ChatGPT

LeetCode Coding Patterns Cheat Sheet (Java)

Sliding Window

When to Use It: - Use the sliding window pattern for problems involving contiguous subarrays or substrings, especially when you need to find an optimal (longest, shortest, or target-sum) sub-sequence in an array or string. - Typically applicable when the problem asks for a subarray/substring that meets a certain condition (such as a sum, average, or containing certain elements) and a brute force approach would involve checking many overlapping subarrays.

Why to Use It: - It greatly reduces time complexity by avoiding re-processing of the array for each subwindow. We maintain a "window" and slide it through the data, updating the state (like sum or counts) in O(1) time as the window moves. - This pattern turns nested-loop problems into single-loop solutions by expanding or shrinking the window based on conditions, making algorithms efficient (often O(n) for array/string traversal).

Example Code (Java): Using a dynamic sliding window to find the smallest subarray with sum ≥ `S`:

```
int windowSum = 0, windowStart = 0;
int minLength = Integer.MAX_VALUE;
for (int windowEnd = 0; windowEnd < arr.length; windowEnd++) {
    windowSum += arr[windowEnd];          // expand the window by adding the
next element
    while (windowSum >= S) {              // shrink window as long as the
condition is met
        minLength = Math.min(minLength, windowEnd - windowStart + 1);
        windowSum -=
arr[windowStart];    // remove the element going out of the window
        windowStart++;
    }
}
```

Sample LeetCode Problems: - Longest Substring Without Repeating Characters
- Minimum Window Substring
- Sliding Window Maximum
- Longest Substring with At Most K Distinct Characters
- Fruit Into Baskets (Longest Subarray with 2 Distinct Characters)
- Find All Anagrams in a String
- Permutation in String
- Minimum Size Subarray Sum

- Max Consecutive Ones III
- Longest Repeating Character Replacement

Two Pointers

When to Use It: - Use two pointers for array or string problems where you need to compare or find pairs of elements. Common scenarios include sorted arrays (find two numbers adding to a target), or partitioning tasks (moving elements around, skipping certain values). - This pattern is effective in problems where one pointer starts at the beginning and another at the end (to converge inwards), or one slow pointer and one fast pointer through the array (to partition or remove elements in-place).

Why to Use It: - It eliminates the need for nested loops by leveraging the sorted order or linear structure to move pointers towards a solution. By moving the pointers from both ends or one after the other, we can find desired pairs or arrangements in O(n) time. - Two pointers are also useful for in-place transformations (like removing duplicates or moving zeros) by having one pointer build the result while the other scans the array.

Example Code (Java): Finding two numbers in a sorted array that add up to a target:

```
int left = 0, right = arr.length - 1;
while (left < right) {
    int sum = arr[left] + arr[right];
    if (sum == target) {
        // found the pair (left, right)
        break;
    } else if (sum < target) {
        left++;    // need a larger sum, move left pointer to the right
    } else {
        right--;   // need a smaller sum, move right pointer to the left
    }
}
```

Sample LeetCode Problems: - Two Sum II - Input Array Is Sorted
- 3Sum
- 4Sum
- Container With Most Water
- Trapping Rain Water
- Remove Duplicates from Sorted Array
- Remove Element (in-place removal)
- Move Zeroes (in-place move)
- Valid Palindrome (check from both ends)
- Squares of a Sorted Array

Fast & Slow Pointers (Tortoise and Hare)

When to Use It: - Use fast & slow pointers when dealing with linked lists or cyclic sequences to detect cycles or find specific nodes. Classic use-cases are detecting a cycle in a linked list or finding the middle of a list (where one pointer moves twice as fast as the other). - Also applicable to numeric problems that form a sequence (e.g., the “happy number” problem) or circular array detection, where advancing pointers at different speeds can reveal a cycle or convergence point.

Why to Use It: - A fast pointer moving at 2x speed and a slow pointer moving at 1x will meet if there is a cycle, because the fast pointer eventually laps the slow pointer. This provides an O(n) cycle detection without extra space. - The pattern is also useful for finding mid-points (fast pointer hitting end means slow is at middle) and for algorithms that need to find an intersection point (e.g., start of a loop in a cycle) efficiently.

Example Code (Java): Detecting a cycle in a linked list and finding the start of the cycle:

```
ListNode slow = head, fast = head;
// Phase 1: Detect cycle
while (fast != null && fast.next != null) {
    slow = slow.next;
    fast = fast.next.next;
    if (slow == fast) {        // cycle detected
        break;
    }
}
if (fast != null && fast.next != null) {
    slow = head;
    while (slow != fast) {     // Phase 2: find cycle start
        slow = slow.next;
        fast = fast.next;
    }
    // slow (or fast) now points to the start of the cycle
}
```

Sample LeetCode Problems: - Linked List Cycle
- Linked List Cycle II (Start of cycle)
- Happy Number (cycle detection in number sequence)
- Find the Duplicate Number (cycle detection in array mapping)
- Middle of the Linked List
- Palindrome Linked List (find middle, then reverse second half)
- Reorder List (find middle, then rearrange using reversed second half)
- Circular Array Loop (cycle detection in an array by index hopping)
- Intersection of Two Linked Lists (two-pointer technique to find meeting node)
- Remove Nth Node From End of List (two pointers with offset to find target)

Merge Intervals (Overlapping Intervals)

When to Use It: - Use the merge intervals pattern when dealing with problems about intervals (ranges with start and end) — such as merging overlapping intervals, inserting an interval, or finding gaps/overlaps in schedules. - Typically applies when you have a list of intervals and need to combine them or analyze their overlaps (e.g., meeting room scheduling, interval covering, etc.).

