A Comprehensive Guide to 15 Core Algorithmic Patterns in C++ for the Modern Technical Interview

Introduction

The landscape of technical interviews at leading technology firms has undergone a significant paradigm shift. The previous emphasis on memorizing a vast catalog of specific problem solutions has become ineffective and is now largely obsolete. Today's interviews are designed to assess a candidate's fundamental problem-solving acumen, particularly their ability to recognize underlying patterns within seemingly novel problems. This approach tests for a deeper, more transferable understanding of data structures and algorithms (DSA). Mastering these foundational patterns allows a candidate to deconstruct complex challenges into familiar components, articulate a clear and optimal thought process, and ultimately, demonstrate the analytical skills required for a successful engineering career.

This report serves as an exhaustive guide to 15 of the most crucial algorithmic patterns that frequently appear in technical interviews. The content is systematically organized into three primary domains of non-linear data structures: Trees, Heaps, and Graphs. Each pattern is presented as a self-contained module, featuring a detailed conceptual framework, strategic guidance for problem recognition, a complete and annotated C++ implementation, a rigorous complexity analysis, and a nuanced discussion of its strategic implications. This guide is structured to facilitate a transition from rote learning to true pattern recognition, equipping engineering candidates with the expertise needed to excel in the modern technical interview.

Part I: Tree-Based Algorithmic Patterns

Trees are foundational, non-linear data structures that represent hierarchical relationships.[2] Their structure, where each node has a single parent (except the root) and can have multiple

children, makes their traversal a non-trivial subject with several distinct and powerful approaches.[4] The following six patterns cover the essential techniques for navigating, validating, and manipulating tree structures.

1. Pattern: Breadth-First Search (BFS) / Level Order Traversal

Conceptual Framework

Breadth-First Search (BFS), when applied to trees, is more commonly known as Level Order Traversal. This pattern involves a systematic, layer-by-layer exploration of the tree's nodes.[6] The traversal begins at the root node (level 0), then visits all nodes at level 1, followed by all nodes at level 2, and so on, until all nodes have been visited.[4] The core mechanism enabling this level-by-level progression is a queue, which adheres to a First-In-First-Out (FIFO) principle. Nodes are added to the queue as they are discovered, and the algorithm processes them in the order they were added, naturally ensuring that all nodes at a given depth are processed before moving to the next depth level.[7]

Problem Recognition

The BFS pattern should be employed in the following scenarios:

- **Explicit Level-Order Requirement:** The problem statement directly asks for a "level-order traversal," "zigzag traversal," or to process the tree "level by level".[2]
- **Shortest Path in Unweighted Trees/Graphs:** When the goal is to find the shortest path (in terms of the number of edges) from a source node to a target node in an unweighted tree or graph. The level-by-level nature of BFS guarantees that the first time a node is reached, it is via the shortest possible path.[10]
- **Level-Specific Operations:** Problems that require operations on all nodes at a specific level, such as finding the maximum value per level, connecting nodes at the same level, or calculating the average of values at each level.[4]

C++ Implementation and Walkthrough

The following C++ code demonstrates a standard implementation of Level Order Traversal. It uses a std::queue to manage the nodes to be visited and returns a vector<vector<int>>, where each inner vector represents a level in the tree.

C++

```cpp
#include <iostream>
#include <vector>
#include <queue>

// Definition for a binary tree node.
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

class Solution {
public:
    std::vector<std::vector<int>> levelOrder(TreeNode* root) {
        std::vector<std::vector<int>> result;
        if (!root) {
            return result;
        }

        std::queue<TreeNode*> q;
        q.push(root);

        while (!q.empty()) {
            int levelSize = q.size(); // Number of nodes at the current level
            std::vector<int> currentLevel;

            for (int i = 0; i < levelSize; ++i) {
                TreeNode* node = q.front();
                q.pop();
                currentLevel.push_back(node->val);
```

```
            if (node->left) {
                q.push(node->left);
            }
            if (node->right) {
                q.push(node->right);
            }
        }
        result.push_back(currentLevel);
    }
    return result;
  }
};

// Helper function to create a new node
TreeNode* newNode(int data) {
  return new TreeNode(data);
}

int main() {
  Solution sol;
  TreeNode *root = newNode(3);
  root->left = newNode(9);
  root->right = newNode(20);
  root->right->left = newNode(15);
  root->right->right = newNode(7);

  std::vector<std::vector<int>> traversal = sol.levelOrder(root);

  std::cout << "Level Order Traversal:" << std::endl;
  for (const auto& level : traversal) {
    std::cout << "[ ";
    for (int val : level) {
      std::cout << val << " ";
    }
    std::cout << "]" << std::endl;
  }

  // Clean up memory
  // (A proper tree deletion function would be needed for a complete application)
  delete root->right->right;
  delete root->right->left;
  delete root->right;
```

```
    delete root->left;
    delete root;

    return 0;
}
```

Walkthrough:

1. An empty result vector and a queue q are initialized. The root node (value 3) is pushed into the queue.
2. The while loop begins. The levelSize is 1. The loop runs once.
3. Node 3 is dequeued, its value is added to currentLevel. Its children, 9 and 20, are enqueued. currentLevel `` is pushed to result.
4. The while loop continues. The queue now contains ``. levelSize is 2. The loop runs twice.
5. Node 9 is dequeued and added to currentLevel. It has no children.
6. Node 20 is dequeued and added to currentLevel. Its children, 15 and 7, are enqueued.
7. currentLevel `` is pushed to result.
8. The while loop continues. The queue now contains ``. levelSize is 2. The loop runs twice.
9. Nodes 15 and 7 are dequeued in turn, their values are added to currentLevel, and they have no children to enqueue.
10. currentLevel `` is pushed to result.
11. The queue is now empty, the while loop terminates, and the final result is returned.

Complexity Analysis

- **Time Complexity: O(N)** where N is the number of nodes in the tree. Each node is enqueued and dequeued exactly once, leading to a linear time complexity.[12]
- **Space Complexity: O(W)** where W is the maximum width of the tree. The space required is determined by the maximum number of nodes that can be in the queue at any one time. In the worst-case scenario of a complete binary tree, the last level contains approximately N/2 nodes, making the space complexity O(N).[8]

Strategic Implications

The BFS algorithm is more than a simple traversal; it is fundamentally a shortest-path algorithm for unweighted graphs. The FIFO nature of the queue ensures that all paths of length k are fully explored before any path of length k+1 is considered. Consequently, the first time a target node is discovered, it is guaranteed to have been found via a path with the

minimum number of edges. This property is the primary reason BFS is the algorithm of choice for finding the shortest path in unweighted scenarios.[10] While a recursive implementation of level-order traversal is possible, the iterative, queue-based approach is generally superior in an interview setting. It is more intuitive, directly models the level-by-level process, and avoids potential stack overflow issues with very deep trees.[13]

2. Pattern: Depth-First Search (DFS) for Trees

Conceptual Framework

Depth-First Search (DFS) is a traversal strategy that explores as far as possible along each branch before backtracking.[4] For tree structures, this "depth-first" philosophy is most naturally and elegantly implemented using recursion, where the function call stack implicitly manages the state needed for backtracking.[2] Unlike the single method of BFS, DFS for binary trees is commonly expressed in three distinct variants, differentiated by the order in which the current node (the "root" of the subtree) is processed relative to its left and right subtrees.[4]

- **Pre-order Traversal (Root → Left → Right):** The current node is processed first, followed by a recursive traversal of the entire left subtree, and then a recursive traversal of the entire right subtree. This "top-down" approach is useful for tasks like creating a copy of a tree, as the root must be created before its children can be attached.[4] It is also used to obtain the prefix notation of an expression tree.[4]
- **In-order Traversal (Left → Root → Right):** The left subtree is traversed completely, then the current node is processed, and finally, the right subtree is traversed. The most critical application of this pattern is on a Binary Search Tree (BST), where an in-order traversal visits the nodes in their natural, non-decreasing sorted order.[4]
- **Post-order Traversal (Left → Right → Root):** Both the left and right subtrees are traversed completely before the current node is processed. This "bottom-up" strategy is essential for operations where children must be fully handled before their parent. A canonical example is deleting a tree, where child nodes must be deallocated before their parent to prevent memory leaks and dangling pointers.[4] It is also used to get the postfix expression of an expression tree.[4]

Traversal Algorithm	Order of Operations	Primary Data Structure	Logical Purpose	Common Application

BFS	Level-by-level	Queue	Shortest Path (unweighted)	Find nodes at level K
Pre-order DFS	Root ⟶ Left ⟶ Right	Stack (Recursive)	Top-down processing	Copying a tree
In-order DFS	Left ⟶ Root ⟶ Right	Stack (Recursive)	Sorted order (for BST)	Validating a BST
Post-order DFS	Left ⟶ Right ⟶ Root	Stack (Recursive)	Bottom-up processing	Deleting a tree

Problem Recognition

DFS is the preferred pattern for problems involving:

- **Path Exploration:** Finding a path from the root to a leaf that satisfies certain conditions, such as in "Binary Tree Path Sum".[6]
- **Structure and Existence Checks:** Determining if a tree is a valid BST, if it contains a specific value, or if it is a subtree of another tree.
- **Subtree Computations:** Any problem where a property of a node depends on the computed properties of its children (e.g., calculating the height or diameter of a tree) often lends itself to a post-order traversal.

C++ Implementation and Walkthrough

The following C++ code provides recursive implementations for all three DFS traversal variants.

C++

```
#include <iostream>
```

```cpp
#include <vector>

// Definition for a binary tree node.
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

class Solution {
public:
    // Pre-order: Root -> Left -> Right
    void preOrderTraversal(TreeNode* root) {
        if (root == nullptr) {
            return;
        }
        std::cout << root->val << " "; // Process root
        preOrderTraversal(root->left);    // Recur on left subtree
        preOrderTraversal(root->right);   // Recur on right subtree
    }

    // In-order: Left -> Root -> Right
    void inOrderTraversal(TreeNode* root) {
        if (root == nullptr) {
            return;
        }
        inOrderTraversal(root->left);     // Recur on left subtree
        std::cout << root->val << " "; // Process root
        inOrderTraversal(root->right);    // Recur on right subtree
    }

    // Post-order: Left -> Right -> Root
    void postOrderTraversal(TreeNode* root) {
        if (root == nullptr) {
            return;
        }
        postOrderTraversal(root->left);    // Recur on left subtree
        postOrderTraversal(root->right);   // Recur on right subtree
        std::cout << root->val << " "; // Process root
    }
};
```

```
// Helper function to create a new node
TreeNode* newNode(int data) {
    return new TreeNode(data);
}

int main() {
    Solution sol;
    /*
          1
        /   \
       2     3
      / \
     4   5
    */
    TreeNode *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);

    std::cout << "Pre-order traversal: ";
    sol.preOrderTraversal(root);
    std::cout << std::endl;

    std::cout << "In-order traversal: ";
    sol.inOrderTraversal(root);
    std::cout << std::endl;

    std::cout << "Post-order traversal: ";
    sol.postOrderTraversal(root);
    std::cout << std::endl;

    // Clean up memory
    delete root->left->right;
    delete root->left->left;
    delete root->right;
    delete root->left;
    delete root;

    return 0;
}
```

Walkthrough:

The key difference between the three functions is the position of the std::cout << root->val << " "; statement.

- In preOrderTraversal, it appears before the recursive calls, resulting in the output: 1 2 4 5 3.
- In inOrderTraversal, it is placed between the calls, yielding: 4 2 5 1 3.
- In postOrderTraversal, it is after both calls, producing: 4 5 2 3 1.

