kintsugi-stack-dsa-cpp: THEORY

> "Data Structures and Algorithms (DSA) should be viewed as essential tools, akin to the finely tuned parts of a Formula 1 car. The act of problem-solving with DSA serves as a crucial platform to exhibit both intelligence and creative thinking. The coding challenges themselves are simply various permutations of external factors; like the weather, track, wind, and rain in an F1 race. Ultimately, what dictates success in both domains; coding and Formula 1; is the mastery of planning, strategizing, maintaining flow, and ensuring precise code orchestration." - Siddhant Bali

- Author: Kintsugi-Programmer

> Disclaimer: The content presented here is a curated blend of my personal learning journey, experiences, open-source documentation, and invaluable knowledge gained from diverse sources. I do not claim sole ownership over all the material; this is a community-driven effort to learn, share, and grow together.

Table of Contents

About

Overview

- **Course Overview: Fundamental Data Structures and Algorithms**

- **Primary Objectives**
  - The course covers all **fundamental data structures and algorithms** required for knowledge in the field.
  - There is a specific **focus on coding interviews**.
- **Target Audience**
  - The content is designed for **beginners**.
  - It is also intended for **anyone who needs a refresher**.
- **Core Areas of Study**
  - The course explores four main aspects of common data structures and algorithms:
    - **Design**: How they are conceptualized.
    - **Implementation**: How they are built.
    - **Tradeoffs**: The pros and cons of different approaches.
    - **Analysis**: How to evaluate their performance.
- **Key Skills for Professional Success**
  - Students will learn to:
    - **Solve problems efficiently**.
    - **Analyze** problems effectively.
    - **Discuss tradeoffs** regarding different solutions.
    - **Communicate ideas** clearly to others.
- **Value and Impact**
  - Mastering these skills can lead to a **difference of hundreds of thousands of dollars in compensation**.
  - The **problem-solving skills** learned are intended to serve a student for their **entire career**.
- **Next Steps**
  - For further details on what the course covers, users should **scroll down**.
  - Users are encouraged to **get started** when they are ready.

Data Structure and Algorithms

Arrays

RAM

- **Introduction to Data Structures and RAM**
  - A **data structure** is defined as a specific way of **structuring data**.
  - In the context of computer science, this data is structured and stored inside of **RAM** (Random Access Memory).
  - RAM serves as the primary location where all **variables** are stored during the execution of code.
  - **Arrays** are the first data structure covered in this study.
  - When an array, such as one containing the values 1, 3, and 5, is used in code, that information must be stored within the RAM.
- **RAM Measurement and Binary Basics**
  - **RAM size** is measured in units called **bytes**.

    - It is common for modern computers to possess approximately **8 gigabytes** of RAM.
    - The term **"giga"** represents approximately **10 to the power of 9**, which is roughly **one billion**.
    - A single **byte** is composed of exactly **eight bits**.
    - A **bit** is a storage position for a single digit, with the restriction that the digit must be either a **0 or a 1**.
    - These zeros and ones constitute the fundamental **language of computers**.
    - Storage is hierarchical: individual bits form groups of bits, which form bytes, which collectively form RAM.
    - This RAM structure is then used to house more **advanced data structures**.

- **Representing Integers in Memory**

    - To store an integer like "1" in RAM, it must be represented in terms of bytes.
    - It is a common standard for **integers** to be represented by **four bytes** rather than a single byte.
    - Since one byte is eight bits, a four-byte integer is represented by **32 bits**.
    - To represent the integer **1** using 32 bits, the computer uses **31 zeros** followed by a single **1** at the very end.
    - Once the data is converted into this byte-based representation, it can be placed into RAM.

- **RAM Structure: Values and Addresses**

    - RAM can be visualized as a **contiguous block of data**.
    - RAM consists of two primary components: **values** and **addresses**.
    - Every value is stored at a **distinct location**, which is referred to as its **address**.
    - In technical diagrams, addresses are often distinguished from values by placing a **dollar sign (`)** in front of the address number.
    - The first address in a sequence is typically **0**.

- **Memory Properties of Arrays**

    - Arrays are considered the **most simple data structure** because they are stored in memory in the same way they are represented conceptually.
    - A defining characteristic of arrays is that they are always **contiguous**.
    - **Contiguous** means that values are stored one after another with **nothing in between** them.
    - For an array, the value 1 is stored first, followed immediately by 3, and then 5, without any gaps in the memory sequence.
    - While a programmer does not always choose the specific starting location of an array in RAM, the values will remain in a continuous block.

- **Data Sizes and Address Increments**

    - Because **32-bit integers** take up **four bytes**, the memory addresses for an integer array must **increment by four** for each new element.

```
Example Integer Array Memory Map:
Address: `0 -> Value: 1
Address: `4 -> Value: 3
Address: `8 -> Value: 5
```

- If the addresses were incremented by only one (`0, 1, `2) for integers, there would not be enough room, as each integer requires a full four-byte slot.
- Other data types, such as **characters** (e.g., 'A', 'B', 'C'), have different size requirements.
- **ASCII characters** typically take up only **one byte** of memory.
- Consequently, the addresses for a character array increment by **one** for each element.

```
Example Character Array Memory Map:
Address: `0 -> Value: 'A'
Address: `1 -> Value: 'B'
Address: `2 -> Value: 'C'
```

- The general rule for storing values contiguously is that the address must increment by the **size of the value** being stored.

- **Theory vs. Practice**
  - Understanding how memory and addresses work provides the necessary **theoretical foundation** for data structures.
  - However, the primary focus of this study is **practical knowledge**.
  - This includes understanding how arrays are used in real-world scenarios, as well as their specific **properties and tradeoffs**.

Storing data in RAM is like a **long row of lockers** in a school hallway; each locker has a unique number (the address), and depending on what you are storing, you might occupy just one locker (a character) or a block of four lockers (an integer), but you always make sure your items are in a row without any empty lockers between them.

Static Arrays

- **Static Arrays Overview**
  - **Definition and Memory Storage**
    - An array is a **contiguous block of data**.
    - In RAM, it is stored exactly this way: as one continuous block.
    - Data structures generally involve two primary operations: **reading** and **writing**.
  - **Reading Data from an Array**
    - **The Concept of Indices**
      - Programmers use variables (e.g., `my array`) to represent the allocated array.
      - To read the first element, the intuitive action is to go to the first memory address and read the value.
      - Programmers do not need to know exact memory addresses for every value because they use **indexes**.
      - **Index 0** is always the first value; it does not start at one,.
      - Subsequent values follow in order: index 1, index 2, and so on.
    - **Efficiency of Reading**

        - Any index in an array can be automatically mapped to a location in memory.
        - This allows for **instant reading** of any value as long as the index is known.
        - This operation is called **Big O of 1** (constant time).
        - Even as an array grows larger, reading a specific index happens in the same amount of time in the worst case.
    - **RAM Properties (Random Access Memory)**
        - The ability to go to an arbitrary address and read it instantly is a property of **Random Access Memory (RAM)**.
        - "Random access" means any part of the memory can be accessed in constant time without starting from the beginning and looking through it sequentially.
        - If you want the third value, you do not need to go through the first and second values first.

- **Looping Through an Array**

    - To read every value from start to finish, a programmer starts at index 0.
    - In code, this is typically represented as `my array[i]`.
    - The process involves:
        1. Starting with index `i = 0`.
        2. Reading the value at `i`.
        3. Incrementing `i` to 1, then 2, etc..
        4. Stopping when `i` equals the **length of the array**.
    - This is usually implemented using a **for loop** or a **while loop**.

- **The Fixed Size Property of Static Arrays**

    - **Static arrays are of fixed size**.
    - If you declare an array of three values, it remains that size.
    - **Challenges with Adding Values:**
        - If you want to add a fourth value (e.g., the number 7) to an array of size three, it must be **contiguous** to the existing data.
        - Example: If the array ends at address 11, the new value must be at address 12.
        - However, programmers cannot always decide where RAM is allocated.
        - The next spot in RAM might be:
            - Occupied by another array.
            - Used by the Operating System for another purpose.
        - If you put the new value anywhere else in memory, the array is no longer contiguous, which breaks the definition of an array.
        - If the values are not contiguous, incrementing an index (e.g., going from index 2 to 3) would not lead the computer to the correct memory spot where the value is stored.
    - **Language Differences:**
        - Languages like **Python or JavaScript** often use **dynamic arrays** as the default, so users rarely encounter these fixed-size limitations,.
        - Static arrays are the specific focus of this limitation.

- **Writing and Removing Data**

- **Initialization:**
  - When an array is first allocated, it might be empty but must store something.
  - Languages may initialize it to all zeros, or it may contain random, arbitrary "garbage" values.
- **Basic Writing:**
  - Adding a value to an empty spot is an **instant operation (Big O of 1)**.
  - Because indices map directly to RAM positions, the computer knows exactly where to write the value.
- **Removing Values:**
  - You cannot technically "delete" or deallocate a single spot in memory within a static array.
  - "Removing" a value involves **overwriting** it,.
  - Common methods include replacing the value with a 0, a -1, or another "irrelevant" marker,.
  - Overwriting/removing is a **Big O of 1** (constant time) operation because it just involves going to that specific memory position and changing it.

- **Inserting at Arbitrary Positions (Beginning or Middle)**

  - Arrays are **ordered values**; the position matters.
  - Inserting at the end (if space is available) is efficient (O(1)).
  - Inserting in the middle or beginning is **inefficient** because the order must be preserved.
  - **Example: Inserting 4 before 5 and 6**
    - If the array has `[5, 6, _]` and you want to insert 4 at the start to get ``:
      1. You cannot just overwrite 5 with 4, or you lose the 5.
      2. You must **shift** the existing values to the right.
      3. Step 1: Move the value at index 1 (6) to index 2 (1 + 1).
      4. Step 2: Move the value at index 0 (5) to index 1 (0 + 1).
      5. Step 3: Insert the value 4 into the now-empty index 0.
  - **Complexity of Shifting:**
    - In a long array, inserting at the beginning requires moving **every single value**.
    - This is a **Big O of N** operation, where N is the number of elements (length) of the array.
    - **Worst Case Scenario:**
      - Big O notation refers to the **worst case**.
      - If you insert in the middle, you only shift the values to the right of that point.
      - If you insert at the very beginning, you shift **all N values**.
      - To generalize for the worst case, we say the time complexity is **Big O of N**.

- **Removing from Arbitrary Positions**

  - Removing a value from the middle and wanting to "close the gap" (so the first value is the new start) requires shifting.
  - **Example: Removing the first element from ``**
    - To make 6 the new first value:
      1. Shift the value at index 1 (6) to the left to index 0 (1 - 1).
      2. Shift the value at index 2 (7) to the left to index 1 (2 - 1).

        - The result is ``. The original memory spot for 7 might still hold a value, but we treat it as non-existent or overwrite it with 0.
    - Like insertion, this is a **Big O of N** operation in the worst case because we may have to shift every subsequent value.

- **Summary of Static Array Operations**

    - **Reading:** `O(1)` (Constant time).
    - **Writing/Overwriting (at a known index):** `O(1)` (Constant time).
    - **Inserting at the end:** `O(1)` (Assuming space is available).
    - **Removing from the end:** `O(1)`.
    - **Inserting in the middle/beginning:** `O(N)` (Worst case due to shifting).
    - **Removing from the middle/beginning:** `O(N)` (Worst case due to shifting).

- **Coding Logic for Access and Loops**

```
// To read an element at index i
my_array[i]

// To loop through an array
for (i = 0; i < length; i++) {
    print(my_array[i]);
}
```

Dynamic Arrays

- DYNAMIC ARRAYS OVERVIEW

    - **Dynamic arrays** are much more common and useful than **static arrays**.
    - They solve the **fixed size problem** inherent in static arrays.
    - **Language Defaults**:
        - In **Python** and **JavaScript**, dynamic arrays are the default array type.
        - In **Java**, they are implemented as an **ArrayList**.
        - In **C++**, they are called a **Vector**.
    - **Initialization**:
        - You do not necessarily have to specify a size when creating one.
        - If no size is specified, it initializes to a **default size**.
        - Example: In Java, the default size of an ArrayList is typically 10, though this number is arbitrary.

- INTERNAL MECHANISMS AND TERMINOLOGY

    - **Capacity (Size)**: The total number of slots allocated in memory (e.g., a size of 3).
    - **Length**: The actual number of elements currently stored in the array (e.g., an empty array of size 3 has a length of 0).
    - **Pointers**:
        - The implementation maintains a **pointer** or variable that tracks the index of the **last element**.

        - Example: If the first element is at index 0 and the next is at index 1, the pointer identifies index 1 as the last element.
        - The **length** is calculated using this pointer; if the pointer is at index 1, the length is 2 (since indices start at 0).

- CORE OPERATIONS

    - **Pushing**:
        - This refers to adding an element to the **end** of the array at the next empty spot.
        - It is a **Big O(1)** time operation because the system knows exactly where the next spot is via the pointer.
    - **Popping**:
        - This refers to removing the last value from the end of the array.
        - It is a **Big O(1)** time operation because the pointer identifies the last element immediately.
        - After removing the value, the pointer shifts left by one index to mark the new end.

- THE RESIZING PROCESS (GROWING THE ARRAY)

    - **The Problem**: When the array runs out of space (e.g., capacity is 3 and you try to push a fourth element), you cannot simply allocate more memory right next to the existing array.
    - **The Solution**: The dynamic array must allocate a brand new array to contain all elements.
    - **Steps for Resizing**:
        1. **Allocation**: A new array is created in a different, random location in memory.
        2. **Capacity Doubling**: The new array's size is typically **double** the original capacity.
            - Example: An original size of 3 becomes size 6. An original size of 6 becomes size 12.
        3. **Copying**: All original values are copied from the old array to the new array starting at the beginning (index 0 to 0, index 1 to 1, etc.).
        4. **Insertion**: The new value (the one that triggered the resize) is added to the new space.
        5. **Deallocation**: The original array is **deallocated** or "freed," telling the operating system it is no longer in use so the memory can be reused.

- TIME COMPLEXITY AND AMORTIZATION

    - **Resizing Complexity**:
        - Allocating new memory takes **O(N)** time, where N is the size of the memory allocated.
        - Copying every value from the old array to the new one is also an **O(N)** operation.
    - **Amortized Time Complexity**:
        - Although a push that triggers a resize takes O(N), resizing is **infrequent**.
        - Because we double the capacity each time, most pushes are simple O(1) operations.
        - **Amortized complexity** is essentially the **average time** of an operation.
        - Pushing a value to a dynamic array has an **amortized time complexity of O(1)**.
    - **Mathematical Explanation (The Power Series)**:
        - If you push 8 elements into an array that started at size 1:
            - Total operations = 8 (last allocation/moves) + 4 + 2 + 1 = 15 operations.
        - This calculation is **dominated by the last term**.
        - The sum of all previous terms will always be less than or equal to the last term.
        - Therefore, the total operations for N elements is always **less than or equal to 2 * N**.

        - In Big O, **O(2 * N)** simplifies to **O(N)**.
        - Since it takes O(N) total time to push N elements, the average (amortized) time per push is **O(1)**.

- BIG O NOTATION AND CONSTANTS

    - **Constants are Ignored**:
        - We do not care about constants multiplied by N (like O(2 * N)) or added to N (like O(N + 8)).
        - Whether the constant is 10, 100, or 1000, it is simplified to O(N).
    - **Why Constants are Ignored**:
        - Big O focuses on **growth rates** as input sizes (N) become very large.
        - We care about how the runtime grows: is it linear (O(N)) or quadratic (O(N^2))?.
        - **Interaction of Functions**:
            - A faster-growing function like `N^2` will always eventually intersect and surpass a slower-growing function like `N` or `2N`, regardless of the starting constant.
            - We ignore small input sizes (like 1, 8, or 1000) because only very large inputs significantly slow down a CPU.

- DYNAMIC ARRAY PERFORMANCE SUMMARY

    - **Accessing an Element**: **O(1)** (can access any index anywhere in constant time).
    - **Push/Pop at the End**: **O(1)** (pushing is O(1) amortized).
    - **Resizing**: Allows for growth even when initial space is full.
    - **Inserting/Removing in the Middle**: **O(N)**.
        - This is the downside of dynamic arrays.
        - You must shift every subsequent value over to make room or fill a gap.
        - There is **no amortization** for this; it is always O(N) in the worst case.

- ANALOGY

    - Imagine a dynamic array as a growing dinner party table. You start with a small table (the original capacity). When more guests arrive than there are chairs, you don't just squeeze them in; you move the entire party to a new room with a table twice as big. While the move itself (copying elements) takes a lot of effort, it happens so rarely compared to guests just sitting down (pushing elements) that on average, seating a guest is still a very quick process.

Stacks

- **Stacks Overview**

    - A stack is a common data structure in programming that typically supports three specific operations.
    - **Core Operations**
        - **Push**: Adding an element to the end of the stack.
        - **Pop**: Removing an element from the end of the stack.
        - **Peek**: Looking at the last element in the stack to read it without removing it, unlike a pop operation.
    - **Efficiency Requirements**

        - These operations must run efficiently in **constant time**.
        - Pushing should be a constant time operation.
        - Popping should be a constant time operation.
        - Peeking at the last element should be a constant time operation.

- **Implementation with Dynamic Arrays**

    - There is no need to design a stack data structure from scratch because **dynamic arrays** satisfy all efficiency requirements.
    - A stack can be implemented using a dynamic array.
    - In standard programming, developers typically use the default dynamic array provided by the language they are using.
    - **Space Management**
        - Because stacks are implemented with dynamic arrays, users do not need to worry about running out of space.
        - The system will automatically allocate more space when needed.

- **Visual and Physical Representation**

    - **Array Representation**: A stack is essentially an array where elements are pushed onto the end.
    - **Vertical Representation**: Some visualize stacks as vertical structures where values are "thrown" into the top.
        - If the stack is empty, the first element goes to the bottom.
        - Subsequent elements are added to the "top," which is considered the "end" of the array.
    - **Tracking Elements**
        - A **pointer** or a count of the number of elements added so far is maintained.
        - This allows the system to know exactly where to push the next element into the next empty spot.

- **The Popping Mechanism**

    - Elements can only be removed from the end of the array (the top of the stack).
    - Removing from the end is an efficient process.
    - **Internal Memory Handling**
        - When an element is popped, the system does not necessarily delete that portion of memory.
        - Instead, it internally updates the marker to indicate a new "end" of the array or a new "top" of the stack.

- **LIFO (Last-In-First-Out) Principle**

    - **Definition**: Stacks are a **LIFO** data structure, meaning "Last In, First Out".
    - **The Reverse Order Property**: The order in which elements are inserted is the reverse of the order in which they are removed.
        - The last element added is the first element removed.
        - The second-to-last element inserted is the second-to-last removed if all elements are popped.
    - **Step-by-Step Example**
        - 1. Push a **1**.

        - 2. Push a **2**.
        - 3. Push a **3**.
        - 4. Pop the **3**.
        - 5. Pop the **2**.
        - 6. Pop the **1**.
    - This sequence demonstrates that the order of removal (3, 2, 1) is the reverse of the insertion order (1, 2, 3).

- **Use Cases**

    - **Reversing Sequences**: Because of the LIFO property, stacks can be used to reverse a sequence of values, such as numbers or a string of characters like "ABC".
    - **Complex Problem Solving**: Stacks are used for many other complex programming use cases and problems.

Analogy for Understanding: Think of a **stack of cafeteria trays**. When you add a new tray, you place it on the very top (Push). When someone needs a tray, they take the one that was most recently placed on top (Pop). The tray at the very bottom of the stack—the first one placed there—will be the very last one to be used.

Linked Lists

Singly Linked Lists

- **Singly Linked Lists Overview**

    - Linked lists have significant overlap with arrays but also many differences.
    - They are comprised of **list nodes**, sometimes simply called **nodes**.

- **List Node Definition and Structure**

    - A list node is an object that encapsulates at least two specific things:
        - **Value**: This can be an integer, a character, another object, or anything else (e.g., the value 4).
        - **Next**: This is a **pointer** that indicates what the next list node in the linked list is.
    - This pointer system is the method for connecting multiple list nodes together to form a linked list.
    - If a `next` pointer points at **null**, it means it points at nothing, signifying the end of the list or a single-node list.

- **Building and Connecting a Linked List**

    - **Initialization**:
        - To build a list, you first create a list node (e.g., a node with the string value "red").
        - By default, the `next` pointer is empty or set to **null**, meaning it is not yet connected to another node.
    - **Memory Allocation**:
        - When a node object is created, it is stored somewhere in memory.
        - Programmers do not necessarily control the exact memory location.
        - Multiple nodes (e.g., "red", "blue", and "green") can be created separately in memory.

    - **Implementation of Connections**:
        - In most programming languages, the `next` pointer is implemented as a **reference** to another list node object.
        - To connect nodes in code, you set the `next` property of one node to the variable representing the next node.
        - **Example**: `ListNode1.next = ListNode2`.
    - **Low-Level Mechanics**:
        - Under the hood, the second node (e.g., "blue") is stored at a specific memory address (e.g., "xx") while the first node is at another (e.g., "y").
        - The pointer tells the system the specific address of the next node.
        - Programmers usually use references to objects and do not need to worry about manual address management.

- **Memory Layout Differences**

    - **Linked Lists**: Nodes may be stored in **random order** in memory and are only connected via pointers.
        - A "red" node can point to a "blue" node even if the blue node is stored "behind" it or elsewhere in a "big mess" in memory.
    - **Arrays**: Arrays are stored in memory in the exact same sequence as they are used by the programmer.

- **Traversing a Linked List**

    - **Goal**: Start at the beginning and loop through every node until the end is reached.
    - **Detection of End**: The end is identified when a node's pointer points to **null**.
    - **Traversal Logic and Code**:
        - A variable named `cur` (current) is initialized to the first node.
        - A `while` loop runs as long as `cur` is not pointing at **null**.

```
cur = head
while cur != null:
    cur = cur.next
```

    - **Step-by-Step Execution Example**:
        1. `cur` starts at the "red" node.
        2. The condition `cur != null` is true, so the loop enters.
        3. `cur` is set to `cur.next`, following the pointer to the "blue" node.
        4. The condition `cur != null` is checked again; it is still true.
        5. `cur` is set to `cur.next`, moving to the "green" node.
        6. The condition is checked; it is still true.
        7. `cur` is set to `cur.next`, which is **null** (the green node points to nothing).
        8. The condition `cur != null` becomes false; the loop terminates.

- **Complexity and Edge Cases**

    - **Infinite Loops**: If a node's `next` pointer points back to a previous node (e.g., the third node points back to the first), the program will loop forever and eventually crash.
    - **Time Complexity of Traversal**: Traversing the entire list is **O(n)**, where **n** is the size of the linked list. This is identical to array traversal.

- **Head and Tail Management**

    - It is helpful to keep track of two specific pointers:
        - **Head**: The first element of the list.
        - **Tail**: The last element of the list.
    - In a list of **size one**, both the Head and Tail pointers point to the same single node.

- **Operations: Insertion at the End**

    - **Scenario**: Adding a new node (e.g., "purple") to the end of the list.
    - **Steps**:
        1. Create the new node.
        2. Set the current `tail.next` to the new node (`tail.next = ListNode4`).
        3. Update the `tail` pointer to the new node (`tail = ListNode4` or `tail = tail.next`).
    - **Time Complexity**: This is a **constant time operation O(1)** because the tail pointer allows direct access without iterating through the list. This is the same as an array.

- **Operations: Removal of Nodes**

    - **Constant Time Removal O(1)**: Removing a node (beginning or end) is **O(1)**, provided there is a pointer to the **previous node** of the one being removed.
    - **Procedure for Middle Removal**:
        - To remove a node between "head" and a third node, set `head.next` to `head.next.next`.
        - This "chains" the fields: `head.next` refers to the second node, and `head.next.next` refers to the third node.
        - By setting the head's next pointer to the third node, the second node is bypassed.
    - **Memory Management**:
        - In languages with **garbage collection**, the system automatically frees the memory of a node once no pointers are pointing to it.
    - **Comparison to Arrays**: Linked list removal is beneficial because it does not require shifting all subsequent elements like an array does.
    - **Arbitrary Removal**: If you must find a specific element first without a pointer, removal takes **O(n)** time because you must search the list.

Analogy for Understanding A singly linked list is like a **scavenger hunt**. Each location (node) contains a prize (value) and a slip of paper (pointer) telling you exactly where to find the next location. You can't jump straight to the third location unless you go to the first and second locations to get the clues. Even if the locations are scattered randomly across a city (memory), the clues keep them in a specific logical order.

Doubly Linked Lists

- **DOUBLY LINKED LISTS**

- **Overview and Structure**
  - A doubly linked list is a variation of a singly linked list.
  - The primary difference is the presence of two pointers, known as **double pointers**.
  - Each node contains:
    - A **next pointer** that points to the next node in the list.
    - A **previous pointer** that points to the previous node in the list.
  - **Structural Example with Three Nodes (1, 2, 3):**
    - The next pointers connect each node to the one following it.
    - The previous pointers connect each node to the one before it.
    - **Boundary Pointers:**
      - The first node has no previous node, so its previous pointer is set to **null**.
      - This null value indicates the beginning of the list.
      - The last node has its next pointer set to **null**, indicating the end of the list.
  - **Navigation Pointers:**
    - Like singly linked lists, doubly linked lists typically use a **head** pointer for the first node and a **tail** pointer for the last node.

- **Operations: Adding a New Node to the End**
  - Adding a node to the end of a doubly linked list is very similar to doing so in a singly linked list.
  - **Example: Adding Node 4 to a list ending at Node 3:**
    1. Take the next pointer of the current tail (Node 3) and assign it to the new node (Node 4).
    2. Set the previous pointer of the new node (Node 4) to point at the current tail (Node 3) to preserve doubly linked properties.
    3. Shift the tail pointer to the new node (Node 4).
  - **Importance of Order:**
    - The order of operations is critical.
    - The pointers must be connected (setting the previous pointer to the current tail) **before** updating the tail pointer to the new node.
  - **Code Logic for Addition:**
    - `tail.next = newNode`
    - `newNode.prev = tail`
    - `tail = tail.next` (or `tail = newNode`)

- **Operations: Removing the Last Node**
  - Removing the last node is easier with a doubly linked list because it allows looking backwards.
  - **Efficiency Advantage:**
    - In a singly linked list, you would have to start at the beginning and iterate forward to reach the second-to-last node.
    - In a doubly linked list, if you have a pointer to the tail, you can simply follow the previous pointer to find the preceding node.
  - **Example: Removing Node 3 from a list of three nodes:**

        1. Start at the tail (Node 3) and follow its previous pointer to reach the previous node (Node 2).
        2. Take the next pointer of Node 2 and set it to **null**.
        3. Update the tail pointer to point at Node 2.
    - **Post-Removal State:**
        - The list size is reduced (e.g., from three nodes to two).
        - Technically, the removed node (Node 3) might still exist in memory with its previous pointer still pointing to Node 2.
        - However, because there are no references to it from the list and the tail has been moved, it is effectively gone.
        - Depending on the programming language, **garbage collection** may delete this node automatically.
    - **Code Logic for Removal:**
        - `node2 = tail.prev`
        - `node2.next = null`
        - `tail = node2`

- **Time Complexity Analysis**

    - **End Operations:**
        - Both inserting and removing a value at the end are **constant time operations (O(1))**.
        - These operations do not require looping through the entire list regardless of its size.
    - **Accessing Elements:**
        - Accessing an arbitrary or random element (e.g., the second, third, or seventh element) is a **linear time operation (O(n))**.
        - You cannot jump to an index; you must follow pointers from the beginning (or end) until you reach the desired element.
    - **Middle Operations:**
        - Inserting or removing from the middle is technically a **constant time operation (O(1))** once you are at the location.
        - Unlike arrays, you do not need to shift all subsequent elements; you only change the pointers to skip or include a node.
        - **The Caveat:** To perform this O(1) removal or insertion, you must first arrive at that element. Since arriving usually requires iterating from the start, the total process is typically a **linear time operation (O(n))**.

- **Comparison: Linked Lists vs. Dynamic Arrays**

    - **Stacks:**
        - Stacks can be implemented with linked lists because they support O(1) push and pop at the end.
        - However, dynamic arrays are much more common for stacks because they offer the same efficiency for end operations while also allowing arbitrary element access.
    - **Summary Table of Complexities:**
        - **Arrays:**

        - Accessing any arbitrary index: **O(1)**.
        - Insert/Remove at end: **O(1)**.
        - Insert/Remove in middle: **O(n)** (requires shifting elements).
    - **Linked Lists (Singly or Doubly):**
        - Accessing arbitrary element: **O(n)**.
        - Insert/Remove at end: **O(1)**.
        - Insert/Remove in middle: **O(n)** (due to the need to iterate to the element first).
- **Utility:**
    - Arrays are generally more useful for problem-solving because efficient arbitrary access is highly important.
    - Linked lists have a slight theoretical benefit for middle operations, but this is often negated by the need to iterate through the list to find the insertion/removal point.

Analogy to Solidify Understanding Think of a **Singly Linked List** like a **one-way train** where each car only knows the car in front of it; if you want to find the person second-to-last, you have to walk from the engine all the way down. A **Doubly Linked List** is like a **train with walkie-talkies between every car**; every car knows who is in front *and* who is behind. If you are at the very last car (the tail) and need to unhook it, you can just talk to the car right behind you to tell it that it's now the new end of the train.

