: Recursively find the closest pair in the left half and the right half. Let this minimum distance be d.
        3. **Combine**: The trickiest step. The closest pair might be a "split pair" (one point on each side of the line). We only need to check points within a narrow strip of width 2d around the dividing line, significantly reducing the number of comparisons.

- **Maximum Subarray Problem**: Find the contiguous subarray with the largest sum.
    - **Kadane's Algorithm**: While this can be solved with D&C, the most famous and efficient (O(n)) solution is Kadane's Algorithm, which uses a dynamic programming approach. It iterates through the array, keeping track of the maximum sum of a subarray ending at the current position and the overall maximum sum found so far.

- **Fast Exponentiation** (Exponentiation by Squaring): A method to calculate xn in O(logn) time.
    - **D&C Approach**:
        - If n is even, xn=(xn/2)2.
        - If n is odd, xn=x⋅(x(n−1)/2)2.
        - The problem is recursively divided in half at each step.

- **Matrix Multiplication**:
    - **Strassen's Algorithm**: A D&C algorithm that multiplies two n×n matrices more efficiently than the standard O(n3) method. It uses a clever trick to perform the multiplication with 7 recursive calls on sub-matrices of size n/2 instead of the usual 8, leading to a complexity of about O(n2.81).

- **Karatsuba Algorithm**: Used for multiplying large integers faster than the classical "grade-school" algorithm (O(n2)).
    - **D&C Approach**: It breaks down the numbers into halves and calculates the product using only three smaller multiplications instead of four, reducing the complexity to about O(n1.58).

Linked Lists vs. Arrays

The main advantages of a **linked list** over an **array** are:

- **Dynamic Size**: Linked lists can grow and shrink dynamically at runtime. Arrays have a fixed size, which must be declared beforehand.
- **Efficient Insertion/Deletion**: Inserting or deleting an element in the middle of a linked list is very fast (O(1)) if you have a pointer to the previous node. For an array, the same operation is slow (O(n)) because all subsequent elements must be shifted.

Real-time Applications of Linked Lists:

- **Web Browsers**: The "previous" and "next" buttons for navigating visited pages.
- **Music Players**: The playlist functionality to go to the next or previous song.
- **Undo Functionality**: In software like text editors, each action is stored in a node, allowing you to easily go back.

Graph Traversal

A **graph** is a non-linear data structure consisting of **nodes** (or vertices) connected by **edges**. It's used to model networks and relationships, like social networks or city road maps.

The two main traversal techniques are:

- **DFS (Depth-First Search)**: Explores a graph by going as deep as possible down a branch before backtracking. It uses a **stack**. Think of it like solving a maze by following one path to its end.
- **BFS (Breadth-First Search)**: Explores the graph layer by layer. It visits all of a node's direct neighbors before moving on to their neighbors. It uses a **queue**. Think of it like the ripple effect from a stone dropped in water.

When to use which:

- **Use BFS when:** You need to find the **shortest path** in an **unweighted** graph, or you want to find all nodes "close" to a starting node.
- **Use DFS when:** You just need to find if a **path exists** between two nodes, you're detecting a **cycle**, or you're doing **topological sorting**.

Merge Sort vs. Quick Sort

Feature	**Merge Sort**	**Quick Sort**
Main Logic	The main work is in **merging** two already sorted sub-arrays.	The main work is in **partitioning** the array around a pivot element.
Time Complexity	**O(nlogn)** in all cases (best, average, worst).	**O(nlogn)** in best/average cases, but **O(n2)** in the worst case (e.g., already sorted array).
Space Complexity	**O(n)** because it requires extra space to create the temporary merged arrays.	**O(logn)** (in-place) as it only uses the recursion call stack space.
Stability	**Stable** (preserves the relative order of equal elements).	**Not Stable**.
When to Prefer?	When a guaranteed worst-case performance and stability are important (e.g., sorting linked lists).	When you want a faster average-case performance and are okay with a rare worst-case scenario.

Export to Sheets

Binary Search Tree (BST) Operations

Insertion

To insert a new value into a BST, you always start at the root:

1. Compare the new value with the current node's value.
2. If the new value is **smaller**, move to the **left child**.

3. If the new value is **larger**, move to the **right child**.
4. Repeat this process until you find a null spot, where you then insert the new node.

Deletion

Deletion is more complex and involves three cases for the node you want to delete:

1. **Case 1: The node is a leaf (no children).** Simply remove it from the tree.
2. **Case 2: The node has one child.** Replace the node with its single child.
3. **Case 3: The node has two children.**
    - Find its **in-order successor** (the smallest value in its right subtree).
    - Copy the value of the successor to the node you want to delete.
    - Delete the successor node (which is a much simpler Case 1 or Case 2 deletion).

Time Complexity

Time Complexity describes how the runtime of an algorithm scales as the size of the input (n) increases. It's not about the exact time in seconds but about the **rate of growth**, expressed using Big O notation.

In the worst-case scenario, the number you're looking for is at the very end of the array, or not in it at all. This means you have to look at **every single element**. If the array has n elements, you perform n comparisons.

- If n=10, you do about 10 operations.
- If n=1,000,000, you do about 1,000,000 operations.

The runtime grows linearly with the input size. Therefore, we say the time complexity of linear search is **O(n)** ("Big O of n").