Complexity Analysis

- **Time Complexity: O(N)** for all three variants, as each node is visited and processed exactly once.
- **Space Complexity: O(H)**, where H is the height of the tree. This space is consumed by the recursion call stack. For a balanced tree, this is O(logN). In the worst-case of a completely skewed tree (resembling a linked list), the height is N, leading to O(N) space complexity.

Strategic Implications

The three DFS traversals are not merely arbitrary sequences but represent distinct logical processing orders for hierarchical data. Recognizing which logical order a problem demands is the key to selecting the appropriate DFS variant. Pre-order embodies a "top-down" logic: process a parent before its descendants. Post-order embodies a "bottom-up" logic: process all descendants before their parent. In-order is unique to binary trees and provides a linear, sorted representation when applied to a BST. This connection between the algorithm's structure and its logical purpose is fundamental. For instance, calculating the size of each subtree requires a post-order traversal; the size of a parent node can only be calculated after the sizes of its left and right subtrees are known.

3. Pattern: Path Sum & Subtree Problems

Conceptual Framework

This pattern is a direct and powerful application of Depth-First Search, tailored for problems that involve aggregating or checking properties along root-to-leaf paths or within subtrees. The core technique involves passing state information down through the parameters of a recursive DFS function. As the traversal descends a path, this state is updated. When a leaf node is reached, the accumulated state is evaluated against a target condition. The natural backtracking of recursion ensures that the state is correctly managed without manual intervention.[6]

Problem Recognition

This pattern is indicated when a problem asks to:

- Find if a root-to-leaf path exists with a specific sum (e.g., "Path Sum").[6]
- Find all root-to-leaf paths that sum to a target value (e.g., "All Paths for a Sum").[6]
- Count the number of paths (not necessarily root-to-leaf) that sum to a target value (e.g., "Count Paths for a Sum").[6]
- Calculate a property of a tree that depends on subtree properties, like "Diameter of Binary Tree" or "Maximum Path Sum" (between any two nodes).

C++ Implementation and Walkthrough

The following C++ code solves the classic "Path Sum" problem: given a binary tree and a target sum, determine if there exists a root-to-leaf path whose node values sum up to the target.

C++

```
#include <iostream>

// Definition for a binary tree node.
struct TreeNode {
    int val;
```

```
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

class Solution {
public:
    bool hasPathSum(TreeNode* root, int targetSum) {
        if (root == nullptr) {
            return false;
        }

        // Check if it's a leaf node and if the path sum equals targetSum
        if (root->left == nullptr && root->right == nullptr && root->val == targetSum) {
            return true;
        }

        // Recursively check the left and right subtrees with the updated targetSum
        bool leftHasPath = hasPathSum(root->left, targetSum - root->val);
        bool rightHasPath = hasPathSum(root->right, targetSum - root->val);

        return leftHasPath |

| rightHasPath;
    }
};

// Helper function to create a new node
TreeNode* newNode(int data) {
    return new TreeNode(data);
}

int main() {
    Solution sol;
    /*
          5
        /   \
       4     8
      /     / \
     11    13  4
    / \       \
   7   2       1
   */
    TreeNode *root = newNode(5);
```

```
    root->left = newNode(4);
    root->right = newNode(8);
    root->left->left = newNode(11);
    root->left->left->left = newNode(7);
    root->left->left->right = newNode(2);
    root->right->left = newNode(13);
    root->right->right = newNode(4);
    root->right->right->right = newNode(1);

    int targetSum = 22;
    if (sol.hasPathSum(root, targetSum)) {
        std::cout << "A path with sum " << targetSum << " exists." << std::endl;
    } else {
        std::cout << "No path with sum " << targetSum << " exists." << std::endl;
    }

    // Clean up memory
    delete root->right->right->right;
    delete root->right->right;
    delete root->right->left;
    delete root->left->left->right;
    delete root->left->left->left;
    delete root->left->left;
    delete root->right;
    delete root->left;
    delete root;

    return 0;
}
```

Walkthrough:
The implementation cleverly avoids passing an accumulating sum downwards. Instead, it subtracts the current node's value from the targetSum at each step.

1. hasPathSum(root=5, targetSum=22) is called. It is not a leaf.
2. It recursively calls hasPathSum(root=4, targetSum=17) and hasPathSum(root=8, targetSum=17).
3. The call for node 4 leads to hasPathSum(root=11, targetSum=13).
4. This leads to hasPathSum(root=7, targetSum=2) and hasPathSum(root=2, targetSum=11).
5. The call for node 7 is a leaf, but 7!= 2, so it returns false.
6. The call for node 2 is a leaf, and 2!= 11, so it returns false.
7. The recursion backtracks. The right path from the root is explored. Eventually, the path 5 → 8 → 4 → 5 is explored (this example tree doesn't have a path sum of 22, let's trace for the path 5 → 4 → 11 → 2 which sums to 22).

8. hasPathSum(root=5, targetSum=22) → hasPathSum(root=4, targetSum=17) → hasPathSum(root=11, targetSum=13) → hasPathSum(root=2, targetSum=2).
9. The call for node 2 is a leaf, and its value 2 equals the remaining targetSum 2. This call returns true.
10. This true value is propagated up the call stack, resulting in the final answer being true.

Complexity Analysis

- **Time Complexity: O(N)**, as the algorithm must visit each node in the tree in the worst case.
- **Space Complexity: O(H)**, where H is the height of the tree, for the recursion stack. This becomes O(N) for a skewed tree and O(logN) for a balanced tree.

Strategic Implications

This pattern exemplifies the use of recursion parameters to maintain path-specific state. A naive approach might involve generating all possible root-to-leaf paths and then summing each one, which is highly inefficient. The recursive DFS approach is superior because it integrates the traversal and the summation into a single, elegant pass. The state (targetSum - root->val) is passed by value in each recursive call. This is a critical detail: it means that when a recursive call returns, the state in the parent's stack frame is unaffected. This perfectly models the backtracking process, as the state automatically reverts to what it was at the parent node, allowing the traversal to proceed down another branch without any manual state cleanup.

4. Pattern: Lowest Common Ancestor (LCA)

Conceptual Framework

The Lowest Common Ancestor (LCA) of two nodes, p and q, in a tree is defined as the lowest (i.e., deepest) node that has both p and q as its descendants. A node is considered a

descendant of itself.[17] The standard and most elegant solution for a general binary tree employs a recursive DFS strategy. The logic hinges on the results of searching the left and right subtrees:

- If p is found in one subtree and q is found in the other subtree of the current node, then the current node must be the LCA.
- If both p and q are found in the same subtree (e.g., the left), then the LCA must also reside within that same subtree, and the result from that subtree's recursive call is propagated upwards.
- If the current node is either p or q, it is a potential LCA and is returned up the call stack.[20]

Problem Recognition

This is a highly specific and common interview pattern. It should be recognized when a problem explicitly asks for the "Lowest Common Ancestor," "LCA," or "First Common Ancestor" of two nodes in a tree.

C++ Implementation and Walkthrough

The following C++ code provides the standard recursive solution for finding the LCA in a binary tree.

C++

```
#include <iostream>
#include <vector>

// Definition for a binary tree node.
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};
```

```
class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        // Base case: if root is null, or if root is one of the nodes, it's the LCA.
        if (root == nullptr |

| root == p |
| root == q) {
            return root;
        }

        // Search for p and q in the left and right subtrees.
        TreeNode* leftLCA = lowestCommonAncestor(root->left, p, q);
        TreeNode* rightLCA = lowestCommonAncestor(root->right, p, q);

        // If p and q are found in different subtrees, then the current root is the LCA.
        if (leftLCA!= nullptr && rightLCA!= nullptr) {
            return root;
        }

        // Otherwise, the LCA is in the subtree where a node was found.
        // If both were in the same subtree, that subtree's LCA would have been returned.
        // If only one was found, that node itself is the LCA (as the other is its descendant).
        return (leftLCA!= nullptr)? leftLCA : rightLCA;
    }
};

int main() {
    Solution sol;
    TreeNode* root = new TreeNode(3);
    root->left = new TreeNode(5);
    root->right = new TreeNode(1);
    root->left->left = new TreeNode(6);
    root->left->right = new TreeNode(2);
    root->left->right->left = new TreeNode(7);
    root->left->right->right = new TreeNode(4);
    root->right->left = new TreeNode(0);
    root->right->right = new TreeNode(8);

    TreeNode* p = root->left; // Node 5
    TreeNode* q = root->right; // Node 1
    TreeNode* lca = sol.lowestCommonAncestor(root, p, q);
    std::cout << "LCA of " << p->val << " and " << q->val << " is " << lca->val << std::endl; // Expected:
```

```
3

    p = root->left; // Node 5
    q = root->left->right->right; // Node 4
    lca = sol.lowestCommonAncestor(root, p, q);
    std::cout << "LCA of " << p->val << " and " << q->val << " is " << lca->val << std::endl; // Expected:
5

    // Clean up memory
    delete root->right->right;
    delete root->right->left;
    delete root->left->right->right;
    delete root->left->right->left;
    delete root->left->right;
    delete root->left->left;
    delete root->right;
    delete root->left;
    delete root;

    return 0;
}
```

Walkthrough (for nodes 5 and 1):

1. lca(root=3, p=5, q=1) is called.
2. It recursively calls lca(root=5, p=5, q=1). This call hits the base case (root == p) and returns node 5. So, leftLCA becomes node 5.
3. It then recursively calls lca(root=1, p=5, q=1). This call hits the base case (root == q) and returns node 1. So, rightLCA becomes node 1.
4. Back in the frame for root=3, both leftLCA (node 5) and rightLCA (node 1) are non-null. The condition if (leftLCA!= nullptr && rightLCA!= nullptr) is true.
5. The function returns root (node 3), which is the correct LCA.

Complexity Analysis

- **Time Complexity: O(N)**, as the algorithm performs a single traversal of the tree, visiting each node at most once.[18]
- **Space Complexity: O(H)**, for the recursion call stack, where H is the height of the tree.

Strategic Implications

The LCA algorithm is a quintessential example of post-order traversal logic. The critical decision—determining if the current node is the LCA—is made *after* the recursive calls to the left and right children have returned. This bottom-up flow of information, where a parent node's status is determined by the results from its children, is the hallmark of post-order thinking. Recognizing this connects the LCA problem to a broader class of algorithms where subtree properties must be computed before a conclusion can be drawn about the parent node (e.g., checking tree balance, finding the maximum path sum).

A crucial variation to note during an interview is the case of a Binary Search Tree (BST). For a BST, the LCA can be found much more efficiently. By starting at the root, if both p and q are smaller than the current node, the LCA must be in the left subtree. If both are larger, it must be in the right subtree. The first node encountered where p and q are on opposite sides (or one of them is the node itself) is the LCA. This optimized approach has a time complexity of O(H), which is a significant improvement over the general O(N) solution.[17]

5. Pattern: Binary Search Tree (BST) Validation

Conceptual Framework

A Binary Search Tree (BST) is a binary tree with a specific ordering constraint that must hold true for every node in the tree: all values in a node's left subtree must be strictly less than the node's value, and all values in its right subtree must be strictly greater.[22] A common and subtle error in validating a BST is to only check a node's value against its immediate parent or children. This is insufficient because the BST property must be satisfied with respect to all ancestors. For example, a node in the right subtree of a left child must still be less than the grandparent node.[22]

Two robust approaches correctly validate a BST:

1. **Range-Based DFS:** This method involves a recursive traversal where, for each node, a valid range [min_val, max_val] is maintained. A node is valid only if its value falls within this range. When recursing to the left child, the upper bound of the range is updated to the parent's value. When recursing to the right child, the lower bound is updated to the parent's value. This ensures that ancestor constraints are propagated down the tree.[24]

2. **In-order Traversal:** A fundamental property of a valid BST is that an in-order traversal of its nodes will yield a strictly increasing sequence of values.[15] This approach performs an in-order traversal and checks if each visited node's value is greater than the value of the previously visited node.[25]

Problem Recognition

This pattern is indicated for problems explicitly named "Validate Binary Search Tree," "Check if a tree is a BST," or similar phrasings.[2]

C++ Implementation and Walkthrough

The range-based DFS approach is often preferred for its clarity in handling constraints. The following C++ code implements this method.