Queues

- Queues
    - Definition and Overview
        - Queues are an important data structure that share similarities with stacks.
        - The defining characteristic of a queue is that it follows the FIFO principle, which stands for "First In First Out".
        - Under FIFO, the first element added to the queue is the first element that is removed.
        - Elements are always removed in the same order they were originally added to the queue.
    - Core Operations
        - Enqueue (NQ)
            - This operation refers to adding elements to the queue.
            - Enqueuing is similar to the "push" operation used in stacks.
            - When you enqueue an element, you add it to the end of the queue.
        - Dequeue (DQ)
            - This operation refers to removing a value from the queue.
            - Unlike a stack, dequeuing involves removing the element from the beginning of the queue rather than the end.
    - Efficiency and Time Complexity
        - A primary goal is to perform queue operations efficiently.
        - Adding an element to the end of the queue should ideally be done in constant time, O(1).
        - Removing an element from the beginning of the queue should also be done in constant time, O(1).
    - Comparison: Arrays vs. Linked Lists
        - Array Implementation
            - Constant time removal from the beginning is not possible with standard arrays.

            - If an element is removed from the beginning of an array, every other element must be shifted over to fill the gap.
            - This shifting process results in a Big O of N (O(n)) time operation.
            - While it is technically possible to build a queue with an array, the implementation is significantly more complicated.
        - Linked List Implementation
            - Linked lists can achieve O(1) constant time for both adding and removing elements.
            - Implementing a queue is much simpler using a linked list compared to an array.
    - Step-by-Step Linked List Implementation
        - Initial State
            - Start with an empty queue where the current pointer is set to null.
            - To manage the queue, two specific pointers are maintained: a head pointer and a tail pointer.
        - Adding the First Element
            - Example: Adding the value "red" to the queue.
            - Create a list node for the value.
            - Both the head and tail pointers are set to point at this initial node.
            - The current pointer is set to this node because it is the only element in the queue.
        - Adding Subsequent Elements
            - Example: Adding the value "blue" to the queue.
            - To insert the new node, take the "next" pointer of the current tail node and set it to point to the new node.
            - The head pointer remains at the first element added.
            - The tail pointer is updated to point at the newest node, which is now the end of the queue.
        - Removing Elements (Dequeue)
            - Removing from the head of the list is a simple and efficient process.
            - To dequeue, take the current pointer (the head) and set it to `current.next`.
            - By moving the pointer to the next node, the original first node is bypassed and can be treated as if it no longer exists in the list.
            - This method allows for constant time removal from the beginning.
    - Usage
        - Queues are a very commonly used data structure in computer science.

Analogy: Think of a queue like a line at a grocery store checkout. The first person to join the line is the first person served and the first to leave (FIFO). If the store had to shift every single person forward physically every time the person at the front left (like an array implementation), it would be very inefficient. Instead, the cashier just looks at whoever is next in line (like moving a head pointer in a linked list), which is a quick and constant process regardless of how long the line is.

Recursion

Factorial

- **Recursion and Factorial Fundamentals**

- Recursion is a vital concept for data structures and algorithms that will be used throughout the course.
- It is often difficult to understand initially, so it is introduced using **one branch recursion** through a math formula: **n factorial**.
- **Definition of n Factorial:**
  - It is a shorthand for writing `n * (n - 1) * (n - 2)` all the way until reaching the base number, which is 1.
  - **Example:** 5 factorial (5!) is `5 * 4 * 3 * 2 * 1`, which equals 120.
  - While recursion is useful for learning, you do not strictly need it for this calculation; a loop is often the easiest way.

- **The Logic of Recursion and Sub-problems**
  - Recursion is centered on the idea of **sub-problems**.
  - A complex problem is broken down into a slightly smaller version of the same problem.
  - **Example Logic:** 5 factorial is the same as `5 * 4!` because 4 factorial is `(4 * 3 * 2 * 1)`.
  - **General Equation:** `n! = n * (n - 1)!`.
  - By using this equation, a big problem (n!) is reduced to a smaller sub-problem ((n - 1)!).

- **Visualizing Recursion with Decision Trees**
  - The best way to represent recursion is through a **decision tree**.
  - In the case of factorials, it is called **one branch recursion** because there is only one decision/path to follow at each step.
  - To compute 5!, the tree shows that we must first solve the sub-problem of 4! before we can multiply it by 5.

- **Recursive Implementation and Code Structure**
  - A recursive function for factorials typically takes an integer and returns an integer.
  - **The Recursive Step:**
    - This is when the function calls itself to solve the sub-problem.
    - In the code, this looks like calling the same function you are currently inside.
    - **Logic:** `return n * factorial(n - 1)`.
  - **The Base Case:**
    - This is the point where the series ends and the multiplication stops.
    - In math, **1 factorial is mapped to the constant 1**, so no further calculation is needed at that point.
    - **Math Note:** 0 factorial is also defined as 1.
    - **Importance:** Without a base case, the function would call itself infinitely, leading to increasingly negative numbers, and it would never return to compute the original result.
  - **Recursive Code Example:**

```
def factorial(n):
    # Base Case
    if n <= 1:
        return 1
```

```
    # Recursive Step
    return n * factorial(n - 1)
```

  - The `n <= 1` condition is used to handle both the base case of 1 and the zero case.

- **Step-by-Step Execution Trace (Using 5 Factorial)**
  - **1. Going Down (The Calls):**
    - Input 5: Not <= 1, so it calls `5 * factorial(4)`.
    - Input 4: Not <= 1, so it calls `4 * factorial(3)`.
    - Input 3: Not <= 1, so it calls `3 * factorial(2)`.
    - Input 2: Not <= 1, so it calls `2 * factorial(1)`.
    - Input 1: **Base Case reached**. It returns 1.
  - **2. Going Back Up (The Returns):**
    - The call for 2! receives 1, calculates `2 * 1`, and returns **2** to its parent.
    - The call for 3! receives 2, calculates `3 * 2`, and returns **6** to its parent.
    - The call for 4! receives 6, calculates `4 * 6`, and returns **24** to its parent.
    - The original call for 5! receives 24, calculates `5 * 24`, and returns the final result: **120**.
- **Complexity Analysis**
  - **Time Complexity:**
    - The complexity is **Big O of N (O(n))**, where n is the input value.
    - This is because there are n total steps or multiplications to perform.
    - This is identical to the time complexity of a standard for or while loop.
  - **Memory (Space) Complexity:**
    - The recursive solution has **Big O of N (O(n)) memory complexity**.
    - This is **worse** than a loop because each function call must be saved in memory.
    - Every function call is "put on hold" while waiting for its sub-problem to return.
    - When the code reaches the base case (1!), all n function calls are "alive" and taking up space in memory at the same time.
    - Each function call stores its own input parameter and tracking information.
- **Iterative (Non-Recursive) Solution**
  - The non-recursive approach is more memory-efficient as it typically uses a single variable.
  - **Iterative Logic:**
    - Initialize a `result` variable to 1.
    - Use a `while` loop that runs as long as `n > 1`.
    - Multiply the `result` by `n`, then decrement `n` by 1 in each iteration.
    - This calculates `n * (n-1) * (n-2)...` until n reaches 1.
  - **Iterative Code Example:**

```
def factorial_iterative(n):
    result = 1
    while n > 1:
        result = result * n
```

```
            n = n - 1
        return result
```

- This approach avoids the O(n) memory overhead of the recursive call stack.

Fibonacci

- **TWO-BRANCH RECURSION: THE FIBONACCI EXAMPLE**
  - **General Mathematical Formula**
    - To calculate the nth Fibonacci number, the formula is: `F(n) = F(n-1) + F(n-2)`.
    - This means you take the `n-1` Fibonacci number and the `n-2` Fibonacci number and add them together to get the nth value.
  - **The Necessity of Base Cases**
    - Without base cases, calculating a Fibonacci number (like the fifth one) would continue forever.
    - **Defined Base Cases**:
      - The zeroth Fibonacci number is 0.
      - The first Fibonacci number is 1.
    - **Example: Calculating the Second Fibonacci Number (F(2))**:
      - Formula: `F(2) = F(2-1) + F(2-2)`.
      - This simplifies to `F(1) + F(0)`.
      - Using the base cases: `1 + 0 = 1`.
      - Therefore, the second Fibonacci number is 1.
    - **Example: Calculating the Third Fibonacci Number (F(3))**:
      - Formula: `F(3) = F(3-1) + F(3-2)`.
      - This simplifies to `F(2) + F(1)`.
      - Since `F(2) = 1` and `F(1) = 1`, the calculation is `1 + 1`.
      - Therefore, the third Fibonacci number is 2.
  - **Recursion vs. Looping**
    - Fibonacci numbers can be calculated in a straightforward way by looping through them, which allows for calculating values as high as desired.
    - The recursive solution is actually less efficient than looping, but it is used to illustrate "two-branch recursion".
  - **The Recursive Decision Tree for F(5)**
    - A decision tree is used to visualize the recursive subproblems.
    - **Level 1**: To compute `F(5)`, the problem is broken into `F(4)` and `F(3)`.
    - **Level 2 (Left Path)**: `F(4)` is broken into `F(3)` and `F(2)`.
    - **Level 2 (Right Path)**: `F(3)` is broken into `F(2)` and `F(1)`.
    - **Further Breakdown**:
      - `F(3)` breaks into `F(2)` and `F(1)`.
      - `F(2)` breaks into `F(1)` and `F(0)`.
    - This process continues until every branch reaches a base case (`0` or `1`).
  - **Code Implementation and Logic**
    - **Base Case Code**:
      - Individual cases: `if n == 0: return 0` and `if n == 1: return 1`.
      - **Condensed version**: `if n <= 1: return n`.

        - This works because if `n` is 1, it returns 1, and if `n` is 0, it returns 0.
    - **Recursive Case Code**:
        - The function must compute the `n-1` and `n-2` values, add them together, and return that integer value.

```
# Representative logic based on the description
if n <= 1:
    return n
return recursive_fib(n - 1) + recursive_fib(n - 2)
```

- **Executing the Recursive Steps for F(5)**
    - The code executes down the tree and returns values back up once base cases are hit.
    - **Calculating F(2)**: Reaches base cases `F(1)` (returns 1) and `F(0)` (returns 0). Adding them (`1 + 0`) returns 1.
    - **Calculating F(3)**: Uses results from `F(2)` (which is 1) and `F(1)` (which is 1). Adding them returns 2.
    - **Calculating F(4)**: Uses results from `F(3)` (which is 2) and `F(2)` (which is 1). Adding them returns 3.
    - **Calculating F(5)**: Uses results from `F(4)` (which is 3) and `F(3)` (which is 2). Adding them returns the final result: 5.
- **Time Complexity Analysis**
    - **Loop Technique**: The time complexity is `O(N)`.
    - **Recursive Technique**: The complexity is much higher and requires analyzing the decision tree.
    - **Tree Height**:
        - The height of the tree (number of levels) is `n`.
        - For `F(5)`, the longest path is `5 -> 4 -> 3 -> 2 -> 1`, resulting in roughly 5 levels.
    - **Branching and Doubling**:
        - Each node generally breaks into two subproblems because there are two branches in the recursive tree.
        - Theoretically, the number of terms doubles at every level: `1 -> 2 -> 4 -> 8...`.
        - The number of values in the last level is roughly `2^n`.
    - **The Power Series and Upper Bounds**:
        - In a series where values double (1, 2, 4, 8), the series is dominated by the last term.
        - The last term (`2^n`) is greater than or equal to all previous terms combined.
        - Therefore, `2^n` serves as an upper bound for the total number of values in the decision tree.
    - **Big O Constants**:
        - Precise values like `n+1` or `n-1` in the exponent do not matter in Big O notation.
        - Multiplying by a constant (like `2 * 2^n`) also results in the same Big O complexity.
        - **Final Complexity**: The recursive solution is bounded by `O(2^n)`.

Sorting

Insertion Sort

- **Introduction to Sorting**
    - Sorting is a fundamental topic used frequently with various algorithms, each having specific tradeoffs.
    - It is most commonly applied to **arrays**, but can also be used for **linked lists** and other data structures.
    - The goal is typically to arrange elements in **ascending order** (increasing, e.g., 1, 2, 3) or **decreasing order** (e.g., 6, 4, 3).
    - **Example target:** Transform an unsorted array into a sorted one like.
- **Main Algorithm Idea: Subproblems**
    - The algorithm works by breaking the problem into subproblems.
    - **Step 1:** Consider the first element of the array. Any array consisting of only one value is considered sorted by default.
    - **Step 2:** Sort the first two values.
    - **Step 3:** Sort the first three values.
    - **Step 4:** Continue this process, increasing the sorted portion by one element each time, until the entire array is sorted.
    - **Implementation:** While this could be solved using recursion, it is more effectively done **iteratively** using loops.
- **The Insertion Process**
    - The name "insertion sort" comes from taking one value and inserting it into an already sorted array.
    - **Iteration start:** The loop starts at the **second element** (index 1) because the first element (the subarray of size one) is already sorted.
    - **Assumption:** At any point in the loop, every value before the current position is already in sorted order.
    - **Goal:** Determine where to put the current value relative to the sorted subarray.
    - **Comparison logic:**
        - Compare the current value to its left neighbor.
        - **If the value is larger:** It stays where it is.
        - **If the value is equal:** It also stays where it is; it will be sorted either way, whether in ascending or descending order.
        - **Important optimization:** If the current value is greater than its immediate left neighbor, it is automatically greater than all values to the left of that neighbor (since they are already sorted). No further comparisons are needed for that iteration.
- **Detailed Step-by-Step Logic**
    - **Pointers:**
        - `i`: The outer loop pointer, starting at index 1 and moving to the end of the list.
        - `j`: The inner pointer used to compare the current value to its neighbors on the left.
    - **Inner Loop Conditions:**

        - The algorithm checks if `j` is within bounds (e.g., `j >= 0`) to avoid index out of bounds errors.
        - It checks if the value at `j + 1` is less than the value at `j`.
    - **The Swap Mechanism:**
        - If the value at `j + 1` is less than the value at `j`, a swap is performed.
        - **Step A:** Place the value originally at `j + 1` into a **temporary variable**.
        - **Step B:** Move the value at `j` to the `j + 1` position.
        - **Step C:** Move the value from the temporary variable to the `j` position.
        - **Step D:** Decrement `j` by one to continue comparing the value with the next neighbor to the left.
    - **Termination:** The inner loop stops when the value is no longer smaller than its left neighbor or when `j` moves out of bounds (becomes less than zero).

- **Supported Data Types**
    - Insertion sort can be applied to any set of values as long as there is a way to compare two values.
    - **Integers:** Sorted numerically.
    - **Characters:** Sorted alphabetically (e.g., A, B, C).
    - **Strings:** Sorted the same way a dictionary sorts words (e.g., "apple" comes before "banana").

- **Stability in Sorting**
    - **Definition of Stable Sorting:** An algorithm that preserves the original relative order of equal values when there is a tie.
    - **Definition of Unstable Sorting:** There is no guarantee that the original relative order of equal values will be preserved.
    - **Example:** In an array, a stable sort ensures the first '7' remains before the second '7' in the final sorted list.
    - **Insertion Sort Stability:** Insertion sort is a **stable** algorithm.
        - During comparison, if two values are equal, the algorithm does not swap them.
        - Because ties do not trigger swaps, the original relative order is maintained.

- **Time Complexity Analysis**
    - **Best-Case Time Complexity:**
        - Occurs when the input array is **already sorted**.
        - The outer loop iterates through the list once.
        - The inner `while` loop condition is never met because every value is already greater than its left neighbor.
        - Result: **Big O(N)** (linear time).
    - **Worst-Case Time Complexity:**
        - Occurs when the input array is in **reverse order**.
        - For every value the algorithm iterates through, the inner loop must execute the maximum number of times to shift the value all the way to the left.
        - The number of operations follows a pattern like 1 + 2 + 3 + ... + (n-1).
        - **Visualizing Worst Case:**

        - If we imagine an N x N grid (N squared), the work done by insertion sort is approximately **half of N squared** (N^2 / 2).
        - In Big O notation, constants like "1/2" are ignored.
    - Result: **Big O(N^2)** (quadratic time).

- **Summary of Characteristics**

    - **Type:** Iterative sorting algorithm.
    - **Stability:** Stable.
    - **Best-Case Runtime:** O(N).
    - **Worst-Case Runtime:** O(N^2).
    - **Tradeoff:** While simple and stable, other sorting algorithms exist with better worst-case time complexities than N squared.

Merge Sort

- Merge Sort Overview

    - Merge sort is one of the most common and efficient sorting algorithms.
    - The main idea is to take an input array and split it into two approximately equal halves.
    - These halves are then split again into smaller approximately equal halves.
    - This splitting continues until the array can no longer be split, resulting in individual elements.
    - The goal is to break the original problem of sorting the entire array into smaller subproblems.
    - Example:
        - For an array like, you cannot split it exactly in half because there is an odd number of values.
        - It is split approximately into a first half (3, 2) and a second half (4).
        - These subarrays are then treated as subproblems to be sorted.

- Divide and Conquer Technique

    - Merge sort uses a technique called "divide and conquer".
    - This means taking the original problem, dividing it into subproblems, and solving those subproblems before solving the original problem.
    - This problem naturally lends itself to recursion because of the way it splits into subproblems until it reaches a base case.
    - It is specifically a "two-branch recursion" because the array is split into two halves at each step.
    - Dividing by two at each step is what makes the algorithm efficient.

- The Base Case

    - The base case occurs when a subarray has a single element.
    - A single element is technically already a sorted array.
    - Length check:
        - The length of a subarray is calculated as: `(ending index - starting index) + 1`.
        - If the length is less than or equal to 1, it is a base case and the array is returned.
        - Example: If an element is at index 0, the calculation is `(0 - 0) + 1 = 1`.

- The Merging Process

    - Once you have two sorted arrays (even if they are single elements), you merge them back together in sorted order.
    - This merging step is why the algorithm is called "merge sort".
    - Step-by-Step Merging Example (Left Side):
        - Start with. Split into individual elements and.
        - Compare the two values: 2 is smaller than 3.
        - Place 2 in the first position and 3 in the second.
    - Two-Pointer Technique:
        - To merge two sorted halves into the original array, use two pointers.
        - One pointer starts at the beginning of the left array and another starts at the beginning of the right array.
        - Smallest values are at the beginning of each sorted array.
        - Compare the values at the pointers.
        - The smaller value is inserted into the next position of the original array.
        - Shift the pointer of the array that provided the value to the next position.
        - A third pointer is used in the output (original) array to track where to insert the next element.
    - Handling Out-of-Bounds:
        - Continue the algorithm until one pointer goes out of bounds (runs out of elements).
        - Once one array is empty, fill in all remaining elements from the non-empty array into the remaining spots of the original array.
        - Example: If the right array has 6, 7, and 8 left after the left array is exhausted, just move 6, 7, and 8 over.

- Implementation Logic

    - The function typically takes the array, a starting index (`s`), and an ending index (`e`).
    - Using indices prevents the need to create new variables for every subarray split.
    - Calculating the Middle:
        - Find the middle index (`m`) by: `(starting index + ending index) / 2`.
        - Most programming languages round this result down.
        - Example: For indices 0 and 1, `(0+1)/2 = 0.5`, which rounds down to 0.
    - Recursive Calls:
        - Call `mergeSort` for the left half: from `s` to `m`.
        - Call `mergeSort` for the right half: from `m + 1` to `e`.
        - After these calls, the two halves are expected to be sorted.
    - Final Merge Call:
        - Merge the left half (`s` to `m`) and the right half (`m + 1` to `e`).
    - Memory Management:
        - During the merge step, extra memory is required.
        - Temporary arrays (copies) of the left and right halves are created so original values are not lost during the overwrite process.

- Time Complexity Analysis

    - Depth of Recursion (Levels):
        - The number of levels is determined by how many times a number `n` can be divided by 2 until it reaches 1.

        - Algebraically: `n / 2^x = 1`, which rearranges to `n = 2^x`.
        - Solving for `x` (the number of levels) using logarithms: `x = LOG_2 n`.
        - Therefore, there are `LOG n` levels in the algorithm.
    - Work Per Level:
        - At each level, the merge step requires iterating through every element of the input array to put them back in order.
        - This results in `O(n)` time complexity for each level.
    - Total Time Complexity:
        - Multiply the number of levels by the work per level: `LOG n * n`.
        - Overall Big O time complexity is `O(n LOG n)`.
        - This is much more efficient than `O(n^2)` algorithms like insertion sort.
        - Comparison: `n LOG n` vs `n`:
            - Decrementing a value (linear) is much slower than repeatedly dividing it by 2.
            - Example: With a value of 8, decrementing takes 8 steps to reach 1 (`8, 7, 6, 5, 4, 3, 2, 1`), while dividing by 2 takes only 3 steps (`8 -> 4 -> 2 -> 1`).

- Space Complexity

    - The memory complexity is `O(n)`.
    - This is because extra memory is needed for the temporary arrays used during the merge step, which is roughly the size of the original input array.

- Stability

    - Merge sort is a **stable** sorting algorithm.
    - A stable algorithm preserves the original order of elements when there is a tie (equal values).
    - To ensure stability in code:
        - When comparing elements from the left and right arrays during a merge, if the values are equal, choose the element from the **left** array.
        - Logic: An element in the left array appeared before the element in the right array in the original order.
        - Code condition: `if left_element <= right_element`, then insert the left element.

Quick Sort

- **Introduction to Quicksort**

    - Quicksort is a common sorting algorithm that shares many similarities with merge sort.
    - Unlike merge sort, which splits arrays into equal halves, quicksort uses a random value called a **pivot** to partition the array.
    - For convenience, the **rightmost value** in the input array is usually selected as the pivot value.
    - The core idea is to iterate through every value before the pivot and compare them to it.
    - Values that are **less than or equal to the pivot** are moved to the **left partition**,.
    - Values that are **greater than the pivot** are moved to the **right partition**.

- **The Partitioning Process**

    - Partitioning can be performed **in place**, meaning it does not require allocating extra memory or arrays,,.

    - The process uses **two pointers**:
        - One pointer scans through the input array,.
        - A second pointer (the left pointer) tracks the position where the next value less than or equal to the pivot should be inserted,.
    - **Step-by-Step Logic**:
        - The left pointer starts at the beginning of the array (index `s`),,.
        - As the iteration pointer moves, if it finds a value less than or equal to the pivot, that value is swapped with the value at the left pointer,.
        - After such a swap, the left pointer shifts one position to the right,.
        - If the current value is greater than the pivot, the iteration pointer moves on, but the left pointer stays put,,.
        - Once the iteration reaches the pivot index, one final swap is performed between the pivot and the value at the current left pointer,.
    - **Outcome of Partitioning**:
        - The pivot value is now in its **correct sorted position**,.
        - Everything to the left of the pivot is less than or equal to it,.
        - Everything to the right of the pivot is greater than it,.
        - Note: The left and right sides are not necessarily sorted yet; they are only partitioned.

- **Walkthrough Example**

    - Input Array: `` with pivot `3`,,.
    - 1. Compare 6 to 3: 6 is not less than or equal to 3. Do nothing. Iteration pointer moves.
    - 2. Compare 2 to 3: 2 is less than or equal to 3. Swap 6 and 2. Array becomes ``. Left pointer moves to the second spot.
    - 3. Compare 4 to 3: 4 is not less than or equal to 3. Do nothing.
    - 4. Compare 1 to 3: 1 is less than or equal to 3. Swap 6 and 1. Array becomes ``. Left pointer moves to the third spot.
    - 5. Reach pivot: Swap pivot `3` with value at left pointer `4`. Array becomes ``.
    - 6. Result: Pivot `3` is in place. Left partition is `, right partition is` .

- **Recursion and Base Case**

    - Quicksort is a recursive algorithm.
    - After the initial partition, quicksort is called recursively on the left side and the right side,.
    - The pivot itself is excluded from these recursive calls because it is already in its final position,.
    - **Base Case**: If the length of a subarray is less than or equal to one, the array is already sorted, and the function returns.
    - The process continues until the subproblems reach individual elements, which are sorted by definition,.

- **Time Complexity**

    - **Average Case: O(n log n)**,.
        - This occurs when the pivot splits the array into roughly equal halves,.
        - Each level requires iterating through the input (O(n)), and there are log n levels,.
    - **Worst Case: O(n^2)**,.

        - This occurs when the input is already sorted (or reverse sorted) and the rightmost element is always chosen as the pivot,.
        - In this scenario, the height of the recursion tree becomes n because the partitions are highly unequal.
    - **Optimizations**:
        - To avoid the worst case, one can use a more sophisticated pivot selection, such as choosing the middle value among the leftmost, rightmost, and center elements.

- **Space Complexity and Stability**

    - **Space Complexity**: Quicksort is more advantageous than merge sort because it does not require extra memory for arrays; it performs swaps **in place**,.
    - **Stability**: Quicksort is generally **not stable**,.
        - It does not preserve the relative order of duplicate values.
        - **Example of Instability**:
            - If you have two "7" values (Red 7 and Yellow 7) and a pivot of 5, the partitioning process may swap them such that Yellow 7 ends up before Red 7, changing their original relative order,.

- **Code Logic for Partitioning (Python-style logic)**

```
# Logic based on the sources
def partition(arr, s, e):
    pivot = arr[e] # Naive way: rightmost element
    left_ptr = s

    for i in range(s, e):
        if arr[i] <= pivot:
            # Swap current element with left_ptr element
            temp = arr[left_ptr]
            arr[left_ptr] = arr[i]
            arr[i] = temp
            left_ptr += 1

    # Final swap of pivot to the correct index
    arr[e] = arr[left_ptr]
    arr[left_ptr] = pivot
    return left_ptr
```

  ,,.

- **Summary Comparison**

    - Better than insertion sort in general efficiency.
    - Average efficiency is comparable to merge sort (O(n log n)), but worst case is worse (O(n^2)),.
    - Most people consider it an efficient algorithm despite the theoretical O(n^2) worst case.

- **Analogy for Understanding**

- Imagine a teacher organizing students by height. The teacher picks one student (the pivot) and tells everyone shorter to stand on the left and everyone taller to stand on the right. The pivot student then stands exactly between those two groups. The teacher then repeats this for the group on the left and the group on the right until every single student is in the correct order.

Bucket Sort

- **BUCKET SORT OVERVIEW**
  - **Definition and Efficiency**
    - Bucket sort is a unique sorting algorithm because it can run in **big O of n** (`O(n)`) time, even in the worst-case scenario.
    - It is considered super efficient compared to other algorithms that run in `O(n LOG n)` time.
  - **The "Forbidden Technique" and Constraints**
    - Bucket sort is referred to as a "**forbidden technique**" or "forbidden jutsu" because it is very rare that you are actually able to use it.
    - It can only be used if the problem has specific constraints on the values being sorted.
    - The primary constraint is that all values must fit within a **finite range**.
    - Example constraint: If you are guaranteed the only values in the array are 0, 1, and 2, you can use bucket sort.
- **RANGE REQUIREMENTS**
  - The range of values is the difference between the minimum and maximum possible values, such as 0 and 2.
  - While standard 32-bit or 64-bit integers have a range (approximately `-2^{31}` to `2^{31}`), this range is usually too large for bucket sort to be practical.
  - Typical ranges that qualify for bucket sort are relatively small, such as 0 to 100, 1,000, 10,000, or even 100,000.
- **THE BUCKET SORT PROCESS**
  - **Creating Buckets**
    - For every single value in the specified range, you create a "bucket".
    - The name "bucket sort" comes from this process of creating a bucket for every possible value.
    - In an example with values 0, 1, and 2, you create an array of three buckets where the indexes represent the values.
    - Values map to positions; for example, 0 maps to index 0, 1 maps to index 1, and 2 maps to index 2.
    - Mapping is not always a direct 1:1 match to the index; for instance, a value like 4 could be arbitrarily mapped to a specific spot in the count array.
  - **Phase 1: Counting Occurrences**
    - You iterate through the input array to count how many times each value appears.
    - Each bucket (index) in your counts array will store the number of times that specific value occurs in the input.
    - **Steps for counting:**
      1. Initialize the `counts` array with zeros for every possible value in the range.

        2. Use an index pointer `i` to move from the beginning to the end of the input array.
        3. For each value encountered, go to that value's index in the `counts` array and increment it by one.
        4. Example: If the first value is 2, increment `counts`. If the next is 1, increment `counts`.
    - You only need to pass through the input array **one time** to finish counting.
- **Phase 2: Overwriting the Input Array**
    - Once counting is complete, the original input values are no longer needed.
    - Unlike other sorting algorithms, bucket sort **does not swap** values; it simply overwrites them.
    - You fill the original array in order: first all the zeros, then all the ones, then all the twos.
    - **Algorithm logic for overwriting:**
        - Use an outer loop with a variable `n` to iterate through every position in the `counts` array.
        - Use an inner loop to iterate a number of times equal to the value stored in `counts[n]`.
        - Use a third pointer `i` (starting at 0) to track where to fill the next value in the original array.
        - **Example step-by-step:**
            - If `counts` is 2, the inner loop runs twice. It puts the value 0 into `arr` and `arr`, then increments `i` each time.
            - Move to the next bucket where `n` is 1. If `counts` is 1, the inner loop runs once, placing the value 1 at `arr` and incrementing `i`.
            - Move to the next bucket where `n` is 2. If `counts` is 3, the inner loop runs three times, placing the value 2 at `arr`, `arr`, and `arr`.
    - When the outer loop pointer `n` goes out of bounds of the counts array, the sorting is finished and the array is returned.