Why to Use It: - Sorting intervals by start time and then merging overlapping ones is an efficient approach (O(n log n) due to sort, then O(n) scan). It simplifies complex interval overlap logic into a single pass merge. - This pattern ensures that you only traverse the list of intervals once after sorting, merging intervals on the fly and thus handling overlaps in linear time.

Example Code (Java): Merging overlapping intervals (assuming `intervals` is a list of int[]{start, end}):

```
Arrays.sort(intervals, (a, b) -> Integer.compare(a[0], b[0]));
List<int[]> merged = new ArrayList<>();
for (int[] interval : intervals) {
    if (merged.isEmpty() || merged.get(merged.size()-1)[1] < interval[0]) {
        merged.add(interval);  // no overlap, add interval
    } else {
        // overlap detected, merge with the last interval
        merged.get(merged.size()-1)[1] = Math.max(merged.get(merged.size()-1)
[1], interval[1]);
    }
}
// merged list contains the merged intervals
```

Sample LeetCode Problems: - Merge Intervals
- Insert Interval
- Non-overlapping Intervals (erase intervals to avoid overlap)
- Meeting Rooms (check if any overlaps exist)
- Meeting Rooms II (minimum number of rooms for all intervals)
- Interval List Intersections
- Minimum Number of Arrows to Burst Balloons
- Car Pooling (can be viewed as interval capacity scheduling)
- Employee Free Time
- Find Right Interval

Cyclic Sort (In-place Sorting for [1..N] Range)

When to Use It: - Use cyclic sort for problems dealing with an array of length `n` containing numbers from a known range (typically `1` to `n` or `0` to `n`), especially when asked to find missing or duplicate numbers. The approach is to place each number in its “correct” index position. - It’s ideal when the array elements are in a constrained range and you want an O(n) time, O(1) space solution by reordering the array in-place instead of using extra data structures.

Why to Use It: - Cyclic sort efficiently sorts the array without comparisons by repeatedly swapping misplaced elements into their correct positions. Once the array is "mostly" sorted this way, finding the first place where the index doesn't match the value reveals a missing number or a duplicate. - This pattern is powerful for detecting anomalies (missing/duplicate) because after in-place sorting, any index `i` that doesn't contain `i+1` indicates a problem (missing or duplicate value) at that position.

Example Code (Java): Sorting an array containing numbers 1 through N in-place (to then find missing/duplicate):

```
int i = 0;
while (i < arr.length) {
    int correctIdx = arr[i] - 1;
    if (arr[i] >= 1 && arr[i] <= arr.length && arr[i] != arr[correctIdx]) {
        // swap arr[i] with arr[correctIdx]
        int temp = arr[i];
        arr[i] = arr[correctIdx];
        arr[correctIdx] = temp;
    } else {
        i++;
    }
}
// After this, any index i where arr[i] != i+1 is a mis-match (missing or
duplicate found)
```

Sample LeetCode Problems: - Missing Number
- First Missing Positive
- Find All Duplicates in an Array
- Find All Numbers Disappeared in an Array
- Set Mismatch
- Find the Duplicate Number
- Kth Missing Positive Number

In-Place Reversal of a Linked List

When to Use It: - Use this pattern when you need to reverse a linked list (either the entire list or a portion of it) without using extra space. Typical scenarios include reversing the whole list, reversing a sub-section of a list, or reversing nodes in groups (k-group reversal). - It is also useful as a sub-step in other problems (for example, checking if a linked list is a palindrome by reversing the second half, or reordering a list by splitting and reversing one part).

Why to Use It: - In-place reversal is memory efficient (O(1) extra space) because it only reassigns pointers, and it's usually linear time. This pattern allows modifications to the list structure on the fly without needing an auxiliary list or stack. - Mastering pointer manipulation for reversal helps in many linked list problems, reducing them to a series of pointer swaps and making the solution clean and in-place.