C++

```
#include <iostream>
#include <climits>

// Definition for a binary tree node.
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

class Solution {
public:
    bool isValidBST(TreeNode* root) {
        // Use long long for min/max to handle edge cases where node val is INT_MIN or INT_MAX
        return validate(root, LONG_MIN, LONG_MAX);
    }
```

```
private:
    bool validate(TreeNode* node, long long min_val, long long max_val) {
        if (node == nullptr) {
            return true;
        }

        // The current node's value must be within the valid range.
        if (node->val <= min_val |

| node->val >= max_val) {
            return false;
        }

        // Recursively validate the left and right subtrees with updated ranges.
        // For the left subtree, the new max is the current node's value.
        // For the right subtree, the new min is the current node's value.
        return validate(node->left, min_val, node->val) &&
               validate(node->right, node->val, max_val);
    }
};

int main() {
    Solution sol;

    // Example 1: Valid BST
    TreeNode* root1 = new TreeNode(2);
    root1->left = new TreeNode(1);
    root1->right = new TreeNode(3);
    std::cout << "Tree 1 is a valid BST: " << (sol.isValidBST(root1)? "True" : "False") << std::endl;

    // Example 2: Invalid BST
    TreeNode* root2 = new TreeNode(5);
    root2->left = new TreeNode(1);
    root2->right = new TreeNode(4);
    root2->right->left = new TreeNode(3);
    root2->right->right = new TreeNode(6);
    std::cout << "Tree 2 is a valid BST: " << (sol.isValidBST(root2)? "True" : "False") << std::endl;

    // Clean up memory
    delete root1->right;
    delete root1->left;
    delete root1;
```

```
    delete root2->right->right;
    delete root2->right->left;
    delete root2->right;
    delete root2->left;
    delete root2;

    return 0;
}
```

Walkthrough (for invalid Tree 2):

1. isValidBST(root=5) calls validate(5, LONG_MIN, LONG_MAX). 5 is within the range.
2. It recursively calls validate(node=1, min_val=LONG_MIN, max_val=5). This is valid.
3. It then calls validate(node=4, min_val=5, max_val=LONG_MAX).
4. In this call, the condition node->val <= min_val (4 <= 5) is true, so the function returns false.
5. This false value propagates up, causing the final result to be false. This correctly identifies that node 4 violates the constraint imposed by its grandparent, node 5.

Complexity Analysis

- **Time Complexity: O(N)**, as each node is visited exactly once.
- **Space Complexity: O(H)**, for the recursion stack depth.

Strategic Implications

The BST validation problem serves as an excellent illustration of how constraints must be propagated through recursive calls. The state passed down the tree (min_val, max_val) is not merely an accumulated value but a set of narrowing constraints that each node must satisfy. The failure of the naive check (node->left->val < node->val) demonstrates that a node's validity is a function of its entire ancestry, not just its immediate parent. Understanding this principle of propagating and tightening constraints is key to solving a wide range of recursive tree problems where context from higher levels of the tree impacts the validity of nodes at lower levels.

6. Pattern: Tree Construction from Traversal Sequences

Conceptual Framework

A unique binary tree can be unambiguously reconstructed if and only if both its in-order traversal and one of its pre-order or post-order traversals are provided.[26] A single traversal is insufficient because it does not provide enough structural information. The reconstruction process relies on the distinct properties of these traversals:

- **Pre-order (or Post-order) Traversal:** This sequence identifies the root of any given subtree. In a pre-order traversal, the root is the *first* element. In a post-order traversal, the root is the *last* element.[27]
- **In-order Traversal:** This sequence provides the spatial arrangement of nodes. Once the root of a subtree is identified, all elements to its left in the in-order sequence belong to the left subtree, and all elements to its right belong to the right subtree.[27]

The pattern involves a recursive algorithm that uses the pre/post-order traversal to pick a root and the in-order traversal to partition the remaining nodes into left and right subtrees, which are then constructed recursively.[27]

Problem Recognition

This pattern is applied to problems that explicitly ask to "Construct Binary Tree from Preorder and Inorder Traversal" or "Construct Binary Tree from Inorder and Postorder Traversal".

C++ Implementation and Walkthrough

The following C++ code demonstrates the construction of a binary tree from its pre-order and in-order traversal sequences. An unordered_map is used to optimize the search for the root's index in the in-order array.

C++

```cpp
#include <iostream>
#include <vector>
#include <unordered_map>
#include <queue>

// Definition for a binary tree node.
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

class Solution {
public:
    TreeNode* buildTree(const std::vector<int>& preorder, const std::vector<int>& inorder) {
        if (preorder.empty() |

| inorder.empty()) {
            return nullptr;
        }

        // Create a map for efficient lookup of inorder indices
        for (int i = 0; i < inorder.size(); ++i) {
            inorderMap[inorder[i]] = i;
        }

        int preorderIndex = 0;
        return build(preorder, preorderIndex, 0, inorder.size() - 1);
    }

private:
    std::unordered_map<int, int> inorderMap;

    TreeNode* build(const std::vector<int>& preorder, int& preorderIndex, int inorderStart, int
inorderEnd) {
        if (inorderStart > inorderEnd) {
            return nullptr;
        }

        // The first element in the current preorder segment is the root
```

```
        int rootVal = preorder[preorderIndex++];
        TreeNode* root = new TreeNode(rootVal);

        // Find the root's index in the inorder traversal to partition
        int inorderRootIndex = inorderMap[rootVal];

        // Recursively build left and right subtrees
        root->left = build(preorder, preorderIndex, inorderStart, inorderRootIndex - 1);
        root->right = build(preorder, preorderIndex, inorderRootIndex + 1, inorderEnd);

        return root;
    }
};

// Helper function to print the tree (level order) for verification
void printLevelOrder(TreeNode* root) {
    if (!root) return;
    std::queue<TreeNode*> q;
    q.push(root);
    while (!q.empty()) {
        TreeNode* node = q.front();
        q.pop();
        std::cout << node->val << " ";
        if (node->left) q.push(node->left);
        if (node->right) q.push(node->right);
    }
    std::cout << std::endl;
}

int main() {
    Solution sol;
    std::vector<int> preorder = {3, 9, 20, 15, 7};
    std::vector<int> inorder = {9, 3, 15, 20, 7};

    TreeNode* root = sol.buildTree(preorder, inorder);

    std::cout << "Level order of constructed tree: ";
    printLevelOrder(root);

    // Memory cleanup would be needed here
    return 0;
}
```

Walkthrough:

1. The inorderMap is built: {9:0, 3:1, 15:2, 20:3, 7:4}.
2. Initial call: build(preorder, preIndex=0, inStart=0, inEnd=4).
3. rootVal is preorder = 3. A root node with value 3 is created. preIndex becomes 1.
4. inorderRootIndex for 3 is 1.
5. Recursive call for left subtree: build(preorder, preIndex=1, inStart=0, inEnd=0).
    - rootVal is preorder = 9. Node 9 created. preIndex becomes 2.
    - inorderRootIndex for 9 is 0.
    - Left call: build(..., inStart=0, inEnd=-1) returns nullptr.
    - Right call: build(..., inStart=1, inEnd=0) returns nullptr.
    - Node 9 is returned. Root 3's left child is set to 9.
6. Recursive call for right subtree: build(preorder, preIndex=2, inStart=2, inEnd=4).
    - rootVal is preorder = 20. Node 20 created. preIndex becomes 3.
    - inorderRootIndex for 20 is 3.
    - This process continues, building the subtrees for 15 and 7, and attaching them to node 20.
7. The final tree is returned.

Complexity Analysis

- **Time Complexity: O(N)**. Building the hash map takes O(N) time. The recursive build function visits each node once, and the lookup for the inorder index is an O(1) operation thanks to the map. Without the map, each lookup would be an O(N) scan, leading to an overall O(N2) complexity.[27]
- **Space Complexity: O(N)**. O(N) is required for the hash map. The recursion stack depth contributes an additional O(H) space.

Strategic Implications

This pattern is a profound exercise in understanding the intrinsic properties of tree traversals. It requires deconstructing the linear sequences back into the hierarchical structure they represent. The pre-order (or post-order) traversal provides the **hierarchical relationship**—identifying the parent-child connections. The in-order traversal provides the **spatial relationship**—distinguishing what belongs in the left subtree versus the right. The ability to articulate this interplay between the two traversal types demonstrates a deep and flexible understanding of tree data structures, which is highly valued in interviews.

Part II: Heap-Based Algorithmic Patterns

Heaps, often implemented as priority queues, are specialized tree-based data structures that maintain a partial order. They do not keep elements fully sorted, but they excel at providing constant-time access to the minimum or maximum element.[29] This property makes them exceptionally efficient for problems centered around priority, ranking, and order statistics.[30]

7. Pattern: Top K Elements

Conceptual Framework

This pattern addresses the common need to find the 'K' largest, smallest, or most frequent elements from a collection without incurring the cost of a full sort.[30] While sorting the entire collection and taking the first K elements works, its

O(NlogN) time complexity is often suboptimal. The more efficient heap-based approach achieves this in O(NlogK) time.

The core of the pattern involves using a heap of a fixed size, K. A slightly counter-intuitive but critical detail is the choice of heap:

- To find the **K largest** elements, a **min-heap** of size K is used.
- To find the **K smallest** elements, a **max-heap** of size K is used.

The heap acts as a dynamic container for the top K elements seen so far. For each new element from the input, it is compared with the top of the heap. If the new element "qualifies" (e.g., is larger than the smallest element in a min-heap tracking the K largest), the top element is removed, and the new element is inserted.[33]

Problem Recognition

This pattern is applicable to a wide range of problems, including:

- "Kth Largest/Smallest Element in an Array".[36]
- "Top K Frequent Elements".[38]
- "K Closest Points to Origin".
- "Merge K Sorted Lists".[31]

C++ Implementation and Walkthrough

The following C++ code solves the "Top K Frequent Elements" problem. It first uses an std::unordered_map to count element frequencies and then an std::priority_queue configured as a min-heap to find the K elements with the highest frequencies.