- **COMPLEXITY ANALYSIS**

    - **Time Complexity**
        - The overall time complexity is **big O of n** (`O(n)`).
        - Although the overwriting phase uses nested loops, it is not `O(n^2)`.
        - The inner loop only executes a total of `n` times across the entire process because the `i` pointer only moves from the beginning of the array to the end once.
        - Total time is `O(n)` for the counting pass plus `O(n)` for the overwriting pass, which equals `O(2n)` and reduces to **`O(n)`** because constants are ignored.
    - **Space Complexity**
        - Bucket sort requires extra memory for the `counts` array.
        - The size of this array is determined by the **range** of values (from `m` to `n`).
        - If the range is a fixed constant size, the extra memory is considered **big O of 1** (`O(1)`) space complexity.

- **STABILITY AND PRACTICALITY**

    - **Stability**
        - Bucket sort is **not a stable** sorting algorithm.

        - It does not care about the relative order of identical values because it overwrites them based on counts rather than moving the original elements.
        - **Making it stable:** It is technically possible to make it stable by using a **linked list** for each bucket.
        - Instead of just counting, you would add each encountered element to a linked list for that bucket, then reconstruct the array from those lists.
        - However, bucket sort is rarely used with anything other than a standard array.
    - **When to Use**
        - Use bucket sort if you are given an input where values are guaranteed to be within a specific, small range.
        - In this specific case, it is more efficient than any other sorting algorithm.
        - For general inputs without range constraints, **Merge Sort** or **Quick Sort** are preferred, with Merge Sort being more common.

- **ALGORITHM SUMMARY (PSEUDOCODE LOGIC)**

```
// Phase 1: Counting
For each value in input_array:
    counts[value] += 1

// Phase 2: Overwriting
i = 0
For n from 0 to counts.length:
    For j from 0 to counts[n]:
        input_array[i] = n
        i += 1
```

- **ANALOGY**

    - Bucket sort is like sorting a collection of colored tokens (red, blue, yellow) by simply counting how many of each you have. Instead of rearranging the actual tokens, you throw them away and lay out new ones in the right order based on your count: all red first, then all blue, then all yellow. You end up with a sorted line, but you don't care which specific red token was which.

Binary Search

Search Array

- Binary Search Overview

    - Binary search is an algorithm that is very closely related to sorting.
    - This algorithm can only run on an input that is already in some type of sorted order.
    - It is much more efficient than looking through every single value in an input.

- Intuition and Concept

    - Linear Search Comparison:
        - In a standard search, you might have to look at every single value to find what you are looking for in the worst case.

        - For example, if you were looking for the number 8 and it was at the very end of the array, you would have to look through all positions before finding it.
    - The Dictionary Example:
        - Massive lists of words are placed in alphabetical order in dictionaries.
        - If you are looking for the word "banana," you might start at the halfway point.
        - If you find words starting with "J" at the halfway point, you know "B" comes before "J".
        - Therefore, you eliminate the second half of the dictionary and only search the first half.
    - The Number Guessing Game:
        - If you need to guess a number between 1 and 100, and you are told if your guess is "too big" or "too small," the best strategy is to guess 50.
        - If 50 is "too small," the answer is between 51 and 100.
        - If 50 is "too big," the answer is between 1 and 49.
        - This approach eliminates half of the possibilities every single time, reducing the search space.
        - Guessing the halfway point is less risky because it guarantees at least half the possibilities are eliminated regardless of the feedback.

- Algorithm Requirements and Goals

    - Input Type: Binary search is most commonly run on an array.
    - Sorted Order: It is very important for the input to be in sorted order.
    - If the input is not sorted, the best possible search time is Big O of N (O(n)), where you look at every element individually.
    - Target: The value you are searching for.
    - Goal: To return the index where the target value appears.
    - Solution Not Found: If the target does not exist in the input, the algorithm usually returns -1 to indicate an invalid index.

- Algorithm Mechanics and Pointers

    - Search Space: The portion of the array currently being considered. At the start, the search space is the entire array.
    - Boundaries (Pointers):
        - Left Pointer (or Low Pointer): Initialized to index 0.
        - Right Pointer (or High Pointer): Initialized to the length of the array minus one.
    - Midpoint Calculation:
        - The middle index (m) is found by adding the two boundaries together and dividing by two.
        - Usually, this result is rounded down.
        - Example: index 0 + index 7 divided by 2 equals 3.5, which rounds down to index 3.

- The Binary Search Logic (Pseudo-code Steps)

    1. Initialize the Left pointer to 0 and Right pointer to length - 1.
    2. Start a loop that continues as long as the Left pointer is less than or equal to the Right pointer.
    3. Calculate the midpointer.
    4. Check the value at the midpointer against the target:

        - Case A: Target is greater than the value at mid.
            - The target must be to the right.
            - Set the Left pointer to mid + 1.
            - This eliminates the mid index and everything to its left.
        - Case B: Target is less than the value at mid.
            - The target must be to the left.
            - Set the Right pointer to mid - 1.
            - This eliminates the mid index and everything to its right.
        - Case C (Else): The target is equal to the value at mid.
            - The target is found.
            - Return the mid index.
    - 5. If the loop finished and the target was not found, return -1.

- Detailed Example Walkthrough: Target 5
    - Setup: A sorted array with indices 0 through 7. Target is 5.
    - Iteration 1:
        - Left is 0, Right is 7. Mid is 3 (Value at index 3 is 4).
        - 5 is greater than 4.
        - Update Left to mid + 1 (Left = 4).
    - Iteration 2:
        - Left is 4, Right is 7.
        - Mid calculation: (4 + 7) / 2 = 5.5, rounded down is 5 (Value at index 5 is 6).
        - 5 is less than 6.
        - Update Right to mid - 1 (Right = 4).
    - Iteration 3:
        - Left is 4, Right is 4. The search space is now a single value.
        - Mid calculation: (4 + 4) / 2 = 4 (Value at index 4 is 5).
        - 5 is equal to 5.
        - Return index 4.

- Example of a Failed Search: Target 9
    - If the search space reaches a single element (e.g., index 7 with value 8) and the target is 9:
        - 9 is greater than 8.
        - Left pointer is updated to mid + 1 (Left = 8).
        - Now Left (8) is greater than Right (7).
        - The search space is empty, and the loop terminates.
        - Return -1.

- Complexity Analysis
    - Time Complexity:
        - With every iteration of the loop, at least half of the search space is eliminated.
        - This continues (n -> n/2 -> n/4...) until the search space is size 1.
        - The number of times a value (n) can be divided by 2 until it equals 1 is represented by the formula: log base 2 of n.
        - Worst-case runtime: O(log n).

        - Log N is much more efficient than the O(N) required for a linear scan.
    - Memory Complexity:
        - Only a few pointers (Left, Right, Mid) are allocated.
        - No additional data structures or variable-length arrays are created.
        - Memory complexity: Big O of 1 (constant space).

To think of binary search simply, imagine you are looking for a specific page in a thick textbook; instead of flipping every single page from the front (Linear Search), you open it exactly to the middle. If the page you need is higher, you ignore the entire first half of the book and repeat the process with the remaining half until you land on the correct page.

Search Range

- **Binary Search Variation: Search Range**
    - This is a variation of the standard binary search algorithm that often appears in interview problems.
    - It is based on the concept of guessing a number within a specific range, such as a range between 1 and 100.
    - **Key Differences from Original Binary Search**
        - You are not necessarily given an array of numbers in sorted order.
        - You are not searching for the index of a specific target value.
        - Instead, you are given a range of values and must find a correct value `n` that satisfies specific conditions.
        - This is a more general case of binary search.
        - While the original case checked if `n` was equal to a target, this version can involve much more complex requirements or computations.
- **The General Template**
    - This range-style binary search is very similar to the original algorithm template.
    - **Inputs**
        - A range of values defined by a lower bound and an upper bound.
        - Important: No array is provided for this search.
    - **Process**
        - Perform the search while the `low` value is less than or equal to the `high` value.
        - Calculate the midway point (`mid`) to eliminate half of the remaining possibilities in each step.
    - **The Helper Function: `isCorrect(n)`**
        - Because there is no explicit target, a helper function (often called a dummy function) decides if the guess is correct.
        - This function takes a single number as an input and computes whether the guess satisfies the requirements.
        - The function must determine not only if a guess is wrong but also if it was "too big" or "too small".
        - **Logic of the Helper Function (Example)**
            - If `n` is greater than the desired value, return `1` (indicates the guess is too big).

    - If `n` is less than the desired value, return `-1` (indicates the guess is too small).
    - If `n` is exactly the desired value, return `0` (indicates the guess is correct).
    - This function can contain any algorithm or computation required by the specific problem.

- **Step-by-Step Example: Finding the Number 10 in a Range of 1 to 100**
    - **Initial State**
        - Range: 1 to 100.
        - Target value (hardcoded in this example): 10.
    - **Iteration 1**
        - Calculate mid: `(1 + 100) / 2 = 50`.
        - Call `isCorrect(50)`. Since 50 is greater than 10, it returns `1` (too big).
        - Update range: Set `high` to `mid - 1` (`50 - 1 = 49`).
        - Half of the possibilities are now eliminated.
    - **Iteration 2**
        - Calculate mid: `(1 + 49) / 2 = 25`.
        - Call `isCorrect(25)`. Since 25 is greater than 10, it returns `1` (too big).
        - Update range: Set `high` to `mid - 1` (`25 - 1 = 24`).
    - **Iteration 3**
        - Calculate mid: `(1 + 24) / 2 = 12`.
        - Call `isCorrect(12)`. Since 12 is greater than 10, it returns `1` (too big).
        - Update range: Set `high` to `mid - 1` (`12 - 1 = 11`).
    - **Iteration 4**
        - Calculate mid: `(1 + 11) / 2 = 6`.
        - Call `isCorrect(6)`. Since 6 is less than 10, it returns `-1` (too small).
        - Update range: Set `low` to `mid + 1` (`6 + 1 = 7`).
    - **Iteration 5**
        - Calculate mid: `(7 + 11) / 2 = 9`.
        - Call `isCorrect(9)`. Since 9 is less than 10, it returns `-1` (too small).
        - Update range: Set `low` to `mid + 1` (`9 + 1 = 10`).
    - **Iteration 6**
        - Calculate mid: `(10 + 11) / 2 = 10.5`, which rounds down to 10.
        - Call `isCorrect(10)`. Since 10 is the target, it returns `0` (correct).
    - **Final Result**
        - The algorithm returns the value 10.
        - The goal was to find the correct value, not an index.

- **Algorithm Complexity**
    - **Time Complexity**
        - The Big O time complexity remains **`O(LOG n)`**.
    - **Definition of `n`**
        - In original binary search, `n` represented the size of the array.
        - In range-style binary search, `n` represents the size of the range of values (e.g., if the range is 1 to 100, `n` is 100).

- **Conclusion**

    - This method allows the core binary search template to be extended to various types of problems by searching a space of values rather than a physical data structure.

Tree

Binary Tree

- Binary Trees Overview
    - General Complexity and Relationship to Other Structures
        - Binary trees are considered the most complicated data structure covered so far.
        - Understanding them requires using almost every topic previously covered in the course.
        - They are similar to linked lists because they utilize nodes containing specific values.
        - These values can be anything: integers, characters, strings, or objects.
    - Basic Structure
        - A node contains a value (for example, the integer 2).
        - Nodes use pointers to connect to each other.
        - As the name "binary" implies, each node has two pointers.
        - While linked lists use "next" and "previous" pointers, binary trees use "left" and "right" pointers.
        - Conventionally, these pointers are drawn pointing downwards to show relationships.
    - Tree Relationships
        - Parent and Child Relationships
            - If a pointer connects one node to another node below it, the lower node is called the child.
            - The node from which the pointer originates is considered the parent node.
            - A parent can have a left child and a right child.
            - Example: If a node with the value 2 points to a node with the value 1 on the left, the node with 1 is the left child of the node with 2.
        - Sibling Relationships
            - Nodes that share the same parent are considered sibling nodes.
            - Example: If a parent node has both a left child (value 1) and a right child (value 3), those two children are siblings.
        - Ancestors and Descendants
            - A descendant is any child or any node that exists beneath a specific node in the tree.
            - For the root node, every other node in the tree is considered a descendant.
            - An ancestor is any node in the parent chain above a specific node, such as its parent or grandparent.
            - The root node has no ancestors because it has no parents.
    - Key Terminology
        - Root Node
            - The single node at the very top of the binary tree is called the root node.
            - Every tree is guaranteed to have exactly one root node.
        - Leaf Nodes
            - Nodes that do not have any children (neither a left nor a right child) are called leaf nodes.
            - Their pointers point to a default value of null.

        - Every binary tree is guaranteed to have at least some leaf nodes.
    - Visualization
        - Computers draw trees "upside down" compared to nature.
        - In nature, roots are at the bottom and leaves are at the top; in computer science, the root is at the top and leaves are at the bottom.
- Rules of Binary Trees
    - No Cycles
        - A binary tree is not allowed to have cycles.
        - You cannot have a pointer from a leaf node point back up to the root node.
        - You cannot connect two adjacent (sibling) nodes together, as this would form a cycle.
        - Even if the pointers were undirected (allowing movement in both directions), there should be no cycles in the graph.
    - Connectivity
        - All nodes in a binary tree must be connected together.
        - You cannot have a "floating" node that is not connected to the rest of the cohesive binary tree.
- Measuring Nodes: Height and Depth
    - Height of a Node
        - Height is measured by finding the longest path from a specific node down to its lowest descendant.
        - It is calculated by counting the number of nodes in that longest path.
        - Example: A single leaf node has a height of 1.
        - Example: If a node has a descendant, and the path from that node to the descendant contains two nodes, the height is 2.
        - Example: If a node has a left path of three nodes and a right path of two nodes, the height of that node is 3 because 3 is the longest path.
        - Note: Some textbooks measure the height of a single node as 0 instead of 1.
    - Depth of a Node
        - Depth is measured as the path from a specific node up to the root node.
        - Measurement Method 1 (Node Count): Count the number of nodes in the path to the root.
            - The depth of the root node is 1.
            - The depth of a child of the root is 2.
            - The depth of a grandchild is 3.
        - Measurement Method 2 (Pointer Count): Some people count the pointers (edges) connecting the nodes instead.
            - In this case, the depth of the root node would be 0.
- Programming Implementation
    - General Structure
        - A tree node is similar to a linked list node.
        - It contains a value property (e.g., an integer) and two pointers for the children.
    - Code Logic
        - When creating a tree node, you initialize the value.
        - The left and right pointers are initially set to null.
        - To add a child, you reassign the null pointer to a new child tree node.

- Example Structure:

```
class TreeNode {
    int value;
    TreeNode left;
    TreeNode right;

    TreeNode(int val) {
        this.value = val;
        this.left = null;
        this.right = null;
    }
}
```

- To connect nodes: The root node's left pointer is assigned to the left child node, and its right pointer is assigned to the right child node.

Binary Search Trees (BST)

- **Core Concept and Sorted Property**
  - Binary search trees, or BSTs for short, are a special type of binary tree that possess a certain sorted property.
  - While they are not literally sorted in the same way an array is, their property allows them to be used in a manner similar to a sorted array, specifically for binary search algorithms.
  - The guarantee of a BST is that for every single node in the tree, every node in the left subtree must have a value less than the root value.
  - Conversely, every single node in the right subtree must have a value greater than the root value.
  - This rule applies to subtrees, which are smaller sections of the entire BST structure.
- **Handling Duplicates and Structure Validation**
  - Generally speaking, binary search trees do not contain duplicate values, so the left and right subtrees contain values strictly less than or strictly greater than the root, respectively.
  - To determine if a structure is a valid BST, every node must be checked against these rules.
  - For example, if a tree has a root of 2, a right child of 3, and a left child of 1, it is valid because 1 < 2 and 3 > 2.
  - However, if that same tree had a node with the value 4 in the left subtree, it would be invalid because 4 is not less than 2.
  - In such a case, the 4 would need to be moved to the right subtree to satisfy the BST requirements.
- **The Recursive Definition**
  - The definition of a binary search tree is recursive, meaning the sorted property must be true for every single node in the tree, not just the root.
  - The property applies to the entire tree as well as every individual subtree.

- From the perspective of any specific node, that node acts as its own tree; even though it has a parent, it must maintain the same sorted properties as any other BST.
- For example, if a node has a value of 3, any value in its left subtree must be less than 3, and any value in its right subtree must be greater than 3.
- If a value like 4 is placed in the left subtree of the node 3, it violates the recursive property because 4 is not less than 3.

- **Searching in a BST**
  - The sorted property allows for efficient searching, achieving the same benefits as a sorted array.
  - In an unsorted structure, finding a target requires looking at every individual element, resulting in `O(n)` time complexity.
  - With a BST, searching can be done in `LOG n` time because the sorted property allows the algorithm to follow the same logic as a binary search.
  - **Step-by-Step Search Example (Target: 5)**
    1. Start at the root node (e.g., value 2).
    2. Compare the target (5) to the root (2).
    3. Since 5 is greater than 2, the search eliminates the root and the entire left subtree.
    4. Move to the right child (e.g., value 3).
    5. Compare the target (5) to the current node (3).
    6. Since 5 is greater than 3, eliminate any left children and move to the right child (e.g., value 4).
    7. Compare the target (5) to the current node (4).
    8. Since 5 is greater than 4, move to the right.
    9. If the next node is null, the search is complete, and the target was not found.
    10. The algorithm would return false if the target is not found and true if it is located.

- **Search Algorithm Implementation**
  - Because binary trees are recursive, a recursive algorithm is the easiest and most common way to traverse and search them.
  - Although recursion is not strictly required, it is preferred due to the tree's structure.
  - Searching a BST is considered "one branch recursion" because, at each node, a comparison is made and the algorithm only proceeds in one direction (left or right).
  - There is no need to explore both branches, which is what makes the search efficient.
  - **Recursive Logic Steps:**
    - The search function accepts the root of a binary tree and a target value.
    - **Base Case:** If the current node is null, return false because the search has run out of nodes without finding the target.
    - **Comparison 1:** Check if the target is greater than the current node's value; if so, recursively call the search function on the right child.
    - **Comparison 2:** Check if the target is less than the current node's value; if so, recursively call the search function on the left child.
    - **Comparison 3:** If the target is neither greater nor less than the current value, it must be equal; return true.
    - The result (true or false) is then returned back up through the parent recursive calls until the original call is resolved.

- **Time Complexity and Balance**
  - The time complexity for searching a BST is `LOG n`, similar to binary search.
  - This efficiency depends on the assumption that every comparison eliminates roughly half of the remaining possibilities.
  - However, the search is only `LOG n` if the tree is "roughly balanced".
  - **Defining a Balanced Tree:**
    - A tree is balanced if, for every single subtree (including the root), the heights of the left and right subtrees differ by no more than one.
    - For example, if a root has a left subtree of height 1 and a right subtree of height 2, it is balanced because the difference is one.
    - If a leaf node has subtrees of height 0 and 0, the difference is zero, and it is balanced.
  - **Unbalanced Trees:**
    - A tree might not be balanced; for instance, it could be a long string of nodes resembling a linked list.
    - In an unbalanced tree, you cannot go "halfway" between nodes, and the search may not eliminate half the possibilities with each step.
    - In the worst case, the time complexity for an unbalanced BST is Big `O(n)`.
  - **General Complexity:**
    - Technically, the time complexity of a search is Big `O(h)`, where `h` is the height of the tree.
    - For an imbalanced tree, `h` equals `n`.
    - For a balanced tree, `h` equals `LOG n` because the structure is multiplied by two at every level.

- **BST vs. Sorted Arrays**
  - A common question is why one should use a BST when a sorted array already allows for `LOG n` searches.
  - The primary downside of sorted arrays is the cost of adding or removing values while maintaining the sorted property.
  - In an array, removing or adding a value requires shifting all subsequent values, which takes `O(n)` time.
  - In a binary search tree, both inserting and deleting values can be achieved in `LOG n` time.
  - This ability to perform insertions and deletions efficiently while maintaining search speed is the main benefit of using a binary search tree.

- **Summary Analogy**
  - Searching a balanced BST is like using a directory where every choice you make immediately throws away half of the remaining pages you don't need to look at. However, if the tree is unbalanced, it is like a directory where every page just tells you to look at the very next page, forcing you to read the whole book one page at many times until you find what you need.

BST Insert and Remove

- **Binary Search Tree (BST) Operations: Insert and Remove**
- **Overview of BST Benefits**

- The primary benefit of binary search trees over sorted arrays is the efficiency of operations.
- We can insert and remove nodes in O(log n) time, provided the tree is roughly balanced.
- In a balanced tree, the height is approximately equal to log n.

- **BST Insertion**
  - **General Strategy**
    - Inserting a node involves traversing the height of the tree.
    - The goal is to find the correct position to insert the node.
    - While there can be multiple valid BST configurations after an insertion, the preferred method is to insert the node as a **leaf node** because it is easier to implement.
    - Advanced data structures, like the **AVL tree**, can implement balanced binary search trees where insert and remove operations automatically result in a balanced tree.
  - **The Insertion Process**
    - Start at the root node.
    - Traverse the tree using the same logic as a search operation.
    - Compare the value to be inserted with the current node's value:
      - If the value is **greater** than the current node, move to the **right subtree**.
      - If the value is **less** than the current node, move to the **left subtree**.
    - Typically, duplicate values do not exist in a BST; if a value already exists, another one is not inserted.
  - **Step-by-Step Examples**
    - **Inserting 6 into a tree with root 4**:
      - Start at the root (4).
      - 6 is greater than 4, so move down to the right.
      - If the right child is null, create a new node with the value 6.
      - The parent node (4) reassigns its right pointer to the newly returned node (6).
    - **Inserting 5 into the same tree**:
      - Start at the root (4). 5 is greater than 4, go right to node 6.
      - 5 is less than 6, so go to the left subtree.
      - The left child of 6 is null, so create a node with value 5.
      - Node 6 assigns its left pointer to node 5.
    - **Inserting 2 into the same tree**:
      - Start at the root (4). 2 is not greater than 4, it is less than 4.
      - Call insert on the left subtree.
      - If the left child is null (base case), create node 2 and return it.
      - Node 4 assigns its left pointer to node 2.
  - **Recursive Implementation and Logic**
    - The function is passed a node (root) and a value.
    - **Base Case**: If the current node is **null**, create the new node and return it.
    - If not at the base case:
      - If the value is greater than the root value, call `insert` on the right subtree and assign the result back to the root's right pointer.
      - Once the insertion is done, the function returns the current node (root) up to its parent to maintain connections.
      - Only one branch (if/else if) executes per recursive call.
  - **Time Complexity of Insert**

        - The complexity is O(height of the tree).
        - Because the process only visits one node per level (either going left or right, never both), it is the same as searching.
        - In a balanced tree, this is **O(log n)**.

- **Finding the Minimum Value**

    - **Purpose**
        - Finding the minimum is required for the `remove` function.
    - **Logic**
        - In a BST, the smallest element is located all the way to the **left**.
        - Because all values less than a node are in its left subtree, traversing left repeatedly leads to the minimum.
    - **Implementation**
        - Assign a pointer to the starting node.
        - While the pointer is not null and the left child is not null, move the pointer to the left child.
        - Once the left pointer is null, the current node is the minimum; return this node.
        - This is not necessarily recursive; it can be done with a simple loop, similar to traversing a linked list.

- **BST Removal**

    - **Overview**
        - Removing a node is more complicated than inserting.
        - The process begins by searching for the node with the target value.
    - **Case 1: Zero or One Child (Simple Case)**
        - This applies if the node to be removed has no children or just a single child.
        - **Zero children**: If a node is a leaf, the function returns `null` to the parent, effectively disconnecting the node.
        - **One child**: If the node has one child, the function returns that child to the parent. The parent's pointer then bypasses the removed node and points directly to the child.
        - Example: Removing node 3 (which has child 2) from parent 4:
            - 3 is less than 4, so we find 3. 3 has node 2 on the left and null on the right.
            - Since the right pointer is null, return the left child (2) to node 4.
            - Node 4's left pointer now points directly to node 2.
    - **Case 2: Two Children (Difficult Case)**
        - This applies when the node to be removed has both a left and a right child.
        - Removing such a node directly would disconnect its descendants.
        - **Resolution Strategy**:
            - Replace the target node's value with a value from one of its descendants to maintain the BST property.
            - The best values to use for replacement are either the **minimum value from the right subtree** or the **maximum value from the left subtree**.
        - **Steps using the Right Subtree Minimum**:
            1. Find the minimum node in the right subtree.
            2. Change the value of the node-to-be-removed to the value of that minimum node.

        - 3. Recursively call the `remove` function on the right subtree to delete the original minimum node.
    - **Logic for Validity**:
        - The minimum value from the right subtree is guaranteed to be greater than all nodes in the left subtree (because it came from the right) and less than or equal to all other nodes in the right subtree.
        - The node being recursively removed (the former minimum) is guaranteed to have **at most one child** (a right child), because if it had a left child, it wouldn't be the minimum. This ensures the recursive call hits Case 1.
- **Example: Removing the Root (4) with children 3 and 6**
    - Node 4 has two children. Find the minimum of the right subtree (e.g., node 5).
    - Update node 4's value to 5.
    - Call `remove(right_subtree, 5)`.
    - Node 5 in the right subtree is found; since it has no left child, its right child (even if null) is returned to its parent (node 6).
    - Node 4 is now effectively node 5, and the original node 5 is gone.
- **Time Complexity of Remove**
    - In the worst case, we traverse the height of the tree twice: once to find the node and its minimum replacement, and once to perform the final removal of the replacement node.
    - Since 2 is a constant, the complexity remains **O(log n)** for a balanced tree.
    - This O(log n) performance for both insertion and removal is the key advantage of BSTs over sorted arrays.

- **Code Logic Summary for Remove**

```
# Logic for removal branching
if value > root.val:
    root.right = remove(root.right, value)
elif value < root.val:
    root.left = remove(root.left, value)
else:
    # Node found
    if not root.left:
        return root.right # Covers 0 or 1 child (right)
    elif not root.right:
        return root.left # Covers 1 child (left)
    else:
        # Case with 2 children
        min_node = findMin(root.right)
        root.val = min_node.val
        root.right = remove(root.right, min_node.val)
return root
```

Analogy for BST Insertion: Think of inserting a new book into a library organized by a specific system. You start at the main entrance (the root) and look at a sign. If your book's category is "higher" than the sign, you go to the right wing; if "lower," you go to the left. You keep following signs at every junction until you find an

empty spot on a shelf (a null leaf position) where the book fits perfectly while keeping the whole library in order.

DFS

- **Introduction to Tree Traversal**
  - **Goal of Traversal**
    - With a sorted array, iterating left to right is simple.
    - We want to achieve the same with a Binary Search Tree (BST): iterate through all values in sorted order.
    - This process is not as simple as an array, but the logic remains going "left to right".
  - **Inorder Traversal Concept**
    - Traversing the tree in sorted order is called "Inorder Traversal".
    - **Recursive Definition**:
      - Start at the root.
      - Smallest values are known to be in the left subtree.
      - Process the entire left subtree first.
      - Process the current value.
      - Process the right subtree (where values are greater).
- **Detailed Step-by-Step Traversal Example**
  - **Scenario**: A tree with root 4, left child 3, left-left child 2.
  - **Execution Trace**:
    - Start at Root (4). Go to left subtree (3).
    - At Node (3), go to left subtree (2).
    - At Node (2), try to go left. It is null (no values less than 2).
    - **Print Node (2)**: This is the first value processed.
    - Check right subtree of (2). It is null. Return to parent (3).
    - **Print Node (3)**: We returned from its left subtree, so now we process 3.
    - Check right subtree of (3). It is null. Return to parent (4).
    - **Print Node (4)**: We returned from its entire left subtree, so now we process 4.
    - Check right subtree of (4). It exists (Node 6). Proceed to 6.
  - **Continuing down the Right Subtree (Node 6)**:
    - Before printing 6, go to its left subtree (Node 5).
    - At Node (5), left is null (base case).
    - **Print Node (5)**.
    - Right of 5 is null (base case). Return to parent (6).
    - **Print Node (6)**: Left side of 6 is done.
    - Go to right subtree of 6 (Node 7).
    - Left of 7 is null. **Print Node (7)**. Right of 7 is null.
  - **Observation**:
    - For every node, the order was: Left -> Current Node -> Right.
    - This mimics going through an array in sorted order using pointers and recursion.
- **Inorder Traversal Implementation**
  - **Code Structure**:

        - It is a recursive function taking a `root` node.
        - **Base Case**: If the current node is `null`, return (nothing to traverse).
        - **Recursive Step 1**: Call function on `node.left`.
        - **Operation**: Print/Process `node.val`.
        - **Recursive Step 2**: Call function on `node.right`.
    - **Time Complexity**:
        - In the worst case (and pretty much every case), we must visit every single node once.
        - Time Complexity: **O(N)**, where N is the size of the tree.
        - This is equivalent to traversing a sorted array; it cannot be less than O(N).

- **Tree Sort (Sorting with BST)**

    - **Concept**:
        - If given random values in an arbitrary order, they can be sorted using a BST.
        - First, build the BST by inserting each value. Then, perform inorder traversal to get a sorted array.
    - **Time Complexity Analysis**:
        - **Building the Tree**:
            - Inserting one value into a balanced BST is O(log N).
            - Doing this for N values is **O(N * log N)**.
        - **Traversing the Tree**:
            - Inorder traversal takes **O(N)**.
        - **Total Complexity**:
            - Equation: O(N log N + N).
            - As N becomes large, N log N grows faster than N.
            - We drop the smaller term. The time complexity is dominated by **O(N log N)**.
    - **Comparison**:
        - This performance is similar to Merge Sort and Quick Sort.

- **Types of Tree Traversals (Depth First Search Variants)**

    - **1. Inorder Traversal** (Discussed above)
        - Order: Left -> Root -> Right.
        - Use: Getting values in sorted order.
    - **2. Pre-order Traversal**
        - **Definition**: Visit the root node *before* visiting its children recursively.
        - **Order**: Root -> Left Subtree -> Right Subtree.
        - **Implementation**: Swap the print line to the top, before the recursive calls.

        ```
        function preorder(node):
            if node is null return
            print node.val        <-- Happened first
            preorder(node.left)
            preorder(node.right)
        ```

    - **3. Post-order Traversal**
        - **Definition**: Visit the children (left and right) *before* visiting the root node.

        - **Order**: Left Subtree -> Right Subtree -> Root.
        - **Implementation**: Swap the print line to the bottom, after the recursive calls.

```
function postorder(node):
    if node is null return
    postorder(node.left)
    postorder(node.right)
    print node.val        <-- Happened last
```

    - **4. Reverse Order Traversal**
        - **Goal**: Iterate values from largest to smallest (e.g., 7, 6, 5, 4...).
        - **Order**: Right Subtree -> Root -> Left Subtree.
        - **Implementation**:
            - Recursively do the right subtree first.
            - Print the value.
            - Recursively do the left subtree.

- **Depth First Search (DFS)**

    - **Definition**:
        - Inorder, Pre-order, and Post-order are all examples of **Depth First Search (DFS)**.
        - As the name implies, the search goes "depth first".
    - **Mechanism**:
        - Start at a node and go as deep as possible in one direction before backing up.
        - Example: Go to the left child, then that node's left child, all the way to the bottom (leaf) first.
        - You do not process a layer at a time; you plunge to the bottom of a specific branch immediately.
        - This behavior applies regardless of whether you process the node before (pre) or after (post) the children.
    - **Time Complexity**:
        - For all DFS traversals (Inorder, Pre, Post, Reverse), the complexity is **O(N)**.

- **Breadth First Search (BFS) Preview**

    - **Concept**:
        - The opposite of DFS is Breadth First Search (BFS).
    - **Mechanism**:
        - Traverses layer by layer.
        - Example: Visit the root (layer 1), then all direct children (layer 2), then all grandchildren (layer 3).
        - It does not reach the bottom immediately like DFS.

BFS

- **Introduction to Breadth-First Search (BFS)**

    - BFS is an algorithm used to traverse trees layer by layer.

    - It can be applied to any tree, regardless of whether it has the sorted property of a binary search tree or not.
    - **Core Concept**:
        - Traverse the closest nodes first, then the next layer of closest nodes, and continue this pattern.
        - This is the opposite of Depth-First Search (DFS), which goes all the way to the bottom before traversing other nodes.
    - **Direction**:
        - Typically, BFS goes from left to right, though this is not strictly necessary.
        - For trees, left to right is the standard approach.
    - **Alternative Name**:
        - BFS is also called "Level Order Traversal" because it processes the tree level by level.

- **Traversal Logic and Challenges**

    - **Example Walkthrough**:
        - If traversing a tree with root 4, left child 3, and right child 6:
            - First, print the root: 4.
            - Move to the next layer and traverse left to right: print 3, then print 6.
        - After printing 6, the algorithm must determine how to reach the children of the previous layer (nodes 2, 5, and 7).
    - **Implementation Strategy**:
        - BFS does not suit recursion well because you do not recursively traverse one branch fully before moving to the next.
        - It requires an **iterative** approach.
    - **Process**:
        - Process a node (e.g., 4).
        - Process all children of that node from left to right (e.g., 3 and 6).
        - Process all children of those nodes (e.g., children of 3, then children of 6).

- **Data Structure Requirement**

    - As nodes are processed, their children must be stored to be processed later.
    - **The Queue**:
        - The required data structure is a **Queue** (First In, First Out).
        - When a node is processed, its children are added to the queue in left-to-right order.
        - Example: After processing 4, add 3 and 6 to the queue.
        - Since 3 was added first, it is processed first, ensuring the current level finishes before the next begins.
    - **Storage**:
        - It is important to store the actual **node** in the queue, not just the value.
        - Storing the node preserves the pointers to the left and right children, which are needed for traversal.

- **Algorithm Implementation (Python)**

    - **Setup**:
        - Use a double-ended queue (called a deque in Python).

        - Initialize the queue with the root node.
            - If the tree is empty, nothing is added.
    - **Level Tracking (Optional but Useful)**:
        - It is possible to count levels (Level 0, Level 1, etc.) while printing.
        - This adds a few extra lines of code but makes the output clearer.
    - **The Code Structure**:

```
# Conceptual Python Code based on description
queue = deque([root])
level = 0
while len(queue) > 0:
    print("Level:", level)
    for i in range(len(queue)):
        curr = queue.popleft()
        print(curr.val)
        if curr.left:
            queue.append(curr.left)
        if curr.right:
            queue.append(curr.right)
    level += 1
```

    - **Step-by-Step Code Execution**:
        - **Outer Loop**: Runs as long as the length of the queue is greater than zero.
        - **Inner Loop**:
            - Takes a "snapshot" of the queue length at the start of the level.
            - Example: If the queue has length 1 (root), the loop runs once.
            - This ensures we only process nodes belonging to the current level before moving to children added during this step.
        - **Operations inside Inner Loop**:
            - **Pop**: Remove element from the left (opposite side of push) to maintain FIFO.
            - **Print**: Output the value of the node.
            - **Add Children**:
                - Check if the left child is non-null. If yes, add to queue.
                - Check if the right child is non-null. If yes, add to queue.
                - Important: Add left before right to maintain left-to-right traversal order.
        - **Level Increment**: After the inner loop finishes (level processed), increment the level counter.

- **Detailed Trace Example**

    - **Tree Structure**: Root (4) -> Children (3, 6) -> Grandchildren (2 from node 3; 5, 7 from node 6).
    - **Initialization**: Queue contains.
    - **Level 0**:
        - Queue length is 1.
        - Pop 4, print it.
        - Add Left (3) to queue.
        - Add Right (6) to queue.

        - Inner loop finishes. Level increments.
    - **Level 1**:
        - Queue contains. Length is 2.
        - **Process 3**:
            - Pop 3 (leftmost).
            - Add Left child (2) to queue.
            - Right child is null (do nothing).
        - **Process 6**:
            - Pop 6.
            - Add Left child (5) to queue.
            - Add Right child (7) to queue.
        - Queue now looks like (children of the previous level).
    - **Level 2**:
        - **Process 2**: Pop 2. No children to add.
        - **Process 5**: Pop 5. No children to add.
        - **Process 7**: Pop 7. No children to add.
    - **Termination**:
        - Queue is empty. Length is not greater than zero. Algorithm stops.

- **Time Complexity Analysis**

    - **Confusion regarding Nested Loops**:
        - Seeing a `for` loop inside a `while` loop often leads people to assume the complexity is `O(n^2)`, but this is incorrect for BFS.
    - **Actual Complexity**: **`O(n)`**.
    - **Reasoning**:
        - Every node is traversed exactly once.
        - For each node, a constant number of operations are performed:
            1. Printing the node.
            2. Appending the node to the queue.
            3. Popping the node from the queue.
        - The total operations equal some constant `C` times `N` (number of nodes), which reduces to `O(n)`.
        - The complexity is determined by the size of the data structure (number of nodes).

- **Summary**

    - BFS is likely the second most common tree algorithm after DFS (or potentially the most common).
    - It is a fundamental algorithm that will appear again in future contexts.

BST Sets and Maps

- **Sets and Maps Overview**

    - Sets and maps are data structures that are commonly implemented using binary search trees.
    - **Sets**
        - A set is essentially a collection of values, for example, 1, 2, 3.

        - While it is similar to having an array, the word "set" usually implies that there is a different underlying data structure being used rather than an array.
        - One implementation of a set could be a binary search tree.
            - If you have values 1, 2, 3 in a binary search tree, the structure would look different than an array.
        - The advantage of implementing a set with a binary search tree instead of an array is that you can search for values, insert values, and remove values in `O(LOG n)` time.
    - **Maps**
        - **Concept and Example**
            - To understand why maps are important, consider the example of a phone book sorted in alphabetical order from A to Z based on the name of the person,.
            - Although phone books are not used much anymore, they serve as a good example.
            - A phone book contains a list of names, such as Alice, Brad, and Colin.
            - These names are listed in alphabetical order.
            - The important detail is that we do not just care about the name itself; every name is mapped to something else.
            - This mapping is where the word "map" comes from.
            - In this example, every name is mapped to a phone number.
                - Alice might map to 1 2 3.
                - Brad might map to 4 5 6.
                - Colin maps to something else.
        - **Key-Value Pairings**
            - The goal is to store information while sorting by the name.
            - For every name, we want to possess other information.
            - This is called a key-value pairing.
            - You map a key to a value, and the data structure is typically sorted by the key.

- **Internal Structure and Nodes**

    - **Difference between Sets and Maps**
        - In the simple case of a set using a binary tree, the nodes just have key values.
        - When using a map, each node has a key, which decides how to structure the binary tree for sorting.
        - However, each node in a map also has an associated value.
    - **The Value Component**
        - In the phone book example, the value would be the phone number.
        - The value can also be its own object if desired.
        - It can contain multiple values or all information about a person, such as their phone number, social security number, and email.
    - **Efficiency**
        - The structure is sorted by the key.
        - Searching for someone in the phone book is efficient because we search by the names.
        - We can search in logarithmic time because the data is sorted by name.
        - Once the name (key) is found, we have access to all the information about the person (the value).

- **Implementation and Interfaces**

    - **Underlying Data Structures**
        - Sets and maps can be implemented using binary search trees.
        - The binary search tree acts as the underlying data structure.
        - This is similar to stacks, where the stack is the interface or data structure allowing push and pop operations, but underneath it is implemented with dynamic arrays.
        - Similarly, queues are implemented with linked lists or sometimes dynamic arrays.
        - Sets and maps can also be implemented with other data structures, such as hashmaps and hash sets, which will be discussed later.
    - **Tradeoffs**
        - Depending on the implementation used (e.g., BST versus hashmap), there will be different tradeoffs.
        - For binary search tree implementations, the time complexities are those previously discussed for BSTs.

- **Language-Specific Terminology and Support**

    - **Terminology**
        - Depending on the programming language, these structures have different names.
        - For binary search trees, a set is typically called an ordered set or a tree set.
        - For maps, it is called an ordered map or a tree map.
    - **Java**
        - Java has a built-in tree map.
        - You can create a tree map where the key is a string and the value is a string.
        - You can also use integers and other types.
    - **C++**
        - C++ has native tree maps, which are simply called "maps".
    - **Python**
        - Python does not really have native tree maps.
        - You can install other packages, such as the `sorted dict` package, which is essentially a tree map under the hood.
    - **JavaScript**
        - JavaScript does not really have native tree maps either.
        - There are other packages available for JavaScript that implement tree maps.

- **Interview Context**

    - **Common Scenarios**
        - In real interviews, it is most common to implement a tree map or run an algorithm on a tree map, such as searching or inserting.
        - It is not super common to need the built-in tree map.
    - **Handling Missing Native Support**
        - If you need a tree map and are using Python or JavaScript, the interviewer will most likely let you assume that an object or interface already exists that you can use.
        - You do not necessarily have to implement it from scratch if that is not the specific problem.
    - **Implementation Questions**
        - Sometimes the interview question itself is to implement the data structure or parts of it from scratch.

- **Key Requirements**
  - The most important thing is understanding the interface a tree map would have.
  - You must understand operations like insert, remove, search, iterate, and in-order traversal.
  - You must also understand the associated time complexities of those operations.

Backtracking

Tree Maze

- **Backtracking Algorithm Fundamentals**
  - **Core Concept**
    - The backtracking algorithm pattern is essentially based on the Depth First Search (DFS) algorithm.
    - In this context, it is applied recursively to a regular binary tree (not a Binary Search Tree).
    - The best way to understand the concept is through an example problem.
  - **Problem Scenario: Tree Maze**
    - **Goal**: Determine if a valid path exists from the root of a tree to a leaf node.
    - **Restriction**: The path must not contain any zeros.
    - **Output**: Return `true` if a path exists; return `false` if it does not.
  - **The Maze Analogy**
    - The process is similar to navigating a maze: you start at a point and try to reach the end.
    - You try a path, and if you hit a dead end, you backtrack (go back up) and try a different route.
    - Because you do not know which path leads to the end, you must recursively try every single possibility.
  - **Brute Force Approach**
    - Backtracking is a "brute force" style of algorithm because it goes through every possibility.
    - **Contrast with Binary Search Trees (BST)**:
      - In a BST, you can use the sorted order property to ignore half the tree (e.g., looking for a value > 4 means ignoring the left subtree).
    - **Regular Binary Trees**:
      - In a regular binary tree, you cannot use sorted properties.
      - You are forced to check both the left and right subtrees; zeros act as "roadblocks" where you cannot pass.

- **Algorithm 1: Boolean Return (True/False)**
  - **Execution Logic**
    - Start at the root node.
    - **Check Root**: Verify the node is not a zero, because if the root is zero, no valid path exists.
  - **Step-by-Step Trace (First Example)**
    - **Step 1**: Start at root (value 4). It is not zero.
    - **Step 2**: Check left child.
      - The left child is 0.
      - Zeros are not allowed, so this path is a dead end.

        - **Backtrack**: Go back up to the parent to try another possibility.
    - **Step 3**: Check right child.
        - The right child is 1 (valid, not zero).
        - It is not a leaf node, so continue.
    - **Step 4**: Check left subtree of the current node (1).
        - The value is 2 (valid).
        - It has no children (Leaf Node).
        - **Result**: Path found. Return `true` up to the parent.
    - **Step 5**: Propagation.
        - The parent node receives `true` from the left subtree.
        - Short-circuit: Since the answer was found in the left subtree, there is no need to look in the right subtree.
        - Return `true` all the way up to the root.
- **Failed Path Example**
    - If a node was changed to 0:
        - You would go left, find it is invalid, and return `false`.
        - You would backtrack and try the right subtree.
        - If the right child is also 0, you run out of options.
        - You return `false` to the parent, eventually returning `false` for the whole tree.
- **Code Logic & Base Cases**
    - The code is recursive and similar to standard tree traversals.
    - **Base Case 1 (Failure)**:
        - If the node is `null` (empty) OR the node value is `0`.
        - Return `false`.
    - **Base Case 2 (Success)**:
        - If the node is a leaf node (no children) and you reached it without hitting a zero.
        - Return `true`.
    - **Recursive Step**:
        - Recursively check the left subtree.
        - If the left subtree returns `true`, then return `true` immediately.
        - If the left subtree returns `false`, recursively check the right subtree.
        - If the right subtree returns `true`, propagate `true` up to the root.
        - If both fail, return `false`.

- **Algorithm 2: Returning the Path Values**

    - **Problem Modification**
        - Instead of just returning `true` or `false`, return the actual values of the valid path (e.g., `4, 1, 2`).
    - **Data Structure Setup**
        - **Function Signature**: `leafPath(root, path)`.
        - **Path Variable**: Passed into the function as an array or dynamic array.
        - **Role**: Acts as a global variable or reference passed to every recursive call; effectively used as a stack.
        - **Assumption**: For simplicity, assume there is at most one valid path (zero or one path).
    - **Algorithm Execution with Stack Operations**
        - **Step 1: Push**

        - Start at the root. If not null and not zero, push the current value (e.g., 4) to the `path` array.
        - Current Path: ``.
    - **Step 2: Recurse Left**
        - Check if children exist. If yes, recurse left passing the same path reference.
        - If the left child is 0, the base case executes and returns `false`.
    - **Step 3: Backtrack & Recurse Right**
        - The left side returned `false`.
        - Recurse to the right subtree.
        - (Example Tree Modified: Right child is 1, which has left child 3 and right child 0).
        - Node 1 is valid. Push 1 to path.
        - Current Path: ``.
    - **Step 4: Deep Recursion**
        - From Node 1, check left subtree (Node 3).
        - Node 3 is valid. Push 3 to path.
        - Current Path: ``.
    - **Step 5: Dead End Identification**
        - Node 3 left child is `null` -> returns `false`.
        - Node 3 right child is `0` -> returns `false`.
        - Result: None of the paths from Node 3 work.
    - **Step 6: The Pop Operation (Crucial Backtracking Step)**
        - Because Node 3 did not lead to a solution, we must remove it from the path.
        - **Action**: Pop 3 from the array.
        - Current Path: ``.
        - This step is vital; otherwise, the final answer would contain incorrect nodes.
    - **Step 7: Find Valid Path**
        - Back at Node 1, try the right subtree.
        - Node 2 is found. Valid value. Push 2.
        - Current Path: ``.
        - Check if Node 2 is a leaf: Yes.
    - **Step 8: Final Return**
        - Return `true` from the leaf.
        - Propagate `true` up the chain.
        - The `path` variable is now populated with ``.

- **Summary and Complexity**

    - **General Application**
        - Backtracking can be used on more than just binary trees.
        - It is similar to other recursive algorithms (like Fibonacci) but serves as a foundation for complex problems.
    - **Key Pattern**
        - Maintain a solution (path).
        - Add values while searching.
        - Pop values (backtrack) when a path turns out to be wrong.
    - **Time Complexity**
        - **Complexity**: `O(N)` (Big O of N).

        - **Reason**: In the worst-case scenario, the algorithm must traverse the entire tree (visit every node).
        - **Definition of N**: N is the size of the input tree.
        - This is standard for backtracking as it is essentially a brute force method running over all possibilities.

Heap Priority Queue

Heap Properties

- **Introduction to Heaps and Priority Queues**
    - **The Concept of Priority Queues**
        - A Priority Queue is a data structure distinct from a regular Queue.
        - In a regular Queue, ordering is First-In-First-Out (FIFO).
        - In a Priority Queue, ordering is based on a priority value rather than arrival time.
    - **Variations of Priority Queues**
        - There are typically two variations based on how priority is defined:
            - **Minimum Priority:** Prioritizes smaller values first.
            - **Maximum Priority:** Prioritizes larger values first.
    - **Example of Priority Queue Behavior**
        - Given the values: 7, 3, 9.
        - **Minimum Priority Scenario:**
            - First pop: 3 (the smallest).
            - Second pop: 7.
            - Third pop: 9.
        - **Dynamic Behavior:**
            - The structure is dynamic; adding values changes the pop order.
            - If we add the value 4 to the remaining items (7, 9):
                - Next pop: 4 (since it is smaller than 7 and 9).
                - Following pop: 9 (assuming 7 was already removed or based on remaining order).
        - **Maximum Priority Scenario:**
            - Given 7, 3, 9.
            - First pop: 9.
            - Second pop: 7.
            - Third pop: 3.
    - **Terminology and Relationship**
        - **Interface vs. Implementation:**
            - "Priority Queue" is the name of the interface.
            - "Heap" is the data structure used "under the hood" to implement the Priority Queue.
        - **Historical Context:** Originally, queues were implemented with linked lists, establishing the separation between the interface name and implementation method.
        - **Usage:** People use the terms "Priority Queue" and "Heap" interchangeably.
        - "Heap" is more commonly used in conversation, likely because it is a shorter word.

- **Binary Heap Fundamentals**

    - **Definition**
        - A Priority Queue is implemented using a Binary Heap.
        - A Binary Heap is a binary tree, but it is **not** a Binary Search Tree (BST).
    - **Distinction from Binary Search Trees (BST)**
        - In a BST, every value in the right subtree is larger than the root, and every value in the left subtree is smaller.
        - In a Binary Heap (specifically a Min Heap example with root 14):
            - The root (14) is smaller than every single descendant in the tree, not just the right side.
            - The strict left/right sorting of a BST does not apply.
    - **Types of Heaps**
        - **Min Heap:** Focuses on minimums. The implementation is exactly the same as a Max Heap, just with logic swapped.
        - **Max Heap:** Focuses on maximums.
        - Note: The source material focuses primarily on the Min Heap as it is generally more common.

- **Heap Property 1: The Structure Property**

    - **Complete Binary Tree Requirement**
        - A Binary Heap is essentially a "Complete Binary Tree".
    - **Rules of a Complete Binary Tree**
        - Every single level in the tree must be completely full, with no holes.
        - **Exception:** The last level is allowed to have holes.
        - **Constraint on the Last Level:** If there are missing nodes (holes) in the last level, they must be at the end.
    - **Insertion Order**
        - Nodes are added from Left to Right.
        - It is not enough for only the last level to have missing nodes; the existing nodes must be filled sequentially.
        - Example: You cannot add a node to the far right if the immediate left position is empty.
        - Intuition: You always add nodes in the "next available position".
    - **Examples of Violations**
        - If a node is missing in the second-to-last level, it is not a complete binary tree.
        - If the last level has nodes but there is a gap (hole) between them (e.g., left child exists, middle missing, right child exists), the structure property is not satisfied.

- **Heap Property 2: The Order Property**

    - **Purpose**
        - The goal of a Heap/Priority Queue is to find the minimum (or maximum) value really quickly and easily.
        - Looking at the root allows finding the priority element in O(1) time.
    - **Min Heap Rule**
        - Recursively, for every node, every value in its left subtree and every value in its right subtree must be greater than (or equal to) the node itself.
        - **Root Condition:** The minimum value among all values must be at the root.
    - **Recursive Verification Example**

        - Given a root of 14:
            - Left child (19) is greater than 14.
            - Right child (16) is greater than 14.
        - Checking the subtree starting at 19:
            - Left child (21) > 19.
            - Right child (26) > 19.
        - Checking the subtree starting at 16:
            - Left child (19) > 16.
            - Right child (68) > 16.
    - **Handling Duplicates**
        - Unlike Binary Search Trees, Heaps allow duplicate values.
        - Example: The tree contains two 19s.
        - Refined Rule: All descendants must be greater than or **equal** to the node.
    - **Max Heap Rule**
        - The opposite of a Min Heap.
        - For every node, all descendants must be **smaller** than the node.
        - The maximum value is at the root.

- **Implementation: Arrays**

    - **Underlying Structure**
        - While visualized as binary trees with nodes and pointers, Heaps are actually implemented using **Arrays**.
    - **Indexing Strategy**
        - **Index 0:** Essentially ignored/skipped. We do not care about the zeroth index.
        - **Index 1:** The Root node is always placed at index 1.
    - **Filling the Array**
        - The array is filled level by level, going from left to right.
    - **Mapping Example**
        - Root (14) -> Index 1.
        - Level 2 Left (19) -> Index 2.
        - Level 2 Right (16) -> Index 3.
        - Level 3 (21, 26, 19, 68) -> Indices 4, 5, 6, 7.
        - Level 4 (65, 30) -> Indices 8, 9.
    - **Why Arrays work**
        - Arrays are valid representations specifically because Heaps are **Complete Binary Trees**.
        - If the tree were a regular binary tree with random holes, an array would not make sense due to gaps.
        - Because it is complete, there are never holes in the array sequence.

- **Mathematical Formulas for Traversal**

    - The decision to skip index 0 is to make the math for finding parents and children cleaner.
    - **Finding a Left Child**
        - Formula: `2 * i` (where `i` is the current index).
        - Example: Root at index 1 -> Left child is at `1 * 2 = 2`.
        - Example: Node at index 2 -> Left child is at `2 * 2 = 4`.
    - **Finding a Right Child**

        - Formula: `2 * i + 1`.
        - Example: Root at index 1 -> Right child is at `(1 * 2) + 1 = 3`.
        - Example: Node at index 2 -> Right child is at `(2 * 2) + 1 = 5`.
    - **Finding a Parent**
        - Formula: `i / 2` (Integer division / Round down).
        - **Logic for Left Child:** Since the left child is `2 * i`, dividing by 2 perfectly reverses the operation to return `i`.
        - **Logic for Right Child:**
            - The right child index is always odd (`2 * i` is even, `+ 1` makes it odd).
            - Example: Index 9. `9 / 2 = 4.5`.
            - Most languages (and math logic here) round down (floor).
            - `4.5` rounds down to `4`.
            - `(2 * i + 1) / 2` effectively eliminates the `+1` remainder and the `*2` factor, leaving `i`.
        - **Programming Note:** In Python, use double slashes `//` for floor division.

- **Traversal Example**
    - **Scenario:** Moving through the heap using array math.
        1. Start at Root (Index 1).
        2. Go Left: `1 * 2` -> Index 2.
        3. Go Right: `2 * 2 + 1` -> Index 5.
        4. Go Left: `5 * 2` -> Index 10.
    - **Bounds Checking:**
        - If the calculated index (e.g., 10) is greater than the last index of the array (size of heap), the node does not exist (it is null).
    - **Conclusion on Traversal:**
        - As long as the structure properties are maintained, array indices allow traversal exactly like pointers.

- **Summary of Importance**
    - These notes cover the structure property (Complete Binary Tree) and the order property (Min/Max rules).
    - These properties are critical; without them, the data structure is no longer a heap.
    - Future topics include inserting and removing values while maintaining these specific properties.

Push and Pop

- **Heap Class Initialization**
    - The constructor initializes the heap as an array containing a single value: zero.
    - This zero acts as a dummy value because the zeroth index is not used; this represents an essentially empty heap.

- **Pushing into the Heap (Insertion)**
    - **Concept and Goals**
        - The term "pushing" is used for inserting a new value into a heap or priority queue.

        - **Structure Property:** To keep the tree contiguous, a new node must be placed in the next available position (the end of the array).
        - **Order Property (Min Heap):** Every node must be smaller than its children; conversely, a child must be greater than its parent.
    - **Step-by-Step Algorithm: Percolate Up**
        - **1. Placement:** Append the new value to the end of the dynamic array.
            - For example, inserting 17 into a heap of length 10 puts it at index 10.
        - **2. Comparison:** Compare the new node with its parent to ensure the order property is met.
            - The parent's index is found by taking the current index `i` and dividing by two (integer division),.
        - **3. Swapping:**
            - If the child is smaller than the parent (e.g., 17 is smaller than its parent), swap the two values in the array.
            - After swapping, the new node moves up the tree.
        - **4. Iteration:**
            - Continue comparing the node with its new parent.
            - If the parent is still larger than the child, swap again.
            - **Optimization Logic:** When moving a node up (e.g., replacing 19 with 17), we do not need to compare it to the other branch's children because the previous occupant (19) was already valid (smaller than its descendants); therefore, the new smaller value (17) is automatically valid against those descendants,.
        - **5. Termination:** The algorithm stops when either:
            - The node reaches the root (index 1) and has no parent.
            - The order property is satisfied (the parent is smaller than or equal to the child).
    - **Code Implementation Details**
        - Calculate the index of the newly inserted element: `len(heap) - 1`.
        - Use a loop to "percolate up" while the order property is violated.
        - Inside the loop:
            - Compare index `i` with parent `i // 2`.
            - If `heap[i] < heap[i // 2]`: Swap the values.
            - Update `i` to `i // 2`.
    - **Time Complexity**
        - The complexity is **log n**, where n is the number of values in the heap.
        - This is determined by the height of the tree, which is guaranteed to be balanced in a complete binary tree.

- **Popping from the Heap (Removal)**

    - **Concept and Goals**
        - Popping removes the priority element, which is always the minimum value located at the root (index 1).
        - Just reading the top element is simple, but removing it is complicated because structure and order properties must be maintained.
    - **Why the Naive Approach Fails**
        - If you remove the root and replace it with the smaller of its two children, you create a "hole" at the next level.