C++

```
#include <iostream>
#include <vector>
#include <unordered_map>
#include <queue>

class Solution {
public:
    std::vector<int> topKFrequent(const std::vector<int>& nums, int k) {
        // Step 1: Count the frequency of each number.
        std::unordered_map<int, int> freqMap;
        for (int num : nums) {
            freqMap[num]++;
        }

        // Step 2: Use a min-heap to keep track of the top k frequent elements.
        // The pair stores {frequency, number}.
        // The custom comparator makes it a min-heap based on frequency.
        using P = std::pair<int, int>;
        std::priority_queue<P, std::vector<P>, std::greater<P>> minHeap;

        for (auto const& [num, freq] : freqMap) {
            minHeap.push({freq, num});
```

```
            if (minHeap.size() > k) {
                minHeap.pop(); // Remove the element with the smallest frequency
            }
        }

        // Step 3: Extract the result from the heap.
        std::vector<int> result;
        while (!minHeap.empty()) {
            result.push_back(minHeap.top().second);
            minHeap.pop();
        }

        // The result is in increasing order of frequency, reverse if needed.
        // For this problem, order doesn't matter.
        return result;
    }
};

int main() {
    Solution sol;
    std::vector<int> nums = {1, 1, 1, 2, 2, 3};
    int k = 2;
    std::vector<int> result = sol.topKFrequent(nums, k);

    std::cout << "Top " << k << " frequent elements are: ";
    for (int num : result) {
        std::cout << num << " ";
    }
    std::cout << std::endl; // Expected output could be 1 2 or 2 1

    return 0;
}
```

Walkthrough:

1. **Frequency Counting:** The freqMap is populated: {1:3, 2:2, 3:1}.
2. **Heap Processing:**
    - {3, 1} is pushed. Heap: [{3, 1}].
    - {2, 2} is pushed. Heap: [{2, 2}, {3, 1}].
    - minHeap.size() is now 2 (equal to k), so no pop occurs.
    - {1, 3} is pushed. Heap: [{1, 3}, {3, 1}, {2, 2}]. size() is 3, which is > k.
    - minHeap.pop() removes the top element, {1, 3}. Heap is now [{2, 2}, {3, 1}].
3. **Result Extraction:** The loop finishes. The min-heap contains the two pairs with the

highest frequencies. The numbers (1 and 2) are extracted and returned.

Complexity Analysis

- **Time Complexity: O(NlogK)**. Building the frequency map takes O(N). Iterating through the unique elements (let's say U unique elements, where U≤N) and performing heap operations takes O(UlogK). In the worst case, U=N, so the complexity is dominated by O(NlogK).[34] This is more efficient than an O(NlogN) sorting approach, especially when K is much smaller than N.
- **Space Complexity: O(N+K)**. In the worst case, the frequency map can store N unique elements. The heap stores at most K elements.[37]

Strategic Implications

The efficiency of this pattern stems from maintaining a heap of size K rather than N. The heap functions as a "gatekeeper" for membership in the "top K" set. A new element does not need to be compared against all K elements in the set; it only needs to challenge the "weakest" member, which is conveniently available at the top of the heap in O(1) time. The use of a min-heap to find the K *largest* elements is a critical, counter-intuitive detail. The min-heap's purpose is to efficiently track the smallest element among the current top K candidates. If a new element is larger than this minimum threshold, the threshold element is discarded, and the new, larger element takes its place, maintaining the heap's invariant of holding the K largest elements seen so far.

8. Pattern: Two Heaps for Dynamic Medians

Conceptual Framework

This advanced heap pattern provides an efficient solution for finding the median of a dataset that is growing over time, such as a stream of incoming numbers.[30] A naive approach of

storing all numbers and re-sorting upon every query is prohibitively slow (

O(NlogN) per query). The Two Heaps pattern maintains the median in O(logN) time per addition and O(1) time per query.[42]

The structure consists of two heaps that partition the entire set of numbers:

1. A **max-heap** (e.g., lower_half) stores the smaller half of the numbers. Its root is the largest element in the smaller half.
2. A **min-heap** (e.g., upper_half) stores the larger half of the numbers. Its root is the smallest element in the larger half.

These heaps are kept balanced so that their sizes differ by at most one. This balance ensures that the median can always be calculated directly from their root elements:

- If the total number of elements is **odd**, one heap will have one more element than the other. The root of the larger heap is the median.
- If the total number of elements is **even**, the heaps will be of equal size. The median is the average of the roots of both heaps.[31]

Problem Recognition

This pattern is the definitive solution for problems involving:

- "Find Median from Data Stream".[44]
- Calculating the median of "running integers" or any continuously updated collection of numbers.[41]

C++ Implementation and Walkthrough

The following C++ code implements a MedianFinder class using two std::priority_queues.

C++

```
#include <iostream>
#include <vector>
```

```cpp
#include <queue>
#include <iomanip>

class MedianFinder {
private:
    // Max-heap for the smaller half of the numbers
    std::priority_queue<int> lower_half;
    // Min-heap for the larger half of the numbers
    std::priority_queue<int, std::vector<int>, std::greater<int>> upper_half;

public:
    MedianFinder() {}

    void addNum(int num) {
        // Add to the max-heap (lower half) by default
        lower_half.push(num);

        // Balance the heaps: ensure every element in lower_half is <= every element in upper_half
        // Move the largest element from the lower half to the upper half.
        upper_half.push(lower_half.top());
        lower_half.pop();

        // Rebalance sizes: ensure size difference is at most 1
        // If upper_half has more elements, move its smallest element to lower_half.
        if (upper_half.size() > lower_half.size()) {
            lower_half.push(upper_half.top());
            upper_half.pop();
        }
    }

    double findMedian() {
        if (lower_half.size() > upper_half.size()) {
            return lower_half.top();
        } else {
            return (lower_half.top() + upper_half.top()) / 2.0;
        }
    }
};

int main() {
    MedianFinder mf;
    mf.addNum(1);
    mf.addNum(2);
```

```
    std::cout << "Median is: " << std::fixed << std::setprecision(1) << mf.findMedian() << std::endl; // 1.5
    mf.addNum(3);
    std::cout << "Median is: " << mf.findMedian() << std::endl; // 2.0
    mf.addNum(4);
    std::cout << "Median is: " << mf.findMedian() << std::endl; // 2.5
    mf.addNum(5);
    std::cout << "Median is: " << mf.findMedian() << std::endl; // 3.0

    return 0;
}
```

Walkthrough of addNum(3):

1. Current state: lower_half (max-heap) = {2}, upper_half (min-heap) = {1}. Incorrect state, let's trace from start.
2. mf.addNum(1): lower_half gets 1. upper_half gets 1, lower_half pops. lower_half gets 1 from upper_half. Final: lower_half={1}, upper_half={}.
3. mf.addNum(2): lower_half gets 2. Becomes {2, 1}. upper_half gets 2, lower_half pops 2. lower_half is now {1}. upper_half is {2}. Sizes are equal, no rebalance. Final: lower_half={1}, upper_half={2}. findMedian() returns (1+2)/2.0 = 1.5.
4. mf.addNum(3): lower_half gets 3. Becomes {3, 1}. upper_half gets 3, lower_half pops 3. lower_half is {1}. upper_half is {2, 3}. upper_half is larger, so lower_half gets 2 from upper_half. Final: lower_half={2, 1}, upper_half={3}. findMedian() returns lower_half.top() = 2.0.

Complexity Analysis

- **Time Complexity:** Adding a number involves a few heap push and pop operations. Each operation on a heap of size ≈N/2 takes O(logN) time. Therefore, addNum has a time complexity of O(logN).[42]
  findMedian only requires accessing the top elements of the heaps, which is an O(1) operation.[42]
- **Space Complexity: O(N)**, as all N numbers from the stream are stored across the two heaps.[43]

Strategic Implications

The Two Heaps pattern is a masterful example of using composite data structures to solve a problem efficiently. It physically models the abstract definition of a median—the value that partitions a sorted set into two equal halves. The max-heap and min-heap act as self-organizing containers for the lower and upper halves of the data stream, respectively. This structure ensures that the two most critical elements for calculating the median—the largest value in the lower half and the smallest value in the upper half—are always instantly accessible at the heaps' roots. The rebalancing logic is the key mechanism that maintains this partition dynamically as new elements arrive. This pattern demonstrates a sophisticated understanding of how data structures can be combined to maintain a complex invariant (the median property) far more efficiently than a single, monolithic structure could.

Part III: Graph-Based Algorithmic Patterns

Graphs are versatile data structures that model networks and interconnected entities, consisting of vertices (nodes) and edges (connections).[45] Unlike trees, graphs can contain cycles and may be disconnected, requiring more robust traversal algorithms. The following patterns are fundamental for solving problems related to connectivity, pathfinding, and dependency resolution in graphs.

9. Pattern: Breadth-First Search (BFS) for Graphs

Conceptual Framework

The principles of BFS for graphs are identical to those for trees: it is a level-by-level traversal algorithm that explores all neighbors of a node before moving to the next level of neighbors.[10] It uses a queue to manage the order of visitation.[48] However, a critical distinction arises when dealing with general graphs: the potential for cycles. To prevent infinite loops and ensure that each vertex is processed only once, graph BFS must employ a tracking mechanism, typically a visited set or boolean array.[45]

Problem Recognition

BFS is the optimal choice for:

- **Finding the Shortest Path in an Unweighted Graph:** BFS naturally finds the shortest path in terms of the number of edges from a source to all other nodes.[10]
- **Connectivity and Network Analysis:** Used in web crawlers to discover pages, in social networks to find friends-of-friends, and to identify all nodes within a connected component.[48]
- **Level-Based Problems:** Any graph problem that requires processing nodes in layers based on their distance from a source.

C++ Implementation and Walkthrough

This C++ code implements BFS on a graph represented by an adjacency list. It uses a queue for traversal and a vector<bool> to track visited vertices.

C++

```
#include <iostream>
#include <vector>
#include <queue>

class Graph {
private:
    int V; // Number of vertices
    std::vector<std::vector<int>> adj; // Adjacency list

public:
    Graph(int vertices) : V(vertices) {
        adj.resize(V);
    }

    void addEdge(int u, int v) {
        adj[u].push_back(v);
        // For an undirected graph, add the reverse edge as well
```

```
        // adj[v].push_back(u);
    }

    void BFS(int startNode) {
        std::vector<bool> visited(V, false);
        std::queue<int> q;

        visited[startNode] = true;
        q.push(startNode);

        while (!q.empty()) {
            int u = q.front();
            q.pop();
            std::cout << u << " ";

            for (int v : adj[u]) {
                if (!visited[v]) {
                    visited[v] = true;
                    q.push(v);
                }
            }
        }
        std::cout << std::endl;
    }
};

int main() {
    Graph g(4);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 3);

    std::cout << "Breadth First Traversal (starting from vertex 2): ";
    g.BFS(2); // Expected: 2 0 3 1

    return 0;
}
```

Walkthrough (starting from vertex 2):

1. visited array is initialized to all false. Queue q is created.
2. startNode 2 is marked as visited and enqueued. q = {2}.
3. Loop starts. 2 is dequeued and printed.
4. Neighbors of 2 are 0 and 3.
5. 0 is not visited. It's marked visited and enqueued. q = {0}.
6. 3 is not visited. It's marked visited and enqueued. q = {0, 3}.
7. Loop continues. 0 is dequeued and printed.
8. Neighbors of 0 are 1 and 2. 1 is not visited; it's marked and enqueued. q = {3, 1}. 2 is already visited.
9. Loop continues. 3 is dequeued and printed. Its only neighbor is 3, which is visited.
10. Loop continues. 1 is dequeued and printed. Its neighbor 2 is visited.
11. The queue is now empty. The loop terminates.

Complexity Analysis

- **Time Complexity: O(V+E)**, where V is the number of vertices and E is the number of edges. Each vertex is enqueued and dequeued exactly once, and every edge is traversed once.[48]
- **Space Complexity: O(V)**. In the worst case, the queue can hold up to V vertices. The visited array also requires O(V) space.

Strategic Implications

The single most important element that distinguishes graph traversal from tree traversal is the visited set. It is the mechanism that gracefully handles the complex, cyclical topologies of general graphs. Forgetting to include a visited check is a common and critical error in interviews, as it leads to an infinite loop for any graph containing a cycle. For example, in a simple graph A <-> B, a BFS starting from A would visit B. From B, it would then re-visit A, which would then re-visit B, ad infinitum. The visited set breaks this symmetry by ensuring that once a node has been enqueued for processing, it is never enqueued again, guaranteeing forward progress and the algorithm's termination with O(V+E) complexity.