        - Shifting nodes up to fill the hole eventually leaves a gap at a random spot, violating the structure property (levels must be full).
    - **The Correct "Genius" Technique**
        - **1. Replacement:** Remove the root node (e.g., 14) and replace it with the **last value** in the array (e.g., 30).
        - **2. Structure Satisfied:** This keeps the array contiguous, satisfying the structure property immediately.
        - **3. Order Violated:** The new root (30) is likely larger than its children, violating the order property.
        - **4. Percolate Down:** Shift the new node down until it fits the order property.
    - **Step-by-Step Algorithm: Percolate Down**
        - Take the current node (originally the last value) and compare it against its left and right children.
        - Find the minimum of the two children (e.g., between 16 and 19, the minimum is 16).
        - Compare the minimum child to the current node.
        - If the minimum child is smaller than the current node:
            - Swap the current node with that minimum child.
            - For example, swap 30 with 16.
        - Repeat this process recursively down the tree until the node is smaller than its children or has no children.
    - **Code Implementation Details**
        - **Empty/Small Heap Checks:**
            - If length is 1: Return null (heap is empty).
            - If length is 2: Remove and return the single value (root),.
        - **Setup for General Case:**
            - Save the root value (result) to a variable.
            - Move the last value of the array to index 1 (the root).
            - Set pointer `i` to 1.
        - **While Loop Logic:**
            - Run loop while a **left child exists** (`2 * i < len(heap)`).
            - **Case A: Swap with Right Child**
                - Conditions:
                    1. Right child exists (`2 * i + 1 < len`).
                    2. Right child is smaller than Left child.
                    3. Current node is greater than Right child.
                - Action: Swap current node with Right child and update `i` to `2 * i + 1`,.
            - **Case B: Swap with Left Child**
                - Conditions: Case A did not run, AND Current node is greater than Left child,.
                - Action: Swap current node with Left child and update `i` to `2 * i`.
            - **Case C: Stop (Break)**
                - Condition: The current node is already in the proper position (smaller than both children).
                - Action: Break out of the loop.
        - **Return:** Finally, return the saved original root value.
    - **Time Complexity**
        - The complexity is **log n**, which is the height of the balanced binary tree,.

        - This cost comes from the percolate down process.

Heapify

- **Heaps vs. Binary Search Trees (BST)**
    - **Advantages of Heaps**
        - **Accessing Min/Max:**
            - In a heap, you can get the minimum or maximum element (depending on implementation) in **constant time**.
            - In a Binary Search Tree (BST), this takes **log n time** because you must traverse all the way to the left of the tree.
        - **Building the Structure:**
            - **BST Construction:** Requires inserting a node every single time. Time complexity is **n log n** (n insertions * log n time per insertion).
            - **Heap Construction:**
                - Can be built by pushing elements one by one (taking n log n time).
                - **Heapify (Build Heap):** A special algorithm that builds a heap in **linear time** (`O(N)`),.
- **The Heapify Algorithm (Build Heap)**
    - **Concept**
        - Takes a set/list of values in no particular order (e.g., an array) and turns them into a heap satisfying heap properties.
        - Time Complexity: `O(N)`.
    - **Initial Structure Property Adjustment**
        - Input arrays often do not satisfy the structure property initially (e.g., having a real value at the 0th index).
        - **Step:** Move the element at the 0th index to the last position.
        - Result: The array satisfies the structure property (e.g., 50 at root, followed by 80, 40, etc.).
    - **Satisfying the Order Property**
        - **Definition:** For a Min Heap, every node must have a value smaller than all of its descendants (recursive property). Max heaps are the exact opposite.
        - **Strategy:** Start at the bottom nodes and work upwards.
            - Leaf nodes (nodes without children) do not require comparison.
            - Approximately half of the nodes are leaf nodes and can be skipped.
- **Heapify Step-by-Step Walkthrough**
    - **Calculating the Starting Point**
        - Get the number of actual nodes (excluding dummy/0-index).
        - **Formula:** `(Length - 1) / 2` (Integer division/rounding down).
        - **Example:**
            - If length is 10 (9 actual nodes), `9 / 2 = 4`.
            - Start at **index 4**. This is the first node that actually has children.
        - Note: You can start at the end and go backward, but calculating the midpoint allows skipping leaf nodes.

- **Percolating Down (The Process)**
  - The algorithm iterates through nodes that have children, performing the "percolate down" operation if necessary (same as when removing/popping nodes).
  - **Example Trace:**
    - **Node at Index 4 (Value 30):**
      - Check: Is 30 smaller than children (90 and 60)?
      - Result: Yes. No swap needed.
    - **Node at Index 3 (Value 40):**
      - Check: Is right child smaller than left? Yes.
      - Check: Is right child smaller than parent (40)? Yes.
      - Action: Swap 40 with the right child.
      - Result: This subtree is now valid.
    - **Node at Index 2 (Value 80):**
      - Check: Has two children. Right child (10) is smaller than left.
      - Check: Is 10 smaller than 80? Yes.
      - Action: Swap 80 with 10.
      - Subtree is now valid.
    - **Node at Index 1 (Root, Value 50):**
      - Check children: 10 is smaller than the other child.
      - Check: Is 10 smaller than 50? Yes.
      - Action: Swap 50 and 10.
      - **Continue Percolating 50:**
        - Compare children at new position. Smaller child is 30.
        - Check: Is 50 bigger than 30? Yes.
        - Action: Swap 50 and 30.
      - **Continue Percolating 50:**
        - Compare children at new position. Smaller child is 60.
        - Check: Is 50 bigger than 60? No.
        - Result: 50 stays. Correct spot found.
- **Final State**
  - The array represents a valid heap.
  - Smallest value (10) is at the root.
  - All nodes (20, 30, 50, etc.) are smaller than their descendants.

- **Code Implementation Logic**

  - **Setup:**
    - Input: An array.
    - Operation: `append` the 0th element to the end to shift positions (satisfies structure).
    - Assign array to be the heap.
  - **Loop Logic:**
    - Calculate starting index: `current = (len(heap) - 1) // 2`.
    - **While Loop:**
      - Create a copy of `current` (e.g., `i`).
      - Perform **Percolate Down** on `i` (checking children, swapping if needed until valid).
      - Decrement `current` by 1 (`current - 1`) to move in reverse order toward the root.

        - **Helper Function:** In a class implementation, "percolate down" logic is usually a helper function because it is also used in `heap_pop`.

- **Time Complexity Analysis: Why O(N)?**
    - **Misconception:** At first glance, it looks like `O(N LOG N)` because we visit every node and percolate down (log n operation).
    - **Intuition:**
        - **Percolating Down vs. Up:**
            - Percolating **up**: Leaf nodes (the majority) must travel the full height of the tree.
            - Percolating **down**:
                - Only the root travels the full height.
                - The bottom level (leaves) travel distance 0.
                - The level above leaves travels distance 1.
            - Shifting down is more efficient because the majority of nodes (at the bottom) do little to no work.
    - **Mathematical Explanation:**
        - In a complete binary tree, roughly `N/2` nodes are at the last level (height 0). Work: `0 * (N/2)`.
        - Roughly `N/4` nodes are at the next level. Work: `1 * (N/4)`.
        - Summation series: The sum of these terms converges to roughly **N**.
    - **Conclusion:** The Big O complexity is **`O(N)`** (linear time). Note: You typically do not need to prove this, just know it.

- **Heap Sort and Searching**
    - **Heap Sort:**
        - Building a heap takes **`O(N)`**.
        - To output a sorted array, you must **pop** every single element.
        - Popping takes **`LOG N`**. Doing this `N` times takes **`N LOG N`**.
        - Total Time: **`N LOG N`** (similar to Merge Sort).
    - **Searching in a Heap:**
        - **Disadvantage:** Heaps are **not good for searching** specific values.
        - **Reason:** Unlike BSTs, there is no strict left/right logic (e.g., a value of 30 could be in either the left or right subtree).
        - **Complexity:** Searching requires checking every node: **`O(N)`**.
        - BST Search Complexity: **`O(LOG N)`**.
        - **Purpose:** Heaps are intended for priority access (Min/Max), not random searching.

- **Usage in Coding Interviews**
    - **Frequency:** Heaps are used more often than BSTs as a utility data structure in problem-solving.
    - **Context:**
        - BST problems usually involve implementing BST logic.
        - Heap problems usually involve using the heap to find min/max values efficiently.
    - **Key Concepts to Know:**
        - **Heapify:** Runs in linear time (`O(N)`).
        - **Push/Pop:** Runs in log time (`O(LOG N)`).

        - **Get Min/Max:** Runs in constant time (`O(1)`).

Hashing

Hash Usage

- **Introduction to Hash Data Structures**
    - Hashmaps and hash sets are likely the most common data structures you will ever use.
    - They are extremely useful for coding interviews, real jobs, and projects.
    - It is more important to understand the application (how to use them) than the specific implementation details.
- **Concept: Sets vs. Maps**
    - **Sets**
        - Sets are essentially a collection of values, such as 1, 2, 3.
    - **Maps**
        - Maps are slightly more complicated than sets.
        - A map consists of a set of keys which are mapped to separate values.
        - **Example:** A phone book is a map where a name (key) is mapped to a phone number (value).
    - **Key Characteristics**
        - The main ideas behind maps are nearly identical to sets.
        - The "key" is used to access values in a hashmap or hash set.
        - In binary search trees, the key is what is used to sort the tree.
        - Maps are generally more common and complex than sets.
- **Comparison: Tree Maps vs. Hashmaps**
    - **Tree Maps**
        - **Insertion:** Runs in `O(LOG n)` time.
            - This is better than maintaining a sorted array, which has a worst-case insertion of `O(n)`.
        - **Removal:** Runs in `O(LOG n)` time.
        - **Searching:** Runs in `O(LOG n)` time using binary search.
        - **Ordering:** Tree maps are ordered, allowing iteration in sorted order.
            - This in-order traversal takes `O(n)` time.
    - **Hashmaps**
        - **Insertion:** Runs in `O(1)` constant time.
        - **Removal:** Runs in `O(1)` constant time.
        - **Searching:** Runs in `O(1)` constant time.
        - **Performance:** Hashmaps generally outperform tree maps significantly regarding these operations.
- **Time Complexity Details for Hashmaps**
    - **Average vs. Worst Case**
        - The `O(1)` complexity is actually the **average-case** time complexity.

        - In the worst case, inserting, removing, and searching can be `O(n)` depending on the implementation.
        - However, in coding interviews and jobs, people assume these operations are constant time `O(1)`.
    - **Trade-offs and Downsides**
        - Hashmaps do not maintain any ordering.
        - You cannot iterate through keys in sorted order in `O(n)` time.
        - To iterate in order, you must take all keys and sort them (e.g., merge sort), which typically takes `N LOG N` time.
        - Despite this downside, the positive aspects (fast insert/search) usually outweigh the negatives, especially when order is not required (like a phone book).

- **Use Case Example: Counting Name Frequencies**

    - **Problem Statement**
        - Given a list (array) of names, count how many times each name appears.
    - **Strategy**
        - Use a hashmap to map every name (Key) to an integer count (Value).
    - **Step-by-Step Execution**
        - **Initialization:** Start with an empty hashmap.
        - **1. Processing "Alice"**
            - Search checks if "Alice" exists (not found).
            - Insert "Alice" into the map with a value of 1.
            - This operation is `O(1)`, whereas a tree map would be `O(LOG n)`.
        - **2. Processing "Brad"**
            - Search checks for "Brad" (not found).
            - Add "Brad" as a key with a value of 1.
        - **3. Processing "Colin"**
            - Search checks for "Colin" (not found).
            - Add "Colin" with a value of 1.
        - **4. Processing "Brad" (Duplicate)**
            - Search finds that "Brad" already exists in the map.
            - Instead of assigning 1, retrieve the current count (1) and increment it by 1.
            - Overwrite the value to become 2.
            - **Note:** Hashmaps cannot contain duplicates; it does not make sense to have multiple copies of the same key.
        - **5. Processing "Dylan"**
            - Search checks for "Dylan" (not found).
            - Add "Dylan" with an initial count of 1.
        - **6. Processing "Kim"**
            - Search checks for "Kim" (not found).
            - Add "Kim" with a count of 1.

- **Algorithm Efficiency Analysis**

    - **Time Complexity**
        - The algorithm runs in **`O(n)`** linear time.
        - For each position in the input array, we performed a few `O(1)` operations.

        - Comparison: A tree map would take `LOG n` for each insertion, resulting in a total time of `N LOG N`.
    - **Space Complexity**
        - The space complexity is **`O(n)`**.
        - In the worst case, every single name is unique, making the map size proportional to `n`.
        - This space usage is roughly the same as a tree map.

- **Implementation Code Logic**

    - **Python Example**
        - Create the map using the syntax `countMap = {}`.
        - In other languages, you usually use a constructor.
        - Iterate through every name in the list of names.
    - **Logic Flow**
        - **Condition:** Check if the name is **not** in the map.
            - Python Syntax: `if name not in countMap`.
            - Other Languages: Use a method like `contains(key)`.
        - **Action (If New):** Assign the name an initial value of 1.
        - **Action (If Exists):** Increment the existing count by 1.
    - This results in a straightforward algorithm running in linear time.

Hash Implementation

- **Introduction to Hashmap Implementation**

    - **General Concept**
        - Implementing a hashmap from scratch is rare in interviews and can become very complicated.
        - It is better to focus on the general concepts (hashing, key-value pairs) as they appear frequently in software engineering.
        - Under the hood, a hashmap is implemented using an **array**.
        - An index in the array maps to a specific key-value pair.
        - Even an empty hashmap has a non-zero size array.
    - **Initialization Example**
        - An empty hashmap might start with an array of size two (indices 0 and 1).
        - The default value in these slots is `null` (or empty), indicating nothing is stored there yet.
    - **Insertion Basics**
        - The operation is often called `put` or `insert`.
        - It typically involves a key (e.g., a string, integer, or object) and a value.
        - The critical requirement is having a way to convert the **key** into an integer.

- **The Hashing Process**

    - **Hashing Function**
        - A hashing function takes a key and converts it into an integer.
        - This integer is used as the index to determine where to store the key-value pair in the array.
        - The array actually stores objects, where each object is a key-value pair.
    - **Converting Characters to Integers**

        - A simple method maps each character to an integer (e.g., A=0, L=11).
        - You sum these integers to get a large result (e.g., resulting in the integer 25).
        - In implementation, ASCII values are often used (e.g., 'B' might be 70, 'b' might be 44).
        - This works well for small alphabet sizes (like 128 characters).
    - **The Modulo Operator (Handling Array Bounds)**
        - The integer resulting from the sum (e.g., 25) is often too large for the specific array size (e.g., size 2).
        - To fix this, use math: take the integer and **mod** it by the size of the array.
        - Modding means dividing and taking the remainder (e.g., 25 / 2 = 12 with a remainder of 1).
        - **Rule:** Modding by the length of the array always results in a valid index.
        - The remainder is always less than the number you are modding by (e.g., mod 5 results in 0–4).
    - **Example**
        - Key: "Alice". Hashed Integer: 25. Array Size: 2.
        - Calculation: 25 mod 2 = 1.
        - "Alice" (key) and "New York City" (value) are inserted at index 1.
    - **Retrieval Efficiency**
        - To `get` "Alice", the same process occurs: Hash "Alice" to 25, mod by 2, get index 1.
        - This allows jumping directly to the index in constant time, **O(1)**.

- **Collisions**

    - **Definition**
        - A collision occurs when two different keys result in the same index after hashing and modding.
        - Example: Inserting "Brad". Hashed integer = 27.
        - Calculation: 27 mod 2 = 1.
        - Problem: Index 1 is already occupied by "Alice".
    - **Inevitability**
        - You cannot entirely avoid collisions; you can only minimize them and work around them.
    - **Minimizing Collisions (Resizing)**
        - If a hashmap is half full (e.g., 1 key in an array of size 2), there is a 50/50 chance of a collision.
        - To improve odds, the array is resized when it becomes half full.
        - Resizing works like dynamic arrays: roughly double the size.
        - Example: After inserting "Alice", the map is half full, so the array is resized to size 4 (indices 0, 1, 2, 3).

- **Rehashing**

    - **The Problem with Resizing**
        - When array size changes, the modulo divisor changes (e.g., from 2 to 4).
        - An index valid in the old array might change in the new array.
        - Example: If "Alice" hashed to 27 (instead of 25), 27 mod 2 = 1. But 27 mod 4 = 3.
        - If you do not move the item, you will look for "Alice" at index 3 but she will still be at index 1.
    - **The Rehash Process**

        - When resizing, you must **recompute the hash** for every single key using the *new* size.
        - Move every element to its new correct position.
        - This is computationally expensive (iterating through every key) but happens infrequently (average case efficient).
    - **Example with Resizing**
        - Map resized to 4. "Brad" (27) is inserted.
        - Calculation: 27 mod 4 = 3. No collision. Brad goes to index 3.
        - "Alice" (25) stays at index 1 because 25 mod 4 is still 1.

- **Handling Collisions**
    - Even after resizing to minimize chances, collisions still happen.
    - Example: Resized to 8. Insert "Colin" (hash 33).
    - Calculation: 33 mod 8 = 1. "Alice" is already at index 1. This is a collision.
    - **Method 1: Chaining (Linked Lists)**
        - Instead of a single pair, store a **linked list** of pairs at every index.
        - Multiple pairs can occupy the same index.
        - **Get Operation with Chaining:**
            - Go to the index (e.g., 1).
            - Traverse the list: Is this Alice? Yes -> ignore. Is this Colin? Yes -> return value "Seattle".
        - **Downside:** Requires maintaining memory/pointers for the list.
    - **Method 2: Open Addressing**
        - If the calculated index is occupied, look for the next available position.
        - **Linear Probing:**
            - Try `index + 1`. If occupied, try `index + 2`, etc..
            - Example: Colin (hash 33, index 1) is blocked by Alice. Try index 2. Empty? Insert Colin at index 2.
        - **Get Operation with Open Addressing:**
            - Look for Colin at index 1. Found Alice. Not Colin.
            - Check next index (2). Found Colin. Return "Seattle".
        - **Determining Non-Existence:**
            - Look for "Dylan" (hash maps to 1). Index 1 is Alice.
            - Check Index 2. Index 2 is Colin.
            - Check Index 3. Index 3 is Empty (Null).
            - Conclusion: Since we hit an empty spot, Dylan does not exist in the map (logic stops).
        - **Downside (Clustering):**
            - Naive open addressing (plus one) leads to clustering, where keys are stored very close together.
            - Better methods exist (e.g., squaring values), but are math-heavy.

- **Optimization: Prime Numbers**
    - For math reasons, maintaining the array size as a **prime number** results in fewer collisions.
    - Instead of size 8, use size 7.
    - When doubling, find the next prime number roughly double the current size (e.g., 7 -> 17 -> 37).
    - Note: Implementing prime number logic perfectly is rare in interviews due to complexity.

- **Code Implementation Breakdown**
  - **Class Structure**
    - Define a class to store the pair: `key` and `value`.
  - **Initialization (`__init__`)**
    - `size`: Tracks number of keys inserted (initially 0).
    - `capacity`: Size of the array (initially 2).
    - `map`: The array itself, initialized with `null` values `[None, None]`.
  - **Hashing Function (`hash`)**
    - Input: Key (string).
    - Logic: Loop through every character, convert to integer (ASCII), sum them into a variable `index`.
    - Final Step: `return index % self.capacity` to ensure the result is within bounds.
  - **Search / Get (`get`)**
    - Compute `index = self.hash(key)`.
    - Loop (Wait for `True` or conditioned on capacity) to handle open addressing.
    - **Check 1:** If `map[index]` is `None` (empty), return `None` (Key not found).
    - **Check 2:** If `map[index].key == key`, return `map[index].value` (Found).
    - **Open Addressing Logic:** If occupied by a different key, increment index: `index = (index + 1) % self.capacity`.
  - **Insertion (`put`)**
    - Compute `index = self.hash(key)`.
    - Loop to find slot:
      - **Case 1 (Empty Slot):** If `map[index]` is `None`:
        - Create new pair.
        - Assign to `map[index]`.
        - Increment `self.size`.
        - **Check Resize:** If `self.size >= self.capacity`, call `self.rehash()`.
        - Return.
      - **Case 2 (Key Exists/Update):** If `map[index].key == key`:
        - Overwrite the value.
        - Return (do not increment size).
      - **Case 3 (Collision):** Slot occupied by different key:
        - `index = (index + 1) % self.capacity` (Try next slot).
  - **Rehash Function (`rehash`)**
    - Save current map to `old_map`.
    - Update `self.capacity` (e.g., `2 * self.capacity`).
    - Create `new_map` with new capacity (filled with `None`).
    - Reset `self.size` to 0 (because `put` will re-increment it).
    - Assign `self.map = new_map`.
    - **Re-insertion:**
      - Loop through every pair in `old_map`.
      - If pair is not `None`:
        - Call `self.put(pair.key, pair.value)`.
        - This automatically recomputes the new hash index based on the new capacity.

- **Summary of Complexity**
    - Operations (`insert`, `remove`, `search`) run in **O(1)** time on average.
    - While the analysis is complex, assuming O(1) is standard for interviews.
    - Understanding the concepts (collisions, rehashing, open addressing) places you ahead of most candidates.

Graphs

Introduction to Graphs

- Introduction to Graphs
    - General Overview
        - Graphs are a really common topic in real coding interviews and are considered pretty challenging.
        - A lot of material needs to be covered regarding graphs.
        - Linked Lists are actually a form of graphs; they are a subset of graphs.
        - Trees (binary trees or other kinds) are also a subset of graphs.
    - Definition and Components
        - A graph is essentially made up of nodes and possibly some pointers connecting them together.
        - Nodes: These can also be referred to as "vertices".
        - Vertices: This is a synonym for nodes; a vertex is essentially the same thing as a node typically.
        - Edges: These are the pointers connecting the nodes together.
- Graph Shapes and Constraints
    - Connectivity and Cycles
        - Graphs can have all kinds of shapes.
        - Unlike binary trees or binary search trees, generic graphs have no restrictions regarding cycles.
        - A cycle occurs when you can follow edges from a node and eventually return to that same node (e.g., A to B to C and back to A).
        - In a generic graph, there are no restrictions on the number of nodes or the edges connecting them.
    - Mathematical Constraints on Edges
        - Technically, the number of edges (E) is less than or equal to the number of vertices (V) squared (`E \le V^2`).
        - V refers to the number of vertices (nodes).
        - Example: If there are 3 nodes, `3^2 = 9`, so the number of edges is less than or equal to 9.
    - Logic Behind the Formula
        - From every single vertex (singular of vertices), there can be a pointer going to every other vertex.
        - A node can also have a pointer going into itself (a self-loop).

        - Since every node can have V edges (where V is the total number of nodes), the maximum is `V * V` or `V^2`.
        - Duplicate edges are typically not considered in graphs.

- Directed vs. Undirected Graphs

    - Directed Graphs
        - A directed graph means that the pointers (edges) have a specific direction.
        - You might be able to go from B to C, but not necessarily from C to A.
        - Trees and Linked Lists are examples of directed graphs.
        - Cycles can form in directed graphs if pointers allow going back and forth between nodes (e.g., A to B and B to A).
    - Undirected Graphs
        - Undirected edges mean you can go in either direction between connected nodes.
        - Drawing style 1: A line with arrows on both ends.
        - Drawing style 2 (More common): A line without any direction indicators.
        - If an undirected graph is drawn with plain lines, it implies movement is possible both ways for every edge.

- Graph Representations

    - Overview

        - There are three common ways to represent graphs.
        - The most common ways for coding interviews are the Matrix and the Adjacency List.
        - The Adjacency Matrix is much less common.

    - Representation 1: Matrix (2D Grid)

        - Basic Structure
            - This is essentially a two-dimensional array used to represent a graph.
            - In languages like Python, it appears as an array of arrays (rows).
            - Example structure:

```
[
  ,

]
```

        - Coordinate Systems
            - Using X and Y for coordinates can get confusing because in math, X usually refers to a row (horizontal) and Y to a column (vertical), which might conflict with array indexing.
            - It is recommended to use R (Row) and C (Column) to avoid confusion during interviews.
        - Accessing Values
            - Everything is indexed by zero.
            - To access a coordinate: `grid[row][column]`.

        - Accessing `grid` returns the entire one-dimensional array at index 1 (the second row).
        - Accessing `grid` accesses the value at column 2 within row 1.
    - Logic as a Graph
        - Values in the matrix (e.g., 0 and 1) can define rules.
        - Example: 0 represents a free space (a node you can move to), and 1 represents a blocked space.
        - Nodes: The free spaces (zeros) act as the nodes.
        - Edges: Movement rules define the edges (e.g., allowed to move Left, Right, Up, Down).
        - Edge Direction: Typically, these edges are undirected (you can move left and then move back right).
        - Null Pointers: If a movement leads out of bounds or to a blocked space, it is essentially a null pointer.
        - Implicit Nature: The matrix does not store explicit pointers; edges exist because of the defined movement rules.

- Representation 2: Adjacency Matrix

    - Basic Structure
        - This is typically a square matrix with dimensions `V * V` (where V is the number of vertices).
        - The dimensions represent the nodes themselves (e.g., Row 0 represents Vertex 0, Column 0 represents Vertex 0).
    - Storing Edges
        - The values inside the matrix represent the existence of edges.
        - 0: There is no edge going from the source vertex to the destination vertex.
        - 1: There is an edge going from the source vertex to the destination vertex.
    - Indexing and Direction
        - If accessing `matrix[i][j]`, `i` represents the starting vertex and `j` represents the ending vertex.
        - Example: `matrix = 1` means there is an edge from Vertex 1 to Vertex 0.
        - This represents a directed edge.
        - To check the reverse direction (0 to 1), you must check `matrix`.
        - Self Loops: An edge from a node to itself is represented by a 1 at the index `[i][i]` (e.g., `matrix`).
    - Space Complexity
        - Space complexity is `O(V^2)` because it requires an entire matrix regardless of the number of actual edges.
        - Even if a graph has very few edges (e.g., 4 edges for 4 nodes), the matrix still takes up `V * V` space.
        - This inefficiency makes it rare to use adjacency matrices compared to other methods.

- Representation 3: Adjacency List

    - Overview
        - This is typically the most common way of representing graphs in coding interviews.

        - It works similarly to Linked Lists and Trees.
    - Node Structure
        - A graph node (vertex) usually contains two things:
            - A value: Can be an integer, character, string, or an object (like a "Person" or "City").
            - Pointers: A way to connect to other nodes.
    - Handling Pointers
        - Unlike Binary Trees (left/right) or Linked Lists (next/prev), generic graphs can have any number of pointers.
        - Implementation: Use a list (array) to store pointers.
        - In Python, this list is often called `neighbors`.
    - Logic
        - The `neighbors` list represents outgoing edges (which nodes this specific node is pointing to).
        - Example 1: If Node 0 has no outgoing edges, its `neighbors` list is empty.
        - Example 2: If Node 1 points to itself and Node 0, its `neighbors` list contains pointers to Node 1 and Node 0.
        - Example 3: If Node 3 points to Node 1, Node 1 is in Node 3's neighbor list (but Node 3 is not necessarily in Node 1's list unless the edge goes back).
    - Space Efficiency
        - This method is more space-efficient than an adjacency matrix.
        - It only contains pointers for nodes that actually exist and are connected.
        - Space corresponds to nodes plus edges, rather than nodes squared.
    - Undirected Graphs in Adjacency Lists
        - If edges are undirected, the connection is stored in both nodes' neighbor lists.
        - Example: If there is a connection between Node 0 and Node 1:
            - Node 1 is in Node 0's `neighbors`.
            - Node 0 is in Node 1's `neighbors`.

Matrix DFS

- **Problem Definition and Goals**
    - The objective is to count the number of unique paths from the top-left of a matrix to the bottom-right.
    - The algorithm used is Depth First Search (DFS), which is very commonly applied to graphs.
    - **Movement Rules**
        - The path can only move along zeros (0).
        - The path cannot move on ones (1) because they are blocked.
        - A single path cannot visit the same cell more than once.
    - **Why avoid revisiting cells?**
        - If we revisit a cell (e.g., going down, left, and back up), we could enter an infinite loop.
        - Without this rule, the answer would basically be infinity, which is not useful.
        - If a cell is already visited on the current path, it is treated as "blocked" for that specific path.
- **Algorithm Strategy: DFS and Backtracking**

    - To count the paths, the algorithm must go through every single possibility.
    - This approach is an example of backtracking, which has a big overlap with DFS.
    - Both DFS and backtracking are recursive algorithms.
    - **Visualizing the Approach**
        - It is best to think about the problem visually as a matter of choices.
        - We start at the top-left position (0, 0).
        - From any position, there are four possible choices: Move Up, Move Left, Move Down, or Move Right.
        - The algorithm explores one path as deep as possible, backtracks, and then explores other paths.

- **Function Parameters and Inputs**
    - The function receives the `grid` (the two-dimensional matrix).
    - The function receives the starting position coordinates (row, column), initially `0, 0`.
    - **Matrix Dimensions**
        - `ROWS` is the length of the grid.
        - `COLS` is the length of the first row (assuming a rectangular grid).
        - These dimensions can be calculated inside the function or passed as parameters; calculating them inside does not change time complexity.
    - **The Visit Hash Set**
        - A `visit` hash set is passed to the recursive function to track cells visited on the current path.
        - This object is reused across recursive calls; a new one is not created every time.
        - Adding and removing from a hash set is a constant time operation.
        - Alternatively, a 2D grid of the same size could be used to mark visited positions.
        - Modifying the input grid (e.g., changing visited cells to 1) is possible but not always safe to assume allowed.

- **Base Cases (Stopping Conditions)**
    - The algorithm has multiple base cases where it must stop and return a value.
    - **1. Out of Bounds**
        - If the row is less than 0 or the column is less than 0, the path is invalid.
        - This can be checked by seeing if the minimum of row and column is less than 0.
        - If the row equals the number of rows (too big), it is out of bounds.
        - If the column equals the number of columns (too big), it is out of bounds.
        - In these cases, return `0` because no path was found.
    - **2. Blocked Position**
        - If the current position is blocked (contains a 1), we are not supposed to be there.
        - We stop searching immediately and return `0`.
    - **3. Already Visited**
        - If the current position is in the `visit` hash set, it means we visited it before on this path.
        - Visiting twice is not allowed, so return `0`.
    - **4. Destination Reached**
        - The destination is defined as the last row (`rows - 1`) and the last column (`cols - 1`).
        - If `r == rows - 1` and `c == cols - 1`, we have found a single valid path.
        - Return `1` to count this path.

- **The Recursive Step (The "Count" Logic)**
  - If no base cases are met, we proceed with the recursive exploration.
  - **Mark as Visited**
    - Add the current coordinate pair to the `visit` hash set.
  - **Explore Directions**
    - Initialize a `count` variable to 0.
    - Call DFS recursively in all four directions:
      - Down: `row + 1`.
      - Up: `row - 1`.
      - Right: `column + 1`.
      - Left: `column - 1`.
    - The `count` variable accumulates the results returned from these four directions.
  - **Backtracking (Unmark Visited)**
    - After exploring all directions from the current cell, we must backtrack.
    - Remove the current position from the `visit` hash set (mark as unvisited).
    - **Why Unmark?**
      - While a single path cannot visit a node twice, different unique paths can share the same nodes.
      - Example: One path might go down-right, and another might go right-down; they are distinct paths even if they cross the same cells.
      - By unmarking, we allow other paths to use this cell later.
  - **Return Result**
    - Return the total `count` to the previous node (the caller).
    - The final result returned to the original caller will be the total number of unique paths.

- **Code Flow Example**
  - Start at (0,0) with an empty visit set.
  - Check base cases (bounds, visited, blocked, destination); if valid, add (0,0) to visit set.
  - Recursively try directions (e.g., Down, Up, Right, Left) one by one.
  - If "Up" is out of bounds, it returns 0.
  - If "Down" is blocked, it returns 0.
  - If "Right" is valid, proceed recursively from the new cell.
  - This process continues deeper until a base case returns 0 or 1.
  - Values (0 or 1) bubble up the recursion tree, summing up at each node.

- **Complexity Analysis**
  - **Time Complexity: O(4^(N * M))**
    - This is an imprecise upper bound based on the decision tree.
    - At every node, there are 4 choices (branches).
    - The maximum height of the decision tree corresponds to the longest possible path, which is the size of the matrix (Rows * Cols, or N * M).
    - Therefore, the complexity is roughly `branches ^ height`, or `4^(N * M)`.
    - This is not efficient, but expected for brute force DFS backtracking.
  - **Space Complexity: O(N * M)**
    - The memory complexity is determined by the recursive call stack.

        - In the worst case, the recursion depth can equal the size of the matrix (`N * M`).
        - Additionally, the `visit` hash set (or grid) stores up to `N * M` elements.

Matrix BFS

- **Matrix BFS (Breadth-First Search) Algorithm Overview**
    - **Introduction**
        - The BFS algorithm on a matrix involves more code than Depth-First Search (DFS).
        - However, it is visually simpler to understand than DFS.
        - Similar to BFS on a tree, the algorithm processes nodes layer by layer.
        - It goes through the first layer, then the second layer, then the third layer, and so on.
    - **Common Use Cases**
        - The most common application is the **shortest path algorithm**.
        - Example: Finding the length of the shortest path from the top-left to the bottom-right of a grid.
        - BFS is the most efficient way to solve this.
        - DFS could be used as a brute force approach to check every possible path and find the shortest one, but it is much less efficient.
    - **Time and Space Complexity**
        - **BFS Time Complexity**: `O(N * M)`.
            - This represents the size of the grid (Rows `*` Columns).
            - In the worst case, the algorithm visits the entire grid.
        - **DFS Time Complexity**: `4^{(N * M)}`.
            - This is significantly less efficient.
        - **Space Complexity**: `O(N * M)`.
            - This is the maximum size that the visit hash set or the queue could reach.
- **Algorithm Setup and Data Structures**
    - **Initial Inputs**
        - Dimensions of the grid.
        - Starting point (e.g., top-left).
        - Destination (e.g., bottom-right).
    - **Required Data Structures**
        - **Visit Hash Set**: Used to track visited nodes, similar to DFS.
        - **Queue**: Used to track the current level/layer of nodes, similar to tree BFS.
    - **Initialization Steps**
        - Initialize the queue with the first node (the starting point).
        - Add the starting point to the visit hash set immediately.
        - This starting point represents all nodes reachable with a path length of zero.

- **Implementation Details**
  - **The Main Loop**
    - Loop runs while the queue is non-empty.
    - Take a snapshot of the length of the queue at the start of the loop.
    - This allows processing all elements belonging to the current layer.
  - **Processing Nodes**
    - Pop elements from the queue.
    - When popping, you retrieve the coordinates of the position.
    - Check if the current position is the destination.
    - If it is the destination, return the length of the path.
  - **Managing Variables**
    - **Length**: Initially set to zero,.
    - The length is incremented by one after processing an entire layer (after the inner loop finishes).
- **Handling Movement and Edge Cases**
  - **Directions**
    - We can move in four directions (neighbors): Left, Right, Up, and Down,.
  - **Edge Cases (Conditionals)**
    - We must check for specific conditions to see if a move is valid. If invalid, we `continue` (skip).
    - **Out of Bounds**:
      - If `min(row, col) < 0`.
      - If `row == number of rows`.
      - If `column == number of columns`.
    - **Blocked**: If the grid value is 1 (or indicates a blockage).
    - **Visited**: If the position is already in the visit hash set.
  - **Coding Strategy for Directions**
    - Writing conditionals four times is repetitive and a "pain to type",.
    - A common strategy is using direction arrays to minimize code.
    - Define change in row (`dr`) and change in column (`dc`) for all four directions (Right, Left, Below, Above).
    - Loop through these directions and add them to the current row and column to get neighbor coordinates.
    - This boilerplate code can also be placed in a helper function,.
- **Step-by-Step Execution Walkthrough**
  - **Layer 1**

- Start at the top-left (Length 0),.
- Check neighbors. Only the Right neighbor is valid.
- Add the valid neighbor to the Queue and the Visit Hash Set.
- Increment path length to 1.

- **Layer 2**

  - Pop the node from the Queue.
  - Check neighbors. Valid moves might be Right and Below.
  - Add both to the Queue and Visit Hash Set.
  - Increment path length to 2.
  - *Note*: If the destination check was performed here and failed, do not return the length yet.

- **Layer 3 and Beyond**

  - If the Queue has multiple items (e.g., 2 items), pop them one by one.
  - **Crucial Logic**:
    - When adding a neighbor to the queue, immediately add it to the visit hash set.
    - This prevents adding the same position to the queue twice if it is reached from two different parent nodes.
    - This ensures the algorithm runs in `N * M` complexity.
  - Continue expanding layer by layer (Length 3, 4, 5, etc.),.

- **Reaching the Destination**

  - The destination check usually occurs *after* popping from the queue.
  - Example: If the popped node is the destination, return the current `length`.
  - If the destination was reached at the end of the layer, the result (e.g., 6) is returned.

- **Code Logic Summary**

  - **Pseudocode Logic**

```
Initialize length = 0
Add start to Queue and Visit Set
While Queue is not empty:
    For i in range(len(Queue)):
        Pop (r, c)
        If (r, c) is destination:
            Return length
        For (dr, dc) in directions:
            new_r = r + dr
            new_c = c + dc
            If (new_r, new_c) is invalid or visited:
                Continue
            Add (new_r, new_c) to Queue
            Add (new_r, new_c) to Visit Set
    Increment length
```

    - *Note*: The example in the source performs the destination check after popping.
    - It is possible to check for the destination when adding to the queue, but the code would need adjustment regarding when the length is incremented.

- **Conclusion**

    - **Pros**:
        - Guarantees finding the shortest path.
        - Never visits the same position twice.
        - Visually easy to follow (layer by layer expansion),.
    - **Cons**:
        - Contains a lot of boilerplate code,.
        - Handling all edge cases and conditionals correctly takes time and practice.
    - **Recommendation**:
        - Practice graph problems to get used to writing the boilerplate efficiently.

Adjacency List

- Adjacency Lists: Introduction and Representation

    - Overview
        - Adjacency lists are generally simpler to run algorithms on compared to matrices.
        - Originally, adjacency lists were discussed as a **Graph Node Class** or object containing:
            - A value (stored as a string, character, or integer).
            - A list of neighbors (pointers pointing to other nodes).
        - These pointers can represent directed or undirected edges.
    - Coding Interview Representation
        - In coding interviews, it is much more common to use a **Hashmap** to represent an adjacency list.
        - This is effective as long as the value or ID of each node is unique (almost always true in interviews).
        - Key-Value Structure:
            - **Key**: The unique identifier of the node (e.g., an integer, character, or string like 'A').
            - **Value**: A list representing the neighbors.
        - Example:
            - If we have nodes 'A' and 'B', the hashmap keys are 'A' and 'B'.
            - If the lists are empty, it means there are no pointers (edges) originating from those nodes.

- Building an Adjacency List

    - Common Interview Scenario
        - Typically, you are not given the adjacency list directly; you must build it from a list of edges.
    - Input Data
        - You are given a list of edges, which are pairs of nodes (Source, Destination).
        - Example: A directed edge from A to B means A points to B.
    - Step-by-Step Construction Algorithm
        - 1. **Initialize Variable**: Create an empty hashmap.

        2. **Iterate Through Edges**: Loop through every pair in the edge list.
            - Identify `source` (first in pair) and `destination` (second in pair).
        3. **Handle New Keys**:
            - Check if the `source` is in the hashmap. If not, add it with an empty array as its value.
            - Check if the `destination` is in the hashmap. If not, add it with an empty array as well.
            - *Note*: We add the destination even if it has no outgoing neighbors to ensure every node seen in the edge list exists in the adjacency list.
        4. **Add Relationship**:
            - Append the `destination` to the `source`'s list of neighbors.
            - Visually, this creates a pointer from Source to Destination.
    - Detailed Example Construction
        - Given Edges: (A, B), (B, C), (B, E), (C, E), (E, D).
        - Step 1: Process (A, B). Add A and B to map. Append B to A's list. (A -> B).
        - Step 2: Process (B, C). B is in map. Add C to map. Append C to B's list. (B -> C).
        - Step 3: Process (B, E). Add E to map. Append E to B's list. (B -> E).
        - Step 4: Process (C, E). Append E to C's list. (C -> E).
        - Step 5: Process (E, D). Add D to map. Append D to E's list. (E -> D).
        - Result: A map containing all nodes and their relationships, equivalent to the visual graph structure.

- Depth First Search (DFS) on Adjacency Lists
    - Overview
        - Running DFS on an adjacency list is easier than on a matrix because there is no need to worry about edge cases like going out of bounds.
        - It is a backtracking algorithm.
    - Problem Statement
        - Count all paths from a **Start Node** (e.g., A) to a **Target Node** (e.g., E).
    - Parameters
        - Adjacency List (Hashmap).
        - Start Node.
        - Target Node.
        - `visit` Hash Set (keeps track of visited nodes along the current path).
    - Algorithm Logic
        1. **Base Case - Visited**: Check if the current node is already in the `visit` set. If yes, return 0 (invalid path).
        2. **Base Case - Target Found**: Check if the current node is the target. If yes, return 1 (found a path).
        3. **Recursive Step**:
            - Initialize a `count` variable to 0.
            - Add the current node to the `visit` set.
            - Iterate through the current node's list of neighbors from the adjacency list.
            - Call DFS recursively on each neighbor.
            - Add the result of the recursive call to `count`.
        4. **Backtracking**:

        - After the loop finishes, remove the current node from the `visit` set.
        - Return the total `count`.
    - Execution Trace (A to E)
        - Start at A. Not visited, not target. Add A to visit. Neighbor: B.
        - Move to B. Not visited, not target. Add B to visit. Neighbors: C, E.
        - **Branch 1**: Call DFS on C (DFS goes deep first).
            - C not visited, not target. Add C to visit. Neighbor: E.
            - Call DFS on E. E is target. **Return 1**.
            - Back at C: Add 1 to count. Loop ends. Remove C from visit. **Return 1** to B.
        - Back at B: Count is 1. Next neighbor: E.
        - **Branch 2**: Call DFS on E.
            - E is target. **Return 1**.
        - Back at B: Add 1 to count. Total count is 2. Loop ends. Remove B from visit. **Return 2** to A.
        - Back at A: Loop ends. **Return 2**.
        - *Observation*: Node D was never visited because the path stopped immediately upon reaching target E.
    - Cycle Detection
        - If the graph had a back-edge (e.g., C points back to B).
        - DFS would go A -> B -> C -> B.
        - At the second B, the algorithm detects B is already in `visit`. Returns 0 immediately to prevent infinite loops.
    - Time Complexity Analysis (DFS)
        - This is a brute force backtracking approach.
        - **Path Length**: In the worst case, the path length is equal to the number of vertices (`V`), acting as the height of the decision tree.
        - **Choices**: Let `N` be the average number of edges per node.
        - **Complexity**: `O(N^V)` (Exponential).
        - As the size of the graph (`V`) grows, the time grows extremely quickly, making this inefficient.

- Breadth First Search (BFS) on Adjacency Lists

    - Overview
        - BFS is much more efficient than exponential DFS.
        - It is used to find the **Shortest Path** from a start node to a target node.
    - Algorithm Setup
        - **Variables**:
            - `length`: Initially 0.
            - `visit`: Hash Set. Add start node immediately.
            - `queue`: Queue data structure. Add start node immediately.
    - Algorithm Logic
        - Loop while the `queue` is not empty.
        - **Snapshot Length**: Get the length of the queue to process the current level/layer entirely.
        - Iterate through the current level:
            - **Pop**: Remove node from queue.
            - **Check Target**: If node equals target, return `length`.
            - **Process Neighbors**:

            - Iterate through neighbors in adjacency list.
            - If neighbor is **not** in `visit`:
                - Add to `visit` set.
                - Add to `queue`.
        - **Increment Layer**: After the inner loop (processing one level) finishes, increment `length` by 1.
    - Execution Trace (A to E)
        - **Level 0 (Length 0)**:
            - Queue: [A]. Pop A. Not target.
            - Neighbor B: Not visited. Add to visit, Add to Queue.
            - Level done. Increment Length to 1.
        - **Level 1 (Length 1)**:
            - Queue: [B]. Pop B. Not target.
            - Neighbors C, E: Both unvisited. Add both to visit, Add both to Queue.
            - Level done. Increment Length to 2.
        - **Level 2 (Length 2)**:
            - Queue: [C, E]. Length is 2.
            - Pop C. Not target. Neighbor E is already visited (skipped).
            - Pop E. **Is Target**. Return `length` (which is 2).
        - Result: Shortest path is length 2.
    - Time Complexity Analysis (BFS)
        - Unlike the matrix complexity (`4 * N * M`), we analyze based on Vertices (`V`) and Edges (`E`).
        - **Graph Size**: `V` (number of vertices).
        - **Edges**: While edges could theoretically be `V^2`, we use `E` to represent actual edges.
        - **Worst Case**: We visit every vertex (add to set/queue) and travel along every edge.
        - **Formula**: `O(V + E)`.
    - Space Complexity Analysis (BFS)
        - In the worst case, we add every node to the `visit` hash set and the `queue`.
        - **Formula**: `O(V)`.

- Limitations and Advanced Topics

    - Weighted Graphs
        - The standard BFS assumes all edges have the same "length" or weight (e.g., 1).
        - If edges have different weights (e.g., one path is length 10, another is length 2+3=5), standard BFS might find the fewer-steps path (length 10) rather than the actual shortest weighted path (length 5).
        - To handle weights, more complicated algorithms like **Dijkstra's Algorithm** are required.
    - Conclusion
        - Graphs are complex structures with significant academic research behind them.
        - Understanding DFS and BFS provides a strong foundation.
        - The shortest path algorithm is essentially a modified BFS.

Dynamic Programming

1-Dimension DP

- **Introduction to 1-Dimension Dynamic Programming (DP)**
  - **The Fibonacci Sequence: A Core Example**
    - The Fibonacci sequence is a classic instance of a one-dimensional dynamic programming problem.
    - **Mathematical Definition**
      - The zeroth Fibonacci number is defined as 0.
      - The first Fibonacci number is defined as 1.
      - An arbitrary nth Fibonacci number is defined as the sum of the (n-1) and (n-2) Fibonacci numbers.
      - **Example**: The second Fibonacci number is the sum of the first and zeroth Fibonacci numbers (1 + 0 = 1),.
    - **Iterative Calculation**
      - Calculating an arbitrary number like the fifth Fibonacci number can be done using a loop.
      - One would calculate the second, then the third, then the fourth, until reaching the fifth.
      - This iterative loop approach runs in **O(n) time**.
  - **The Brute Force Recursive Approach**
    - **Recursive Decision Tree**
      - To calculate the fifth Fibonacci number (F(5)), the algorithm must find the fourth (F(4)) and third (F(3)) numbers.
      - This process continues, drawing out a decision tree until the base cases are reached.
      - **Base Cases**: The recursion stops at F(1) and F(0).
      - For F(4), the algorithm must find F(3) and F(2); for F(3), it needs F(2) and F(1).
    - **Recursive Code Logic**,

```
def fibonacci(n):
    if n <= 1:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)
```

      - If the input n is 0 or 1, the function returns n itself.
      - Otherwise, it makes two recursive calls: one for the left branch (n-1) and one for the right branch (n-2).
    - **Efficiency Issues and Repeated Work**,
      - The height of this decision tree is approximately equal to n.
      - The number of branches at each level is two.
      - A rough upper bound for the size of the tree is roughly equal to n, though the tree is not completely filled.
      - This approach involves significant **repeated work**.
      - For example, while calculating F(5), the subtree for F(2) is drawn and calculated three separate times.

        - Similarly, the larger subtree for F(3) is calculated twice.

- **Top-Down Dynamic Programming (Memoization)**,

    - **Concept of Memoization**
        - Memoization is an optimisation technique used to eliminate repeated work by caching results.
        - When a sub-problem is solved for the first time, its result is stored.
        - If the same sub-problem is encountered again, the stored value is returned immediately instead of recreating the recursive subtree.
    - **Implementation with Caching**
        - Most commonly, a **hashmap** is used to represent the cache, though an array can also be used.
        - The hashmap maps an integer (the Fibonacci index) to its calculated Fibonacci value,.
        - Checking if a value exists in the hashmap is an **O(1) operation**.
    - **Step-by-Step Execution for F(5)**,,,
        - Start with an empty cache and call the function for F(5).
        - The algorithm traverses down: F(5) -> F(4) -> F(3) -> F(2).
        - At F(2), it calls base cases F(1) (returns 1) and F(0) (returns 0).
        - F(2) is calculated as 1 + 0 = 1, and this result is **stored in the cache** (key: 2, value: 1),.
        - Returning to F(3), the algorithm calculates F(1) (base case returns 1). F(3) = 1 (from F(2)) + 1 = 2. This is cached (key: 3, value: 2).
        - Returning to F(4), the algorithm needs F(3) (already calculated) and F(2). It checks the cache for F(2), finds the value 1, and returns it immediately without recursive calls.
        - F(4) = 2 + 1 = 3, which is then cached,.
        - Finally, for F(5), the algorithm uses F(4) and checks the cache for F(3) (finding value 2). F(5) = 3 + 2 = 5.
    - **Complexity and Classification**
        - This approach reduces the decision tree to a linear size.
        - The time complexity is **O(n)**.
        - It is called **Top-Down DP** because it starts at the top of the decision tree and works downwards.

- **Bottom-Up Dynamic Programming**

    - **The "True" Dynamic Programming Approach**
        - This approach does not use recursion.
        - Instead of starting at the top (F(n)) and going down, it starts at the base cases (bottom) and works upwards,.
        - It is often considered the "true" dynamic programming method by some practitioners.
    - **Iterative Logic and Array Implementation**,
        - For n=5, the process starts at the base cases: F(0)=0 and F(1)=1,.
        - **F(2)**: 0 + 1 = 1.
        - **F(3)**: 1 + 1 = 2.

        - **F(4)**: 1 + 2 = 3.
        - **F(5)**: 2 + 3 = 5.
        - **Time Complexity**: Big O of N (O(n)) because it iterates n times.
        - **Memory Complexity**: Big O of N (O(n)) if maintaining a full array.
    - **Memory Optimisation (Constant Space)**,,
        - It is not necessary to maintain the entire array; only the two previous values are needed at any time to calculate the next value.
        - By using a small array of size two or temporary variables, the memory complexity becomes **constant (O(1))**,.
        - **The process**:
            - Initially, store 0 and 1.
            - Calculate the new value and overwrite the old one in a specific position, while saving the necessary previous value in a temporary variable.
            - This continues until the loop finishes at n, returning the final calculated value.
        - This is the most efficient way to solve the problem.

- **Key Concepts of Dynamic Programming**,

    - **Problem Breakdown**
        - DP is a technique that takes a large, complex problem and simplifies it.
        - It involves breaking a problem down into smaller **sub-problems** (e.g., finding F(4) to find F(5)).
        - Most DP problems can be represented by mathematical equations defining these sub-problems.
    - **Dimensionality**
        - This is a **one-dimensional** case because the solution space is a one-dimensional array (indices 0 to n).
        - This implies the existence of two-dimensional dynamic programming problems where the solution space is more complex.

An analogy for Dynamic Programming is building a staircase where you can only reach a higher step by standing on the ones directly below it; rather than jumping from the top and trying to figure out where the ground is (recursion), you start firmly on the ground floor and lay each brick one by one until you reach the desired height.

2-Dimension DP

- **2-Dimension Dynamic Programming**
    - **The Counting Paths Problem**
        - The task is to count the number of paths from the top-left of a two-dimensional grid to the bottom-right.
        - In this specific version, there are no blockers (no zeros or ones acting as obstacles).
        - A simplifying constraint is that movement is only allowed **down or to the right**.
        - Because every move (down or right) brings you closer to the goal, every single path to the result will be the **exact same length**.
    - **Recursive Brute Force Approach**

- The problem can be solved recursively using **depth-first search (DFS)**, though this is inefficient.
- The grid itself is not a parameter for the function; instead, the dimensions of the grid are what matter.
- For a square grid of 4x4, the function is called with the starting coordinates of (0, 0) and dimensions 4x4.
- **Base Cases for Recursion:**
  - **Out of Bounds:** If the row (`r`) equals the total number of rows (e.g., 4) or the column (`c`) equals the total number of columns, the path is invalid. This returns a value of **0**.
  - **Goal Reached:** If the current row (`R`) is equal to `rows - 1` and the current column is equal to `columns - 1`, the path is valid. This returns a value of **1**.
- **Decision Tree and Complexity:**
  - At every position, there are two choices: move down (`row + 1`) or move right (`column + 1`).
  - This creates a decision tree with **two branches** at every node.
  - The height of the decision tree is approximately the number of rows plus the number of columns (`n + m`).
  - The time complexity for this brute force method is `2^{(n + m)}`.

- **Top-Down Dynamic Programming (Memoization)**
  - Efficiency can be improved by noticing **repeated work**; many sub-problems are solved multiple times.
  - For example, to solve the problem for one cell, you must solve the two cells below and to the right of it. Those cells in turn solve their neighbours, leading to exponential repetition.
  - **Caching** (memoization) reduces the time complexity to the **size of the grid** (`n * m`).
  - **Cache Implementation:**
    - The cache is a two-dimensional array of the same dimensions as the grid, or a hashmap.
    - In Python, a 4x4 grid of zeros can be created by taking an array with one zero `, multiplying it by four to get` , and then creating an outer array with four copies of that inner array.
  - **Execution Logic:**
    - **Crucial Step:** You must check if a move is **out of bounds** before checking if it is in the cache to avoid indexing errors or exceptions.
    - If the position is not in the cache and not a base case, the function recursively calls itself for `r + 1` and `c + 1`.
    - It uses a single cache object passed by reference throughout all recursive calls.
    - The result for a cell is the sum of the results from the move down and the move to the right.
    - In a 4x4 grid, the final result calculated at the starting point (0, 0) is **20 unique paths**.
- **Bottom-Up Dynamic Programming (True DP)**
  - This approach starts at the **base case** (the bottom-right goal) and works backwards to the starting point.

        - It typically requires less code than the recursive approach but can be harder to conceptualise.
        - **Filling the Grid Logic:**
            - The order of solving sub-problems is intentional: start at the last row and move left, then move to the next row up and move left.
            - We place a **1** at the goal position as a base case to allow the maths to work correctly; otherwise, all values would remain zero.
            - To calculate a value at a position, you add the value from the cell **below** it and the cell to its **right**.
            - Every cell in the bottom row and the right-most column will have a value of **1** because there is only one way to reach the goal from those straight-line paths.
    - **Space-Optimised Bottom-Up DP**
        - It is not necessary to store the entire 2D grid in memory.
        - To calculate values for a current row, you only need the values from the **row immediately below it** (the previous row).
        - At any point, only **two rows** are needed in memory: the `current_row` and the `previous_row`.
        - **Complexity:**
            - **Time Complexity:** `O(n * m)` because you still iterate through the entire grid.
            - **Space Complexity:** `O(m)`, where `m` is the number of columns, because you only store two rows (constants are ignored in Big O).
        - **Step-by-Step Calculation (4x4 Grid Example):**
            1. Initialize a `previous_row` of all zeros.
            2. Create a `current_row` where the last position is set to **1**.
            3. Iterate through the grid from the last row upwards and from the second-to-last column to the left.
            4. In the bottom row, calculations result in all **1s**.
            5. For the next row up: the last cell is 1; the next is `1 (below) + 1 (right) = 2`; then `1+2=3`; then `1+3=4`.
            6. For the row above that: values become **1, 3, 6, 10**.
            7. For the top row: values become **1, 4, 10, 20**.
            8. The `current_row` is assigned to `previous_row` after each row is completed.
            9. The final answer is the zeroth value in the final `previous_row`, which is **20**.

To understand this, imagine building a brick wall from the bottom up; you only need the layer of bricks directly beneath the one you are currently laying to provide support, so you don't need to keep the blueprints for the entire foundation in your hands once that specific layer is finished.

Bit Operator

Bit Operator

- **OVERVIEW OF BIT MANIPULATION**
    - Bit manipulation is the final topic covered in the course.
    - It is not considered a super important topic for coding interviews, but it does come up occasionally.

- It is important to understand the basics and have a fundamental understanding of how bits work, even beyond interview contexts.
- Computers use zeros and ones (bits) exclusively under the hood, though this is usually abstracted away for the user.

- **BASIC LOGICAL OPERATIONS**

  - **Bitwise AND (&)**
    - Most languages, including Python, use the ampersand character `&` for this operation.
    - Logic: Both bits must be 1 for the result to be 1.
    - If either bit is 0, the result is 0.
    - Truth Table:

```
0 & 0 = 0
0 & 1 = 0
1 & 0 = 0
1 & 1 = 1
```

  - **Logic OR (|)**
    - The character for logic OR is the pipe `|`.
    - Logic: Only a single bit has to be 1 for the result to be 1.
    - If neither bit is 1, the result is 0.
    - Truth Table:

```
0 | 0 = 0
0 | 1 = 1
1 | 0 = 1
1 | 1 = 1
```

  - **Exclusive OR (XOR / ^)**
    - This operation comes up occasionally in coding interviews.
    - The character for XOR is the up-caret `^`.
    - Logic: The result is 1 **only** if exclusively one of the bits is 1.
    - If both bits are 1, or both bits are 0, the result is 0.
    - Truth Table:

```
0 ^ 0 = 0
0 ^ 1 = 1
1 ^ 0 = 1
1 ^ 1 = 0
```

  - **Negation (NOT / ~)**
    - The negation operator is typically the tilde `~`.
    - It takes any bit and flips it to its opposite value.

        - Logic:
            - Negation of 1 is 0.
            - Negation of 0 is 1.

- **BINARY REPRESENTATION AND BASES**

    - **Base 10 (Decimal)**
        - This is the standard system where the maximum value of a digit is nine.
        - When you reach ten, you add a new digit.
        - Places are powers of 10:
            - Ones place (`10^0`).
            - Tens place (`10^1`).
            - Hundreds place (`10^2`).
    - **Base 2 (Binary)**
        - When you reach the value of two, you add a new digit.
        - For example, the number 2 is represented as `10` in binary.
        - Places are powers of 2:
            - Ones place (`2^0`).
            - Twos place (`2^1`).
            - Fours place (`2^2`).
            - Eights place (`2^3`).
    - **Integer Storage**
        - Programming languages usually use 32-bit integers.
        - A small number like 23 would be represented with its binary bits preceded by many leading zeros to fill the 32-bit requirement.

- **BIT SHIFTING**

    - **Left Shift (<<)**
        - Shifting to the left moves every bit one position to the left.
        - The vacated spot on the right is replaced with a 0.
        - If a bit is shifted off the left end, it is dropped or deleted; it does not circle back.
        - **Mathematical Rule:** Shifting an integer to the left by one is the same as multiplying it by two.
    - **Right Shift (>>)**
        - Shifting to the right moves every bit one position to the right.
        - The vacated spot on the left is replaced with a 0.
        - **Mathematical Rule:** Shifting an integer to the right by one is the same as dividing it by two.
        - If the number is odd, the result is rounded down.