10. Pattern: Depth-First Search (DFS) for Graphs

Conceptual Framework

Similar to its tree counterpart, DFS for graphs explores as deeply as possible along each path before backtracking.[10] It can be implemented recursively, leveraging the call stack, or iteratively with an explicit stack.[53] Just as with graph BFS, a

visited set is indispensable for handling cycles and ensuring each vertex is visited only once.[45]

Problem Recognition

DFS is a versatile tool for a wide range of graph problems, including:

- **Pathfinding and Connectivity:** Determining if a path exists between two vertices or finding all vertices in a connected component.[10]
- **Cycle Detection:** Identifying cycles in both directed and undirected graphs.[10]
- **Topological Sorting:** As a core subroutine for ordering vertices in a Directed Acyclic Graph (DAG).

C++ Implementation and Walkthrough

The following is a standard recursive C++ implementation of DFS for a graph represented by an adjacency list.

C++

```
#include <iostream>
#include <vector>
#include <list>

class Graph {
private:
```

```
    int V; // Number of vertices
    std::vector<std::list<int>> adj; // Adjacency list

    void DFSUtil(int v, std::vector<bool>& visited) {
        visited[v] = true;
        std::cout << v << " ";

        for (int neighbor : adj[v]) {
            if (!visited[neighbor]) {
                DFSUtil(neighbor, visited);
            }
        }
    }

public:
    Graph(int vertices) : V(vertices) {
        adj.resize(V);
    }

    void addEdge(int u, int v) {
        adj[u].push_back(v);
    }

    void DFS(int startNode) {
        std::vector<bool> visited(V, false);
        DFSUtil(startNode, visited);
        std::cout << std::endl;
    }
};

int main() {
    Graph g(4);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 3);

    std::cout << "Depth First Traversal (starting from vertex 2): ";
    g.DFS(2); // Expected: 2 0 1 3

    return 0;
```

```
}
```

Walkthrough (starting from vertex 2):

1. DFS(2) is called. visited array is initialized to false. DFSUtil(2, visited) is called.
2. 2 is marked as visited and printed.
3. The first neighbor of 2 is 0. It is not visited. DFSUtil(0, visited) is called.
4. 0 is marked visited and printed. Its first neighbor is 1. DFSUtil(1, visited) is called.
5. 1 is marked visited and printed. Its neighbor 2 is already visited. The function for 1 returns.
6. Back in the call for 0, its next neighbor is 2, which is visited. The function for 0 returns.
7. Back in the call for 2, its next neighbor is 3. It is not visited. DFSUtil(3, visited) is called.
8. 3 is marked visited and printed. Its neighbor 3 is visited. The function for 3 returns.
9. All neighbors of 2 have been explored. The initial call to DFSUtil returns.

Complexity Analysis

- **Time Complexity: O(V+E)**. Each vertex is visited once, and all its outgoing edges are checked once.[57]
- **Space Complexity: O(V)**. This is for the visited array and, in the worst-case scenario of a path graph, the recursion stack depth can be V.

Strategic Implications

For detecting cycles in a **directed graph**, a simple boolean visited array is insufficient. It can only tell if a node has ever been visited, but not if it is currently in the recursion stack of the current exploration path. A back edge, which indicates a cycle in a directed graph, is an edge from a node u to an ancestor v in the DFS tree. To detect this, a three-state system is required: UNVISITED, VISITING, and VISITED. A cycle is detected if a DFS traversal encounters a node that is currently in the VISITING state. This is often implemented with two boolean arrays: visited (tracks nodes ever visited) and recursionStack (tracks nodes in the current path).[56] Encountering a neighbor that is

true in recursionStack confirms a cycle. This distinction between cycle detection in undirected vs. directed graphs is a frequent source of interview questions and demonstrates a deeper understanding of the DFS algorithm's nuances.

11. Pattern: Matrix Traversal (Number of Islands)

Conceptual Framework

This pattern is a specific and highly common application of graph traversal (either DFS or BFS) to problems presented on a 2D grid or matrix.[6] The matrix itself is treated as an implicit graph:

- **Nodes:** Each cell (i, j) in the matrix is a node.
- **Edges:** Edges implicitly exist between adjacent cells (horizontally and vertically, and sometimes diagonally) that share a common property, such as being 'land' cells.[45]

The "Number of Islands" problem is the canonical example, where the goal is to count the number of disconnected groups of 'land' cells. This is equivalent to counting the number of connected components in the implicit graph.[59] The standard solution involves iterating through every cell of the matrix. If an unvisited 'land' cell is found, it signifies a new island. A traversal (DFS or BFS) is then launched from that cell to find and mark all other cells belonging to the same island, preventing them from being counted again.[59]

Problem Recognition

This pattern should be identified for any grid-based problem involving:

- Counting or identifying connected regions (e.g., "Number of Islands," "Biggest Island").[6]
- Modifying connected regions (e.g., "Flood Fill," "Coloring a Border").
- Finding paths or exploring possibilities within a maze or on a game board.

C++ Implementation and Walkthrough

The following C++ code solves the "Number of Islands" problem using DFS. It modifies the grid in-place by "sinking" the island (changing '1's to '0's) to mark cells as visited, thus optimizing space.

C++

```
#include <iostream>
#include <vector>

class Solution {
public:
    int numIslands(std::vector<std::vector<char>>& grid) {
        if (grid.empty() |

| grid.empty()) {
            return 0;
        }

        int rows = grid.size();
        int cols = grid.size();
        int islandCount = 0;

        for (int i = 0; i < rows; ++i) {
            for (int j = 0; j < cols; ++j) {
                if (grid[i][j] == '1') {
                    islandCount++;
                    dfs(grid, i, j);
                }
            }
        }
        return islandCount;
    }

private:
    void dfs(std::vector<std::vector<char>>& grid, int r, int c) {
        int rows = grid.size();
        int cols = grid.size();

        // Boundary checks and check if it's water or already visited (sunk)
        if (r < 0 |

| r >= rows |
| c < 0 |
| c >= cols |
| grid[r][c] == '0') {
            return;
```

```
        }

        // Mark the current cell as visited by sinking it
        grid[r][c] = '0';

        // Explore all 4 adjacent cells
        dfs(grid, r + 1, c); // Down
        dfs(grid, r - 1, c); // Up
        dfs(grid, r, c + 1); // Right
        dfs(grid, r, c - 1); // Left
    }
};

int main() {
    Solution sol;
    std::vector<std::vector<char>> grid = {
        {'1', '1', '0', '0', '0'},
        {'1', '1', '0', '0', '0'},
        {'0', '0', '1', '0', '0'},
        {'0', '0', '0', '1', '1'}
    };

    std::cout << "Number of islands: " << sol.numIslands(grid) << std::endl; // Expected: 3

    return 0;
}
```

Walkthrough:

1. The main function iterates through the grid. At (0,0), grid is '1'.
2. islandCount becomes 1. dfs(grid, 0, 0) is called.
3. The DFS explores all connected '1's starting from (0,0), which are (0,1), (1,0), and (1,1), changing them to '0's.
4. The main loop continues. It skips over the now '0' cells. At (2,2), it finds another '1'.
5. islandCount becomes 2. dfs(grid, 2, 2) is called, which sinks that single '1'.
6. The main loop continues. At (3,3), it finds a '1'.
7. islandCount becomes 3. dfs(grid, 3, 3) is called, which sinks (3,3) and (3,4).
8. The loops complete. The final islandCount of 3 is returned.

Complexity Analysis

- **Time Complexity: O(M×N)**, where M and N are the dimensions of the grid. Each cell is visited at most a constant number of times.[60]
- **Space Complexity:** In this in-place modification version, the space complexity is determined by the recursion depth of the DFS. In the worst case (a serpentine island that covers the entire grid), this can be O(M×N). If an auxiliary visited matrix were used, the space would be explicitly O(M×N).[59]

Strategic Implications

The Matrix Traversal pattern is a powerful lesson in abstraction. The core insight is recognizing that a grid is simply a specific representation of a graph. This mental leap allows the direct application of standard, well-understood graph traversal algorithms like DFS and BFS to solve a whole class of problems. The general meta-pattern for counting connected components is: iterate through all potential starting nodes (cells), and for each unvisited one, launch a full traversal to explore and mark its entire component, incrementing a counter each time a new traversal is launched. This template is highly reusable for any problem that requires partitioning a space into connected regions.

12. Pattern: Topological Sort

Conceptual Framework

Topological Sort produces a linear ordering of vertices in a **Directed Acyclic Graph (DAG)**. The ordering is such that for every directed edge from vertex u to vertex v, u appears before v in the sequence.[10] This type of ordering is impossible if the graph contains a cycle, as a cycle represents a circular dependency (e.g., A depends on B, and B depends on A) that cannot be linearly ordered.[61] A graph may have multiple valid topological sortings.

There are two primary algorithms for performing a topological sort:

1. **DFS-based Approach:** This is a common and elegant method. A DFS is performed on the graph. The topological sort is the reverse of the order in which vertices finish their recursive calls (i.e., a reversed post-order traversal). As each vertex's DFS call completes (meaning all its descendants have been visited), it is pushed onto a stack. After the entire

graph has been traversed, popping from the stack yields a valid topological order.[61]

2. **Kahn's Algorithm (BFS-based):** This iterative approach uses a queue and tracks the "in-degree" (number of incoming edges) of each node. Initially, all nodes with an in-degree of 0 are added to the queue. The algorithm then repeatedly dequeues a node, adds it to the topological order, and "removes" its outgoing edges by decrementing the in-degree of its neighbors. If a neighbor's in-degree becomes 0, it is added to the queue.[64]

Problem Recognition

This pattern is the definitive solution for problems involving dependencies, prerequisites, or a required sequence of events. Common examples include:

- **Course Scheduling:** A set of courses where some are prerequisites for others.[11]
- **Task Dependencies:** In build systems or project management, where certain tasks must be completed before others can begin.[9]
- **Alien Dictionary:** Deducing the alphabetical order of a new language from a list of lexicographically sorted words.[11]

C++ Implementation and Walkthrough

The DFS-based approach is frequently implemented in interviews due to its concise recursive structure. The following C++ code demonstrates this method.

C++

```
#include <iostream>
#include <vector>
#include <stack>
#include <list>
#include <algorithm>

class Graph {
private:
```

```
    int V;
    std::vector<std::list<int>> adj;

    void topologicalSortUtil(int v, std::vector<bool>& visited, std::stack<int>& s) {
        visited[v] = true;

        for (int neighbor : adj[v]) {
            if (!visited[neighbor]) {
                topologicalSortUtil(neighbor, visited, s);
            }
        }
        // After visiting all descendants, push current vertex to stack
        s.push(v);
    }

public:
    Graph(int vertices) : V(vertices) {
        adj.resize(V);
    }

    void addEdge(int u, int v) {
        adj[u].push_back(v);
    }

    void topologicalSort() {
        std::stack<int> s;
        std::vector<bool> visited(V, false);

        for (int i = 0; i < V; ++i) {
            if (!visited[i]) {
                topologicalSortUtil(i, visited, s);
            }
        }

        std::cout << "Topological Sort: ";
        while (!s.empty()) {
            std::cout << s.top() << " ";
            s.pop();
        }
        std::cout << std::endl;
    }
};
```

```
int main() {
    Graph g(6);
    g.addEdge(5, 2);
    g.addEdge(5, 0);
    g.addEdge(4, 0);
    g.addEdge(4, 1);
    g.addEdge(2, 3);
    g.addEdge(3, 1);

    g.topologicalSort(); // Expected: 5 4 2 3 1 0 (or another valid order)

    return 0;
}
```

Walkthrough:

1. The main loop starts DFS from unvisited nodes. Let's say it starts with 0. DFS(0) is called. 0 has no outgoing edges, so 0 is pushed to the stack. Stack: {0}.
2. Next unvisited is 1. DFS(1) is called. No outgoing edges, 1 is pushed. Stack: {1, 0}.
3. Next is 2. DFS(2) is called. It has a neighbor 3. DFS(3) is called.
4. DFS(3) has neighbor 1, which is visited. So DFS(3) finishes, and 3 is pushed. Stack: {3, 1, 0}.
5. Back in DFS(2), all neighbors are done. 2 is pushed. Stack: {2, 3, 1, 0}.
6. This continues until all nodes are visited. The final stack will produce a valid topological order when popped.