- **ALGORITHM: COUNTING ONE BITS**

    - **Goal:** Given an integer, count the number of 1s in its binary representation.
    - **Example:** The integer 23 (base 10) is `10111` in binary.
    - **Steps:**
        1. Declare a `count` variable starting at zero.
        2. Run a loop while the integer `n` is greater than zero.

3. Check the "ones place" of the current number by using bitwise AND with 1 (`n & 1`).
    - The value 1 is used because it has all zeros except for the last bit.
    - If the result of `n & 1` is 1, it means there is a 1 in the ones place.
4. If `n & 1` is true, increment the `count`.
5. Shift the integer `n` to the right by one (`n >> 1`) to discard the bit that was just checked.
6. Repeat until `n` becomes zero.

- **Tracing the example (Integer 23 / `10111`):**
    - `10111 & 1` is 1 `->` Count becomes 1. Shift right to get `1011`.
    - `1011 & 1` is 1 `->` Count becomes 2. Shift right to get `101`.
    - `101 & 1` is 1 `->` Count becomes 3. Shift right to get `10`.
    - `10 & 1` is 0 `->` Count stays 3. Shift right to get `1`.
    - `1 & 1` is 1 `->` Count becomes 4. Shift right to get `0`.
    - Loop ends; result is 4.

- **FINAL NOTES ON BIT MANIPULATION**

    - These problems often rely on specific "tricks," which is why some people feel they are not the best coding interview questions.
    - You either know the trick or you don't, which can limit the effectiveness of the question for evaluating general coding ability.
    - However, mastering these basics is useful for general programming tasks outside of interviews.

Analogy for Understanding Bit Shifting: Think of bit shifting like a queue of people sitting in a row of chairs. **Left shifting** is like everyone moving one chair to the left; the person at the far left falls off the end and disappears, while a new person (a zero) sits in the empty chair on the right. **Right shifting** is the same process in reverse: everyone moves right, the person on the far right disappears, and a new zero sits in the chair on the far left. Just as adding a zero to the end of a decimal number (`15 -> 150`) multiplies it by 10, adding a zero to the end of a binary number multiplies it by 2.

Advanced Algorithms

Arrays

Kadanes Algorithm

- **Kadane's Algorithm Overview**

    - This algorithm is named after a person and, while not considered a hard algorithm compared to others, it is fundamental and important due to the many lessons it offers.
    - It shares significant overlap with other algorithmic patterns, specifically **two pointers** and **sliding window**.
    - Technically, it is classified as both a **dynamic programming** and a **greedy algorithm**.

- **The Problem Statement**

    - The input is an array of integers which can be positive or negative.
    - The objective is to return a **non-empty subarray** from the array that has the largest possible sum.

    - Definition of a **subarray**: It must be a **contiguous** subarray, meaning you cannot have gaps or "holes" between elements.
    - Variation - **All Negative Values**:
        - If the array contains only negative numbers, you cannot return zero because the subarray must be non-empty.
        - In this case, you return the largest individual value (even if it is negative).
        - For example, in an all-negative array where -1 is the largest value, -1 is the sum returned; making the subarray larger would only result in a more negative sum.
    - Result: The algorithm returns the **sum** itself, not necessarily the subarray.

- **The Brute Force Solution**

    - The simplest approach is to calculate the sum of every possible subarray and track the maximum.
    - **Logic**:
        - Iterate through every possible starting position (i).
        - For each starting position, iterate through every possible ending position (j) to find every subarray starting at i.
        - This involves a nested loop structure.
    - **Time Complexity**: $O(n^2)$, where n is the size of the array, because you check every subarray starting at every position.
    - **Initialization**:
        - The `maxSum` is initialized to the first element of the array because the subarray must be non-empty.
        - If empty subarrays were allowed, `maxSum` could be initialized to zero.
        - Some programmers initialize `maxSum` to negative infinity or the minimum possible integer for the language used.
    - **Edge Cases**:
        - The algorithm assumes the input array is non-empty; if it were empty, finding a non-empty subarray would be impossible, and the code might throw an exception or return immediately.
    - **Implementation Detail**:
        - Two pointers (i and j) determine the boundaries of the current subarray.
        - A `currentSum` variable tracks the sum as the inner loop progresses.
        - If `currentSum > maxSum`, then `maxSum` is reassigned.

- **Optimization and Logic of Kadane's Algorithm**

    - To improve on $O(n^2)$, redundant work must be identified.
    - **Core Insight**:
        - Positive numbers contribute to a larger sum, while negative numbers make the sum smaller.
        - If a subarray has a positive sum, it is worth including in the next calculation because it contributes to the total.
        - Even if a subarray contains a negative number, it is worth keeping if the overall sum remains positive and allows for a larger positive number later that "counteracts" the negative one.

        - **When to Discard**: You should never add a negative previous sum to a current value because a negative value will never increase the total sum.
    - **Comparison Choice**: The "current sum" at any given position is the largest subarray sum ending at that position. You choose the larger value between the current element alone or the current element plus the previous sum.

- **The Linear Time Algorithm ($O(n)$)**
    - This version does not require nested loops.
    - **Variables**:
        - `maxSum`: Initialized to the first element.
        - `currentSum`: Tracks the sum of the current window.
    - **Step-by-Step Logic**:
        - Iterate through every number `n` in the input array.
        - **Reset Check**: Before adding `n` to `currentSum`, check if `currentSum` is negative.
        - If `currentSum < 0`, reset it to zero. This ensures you are not carrying over a "debt" that decreases the sum of the current element.
        - Add `n` to `currentSum`.
        - Update `maxSum` if `currentSum` is now greater than the existing `maxSum`.
    - **Example Walkthrough (Array: [4, -1, 2, -7, 3, 4])**:
        - Start: `maxSum = 4`, `currentSum = 0`.
        - Element 4: `currentSum` becomes 4. `maxSum` stays 4.
        - Element -1: `currentSum` becomes 3. `maxSum` stays 4.
        - Element 2: `currentSum` (3 + 2) becomes 5. `maxSum` updated to 5.
        - Element -7: `currentSum` (5 - 7) becomes -2. `maxSum` stays 5.
        - Element 3: `currentSum` was negative, so it resets to 0, then adds 3. `currentSum = 3`. `maxSum` stays 5.
        - Element 4: `currentSum` (3 + 4) becomes 7. `maxSum` updated to 7.
        - End: Return 7.

- **Sliding Window Variation (Returning Indices)**
    - This variation is used if the problem asks for the start and end indices of the subarray rather than just the sum.
    - **Pointers**:
        - `L` (Left pointer): Represents the start of the window.
        - `R` (Right pointer): Represents the end of the window.
        - Use a capital 'L' in code to avoid confusion with the number '1'.
        - `maxLeft` and `maxRight`: Store the indices of the window that produced the `maxSum`.
    - **Rules**:
        - `L` should never cross `R`.
        - If `L == R`, the subarray has a length of 1.
        - The window can grow or slide.
    - **Logic for Index Tracking**:
        - Initialize `L = 0` and iterate `R` from 0 to the end of the array.
        - If `currentSum < 0`:
            - Reset `currentSum = 0`.
            - Move `L` to the current position of `R`. This starts a "fresh" window.

        - Add the value at `R` to `currentSum`.
        - If `currentSum > maxSum`:
            - Update `maxSum`.
            - Set `maxLeft = L` and `maxRight = R`.
    - **Example Context**: If the max sum of 7 was found between index 4 and 5, the algorithm would return those specific pointers.

- **Complexity Summary**
    - **Time Complexity**: Linear $O(n)$ because it only requires a single pass through the array.
    - **Space Complexity**: Efficient, as it only tracks a few variables (`maxSum`, `currentSum`, and pointers).

- **Analogy for Understanding**
    - Think of the `currentSum` as a traveler's bank account while walking across a path. Positive numbers are gifts of money, and negative numbers are tolls. If the traveler's account drops below zero (becomes negative), they are better off declaring bankruptcy (resetting to zero) and starting fresh at the next town rather than carrying that debt forward to future gifts. [This analogy is provided for clarity based on source logic in].

Sliding Window Fixed

- **Problem Definition**
    - The goal is to return **true** if there are **two elements** within a window of size **K** that are equal.
    - "Within a window of size K" means the window can be **at most** of length K.
    - The two elements must be **different elements** at different positions.
    - You cannot say a window has two identical elements if you are just looking at the same position twice.

- **Brute Force Approach**
    - **Method**: Look at every possible window of size K in the array.
    - **Example with K = 2**:
        - Examine the first window of two values: Check if they are the same.
        - If not, move to the next window of two values and check again.
        - Continue this process until the end of the array is reached.
        - If a window is found where the two values are equal, return true.
    - **Window Constraints**:
        - If a duplicate exists at a distance of 3, but K is fixed at 2, that duplicate will not be found because the window is too small.
        - Brute force involves manually comparing the left element to every other element in that specific window.

- **Pointer-Based Implementation (Brute Force Logic)**
    - **Left Pointer**: Represents the **beginning** of the window.
    - **Right Pointer**:
        - Initialized to `left + 1`.

        - It should **never** be at the same position as the left pointer to avoid comparing an element to itself.
        - It is used to check if the value at the left pointer is equal to any other values within the window.
    - **Shifting the Window**:
        - Once the right pointer reaches the end of the window size (larger than K), the **left pointer is shifted** over.
        - This shift creates a new window.
    - **Loop Structure**:
        - **Outer Loop**: Runs as many times as there are elements in the array (**N**).
        - **Inner Loop**: Runs roughly **K** times (technically K - 1).
    - **Boundary Management**:
        - To avoid going out of bounds, the right pointer iterates from `left + 1` up to the **minimum** of the array length and `left + K`.
    - **Small Window Edge Case**:
        - Even if there aren't enough values to fill a window of size K (e.g., at the very end of the array), the window is still **valid** and must be checked.

- **Complexity Analysis (Brute Force)**
    - **Time Complexity**: **O(N * K)**.
    - **Variable Relationship**: **K** will always be less than or equal to **N** because a window cannot be larger than the array itself.
    - **Performance**: This is acceptable for small K values but becomes inefficient as the window size (K) increases.

- **Optimized Sliding Window (Hash Set)**
    - **Improvement**: Duplicate detection can be optimized using **hashing**.
    - **Rolling Hash Set**:
        - Maintain a hash set containing all values currently within the window.
        - This avoids manually comparing every element to every other element.
    - **Algorithm Steps**:
        1. Use a **single loop** where the right pointer iterates through every position.
        2. Before adding a new value to the set, check if it **already exists** in the hash set.
        3. **Constant Time Operation**: Checking the hash set for a value is a constant time operation.
        4. If the value exists in the set, a duplicate within the window has been found; **return true**.
        5. If the value does not exist, **add it** to the hash set.
        6. **Window Maintenance**:
            - If the pointers representing the window exceed the size **K**, remove the **leftmost value** from the hash set.
            - **Increment the left pointer** to shift the window.
    - **Conclusion of Algorithm**:
        - The process continues until the end of the array, even handling the smaller windows at the finish.
        - If the entire array is processed and no duplicates are found within any window of size K, **return false**.

- **Comparison to Variable Windows**
    - This is a **fixed-size** sliding window case.
    - It differs from **variable-length** windows (like the sliding window variation of **Kadane's algorithm**) where the window size grows or shrinks dynamically based on certain conditions.
- **Analogy for Understanding**
    - Imagine you are looking at a long row of houses through a **fixed-size binoculars frame** that only lets you see 3 houses at a time. To find if two houses have the same color, you check the 3 houses currently in view. Instead of comparing house #1 to #2 and #3 every single time you move, you keep a **list (the Hash Set)** of the colors currently in your view. As you slide your view one house to the right, you cross off the color of the house that just left your view on the left and add the color of the new house appearing on the right. If the new house's color is already on your list, you've found a match within your view.

Sliding Window Variable

- **Variable Length Sliding Window Problems**
    - **Basic Example: Longest Subarray with the Same Value**
        - **Problem Description**
            - The goal is to find the length of the longest subarray where every position contains the same value.
            - Essentially, you are looking for the longest string of consecutive duplicates.
            - Example array: ``.
                - There is a single four.
                - There are two twos.
                - There are three threes.
                - The longest window is the one containing three threes, so the answer is three.
        - **Sliding Window Approach (Manual Trace)**
            - Initialize the **left (L)** and **right (R)** pointers at the same position (index 0).
            - A single value is considered a valid window.
            - Increment the right pointer.
            - If the value at the right pointer is not equal to the value at the left pointer, the window is invalid.
            - To make the window valid again, increment the left pointer until the values at both pointers are equal.
            - In this specific case, the left pointer only needs to move once to meet the right pointer's position.
            - Continue incrementing the right pointer and check if the values match the left pointer's value.
            - Keep track of the **maximum length** seen so far.
            - When the right pointer goes out of bounds, the process stops.

- **Code and Optimization**
  - Initialize `longest_length` to 0.
  - Initialize the left pointer to 0.
  - Iterate through the array with a **for loop** using the right pointer (initialized to 0).
  - Growth and Shrinkage:
    - Grow the window until it becomes invalid.
    - Shrink the window from the left side when a new, different value is found.
  - **Optimization Detail**: Instead of incrementing the left pointer by one at a time, you can immediately set the **left pointer equal to the right pointer**. This is because if the right pointer found a new value, all values between the previous left pointer and the current right pointer were identical duplicates.
  - This optimization does not change the overall time complexity.
  - For such a simple problem, an inner loop is not strictly necessary.
- **Calculating Window Length**
  - If the left pointer is at index 0 and the right pointer is at index 2, the length is 3.
  - The formula is: `(right_pointer - left_pointer) + 1`.
  - Example: `2 - 0 = 2`. Adding 1 gives the correct length of 3.

- **Advanced Example: Minimum Length Subarray with Target Sum**
  - **Problem Description**
    - Given an array of **all positive integers**, find the **minimum length** subarray where the sum is **greater than or equal to** a given **target value**.
    - Example target: 6.
    - Detail Importance: The fact that all values are **positive** is a hint for the solution.
  - **Brute Force vs. Sliding Window**
    - Brute Force: Check every single subarray, total them up, and keep track of the minimum length among those that meet or exceed the target.
    - Sliding Window Logic:
      - Because all values are positive, growing the window will only increase the sum.
      - Once a window satisfies the target (sum >= target), adding more values will only make the window larger, not smaller.
      - Therefore, once the target is reached, you must **shrink the window** from the left to find a potentially smaller valid window.
  - **Manual Trace (Target = 6)**
    - 1. Grow the window: Add values until the sum is 6 (e.g., ``). Current min length is 3.
    - 2. To seek a smaller window, move the left pointer to the right.
    - 3. Update the sum: Subtract the value removed from the left from the total sum. (Example: `6 - 2 = 4`).

        4. Move the right pointer to add the next value. (Example: Add 2, sum becomes 6 again). Min length remains 3.
        5. Shrink left again: (Example: `6 - 3 = 3`).
        6. Move right: (Example: Add 4, sum becomes 7). Sum is >= 6, so window is valid.
        7. Shrink left: (Example: `7 - 1 = 6`). This is a valid window of **length 2**. This is the new minimum.
        8. Continue shrinking/growing until the right pointer is out of bounds.

    - **Implementation and Logic**

        - **Initialization**:
            - `left_pointer` = 0.
            - `total` (or `window_sum`) = 0.
            - `length` = **positive infinity** (or `array_length + 1`) to ensure the first valid window reduces it via a `min` function.
        - **Return Value**: If the length remains positive infinity at the end, return 0 (indicating no valid subarray was found). Otherwise, return the calculated minimum length.
        - **Algorithm Steps**:
            1. Use an outer loop to shift the right pointer.
            2. Add the value at the right pointer to the `total`.
            3. Use an **inner while loop** that runs as long as `total >= target`.
            4. Inside the while loop:
                - Check if the current window is smaller than the current `min_length` and update if necessary.
                - Subtract the value at the left pointer from `total`.
                - Increment the `left_pointer`.

    - **Time Complexity Analysis**

        - **Big O Notation**: The time complexity is **O(n)** (linear time).
        - **Why it is not $O(n^2)$**:
            - While there are nested loops (a for loop and a while loop), the inner loop does not execute N times for every iteration of the outer loop.
            - The right pointer visits every position exactly once.
            - The left pointer also visits positions only as it is incremented.
            - The left pointer only moves forward and never reaches the end of the array more than once.
            - In the worst case, the complexity is roughly **2 * n** (each element added once by R and removed once by L), which reduces to **O(n)**.
            - The inner loop executes a **total of N times across the entire program execution**, not per outer loop iteration.

    - **General Pattern**

        - Variable length sliding window solutions typically use an **inner while loop**.
        - This structure remains efficient and is a standard pattern for these types of problems.

- **Analogy for Understanding**

- Imagine an **accordion** being pulled across a floor. The front end (right pointer) moves forward to cover more ground. If the accordion gets too long or reaches a specific goal, you pull the back end (left pointer) forward to shorten it. Even though you are moving both the front and the back, each part of the accordion only travels the length of the room once. In the end, the total movement is proportional to the size of the room, not the square of it.

Two Pointers

- **Two Pointers and Sliding Window Relationship**
  - Two pointers is a **very large and general topic** that covers many of the algorithms previously discussed.
  - **Sliding window** is essentially a **subset** of two-pointer problems.
  - The distinction between the two is generally based on the focus of the algorithm:
    - In **two-pointer algorithms**, the primary focus is on the **two individual elements** that the pointers are pointing at.
    - In **sliding window problems**, the focus is on the **entire window** between the pointers, such as every unique value or the total sum of all values within that window.
  - Despite this distinction, two pointers and sliding window are considered **two sides of the same coin**.

- **Simple Example: Checking for Palindromes**
  - **Definitions and Logic**
    - A **palindrome** is defined as something that remains the **same when reversed**.
    - For this algorithm, **arrays and strings are essentially the same**, so the data type (integers, characters, etc.) does not matter.
    - An example of a palindrome is `1, 2, 2, 1`; reversing it results in `1, 2, 2, 1`, which is identical to the original.
  - **Two-Pointer Approach vs. Brute Force**
    - While one could solve this by building a new array in reverse, using **two pointers** is easier because it requires **no extra space**.
    - The process begins by **initializing a left pointer** at the very beginning and a **right pointer** at the very end of the array or string.
    - The algorithm **compares the two individual elements** at each pointer.
    - If the elements are equal, the **left pointer is incremented** (shifted right) and the **right pointer is decremented** (shifted left) to continue comparing.
    - If any two elements are **not equal** (e.g., comparing a 2 to a 3), the algorithm immediately returns **false**, as it is not a palindrome.
  - **Stopping Conditions**
    - The process continues until the **pointers cross each other**.
    - Once they cross, the algorithm can stop because every pair has been checked.
    - Checking further would only repeat comparisons that have already been made.
    - If the pointers cross and all pairs were equal, the input is a palindrome and the algorithm returns **true**.

- **Complex Example: Two Sum with Sorted Input**
  - **Problem Constraints**

        - The input is a **sorted array**.
        - The goal is to return the **indices of two separate elements** (different positions) that **sum up to a target value**.
        - It is assumed that there is **exactly one solution**.
    - **Brute Force vs. Optimized Strategy**
        - A **brute force solution** involves comparing every possible pair of elements, which results in an **O(N²)** time complexity.
        - Because the array is **sorted**, a more efficient two-pointer approach can be used.
        - Initializing pointers at the **extremes** (smallest value at the left, largest at the right) provides useful information.
    - **Shifting Logic Based on Sum**
        - **Sum is too large:** If the sum of the smallest and largest elements is **greater than the target**, the largest element cannot be part of the solution. This is because the largest element added to even the smallest possible value is still too big; therefore, adding it to any other value would also be too big. In this case, the **right pointer is decremented** to decrease the sum.
        - **Sum is too small:** If the sum is **smaller than the target**, the smallest element cannot be part of the solution. Since the array is sorted, all other available values are less than or equal to the current right element; thus, nothing added to the current smallest element will ever reach the target. In this case, the **left pointer is incremented** to increase the sum.
    - **Walkthrough Example (Target = 7)**
        - Suppose pointers are at `-1` (left) and `9` (right). Sum is `8`. This is **greater than 7**, so the right pointer moves left.
        - New pair is `-1` and `8`. Sum is `7`. This is the target, and indices are returned.
        - If the pair was `2` (left) and `7` (right), the sum is `9`. This is **too big**, so the right pointer moves to `4`.
        - The new pair is `2` and `4`. Sum is `6`. This is **too small**, so the left pointer moves to `3`.
        - Finally, `3 + 4 = 7`, which is the solution.

- **Algorithm Mechanics and Complexity**

    - **Initialization and Loop**
        - Pointers are initialized at the edges of the array.
        - The loop continues until the pointers meet or cross.
        - Because a solution is guaranteed in this specific problem, the loop will never exit without finding one.
    - **Time Complexity**
        - This algorithm is **O(N)**.
        - In the worst case, the pointers start at opposite ends and shift until they meet, meaning every position in the input array is looked at exactly once.
    - **Pattern Name**
        - This specific pattern of moving pointers from the ends toward each other based on conditional logic is sometimes called the **"shrinking window,"** though this name is not standard in the industry.

- **Summary Implementation Logic**

```
# General Two-Pointer Template for Two Sum (Sorted)
left = 0
right = len(array) - 1

while left < right:
    current_sum = array[left] + array[right]
    if current_sum == target:
        return [left, right]
    elif current_sum > target:
        right -= 1 # Decrease the sum
    else:
        left += 1  # Increase the sum
```

To understand the two-pointer approach on a sorted array, imagine you are **adjusting the temperature of a shower with two separate knobs**: one for freezing cold and one for scalding hot. If the water is too hot (sum > target), you turn down the hot knob (decrement right pointer). If the water is too cold (sum < target), you turn up the cold knob (increment left pointer). You keep adjusting from the extremes until you hit the perfect temperature.

Prefix Sums

- **Prefix Sums Pattern Overview**
    - **Core Definitions and Concepts**
        - **Prefix Sums** are a common pattern used with arrays that typically do not require complex data structures or advanced algorithms.
        - A **prefix** of an array is defined as any contiguous subarray that starts at the very beginning of that array.
        - Examples of prefixes:
            - A subarray containing only the first element.
            - A subarray containing the first two elements.
            - The entire array itself is technically a prefix.
        - Non-examples of prefixes:
            - A subarray that does not start at the beginning of the array.
            - A block of elements that starts at the beginning but is not contiguous.
        - A **prefix sum** refers to calculating the sum of every prefix within a given array to improve efficiency.
    - **Calculating Prefix Sums Step-by-Step**
        - To calculate prefix sums, you take the sum of a prefix and add the next element to it, avoiding repeated work.
        - **Example Array Walkthrough**: [2, -1, 3, -3, 4]
            - The first prefix sum is **2**.
            - The second prefix sum (first two values) is 2 + (-1) = **1**.
            - The third prefix sum (first three values) is the previous sum (1) + 3 = **4**.
            - The fourth prefix sum (first four values) is the previous sum (4) + (-3) = **1**.
            - The fifth prefix sum (entire array) is the previous sum (1) + 4 = **5**.
    - **Variations of the Pattern**

    - **Prefix Products**: This follows the same logic as prefix sums but involves calculating the product of elements instead.
        - Example for [2, -1, 3]: The first product is 2, the second is 2 * -1 = -2, and the third is -2 * 3 = -6.
    - **Postfix Sums**: This is the "opposite" of a prefix; it consists of every contiguous subarray starting from the end of the array.
    - Prefixes and postfixes are considered "two sides of the same coin" because they both eliminate repeated work.
- **The Subarray Sum Query Problem**
    - A classic application of this pattern is designing a data structure that can query the sum of a subarray between two indices: **left** and **right**.
    - **Naive Implementation**:
        - This approach does not use prefixes; it simply iterates through every position between the indices, adds them to a total, and returns that total.
        - In the worst case, this is an **O(n)** time operation every time the function is called.
    - **Prefix Sum Implementation**:
        - Precomputing prefix sums eliminates nearly all repeated work.
        - While prefix sums only represent subarrays starting at the beginning, they can be used to calculate any arbitrary subarray sum.
        - **Calculation Logic**: To find a specific subarray sum, take the prefix sum up to the `right` index and subtract the prefix sum of the elements to the left of the `left` index.
        - This allows for **O(1)** constant time queries.
- **Numerical Examples of Subarray Queries**
    - **Example 1**: To find a subarray sum that equals -1, the source takes a prefix sum of 1 and subtracts the previous prefix sum of 2 (1 - 2 = -1).
    - **Example 2**: To find the sum of a subarray at the end of the array, take the prefix sum of the entire array (5) and subtract the prefix sum of the first three values (4) (5 - 4 = 1).
    - The source notes that looking at the values individually (-3 and 4), the sum is indeed 1.
- **Using Postfix Sums for Queries**
    - This problem can be solved identically using postfix sums.
    - A value in a postfix array represents the sum from that index to the end of the array.
    - To find a subarray sum with postfixes, take the postfix value at the `left` index (representing the sum to the end) and subtract the postfix value starting after the `right` index.
- **Code Implementation and Data Structure Design**
    - **Initialization (Constructor)**:
        - Given an array `nums`, initialize the data structure by computing the prefix sums once.
        - Create a member variable array called `prefix` and a `total` variable initialized to zero.
        - Loop through every number in `nums`, add it to the `total`, and append that `total` to the `prefix` array.
        - This precomputation takes **O(n)** linear time.
    - **Range Sum Query Function**:
        - This function takes `left` and `right` indices and returns the sum in **O(1)** time.

        - **Right Prefix Sum**: This is retrieved directly from the `prefix` array at the `right` index.
        - **Left Prefix Sum**: This is the value to be subtracted, found at the `left - 1` index of the `prefix` array.
        - **Edge Case (Index 0)**: If the `left` index is 0, the `left - 1` index is out of bounds.
        - If `left` is 0, the default value to subtract should be 0 because the `right` prefix sum already represents the entire sum needed.
        - **Implementation Logic**:

```
# Logical representation using a ternary operator
leftPrefix = prefix[left - 1] if left > 0 else 0
return prefix[right] - leftPrefix
```

    - **Efficiency Analysis**:
        - While the naive constructor is O(1) and prefix sum constructor is O(n), the prefix sum approach is superior if the range sum function is called frequently.
- **Conclusion on the Pattern**
    - Prefix sums are one of the most common and important patterns in coding interviews and data structure design.
    - The pattern is versatile and open-ended, applying to sums, products, and various other precomputed range queries.

- **Analogy for Understanding**

    - Think of prefix sums like the distance markers on a highway. If you want to know the distance between Exit 10 and Exit 50, you don't need to remeasure the road; you simply take the distance at marker 50 and subtract the distance at marker 10.

Linked Lists

Fast and Slow Pointers

- **Fast and Slow Pointer Algorithm**

    - The algorithm is also commonly referred to as **Floyd's Tortoise and Hare algorithm**.
    - In this analogy, the **tortoise** is the **slow pointer** and the **hair** is the **fast pointer**.
    - It is a two-pointer technique usually used for **linked lists** and can be applied to many different areas.

- **Example 1: Finding the Middle of a Linked List**

    - This is a trivial example that can be solved without this specific algorithm.
    - **Standard Approach**:
        - Count the length of the linked list (e.g., 1, 2, 3, 4, 5).
        - Find the midway point based on that length.
        - For odd-length lists, the midway point is straightforward.
        - For even-length lists, there are two middle nodes. This specific algorithm assumes the **second node** of the two is the middle.

        - This standard approach takes **O(N) time** because it requires iterating once to find the length and then iterating halfway again.
    - **Fast and Slow Pointer Approach**:
        - This is also a linear time (**O(N)**) algorithm, so it does not improve time complexity, but it is a fundamental application of the technique.
        - **Initialization**: Initialize both a fast pointer and a slow pointer at the **head** of the linked list.
        - **Movement**:
            - On each iteration, shift the **fast pointer by two spaces**.
            - Shift the **slow pointer by one space** each time.
            - The fast pointer is exactly twice as fast as the slow pointer.
        - **Termination**: Continue until one pointer reaches the end. The fast pointer will reach the end first.
        - **Definition of "End"**: The fast pointer is at the end when it is at the **last node** or it is **equal to null**.
        - **Result**: Because the slow pointer travels half as fast, it will have traveled half the length of the list when the fast pointer reaches the end.
        - **Odd Length Example**: The slow pointer ends up perfectly at the middle node.
        - **Even Length Example**:
            - If we have 4 nodes (1, 2, 3, 4) and 4 points to null.
            - 1st iteration: Fast moves two spots, slow moves one.
            - 2nd iteration: Fast moves two spots (reaches null), slow moves one more spot to the second node of the middle pair.
        - **Variant**: If you want the first node to be the middle in an even-length list, you can augment the algorithm by initializing the slow pointer at the head and the fast pointer at the node after the head.
    - **Code Implementation**:

```
# Initialize two pointers at the head
slow = head
fast = head

# While fast pointer and the next node are not null
while fast is not None and fast.next is not None:
    slow = slow.next          # Increment slow by one
    fast = fast.next.next     # Increment fast by two

# When fast reaches the end, slow is at the middle
return slow
```

        - This algorithm handles **edge cases**, such as an **empty (null) linked list**. If the head is null, the loop never executes, and it returns slow (null).