Complexity Analysis

- **Time Complexity: O(V+E)**. The algorithm is fundamentally a DFS traversal of the entire graph, so its complexity is linear in the number of vertices and edges.[63]
- **Space Complexity: O(V+E)**. This includes O(V+E) for the adjacency list representation, O(V) for the visited array, and O(V) for the recursion stack and the result stack.

Strategic Implications

Topological sort is the algorithmic formalization of dependency resolution. The DFS-based solution works because of its post-order nature. A vertex is pushed onto the result stack only *after* all of its descendants (i.e., all the tasks that depend on it) have been fully explored and

pushed onto the stack. Consequently, a node with no outgoing dependencies will be pushed first, and a source node with no incoming dependencies will be processed last by the recursive utility, placing it at the top of the stack. When the stack is popped, this source node comes out first, correctly starting the linear ordering. Understanding this connection between the post-order traversal mechanism and the logic of dependency chains is key to mastering this pattern.

13. Pattern: Disjoint Set Union (DSU) / Union-Find

Conceptual Framework

A Disjoint Set Union (DSU) data structure, also known as Union-Find, is a specialized data structure designed to manage a collection of disjoint (non-overlapping) sets.[65] It excels at performing two primary operations with near-constant time complexity on an amortized basis:

1. **find(i):** Determines the representative (or root) of the set to which element i belongs. This is used to check if two elements are in the same set.
2. **union(i, j):** Merges the two sets containing elements i and j into a single set.

The remarkable efficiency of DSU comes from two key optimizations:

- **Union by Rank (or Size):** During a union operation, instead of arbitrarily merging sets, the tree with the smaller rank (a proxy for height) is attached to the root of the tree with the larger rank. This helps keep the trees shallow and prevents the formation of long, chain-like structures that would degrade performance.[66]
- **Path Compression:** During a find operation, after the root of the set is found, the path from the queried element to the root is "compressed" by making every node on that path a direct child of the root. This dramatically flattens the tree structure, speeding up future find operations for those elements and their descendants.[66]

Problem Recognition

The DSU pattern is ideal for problems involving:

- **Dynamic Connectivity:** Tracking connectivity between nodes as edges are added to a

graph.

- **Cycle Detection in Undirected Graphs:** It is the most efficient way to detect cycles. When considering adding an edge (u, v), if find(u) is the same as find(v), it means u and v are already in the same connected component, and adding the edge would form a cycle. This is the core of Kruskal's MST algorithm.
- **Grouping and Equivalence Classes:** Any problem that requires partitioning elements into groups where all elements in a group are equivalent according to some transitive property.

C++ Implementation and Walkthrough

The following C++ code provides a robust implementation of the DSU data structure, incorporating both union by rank and path compression.

C++

```
#include <iostream>
#include <vector>
#include <numeric>

class DSU {
private:
    std::vector<int> parent;
    std::vector<int> rank;

public:
    DSU(int n) {
        parent.resize(n);
        std::iota(parent.begin(), parent.end(), 0); // Each element is its own parent
        rank.assign(n, 0); // All ranks are initially 0
    }

    // Find with Path Compression
    int find(int i) {
        if (parent[i] == i) {
            return i;
        }
```

```
        return parent[i] = find(parent[i]); // Path compression
    }

    // Union by Rank
    void unite(int i, int j) {
        int root_i = find(i);
        int root_j = find(j);

        if (root_i!= root_j) {
            if (rank[root_i] < rank[root_j]) {
                parent[root_i] = root_j;
            } else if (rank[root_i] > rank[root_j]) {
                parent[root_j] = root_i;
            } else {
                parent[root_j] = root_i;
                rank[root_i]++;
            }
        }
    }
};

int main() {
    DSU dsu(5); // 5 elements: 0, 1, 2, 3, 4

    dsu.unite(0, 1);
    dsu.unite(2, 3);

    std::cout << "Are 0 and 1 in the same set? " << (dsu.find(0) == dsu.find(1)? "Yes" : "No") <<
std::endl;
    std::cout << "Are 0 and 2 in the same set? " << (dsu.find(0) == dsu.find(2)? "Yes" : "No") <<
std::endl;

    dsu.unite(0, 4);
    dsu.unite(1, 2);

    std::cout << "Are 0 and 2 in the same set now? " << (dsu.find(0) == dsu.find(2)? "Yes" : "No") <<
std::endl;
    std::cout << "Representative of set containing 3: " << dsu.find(3) << std::endl;

    return 0;
}
```

Walkthrough:

1. **Initialization:** parent = {0, 1, 2, 3, 4}, rank = {0, 0, 0, 0, 0}.
2. unite(0, 1): find(0) is 0, find(1) is 1. Ranks are equal. parent becomes 0, rank becomes 1.
3. unite(2, 3): find(2) is 2, find(3) is 3. Ranks are equal. parent becomes 2, rank becomes 1.
4. unite(1, 2): find(1) returns 0. find(2) returns 2. rank (1) is equal to rank (1). parent is set to 0, and rank is incremented to 2. Now {0, 1, 2, 3, 4} are all in the same set with root 0.

Complexity Analysis

- **Time Complexity:** With both path compression and union by rank/size, the amortized time complexity for both find and union operations is nearly constant, formally expressed as O(α(N)), where α(N) is the extremely slow-growing inverse Ackermann function. For any practical input size N, α(N) is less than 5, making the operations effectively constant time.[65]
- **Space Complexity: O(N)** to store the parent and rank arrays for N elements.

Strategic Implications

The DSU data structure is a testament to the power of algorithmic optimization. The naive implementation of union-find can result in linear-time operations. However, the synergistic combination of two simple heuristics—path compression and union by rank—reduces this to nearly constant time. This dramatic improvement is a result of how the optimizations work together: union by rank prevents the creation of tall, spindly trees, ensuring that paths are short to begin with, while path compression actively flattens the trees during find operations, making subsequent accesses even faster. Understanding that these two seemingly minor tweaks have such a profound impact on performance is a mark of a sophisticated algorithmic thinker.

14. Pattern: Minimum Spanning Tree (MST) - Kruskal's Algorithm

Conceptual Framework

A Minimum Spanning Tree (MST) of a connected, weighted, and undirected graph is a subgraph that connects all the vertices together with the minimum possible sum of edge weights, without forming any cycles.[67] Kruskal's algorithm is a classic greedy algorithm for finding an MST.[67] Its strategy is simple and intuitive:

1. Create a list of all edges in the graph and sort them in non-decreasing order of their weight.
2. Initialize an empty MST.
3. Iterate through the sorted edges. For each edge, if adding it to the MST does not create a cycle, add it.
4. Stop when the MST has V−1 edges, where V is the number of vertices.

The algorithm's efficiency hinges on a fast way to detect cycles. This is where the Disjoint Set Union (DSU) data structure is employed. Each connected component in the growing MST is maintained as a set in the DSU. An edge (u, v) creates a cycle if and only if u and v are already in the same set.[67]

Problem Recognition

This pattern is the solution for problems that ask to:

- Find the minimum cost to connect all points in a network (e.g., connecting cities with roads, computers with cables, or islands with bridges).
- Any problem that can be modeled as finding the cheapest way to make a weighted, undirected graph fully connected.

C++ Implementation and Walkthrough

The following C++ implementation of Kruskal's algorithm uses the DSU class from the previous pattern and std::sort to order the edges.

C++

```cpp
#include <iostream>
#include <vector>
#include <numeric>
#include <algorithm>

// DSU class from the previous pattern
class DSU {
private:
    std::vector<int> parent;
    std::vector<int> rank;
public:
    DSU(int n) {
        parent.resize(n);
        std::iota(parent.begin(), parent.end(), 0);
        rank.assign(n, 0);
    }
    int find(int i) {
        if (parent[i] == i) return i;
        return parent[i] = find(parent[i]);
    }
    void unite(int i, int j) {
        int root_i = find(i);
        int root_j = find(j);
        if (root_i!= root_j) {
            if (rank[root_i] < rank[root_j]) parent[root_i] = root_j;
            else if (rank[root_i] > rank[root_j]) parent[root_j] = root_i;
            else { parent[root_j] = root_i; rank[root_i]++; }
        }
    }
};

struct Edge {
    int u, v, weight;
};

bool compareEdges(const Edge& a, const Edge& b) {
    return a.weight < b.weight;
}

class Graph {
private:
    int V;
    std::vector<Edge> edges;
public:
```

```
    Graph(int vertices) : V(vertices) {}

    void addEdge(int u, int v, int weight) {
        edges.push_back({u, v, weight});
    }

    int kruskalMST() {
        // Step 1: Sort all edges by weight
        std::sort(edges.begin(), edges.end(), compareEdges);

        DSU dsu(V);
        int mst_weight = 0;
        std::cout << "Edges in MST:" << std::endl;

        // Step 2 & 3: Iterate and add edges if they don't form a cycle
        for (const auto& edge : edges) {
            if (dsu.find(edge.u)!= dsu.find(edge.v)) {
                dsu.unite(edge.u, edge.v);
                mst_weight += edge.weight;
                std::cout << edge.u << " - " << edge.v << " (weight " << edge.weight << ")" << std::endl;
            }
        }
        return mst_weight;
    }
};

int main() {
    Graph g(4);
    g.addEdge(0, 1, 10);
    g.addEdge(0, 2, 6);
    g.addEdge(0, 3, 5);
    g.addEdge(1, 3, 15);
    g.addEdge(2, 3, 4);

    int weight = g.kruskalMST();
    std::cout << "Total weight of MST: " << weight << std::endl;

    return 0;
}
```

Walkthrough:

1. Edges are sorted by weight: (2,3,4), (0,3,5), (0,2,6), (0,1,10), (1,3,15).

2. Initialize DSU for 4 vertices.
3. Process edge (2,3) weight 4: find(2)!= find(3). Add edge, unite(2,3). mst_weight = 4.
4. Process edge (0,3) weight 5: find(0)!= find(3). Add edge, unite(0,3). mst_weight = 4+5 = 9.
5. Process edge (0,2) weight 6: find(0) == find(2). They are in the same set. Adding this edge would form a cycle (0-3-2). Discard.
6. Process edge (0,1) weight 10: find(0)!= find(1). Add edge, unite(0,1). mst_weight = 9+10 = 19.
7. The MST now has V−1=3 edges. The algorithm can terminate. The total weight is 19.

Complexity Analysis

- **Time Complexity: O(ElogE) or O(ElogV)**. The dominant operation is sorting the E edges. The loop that follows iterates through E edges, and each DSU operation is nearly constant time. Since the number of edges E can be at most O(V2), logE is in the same order as logV, making the complexities equivalent.[70]
- **Space Complexity: O(V+E)** to store the graph edges and the DSU data structure.

Strategic Implications

The correctness of Kruskal's greedy approach is non-trivial and is guaranteed by a graph theory principle known as the "cut property." This property states that for any partition (or "cut") of a graph's vertices into two disjoint sets, the minimum-weight edge that crosses between these two sets must be part of every MST. At each step, Kruskal's algorithm considers an edge (u, v) where u and v are in different components of the growing forest. These components define a cut. Because the edges are processed in sorted order, the selected edge (u, v) is guaranteed to be the minimum-weight edge crossing that specific cut. Therefore, adding it is a "safe move" that will not prevent the formation of an optimal MST. This theoretical underpinning is what justifies the simple, elegant, and greedy nature of the algorithm.

15. Pattern: Shortest Path in Weighted Graphs - Dijkstra's Algorithm

Conceptual Framework

Dijkstra's algorithm finds the shortest path from a single source vertex to all other vertices in a **weighted graph where all edge weights are non-negative**.[71] It is a greedy algorithm that works by iteratively building a set of vertices for which the shortest path from the source is known.

The algorithm maintains a set of unvisited vertices and a distance array, dist, initialized to infinity for all vertices except the source, which is 0. At each step, it selects the unvisited vertex u with the smallest dist value. This selection is made efficient by using a min-priority queue. Once u is selected, its distance is considered final. The algorithm then performs a "relaxation" step for all of u's neighbors v: if the path to v through u (i.e., dist[u] + weight(u,v)) is shorter than the currently known dist[v], then dist[v] is updated.[71]

Problem Recognition

Dijkstra's algorithm is the standard solution for:

- Finding the shortest path in networks like road systems (GPS navigation), computer networks (IP routing), or flight paths, provided the costs (weights) are non-negative.[10]
- Any problem that can be modeled as finding the minimum-cost path from a single source in a weighted, non-negative-edge graph.[11]

C++ Implementation and Walkthrough

This C++ implementation uses an adjacency list to represent the graph and an std::priority_queue to efficiently select the vertex with the minimum distance.

C++

```
#include <iostream>
```

```cpp
#include <vector>
#include <queue>
#include <climits>

using iPair = std::pair<int, int>; // {distance, vertex}

class Graph {
private:
    int V;
    std::vector<std::list<iPair>> adj; // {neighbor, weight}

public:
    Graph(int vertices) : V(vertices) {
        adj.resize(V);
    }

    void addEdge(int u, int v, int weight) {
        adj[u].push_back({v, weight});
        adj[v].push_back({u, weight}); // For undirected graph
    }

    void dijkstra(int src) {
        // Min-priority queue to store {distance, vertex}
        std::priority_queue<iPair, std::vector<iPair>, std::greater<iPair>> pq;

        std::vector<int> dist(V, INT_MAX);
        dist[src] = 0;
        pq.push({0, src});

        while (!pq.empty()) {
            int u = pq.top().second;
            pq.pop();

            for (const auto& neighborPair : adj[u]) {
                int v = neighborPair.first;
                int weight = neighborPair.second;

                // Relaxation step
                if (dist[v] > dist[u] + weight) {
                    dist[v] = dist[u] + weight;
                    pq.push({dist[v], v});
                }
            }
```

```
        }

        std::cout << "Vertex distances from source " << src << ":" << std::endl;
        for (int i = 0; i < V; ++i) {
            std::cout << i << "\t\t" << (dist[i] == INT_MAX? "INF" : std::to_string(dist[i])) << std::endl;
        }
    }
};

int main() {
    int V = 9;
    Graph g(V);
    g.addEdge(0, 1, 4);
    g.addEdge(0, 7, 8);
    g.addEdge(1, 2, 8);
    g.addEdge(1, 7, 11);
    g.addEdge(2, 3, 7);
    g.addEdge(2, 8, 2);
    g.addEdge(2, 5, 4);
    g.addEdge(3, 4, 9);
    g.addEdge(3, 5, 14);
    g.addEdge(4, 5, 10);
    g.addEdge(5, 6, 2);
    g.addEdge(6, 7, 1);
    g.addEdge(6, 8, 6);
    g.addEdge(7, 8, 7);

    g.dijkstra(0);

    return 0;
}
```

Walkthrough (starting from source 0):

1. dist array is {0, INF, INF,...}. pq contains {{0, 0}}.
2. Pop {0, 0}. Vertex u is 0.
3. Neighbors of 0 are 1 and 7.
    - Relax edge (0,1): dist becomes 0+4=4. Push {4, 1} to pq.
    - Relax edge (0,7): dist becomes 0+8=8. Push {8, 7} to pq.
4. pq top is {4, 1}. Pop it. Vertex u is 1.
5. Neighbors of 1 are 0, 2, 7.
    - Relax edge (1,2): dist becomes 4+8=12. Push {12, 2}.
    - Relax edge (1,7): dist is 8, 4+11=15 is not smaller. No update.

6. This process continues, always extracting the unvisited node with the smallest known distance from the priority queue and relaxing its edges, until all reachable nodes have been finalized.

Complexity Analysis

- **Time Complexity: O(ElogV)**. With a binary heap implementation of the priority queue, there are at most E "decrease-key" operations (pushing an updated distance to the queue) and V "extract-min" operations (popping from the queue). Both operations take O(logV) time.[71]
- **Space Complexity: O(V+E)** for the adjacency list, distance array, and the priority queue, which can store up to V vertices in the worst case.

Strategic Implications

Dijkstra's algorithm can be conceptualized as a "weighted BFS." While a standard BFS expands its frontier uniformly one layer at a time, Dijkstra's expands its frontier non-uniformly, always pushing outward from the vertex that is currently closest to the source. The min-priority queue is the essential mechanism that enables this intelligent, greedy selection of the next vertex to explore. The algorithm's correctness is critically dependent on the assumption of **non-negative edge weights**. This guarantees that once a vertex u is extracted from the priority queue and its distance is finalized, no shorter path to u will ever be found. Any alternative path would have to pass through another unvisited vertex v, but since u was chosen, dist[u] <= dist[v]. With non-negative weights, any path through v to u can only be longer. This assumption is the cornerstone of the algorithm's greedy strategy and a vital point to articulate in an interview. For graphs with negative edges, algorithms like Bellman-Ford must be used.[10]

Conclusion

The 15 algorithmic patterns detailed in this report represent the foundational toolkit for modern software engineering interviews. Mastery of these patterns—from the hierarchical traversals of trees, to the priority-based logic of heaps, to the complex connectivity problems of graphs—enables a candidate to move beyond problem-specific solutions and toward a

more profound and adaptable problem-solving methodology. The emphasis is not on memorizing C++ code but on internalizing the underlying logic of each pattern: understanding *why* BFS finds the shortest path, *why* a post-order traversal is used for deletion, and *why* Dijkstra's algorithm requires non-negative weights. This deeper comprehension allows for the confident deconstruction of novel problems into familiar, solvable components. The following table provides a high-level summary for quick reference and review.

Pattern Name	Category	Core Idea	Time Complexity	Space Complexity	Key Use Case / Problem Cues
BFS / Level Order	Tree	Level-by-level traversal using a queue.	O(N)	O(W)	Shortest path in unweighted tree/graph, level-specific operations.
DFS (Pre/In/Post)	Tree	Explore one branch completely before backtracking (recursive).	O(N)	O(H)	Pathfinding, subtree processing, BST sorting (in-order).
Path Sum	Tree	Use DFS recursion parameters to track state along a path.	O(N)	O(H)	Find root-to-leaf paths with a given sum or property.
Lowest Common Ancestor	Tree	Recursive DFS to find the deepest shared ancestor.	O(N)	O(H)	Explicit LCA problems; post-order logic.

BST Validation	Tree	Recursively check if nodes adhere to min/max range constraints.	O(N)	O(H)	Determine if a tree follows BST ordering rules.
Tree Construction	Tree	Use pre/post-order for roots and in-order for subtrees.	O(N)	O(N)	Rebuild a tree from its pre/in or post/in traversal sequences.
Top K Elements	Heap	Use a min-heap of size K to find the K largest elements.	O(NlogK)	O(N+K)	Find Kth largest/smallest/most frequent elements.
Two Heaps (Median)	Heap	Balance a max-heap (lower half) and min-heap (upper half).	O(logN) add, O(1) find	O(N)	Find the median of a dynamic data stream.
BFS for Graphs	Graph	Level-by-level traversal using a queue and a visited set.	O(V+E)	O(V)	Shortest path in an unweighted graph, connected components.
DFS for Graphs	Graph	Explore paths fully before	O(V+E)	O(V)	Cycle detection, pathfinding,

		backtracking, using a visited set.			connectivity.
Matrix Traversal	Graph	Apply DFS/BFS to an implicit graph represented by a 2D grid.	O(M×N)	O(M×N)	"Number of Islands," "Flood Fill," maze solving.
Topological Sort	Graph	Linear ordering of DAG nodes based on dependencies.	O(V+E)	O(V+E)	Task scheduling, course prerequisites, dependency resolution.
Disjoint Set Union	Graph	Efficiently manage disjoint sets with find/union operations.	O(α(N))	O(N)	Cycle detection in undirected graphs, dynamic connectivity.
Kruskal's MST	Graph	Greedily add the smallest edges that do not form a cycle.	O(ElogE)	O(V+E)	Find the minimum cost to connect all vertices.
Dijkstra's Algorithm	Graph	Find shortest paths from a source with a	O(ElogV)	O(V+E)	Shortest path in weighted graphs with non-negati

		priority queue.			ve edges.

Works cited

1. Tree cheatsheet for coding interviews | Tech Interview Handbook, accessed on September 12, 2025, https://www.techinterviewhandbook.org/algorithms/tree/
2. Introduction to Tree Data Structure - GeeksforGeeks, accessed on September 12, 2025, https://www.geeksforgeeks.org/dsa/introduction-to-tree-data-structure/
3. Tree Traversal Techniques - GeeksforGeeks, accessed on September 12, 2025, https://www.geeksforgeeks.org/dsa/tree-traversals-inorder-preorder-and-postorder/
4. Tree traversal - Wikipedia, accessed on September 12, 2025, https://en.wikipedia.org/wiki/Tree_traversal
5. Mastering the 20 Coding Patterns for Interviews - Design Gurus, accessed on September 12, 2025, https://www.designgurus.io/blog/grokking-the-coding-interview-patterns
6. 4 Types of Tree Traversal Algorithms - Built In, accessed on September 12, 2025, https://builtin.com/software-engineering-perspectives/tree-traversal
7. Binary Tree Level Order Traversal in C++ - GeeksforGeeks, accessed on September 12, 2025, https://www.geeksforgeeks.org/cpp/binary-tree-level-order-traversal-in-cpp/
8. 20 Essential Coding Patterns to Ace Your Next Coding Interview - DEV Community, accessed on September 12, 2025, https://dev.to/arslan_ah/20-essential-coding-patterns-to-ace-your-next-coding-interview-32a3
9. All Graph Algorithms in Data Structure (With Complexity & Techniques) - WsCube Tech, accessed on September 12, 2025, https://www.wscubetech.com/resources/dsa/graph-algorithms
10. 5 Graph Patterns To Ace Coding Interviews | HackerNoon, accessed on September 12, 2025, https://hackernoon.com/5-graph-patterns-to-ace-coding-interviews
11. Level Order Tree Traversal | GeeksforGeeks - YouTube, accessed on September 12, 2025, https://www.youtube.com/watch?v=kQ-aoKbGKSo
12. Level Order Traversal (Breadth First Search or BFS) of Binary Tree ..., accessed on September 12, 2025, https://www.geeksforgeeks.org/dsa/level-order-tree-traversal/
13. Mastering Binary Tree Traversals: A Comprehensive Guide | by Adam DeJans Jr. - Medium, accessed on September 12, 2025, https://medium.com/plain-simple-software/mastering-binary-tree-traversals-a-comprehensive-guide-d7203b1f7fcd
14. Binary Search Tree in C++ - GeeksforGeeks, accessed on September 12, 2025, https://www.geeksforgeeks.org/cpp/cpp-binary-search-tree/
15. 8.6. Tree Traversals — Problem Solving with Algorithms and Data Structures,