- **Example 2: Cycle Detection**
    - A cycle occurs if a node (e.g., node 5) points back to a previous node (e.g., node 3).
    - In a cycle, a standard traversal loop would run forever.

- **Goal**: Given an arbitrary linked list, determine if it contains a cycle.
- **Hash Set Approach**:
  - Traverse the list and add the pointer of every node to a **hash set**.
  - If you visit a node that is already in the hash set, you have found a duplicate, which means there is a cycle.
  - If you reach null, there is no cycle.
  - **Complexity**: O(N) time, but also **O(N) memory** because of the hash set.
- **Fast and Slow Pointer (Floyd's) Approach**:
  - This eliminates the extra memory need, resulting in **O(1) space**.
  - **Algorithm**:
    - Initialize both pointers at the head.
    - Move the fast pointer by two spaces and the slow pointer by one space.
    - If the pointers **intersect (equal each other)**, return true; a cycle exists.
  - **Why it works**:
    - In a straight line, the fast pointer will always be twice as far ahead as the slow pointer.
    - In a loop, the fast pointer will eventually "catch up" and overlap the slow pointer from behind.
  - **Guarantee of Intersection**:
    - Once both pointers enter a cycle of length C, there is a distance between them.
    - Every iteration, the slow pointer moves 1 and the fast pointer moves 2. This means the **distance between them shrinks by 1** every iteration.
    - Eventually, the distance becomes zero, and they intersect.
  - **Time Complexity**:
    - It is guaranteed to happen in **linear time**.
    - The slow pointer will not have to traverse longer than the entire length of the list before an intersection occurs.
    - The fast pointer might traverse twice the length, but the overall complexity remains **O(N)**.
- **Code Implementation**:

```
slow = head
fast = head

while fast is not None and fast.next is not None:
    slow = slow.next
    fast = fast.next.next

    # If they meet, a cycle is detected
    if slow == fast:
        return True

# If fast reaches out of bounds, no cycle exists
return False
```

- **Example 3: Finding the Head of the Cycle**

- **Goal**: If a cycle exists, return the specific node that is the **head (start)** of that cycle.
- **The Algorithm**:
    1. Perform the standard cycle detection: move the fast pointer by two and the slow pointer by one until they intersect.
    2. If the fast pointer reaches null, there is no cycle; return null.
    3. If they intersect, leave the original slow pointer (**S1**) at the intersection point.
    4. Create a **second slow pointer (S2)** and initialize it at the **head** of the linked list.
    5. Increment both S1 and S2 by **one space each** simultaneously.
    6. The point where S1 and S2 intersect is guaranteed to be the start of the cycle.
- **Mathematical Proof**:
    - Define **P** as the portion of the list before the cycle.
    - Define **C** as the length of the cycle.
    - Define the intersection point as **X** distance from the head of the cycle.
    - The remaining portion of the cycle after the intersection point is **C - X**.
    - **Slow pointer distance**: Travels P and then some portion of the cycle ($C - X$).
    - **Fast pointer distance**: Travels P, a full cycle C, and then the same portion of the cycle ($C - X$).
    - Because the fast pointer is twice as fast: $2 \times (\text{Slow Distance}) = \text{Fast Distance}$.
    - $2(P + C - X) = P + C + (C - X)$.
    - Simplified: $2P + 2C - 2X = P + 2C - X$.
    - Subtract P from both sides and $2C$ from both sides: $P - 2X = -X$.
    - Solve for P: **$P = X$**.
    - This proves the distance from the head of the list to the start of the cycle (P) is equal to the distance from the intersection point to the start of the cycle (X).
- **Code Implementation**:

```
slow = head
fast = head

# Phase 1: Detect Intersection
while fast is not None and fast.next is not None:
    slow = slow.next
    fast = fast.next.next
    if slow == fast:
        break
else:
    # If the loop finished without a break, no cycle exists
    return None

# Phase 2: Find the head of the cycle
slow2 = head
while slow != slow2:
    slow = slow.next
    slow2 = slow2.next

# Both pointers now point to the start of the cycle
return slow
```

- While the proof is mathematically complex, the algorithm itself is simple and worth memorizing for interviews.

Analogy: Imagine a race on a track that has a straight path leading into a circular loop. If a fast runner (the hair) and a slow runner (the tortoise) start together, the fast runner will enter the loop first and eventually "lap" the slow runner. The moment they are at the exact same spot on the loop is the intersection used to detect the cycle.

Tries

Trie

- **Trie (Prefix Tree) Comprehensive Notes**
- **Introduction to the Trie Data Structure**
  - **Definition and Names**:
    - A "trie" is a specific type of tree structure.
    - A better and more descriptive name for it is a **prefix tree**.
  - **Primary Goal**:
    - The goal is to insert words (strings of characters) in constant time.
    - It is specifically designed for strings, rather than integers or other value types.
  - **Performance**:
    - **Insertion**: Can be considered O(1) or more accurately **O(N)**, where N is the length of the word, because the algorithm must iterate through every character.
    - **Search**: Allows for checking the existence of a word very efficiently in constant time (O(N) relative to word length).
- **Trie vs. Hash Sets**
  - **Capabilities of Hash Sets**: A hash set can easily achieve constant time insertion and searching for full words.
  - **The Limitation of Hash Sets**: Hash sets cannot efficiently **search for a prefix**.
  - **The Prefix Search Problem**:
    - In a hash map or hash set, searching for a prefix (e.g., finding if any words start with "app") would require checking every single string in the structure in the worst case.
    - In a **prefix tree**, searching for a prefix is done in **O(1)** relative to the size of the prefix itself.
    - Example: If the word "apple" is inserted, a trie can quickly confirm that the prefix "app" exists.
- **Applications and Importance**
  - **Search Engines**: Prefix trees are used by search engines like Google.
  - **Autocomplete/Suggestions**: When a user types a few characters, the trie identifies which strings match that prefix to provide suggestions.
  - **Coding Interviews**: It is a common data structure for interviews and is easier to implement than many expect.

- **Structure of a Trie**
  - **Character-Based Nodes**: A trie is a tree of characters where each node represents a character.
  - **Character Sets**:
    - Usually, the restriction is lowercase letters from **A through Z** (26 characters).
    - It can be expanded to include uppercase letters or larger alphabets without changing the fundamental structure.
  - **The Root Node**:
    - The trie has a single **root node** that is essentially empty.
    - Alternatively, you can think of it as having 26 root nodes, one for each possible starting character.
  - **Branching Logic**:
    - All words starting with 'A' follow the path down the 'A' branch.
    - Each node can have up to 26 children (A through Z).
    - A node might have anywhere from 0 to 26 children depending on what has been inserted.
- **Trie Node Implementation Details**
  - **Child Management**:
    - Using individual variables or left/right pointers for 26 children is inefficient.
    - An easier and more flexible method is using a **hashmap** for children.
  - **Flexible Design**:
    - While an array of size 26 could be used, a hashmap makes the trie flexible for different alphabets (like uppercase A-Z).
  - **The TrieNode Class Components**:
    - **Children**: An empty hashmap initially. The key is the character (e.g., 'a'), and the value is the corresponding TrieNode.
    - **Word End Indicator**: A boolean value (e.g., `wordEnd`) to determine if a character marks the end of a complete word. This is initially set to **false**.
  - **Redundancy Note**:
    - Trie nodes do **not** actually need to store the character itself.
    - The character is stored as a **key in the parent's children hashmap**. Accessing `children['a']` provides all the necessary information.
- **Core Algorithms**
  - **1. Insert Operation**
    - **Step 1**: Start at the root node (the "current pointer").
    - **Step 2**: Iterate through every character in the input word.
    - **Step 3**: Check if the character is already a child of the current node.
    - **Step 4**: If the character does not exist, create a new **TrieNode** and add it to the hashmap using the character as the key.
    - **Step 5**: Shift the current pointer to the node of the character just checked or created.
    - **Step 6**: Repeat until the end of the word.
    - **Step 7**: Mark the `wordEnd` variable of the final node as **true** (often represented visually as a green circle).
  - **2. Search Operation (Full Word)**

        - **Step 1**: Start at the root node.
        - **Step 2**: Iterate through every character of the search word.
        - **Step 3**: Check if the character exists as a child. If it does not exist at any point, return **false** immediately.
        - **Step 4**: If the character exists, move the pointer to that child node and continue.
        - **Step 5**: After iterating through all characters, return the value of the current node's `wordEnd` boolean.
        - **Example**: If "apple" is in the trie but "a" is searched, it returns **false** because the 'a' node was never marked as the end of a word.
    - **3. StartsWith Operation (Prefix Search)**
        - This is the main purpose of the trie.
        - **Step 1**: Takes a prefix as input.
        - **Step 2**: Start at the root and iterate through the prefix characters.
        - **Step 3**: Use the children hashmap to move through nodes efficiently.
        - **Step 4**: If a character in the prefix is not found, return **false**.
        - **Step 5**: If the loop completes (meaning all characters in the prefix were found), return **true**.
        - **Distinction**: Unlike the search function, this returns true even if the last node's `wordEnd` is false, as long as the path exists.

- **Efficiency and Branching Example**
    - **Avoiding Redundancy**:
        - If "apple" is inserted and then "ape" is inserted, the trie uses the existing 'a' and 'p' nodes.
        - It only creates a new node for the 'e' in "ape" starting from the second position.
        - This ensures all words starting with the same prefix follow a single branch, making searching efficient.
    - **Efficiency**:
        - Every operation (insert, search, startsWith) involves a single pass through the characters of the input.
        - Hashmap lookups for children are performed in constant time.

- **Summary of Node Logic**

```
# Conceptual Structure of a Trie Node
class TrieNode:
    def __init__(self):
        # Hashmap to store children: {char: TrieNode}
        self.children = {}
        # Boolean to mark the end of a word
        self.isWordEnd = False
```

Analogy for Understanding: Think of a Trie like a **structured dictionary** where you don't look up a whole word at once, but follow a **path of breadcrumbs**. Each letter is a signpost pointing to the next. If you are looking for the word "apple," you follow the 'A' path, then 'P', and so on. If you reach the end of your word

and find a "Finished" sign (the `wordEnd` boolean), the word exists. If you are just checking for a prefix, you only care that the path exists, not necessarily that there is a "Finished" sign at the end of it.

Union Find

- **Union Find Data Structure Overview**
    - Union Find, also known as the **Disjoint Sets** data structure, is an underrated tool.
    - It is technically a **tree data structure**, though it is most commonly applied to **generic graphs**.
    - While implementation can seem complicated due to its application to graphs, it is actually easier to implement than expected.
    - Its primary strength is dealing with **disjoint sets** where a graph may have one or more **connected components**,.
    - In a typical graph, all nodes might not be connected; Union Find helps identify these separate entities.
- **Key Applications and Comparisons**
    - **Counting Connected Components**: It is used to determine how many separate groups of connected nodes exist in a graph.
    - **Cycle Detection**:
        - It can determine if a graph contains cycles.
        - If an edge is added between two nodes that are already part of the same connected component, Union Find detects a cycle.
    - **Comparison to DFS**:
        - A generic Depth First Search (DFS) algorithm can often satisfy the same operations (cycle detection and counting components).
        - Because DFS is common, Union Find is seen less frequently.
        - However, Union Find is sometimes **more efficient** and occasionally required,.
- **Conceptual Implementation**
    - **Forest of Trees**: Union Find is represented as a "forest of trees," meaning it consists of a collection of multiple trees.
    - **Initialization**:
        - Initially, every node is assumed to be a disjoint set (its own tree).
        - For each node, the structure only stores one piece of information: the **parent** of that node.
        - In code, it is common to initialize every node as its own parent (`parent[i] = i`),.
    - **Connecting Nodes (Union)**:
        - When an edge exists between two nodes (e.g., node 1 and node 2), the two sets are connected.
        - To connect them, one node is set as the parent of the other node.
        - For example, if node 1 and 2 are connected, node 2 can become a child of node 1.
- **The Find Operation**
    - The `find` function is used to determine the **root parent** of a node,.

- **Traversing the Tree**: To find the root, you start at a node and follow the parent pointers up the chain until you reach a node that is its own parent.
- **Example**: To find the root of node 4, you move to its parent, then that parent's parent, until you cannot go any higher.
- This root represents the "set" that the node belongs to.

- **The Union Operation**
  - To union two nodes (e.g., n1 and n2), you must first find the **root parent** of both,.
  - If both nodes have the **same root parent**, they are already in the same set, and a union is not performed (this indicates a cycle),.
  - If they have different roots, the root of one tree is set as the child of the root of the other tree, merging the two sets.
  - **Union by Rank (Height)**:
    - Arbitrarily picking which root becomes the parent can result in imbalanced trees (like a linked list), making the `find` operation inefficient,.
    - **Rank** refers to the **height** of the tree,.
    - The more efficient method is to take the root of the tree with the **smaller height** and make it a child of the root of the tree with the **larger height**,.
    - If the heights are equal, one is chosen arbitrarily to be the parent, and its rank is increased by one,.

- **Path Compression Optimization**
  - Path compression is a single-line optimization for the `find` function,.
  - As the algorithm traverses up the tree to find the root, it **shortens the chain** by pointing nodes directly to their "grandparent".
  - This does not improve the time complexity of the very first `find` operation, but it makes every subsequent `find` on those nodes much faster by creating a direct link toward the root,.

- **Code Structure and Logic**
  - **Data Structures**:
    - A **hashmap** or **array** is used to store the `parent` of each node.
    - A second hashmap or array is used to store the `rank` (height) of each tree.
  - **Constructor**:
    - Takes the size `n` (number of nodes).
    - Initializes `parent[i] = i` and `rank[i] = 0` (or 1, as long as consistent) for all nodes.
  - **Find Logic**:

```
# Conceptual logic for Find with Path Compression
def find(n):
    p = parent[n]
    while p != parent[p]:
        # Path compression: set parent to grandparent
        parent[p] = parent[parent[p]]
        p = parent[p]
    return p
```

,

- **Union Logic**:
  - Find root `p1` of `n1` and root `p2` of `n2`.
  - If `p1 == p2`, return `false` (cycle detected/already connected).
  - If `rank[p1] > rank[p2]`, set `parent[p2] = p1`.
  - If `rank[p2] > rank[p1]`, set `parent[p1] = p2`.
  - If `rank[p1] == rank[p2]`, set `parent[p1] = p2` and `rank[p2] += 1`,.

- **Time Complexity**

  - **Naive Case**: Without path compression or union by rank, `find` can take **O(n)** time because the tree can become imbalanced.
  - **With One Optimization**: Implementing either path compression OR union by rank reduces the time complexity of `find` to **O(log n)**,.
  - **With Both Optimizations**:
    - Using both path compression and union by rank results in a time complexity based on the **Inverse Ackermann function**.
    - This function grows so slowly that it is essentially **constant time O(1)** for all practical values of `n`.
  - **Total Complexity**: For a graph with `M` edges, the total time complexity is effectively **O(M)** when both optimizations are used.
  - This efficiency is why Union Find can outperform DFS for specific graph problems.

- **Analogy for Understanding**

  - Think of Union Find like several different families (sets) merging through marriage. When two people from different families marry, one entire family (the smaller one) joins the other. Path compression is like everyone in the family getting the direct phone number for the family patriarch (the root), rather than having to call their parents, who then call their grandparents, to get information.

Segment Tree

- **Motivation for Segment Trees**

  - The segment tree is considered the **most complicated tree structure** discussed so far.
  - It is designed to solve problems involving an **array of values** where two main operations are required.
  - The first operation is **updating a value** at a specific index.
  - In a standard array, an update is very efficient and can be done in **constant time (O(1))** by overwriting the value.
  - The second operation is **querying a range** between a left and right index inclusive to get a result, such as a **sum**.
  - This sum query is the textbook problem for segment trees.
  - In a standard array, a range query takes **linear time (O(n))** in the worst case because you must iterate from left to right.
  - While **prefix sums** can make queries constant time, this optimization only works if the array values never change.
  - If updates are allowed, an array cannot achieve constant time queries; it remains **O(n)**.

    - A **segment tree** allows both **updates and range queries to be performed in logarithmic time (O(log n))**.

- **Trade-offs and General Structure**

    - The update operation in a segment tree is actually a **downside** compared to an array because it moves from O(1) to **O(log n)**.
    - This adds overhead to updates, but the trade-off is a much more **efficient query** compared to the linear time required by an array.
    - Having O(log n) for both is better than having one O(1) and one O(n) operation if you are frequently performing both tasks.
    - The name "segment tree" comes from the fact that it **breaks an array into segments**.
    - The **root node represents the entire array range**.
    - As you move down to the children, the array is broken into **smaller and smaller portions**.
    - The **left half** of an array segment belongs to the left child, and the **right half** belongs to the right child.
    - When updating a value, you cannot just update one node; you must update the **entire chain of ancestors** (the leaf, its parent, and so on up to the root) that contains that index.
    - This chain of updates is why the update operation takes **O(log n)** time.

- **Building a Segment Tree**

    - To build a tree for an array with indices **0 through 5**, the **root represents the range**.
    - The construction uses a **divide and conquer** approach similar to **merge sort**.
    - To find the boundaries for children, you calculate the **midpoint (m)** by adding the left (L) and right (R) indices and dividing by two, rounding down.
    - For the range, the midpoint is **(0 + 5) / 2 = 2**.
    - The **left child** boundaries are the current left index to the midpoint: **[L, m]** (e.g.,).
    - The **right child** boundaries are the midpoint plus one to the current right index: **[m + 1, R]** (e.g.,).
    - This process repeats **recursively**:
        - Range splits into and (midpoint is 1).
        - Range splits into and (midpoint is 0).
        - Range splits into and (midpoint is 4).
        - Range splits into and (midpoint is 3).
    - A **base case** occurs when a range has a length of one (e.g.,), meaning it cannot be broken up further and has no children.
    - Each node stores a value representing its range, such as a **sum**.
    - **Leaf nodes** (base cases) take their sum directly from the array values at that index (e.g., index 0 = 5, index 1 = 3, index 2 = 7, index 3 = 1, index 4 = 4, index 5 = 2).
    - As recursion "pops back up," the sum for a parent node is calculated by **adding the sums of its two children**.
    - For example, if children are 5 and 3, the parent sum is **8**; if children are 8 and 7, the parent sum is **15**.
    - The **root node's sum** will eventually equal the sum of the entire array (e.g., 22).

- **Implementation Details**

- A **SegmentTree class** can be used for implementation.
- Each node in the class contains:
  - `total`: The range sum (or other value being tracked).
  - `L` and `R`: The **boundaries** of the range the node represents.
  - `left` and `right`: **Pointers** to the child nodes.
- A **static build method** can construct the tree in **linear time (O(n))**.
- The build process is **bottom-up**:
  - If `L == R`, create a leaf node with the array value and return it.
  - Otherwise, calculate the **middle**, create a temporary node with a sum of 0, and recursively build the left and right children.
  - After children are built, set the node's total to `left.total + right.total`.
- While segment trees are similar to **heaps** (full binary trees), they are harder to implement with arrays because they can have **gaps** in the tree structure.
- Therefore, using **nodes and pointers** is often preferred.

- **Update Operation**

  - The update operation takes an **index and a new value** and runs in **O(log n)** time.
  - You must find the **leaf node** representing the specific index by searching recursively.
  - If the target index is **greater than the midpoint** of the current node, you move to the **right child**.
  - If the target index is **less than or equal to the midpoint**, you move to the **left child**.
  - Once you reach the **base case** (the leaf node where `L == R`), you overwrite its value.
  - Because the value has changed, you must update the sums of **all ancestors** in the chain back to the root.
  - As the recursion returns to each parent, you recalculate its sum by **adding the current sums of its two children**.
  - This is a **constant time operation** at each level of the tree.
  - For example, if you update index 3 from 1 to 4:
    - The leaf for index 3 becomes 4.
    - Its parent updates from 5 to **8** (4 + 4).
    - Its grandparent updates from 7 to **10** (8 + 2).
    - The root updates from 22 to **25** (15 + 10).
  - The tree is always **balanced** because it is split into equal halves, ensuring the height is always **log n**.

- **Range Query Operation**

  - The goal is to calculate the **range sum** between a left and right index inclusive.
  - If the query range **exactly matches** the node's range, you simply return the node's sum.
  - To find the sum for a specific range, you calculate the **midpoint** and determine where the range lies.
  - There are **three possibilities** for where the queried range lies relative to the current node:
    1. **Entirely in the right subtree**: This happens if the query's **left index is greater than the midpoint**.
    2. **Entirely in the left subtree**: This happens if the query's **right index is less than or equal to the midpoint**.
    3. **Overlaps both subtrees**: This happens if the query range spans across the midpoint.

    - If the range is only on one side, you only recurse down that side of the tree.
    - If the range **overlaps both sides**, you call the query function **recursively on both children**.
        - For the left child, you query the range from the original **left to the midpoint**.
        - For the right child, you query the range from the **midpoint plus one to the original right**.
        - You add the results from both sides together and return the total.
    - **Optimization**: You often find the answer before reaching the leaf nodes because a node might represent the exact sub-range you need.
    - For example, to find the sum of range:
        - At root, the range overlaps both sides (midpoint 2).
        - Left child is queried for index 2; it returns **7**.
        - Right child is queried for range; it returns **5** (the pre-calculated sum for that segment).
        - The total returned is 7 + 5 = **12**.
    - The **time complexity is O(log n)** because at any level of the tree, you look at at most a **constant number of nodes (around 4)**.

- **Analogy**

    - A segment tree is like a **corporate management structure**. The CEO (root) knows the total productivity of the entire company. Each department head (child node) knows the productivity of their specific branch. If one employee (leaf node) changes their output, the employee's manager must update their total, then the department head must update theirs, and finally the CEO updates the company total. When the CEO needs to know the productivity of a specific group of departments, they don't ask every employee; they just ask the few managers who already have the totals for those specific groups.

Iterative DFS

- **Iterative DFS on Binary Trees**

    - **Overview of Iterative DFS**

        - Binary trees are familiar data structures where Depth-First Search (DFS) is a common algorithm.
        - While DFS is easy to implement using recursion, it is more complex to implement iteratively without recursion.
        - Iterative DFS is performed by emulating how the recursive function actually runs using an explicit stack,.
        - The three common traversals for a tree are:
            - **In-order traversal**: The easiest to do iteratively.
            - **Pre-order traversal**: Similar to in-order but with a different processing sequence,.
            - **Post-order traversal**: The hardest to implement iteratively.
        - **Traversing** a node means performing an operation on it, such as printing its value.

    - **In-Order Traversal**

        - **Recursive Concept**
            - The goal is to traverse the entire left subtree, then the root node, and then the right subtree,.
            - In recursion, when a null node is reached, the function simply returns.

        - Returning from a null node pops the execution back to the previous node in the recursive call stack.
    - **Iterative Implementation via Explicit Stack**
        - Instead of a call stack, a stack data structure is explicitly declared and used the same way.
        - **Tree Nodes** (or pointers to them) are added to the stack, not just their literal values.
    - **Step-by-Step Example (Nodes 1, 2, 3, 4, 5)**
        - Start at the root node (1). Before printing, the left subtree must be processed.
        - Push 1 onto the stack, then move to the left child (2).
        - Push 2 onto the stack, then move to the left child (4),.
        - Push 4 onto the stack, then move to its left child, which is null.
        - When null is reached, do not push anything; instead, pop from the stack.
        - Pop 4 and print it. Since 4's left subtree was null and 4 is now printed, move the pointer to the right child of 4,.
        - The right child of 4 is null. Pop from the stack again to get the parent node.
        - Pop 2 and print it. This indicates the left subtree of 2 is finished and 2 itself is processed.
        - Move to the right child of 2. It is null, so pop from the stack again.
        - Pop 1 and print it. The left subtree of 1 is now fully traversed.
        - Set the pointer to the right child of 1 (node 3) and run DFS on this subtree.
        - Since 3 is not null, push it to the stack and go to its left child (null).
        - Since the left child is null, pop 3 and print it.
        - Shift the pointer to the right child of 3 (node 5). Since 5 is not null, push it to the stack and go to its left child (null).
        - Pop 5 and print it, then go to the right child (null).
        - The algorithm stops when the pointer is null and the stack is empty.
    - **Code and Logic**
        - Initialize an empty stack and set a `current` pointer to the root.
        - The loop continues as long as `current` is not null OR the stack is not empty.
        - If `current` is not null:
            - Push `current` to the stack and move `current` to the left child.
        - If `current` is null (else case):
            - Pop a node from the stack, print its value, and set `current` to the popped node's right child,.
    - **Complexity**
        - **Time Complexity**: O(N) because every node is traversed.
        - **Memory Complexity**: O(H), where H is the height of the tree. In the worst case (an imbalanced tree acting like a chain), this is O(N).

- **Pre-Order Traversal**

    - **Iterative Process**
        - Pre-order means processing the current node before processing its children (Node -> Left Subtree -> Right Subtree).
        - Print the current node immediately upon visiting it.

        - To ensure the right subtree can be accessed later, push the right child onto the stack before moving the pointer to the left child.
    - **Step-by-Step Example**
        - Start at root (1). Print 1. Push its right child (3) to the stack and move the pointer to the left child (2),.
        - At node 2, print 2. Its right child is null, so nothing is pushed. Move pointer to left child (4),.
        - At node 4, print 4. Its right child is null. Move pointer to left child (null).
        - When the pointer is null, pop from the stack.
        - Pop 3 and print it. Push its right child (5) to the stack and move pointer to the left child (null).
        - Pointer is null, so pop 5 and print it.
    - **Code and Logic**
        - Similar loop structure: `while (curr != null || !stack.isEmpty())`.
        - If `current` is not null:
            - Print the node value.
            - If the right child exists, push it to the stack.
            - Move `current` to the left child.
        - If `current` is null:
            - Pop from the stack to set the new `current` pointer.
    - **Complexity**
        - **Time Complexity**: O(N).
        - **Space Complexity**: O(H), which can be O(N) in the worst case.

- **Post-Order Traversal**

    - **The Difficulty of Post-Order**
        - Post-order requires traversing the left subtree, then the right subtree, and finally the node (Left -> Right -> Node).
        - Using a single stack is difficult because you visit the same node multiple times and need a way to know when to process the right child versus the node itself,.
    - **The Two-Stack / Visit-Stack Solution**
        - One stack stores the nodes; another stack (the "visit stack") marks whether a node has been visited before using boolean values (True/False).
        - Alternatively, this can be implemented as a single stack of pairs (node, boolean).
    - **The Algorithm Logic**
        - Initialize the stack with the root node and mark it as `False` (not visited),.
        - While the stack is not empty:
            - Pop the node and its visited status.
            - If the node is null, do nothing.
            - If `visited` is `True`:
                - Print the node value.
            - If `visited` is `False`:
                - Push the node back onto the stack but mark it as `True`,.
                - Push the right child onto the stack marked as `False`,.
                - Push the left child onto the stack marked as `False`,.

        - Pushing the right child before the left child ensures the left child is popped and processed first.
    - **Step-by-Step Example**
        - Start with root (1) as `False`. Pop 1. Since it's `False`, push 1 as `True`, then push 3 as `False`, then 2 as `False`,.
        - Pop 2 (`False`). Push 2 as `True`, push its right child (null) as `False`, push its left child (4) as `False`.
        - Pop 4 (`False`). Push 4 as `True`, push its children (null, null) as `False`.
        - Pop null nodes and do nothing. Pop 4 (`True`) and print it.
        - Pop null (right child of 2). Pop 2 (`True`) and print it.
        - Pop 3 (`False`). Push 3 as `True`, push 5 as `False`, push left child (null) as `False`.
        - Pop null. Pop 5 (`False`). Push 5 as `True`, push its children (null, null) as `False`.
        - Pop nulls. Pop 5 (`True`) and print it.
        - Pop 3 (`True`) and print it.
        - Pop 1 (`True`) and print it.
    - **Complexity**
        - **Time Complexity**: O(N). Each node is pushed and popped twice, resulting in O(2*N), which reduces to linear time.
        - **Space Complexity**: O(H), worst case O(N).

- **General Applications and Recommendations**

    - A common application of iterative DFS is implementing a **Binary Search Tree (BST) iterator**.
    - A BST iterator allows fetching values in order one by one (e.g., getting the first two values and pausing) without traversing the entire tree at once.
    - If recursion is allowed, it is generally recommended over iterative methods because recursive DFS is easier to implement.

Others

Prompts

Notes Formatting

```
Create super depth notes in Markdown (.md) format with 100% information
preserved, no loss. Use simple grammar and keep everything clear, direct,
and well-structured. using headings, subheadings,paragraphs, statements and
code blocks when needed. Include every detail, definition, example, and
step exactly from the source. transform the given content into clean,
readable .md format.
and no #, just nested - lines plaintext
```

LaTeX math symbols/commands -> Markdown-friendly alternatives

```
\to => ->
\times => *
$ => ** or `
\log => LOG
\rightarrow => ->
\cdot => *
\frac => \
```

End-of-File

The kintsugi-stack repository, authored by Kintsugi-Programmer, is less a comprehensive resource and more an Artifact of Continuous Research and Deep Inquiry into Computer Science and Software Engineering. It serves as a transparent ledger of the author's relentless pursuit of mastery, from the foundational algorithms to modern full-stack implementation.

> Made with 💚 Kintsugi-Programmer