accessed on September 12, 2025, https://cs.berea.edu/cppds/Trees/TreeTraversals.html
16. LCA in BST - Lowest Common Ancestor in Binary Search Tree - GeeksforGeeks, accessed on September 12, 2025, https://www.geeksforgeeks.org/dsa/lowest-common-ancestor-in-a-binary-search-tree/
17. Lowest Common Ancestor in a Binary Tree - GeeksforGeeks, accessed on September 12, 2025, https://www.geeksforgeeks.org/dsa/lowest-common-ancestor-binary-tree-set-1/
18. Lowest Common Ancestor of a Binary Tree - LeetCode, accessed on September 12, 2025, https://leetcode.com/problems/lowest-common-ancestor-of-a-binary-tree/
19. Lowest Common Ancestor for two given Nodes - Tutorial - takeUforward, accessed on September 12, 2025, https://takeuforward.org/data-structure/lowest-common-ancestor-for-two-given-nodes/
20. Lowest Common Ancestor of a Binary Search Tree - LeetCode, accessed on September 12, 2025, https://leetcode.com/problems/lowest-common-ancestor-of-a-binary-search-tree/
21. 98. Validate Binary Search Tree - LeetCode, accessed on September 12, 2025, https://leetcode.com/problems/validate-binary-search-tree/
22. Binary Search Tree(BST) - Programiz, accessed on September 12, 2025, https://www.programiz.com/dsa/binary-search-tree
23. Check if a Binary Tree is BST or not - GeeksforGeeks, accessed on September 12, 2025, https://www.geeksforgeeks.org/dsa/a-program-to-check-if-a-binary-tree-is-bst-or-not/
24. Check for BST | Problem of the Day | GeeksForGeeks - YouTube, accessed on September 12, 2025, https://www.youtube.com/watch?v=fJ5uBzl3na4
25. Binary Tree : pre order , inorder and post order - Stack Overflow, accessed on September 12, 2025, https://stackoverflow.com/questions/30636468/binary-tree-pre-order-inorder-and-post-order
26. Construct Tree from given Inorder and Preorder traversals ..., accessed on September 12, 2025, https://www.geeksforgeeks.org/dsa/construct-tree-from-given-inorder-and-preorder-traversal/
27. GeeksForGeeks | Construct Binary Tree from Inorder and Postorder - YouTube, accessed on September 12, 2025, https://www.youtube.com/watch?v=Ow9-bsE3-Cw
28. Heap (data structure) - Wikipedia, accessed on September 12, 2025, https://en.wikipedia.org/wiki/Heap_(data_structure)
29. Heap Data Structure Patterns - Medium, accessed on September 12, 2025, https://medium.com/@praveenkumar505987/heap-data-structure-patterns-64a0

7f979273
30. Heap Data Structures Explained: Applications, Problem-Solving Patterns, and Real-World Examples | by Architect Algos | ArchitectAlgos, accessed on September 12, 2025, https://www.architectalgos.com/heap-data-structures-explained-applications-problem-solving-patterns-and-real-world-examples-6256e4b8b600
31. Heap Data Structure Guide: Theory, Implementation & Interview Qs - Get SDE Ready, accessed on September 12, 2025, https://getsdeready.com/heap-data-structure-guide/
32. Find k largest elements in an array - GeeksforGeeks, accessed on September 12, 2025, https://www.geeksforgeeks.org/dsa/k-largestor-smallest-elements-in-an-array/
33. Find the top K elements in O(N log K) time using heaps - Stack Overflow, accessed on September 12, 2025, https://stackoverflow.com/questions/49217910/find-the-top-k-elements-in-on-log-k-time-using-heaps
34. How to find the kth largest element without sorting - Quora, accessed on September 12, 2025, https://www.quora.com/How-can-I-find-the-kth-largest-element-without-sorting
35. Kth Largest Element in an Array - LeetCode, accessed on September 12, 2025, https://leetcode.com/problems/kth-largest-element-in-an-array/
36. Kth Largest Element in an Array - GeeksforGeeks, accessed on September 12, 2025, https://www.geeksforgeeks.org/dsa/kth-largest-element-in-an-array/
37. Top K Frequent Elements - LeetCode, accessed on September 12, 2025, https://leetcode.com/problems/top-k-frequent-elements/
38. Top K Frequent Elements in an Array - GeeksforGeeks, accessed on September 12, 2025, https://www.geeksforgeeks.org/dsa/find-k-numbers-occurrences-given-array/
39. For finding top K elements using heap, which approach is better - NlogK or KLogN?, accessed on September 12, 2025, https://stackoverflow.com/questions/64015196/for-finding-top-k-elements-using-heap-which-approach-is-better-nlogk-or-klogn
40. Find Median from Running Data Stream - GeeksforGeeks, accessed on September 12, 2025, https://www.geeksforgeeks.org/dsa/median-of-stream-of-integers-running-integers/
41. 295. Find Median from Data Stream - In-Depth Explanation, accessed on September 12, 2025, https://algo.monster/liteproblems/295
42. Median of Stream of Running Integers using STL - GeeksforGeeks, accessed on September 12, 2025, https://www.geeksforgeeks.org/dsa/median-of-stream-of-running-integers-using-stl/
43. Find Median from Data Stream - LeetCode, accessed on September 12, 2025, https://leetcode.com/problems/find-median-from-data-stream/

44. Graph cheatsheet for coding interviews - Tech Interview Handbook, accessed on September 12, 2025, https://www.techinterviewhandbook.org/algorithms/graph/
45. Graph Algorithms - GeeksforGeeks, accessed on September 12, 2025, https://www.geeksforgeeks.org/dsa/graph-data-structure-and-algorithms/
46. Breadth First Search Tutorials & Notes | Algorithms - HackerEarth, accessed on September 12, 2025, https://www.hackerearth.com/practice/algorithms/graphs/breadth-first-search/tutorial/
47. BFS in C++: Breadth-First Search Algorithm & Program - FavTutor, accessed on September 12, 2025, https://favtutor.com/blogs/bfs-breadth-first-search-cpp
48. C++ Program for BFS Traversal - GeeksforGeeks, accessed on September 12, 2025, https://www.geeksforgeeks.org/cpp/cpp-program-for-bfs-traversal/
49. Breadth First Search or BFS for a Graph - GeeksforGeeks, accessed on September 12, 2025, https://www.geeksforgeeks.org/dsa/breadth-first-search-or-bfs-for-a-graph/
50. Breadth-First Search (BFS) - CelerData, accessed on September 12, 2025, https://celerdata.com/glossary/breadth-first-search-bfs
51. Depth First Search (DFS) Algorithm - Programiz, accessed on September 12, 2025, https://www.programiz.com/dsa/graph-dfs
52. Iterative Depth First Traversal of Graph | GeeksforGeeks - YouTube, accessed on September 12, 2025, https://www.youtube.com/watch?v=Lpmodhe8Bq0
53. Depth First Search (DFS) Algorithm - Tutorials Point, accessed on September 12, 2025, https://www.tutorialspoint.com/data_structures_algorithms/depth_first_traversal.htm
54. Depth First Search or DFS for a Graph - GeeksforGeeks, accessed on September 12, 2025, https://www.geeksforgeeks.org/dsa/depth-first-search-or-dfs-for-a-graph/
55. "Mastering Graph Algorithms: A Comprehensive DSA Graph Common Question Patterns CheatSheet" - Discuss - LeetCode, accessed on September 12, 2025, https://leetcode.com/discuss/post/3900838/mastering-graph-algorithms-a-comprehensi-xyws/
56. C++ Program for DFS Traversal - GeeksforGeeks, accessed on September 12, 2025, https://www.geeksforgeeks.org/cpp/cpp-program-for-dfs-traversal/
57. Depth First Traversal for a Graph | GeeksforGeeks - YouTube, accessed on September 12, 2025, https://www.youtube.com/watch?v=Y40bRyPQQr0
58. Number of Islands - GeeksforGeeks, accessed on September 12, 2025, https://www.geeksforgeeks.org/dsa/find-the-number-of-islands-using-dfs/
59. Size of Islands in C++ (Time Complexity: O(m*n)) /homework - AlgoCademy, accessed on September 12, 2025, https://algocademy.com/link/?problem=size-of-islands&lang=cpp&solution=1
60. Topological Sorting - GeeksforGeeks, accessed on September 12, 2025, https://www.geeksforgeeks.org/dsa/topological-sorting/
61. Topological Sort Algorithm | DFS: G-21 - Tutorial - takeUforward, accessed on September 12, 2025,

https://takeuforward.org/data-structure/topological-sort-algorithm-dfs-g-21/
62. C++ Program for Topological Sorting - GeeksforGeeks, accessed on September 12, 2025, https://www.geeksforgeeks.org/cpp/cpp-program-for-topological-sorting/
63. Topological Sort Algorithm - Interview Cake, accessed on September 12, 2025, https://www.interviewcake.com/concept/java/topological-sort
64. Disjoint-set data structure - Wikipedia, accessed on September 12, 2025, https://en.wikipedia.org/wiki/Disjoint-set_data_structure
65. Union By Rank and Path Compression in Union-Find Algorithm ..., accessed on September 12, 2025, https://www.geeksforgeeks.org/dsa/union-by-rank-and-path-compression-in-union-find-algorithm/
66. Kruskal's Minimum Spanning Tree (MST) Algorithm - GeeksforGeeks, accessed on September 12, 2025, https://www.geeksforgeeks.org/dsa/kruskals-minimum-spanning-tree-algorithm-greedy-algo-2/
67. Kruskal's algorithm - Wikipedia, accessed on September 12, 2025, https://en.wikipedia.org/wiki/Kruskal%27s_algorithm
68. Kruskal's Minimum Spanning Tree (MST) Algorithm - GeeksforGeeks | Community Highlights & Summary | Glasp, accessed on September 12, 2025, https://glasp.co/discover?url=www.geeksforgeeks.org%2Fkruskals-minimum-spanning-tree-algorithm-greedy-algo-2%2F
69. Kruskal's Minimum Spanning Tree using STL in C++ - GeeksforGeeks, accessed on September 12, 2025, https://www.geeksforgeeks.org/dsa/kruskals-minimum-spanning-tree-using-stl-in-c/
70. Dijkstra's Algorithm to find Shortest Paths from a Source to all ..., accessed on September 12, 2025, https://www.geeksforgeeks.org/dsa/dijkstras-shortest-path-algorithm-greedy-algo-7/
71. Dijkstra - finding shortest paths from given vertex - Algorithms for Competitive Programming, accessed on September 12, 2025, https://cp-algorithms.com/graph/dijkstra.html
72. C / C++ Program for Dijkstra's shortest path algorithm | Greedy Algo-7 - GeeksforGeeks, accessed on September 12, 2025, https://www.geeksforgeeks.org/cpp/c-program-for-dijkstras-shortest-path-algorithm-greedy-algo-7/
73. Dijkstra's Algorithm - Programiz, accessed on September 12, 2025, https://www.programiz.com/dsa/dijkstra-algorithm