Example Code (Java): Reversing a linked list in-place (iterative approach):

```
ListNode prev = null;
ListNode curr = head;
while (curr != null) {
    ListNode nextNode = curr.next;
    curr.next = prev;    // reverse the link
    prev = curr;
    curr = nextNode;
}
// prev now points to the new head of the reversed list
```

Sample LeetCode Problems: - Reverse Linked List
- Reverse Linked List II (reverse a sublist between two positions)
- Reverse Nodes in k-Group
- Swap Nodes in Pairs (swap adjacent nodes, a form of local reversal)
- Palindrome Linked List (reverse second half to compare)
- Reorder List (reverse second half and merge)
- Rotate List (can be done by finding tail and adjusting pointers)
- Swapping Nodes in a Linked List (find two nodes and swap their values/pointers)

Breadth-First Search (BFS)

When to Use It: - Use BFS for tree or graph problems where you need to traverse level by level (in trees, this is Level Order Traversal) or find the shortest path in an **unweighted** graph (or a grid). It's ideal for scenarios where exploring neighbors in waves makes sense (e.g., finding the minimum number of moves or steps to reach a target). - BFS is also used in problems that naturally unfold in layers—like all nodes distance 1 from the source, then distance 2, etc. (e.g., word ladder transformations, puzzles, or multi-source scenarios like rotten oranges spreading).

Why to Use It: - BFS guarantees the shortest path (minimum number of edges) from the start node to any other reachable node in an unweighted graph or grid because it explores neighbors level by level. In a tree, it systematically visits nodes in increasing depth order which is useful for operations that require processing by levels. - It uses a queue to maintain the frontier of exploration, ensuring that nodes are processed in the order they are discovered (first-in-first-out), which is ideal for breadth-wise expansion.

Example Code (Java): Level-order traversal of a binary tree using BFS:

```
Queue<TreeNode> queue = new LinkedList<>();
queue.offer(root);
while (!queue.isEmpty()) {
    TreeNode current = queue.poll();
    // process current node (e.g., add current.val to result list)
    if (current.left != null)  queue.offer(current.left);
```

```
    if (current.right != null) queue.offer(current.right);
}
```

Sample LeetCode Problems: - Binary Tree Level Order Traversal
- Binary Tree Zigzag Level Order Traversal
- Minimum Depth of Binary Tree
- Binary Tree Right Side View
- Rotting Oranges (minimum minutes to rot all oranges)
- Word Ladder (shortest transformation sequence length)
- Open the Lock (minimum turns to open lock)
- Shortest Path in Binary Matrix
- Bus Routes (least number of buses to destination)
- Course Schedule (detect cycle or find order using Kahn's algorithm)

Depth-First Search (DFS)

When to Use It: - Use DFS for tree and graph traversal when you need to explore as far as possible down one path before backtracking. It's natural for problems that require exploring all possibilities or paths (like finding all root-to-leaf paths in a tree, or connected components in graphs). - DFS is also the basis for backtracking (exhaustive search) and is used in scenarios like maze solving, puzzle solving, or anytime you need to recursively explore combinations or partitions.

Why to Use It: - DFS uses a stack (implicitly via recursion or an explicit stack) to dive deep. This is useful for exploring complete paths or making sure you traverse all nodes. It's simpler to implement recursively for tree traversals and can be more memory-efficient than BFS if the solution doesn't require tracking breadth. - In graphs, DFS can help detect cycles, perform topological sorting (via DFS post-order), and is the foundation of algorithms like Tarjan's (for strongly connected components) or solving puzzles by trying and backtracking.

Example Code (Java): Recursive DFS traversal on a graph:

```
void dfs(Node node, Set<Node> visited) {
    if (visited.contains(node)) return;
    visited.add(node);
    // process node (e.g., add to result)
    for (Node neighbor : node.neighbors) {
        dfs(neighbor, visited);
    }
}
```

(In a tree, you can call `dfs(node.left)` *and* `dfs(node.right)` *accordingly.)*

Sample LeetCode Problems: - Binary Tree Paths (all root-to-leaf paths)
- Path Sum / Path Sum II (has a root-to-leaf path with given sum / list all such paths)
- Number of Islands (DFS on a grid to mark connected lands)

- Max Area of Island (DFS to find area of connected region)
- Clone Graph (DFS to copy graph structure)
- Pacific Atlantic Water Flow (DFS from ocean borders)
- Surrounded Regions (capture regions by DFS from borders)
- All Paths From Source to Target (explore all paths in DAG)
- Course Schedule II (DFS for topological sort order)
- Word Search (DFS/backtracking in a grid of letters)

Topological Sort (Graph Ordering)

When to Use It: - Use topological sort for problems that ask for an order of tasks given dependencies (prerequisites). Typical scenarios include course scheduling, task scheduling, or determining a valid order to do things when some tasks must come before others (i.e., a Directed Acyclic Graph of dependencies). - This pattern applies only to directed acyclic graphs (DAGs). If the graph has a cycle, a topological order doesn't exist (which often translates to the problem having "no solution" or detecting an inconsistency).

Why to Use It: - Topological sorting gives a linear ordering of vertices such that for every directed edge (U -> V), U comes before V. It's useful to resolve dependency constraints in an order that respects all prerequisites. - Two common methods: **Kahn's algorithm (BFS-based)** uses indegree counts and repeatedly picks off nodes with indegree 0, and **DFS-based** which performs a post-order traversal and builds the order backwards. Both achieve O(V + E) time, providing an efficient way to order tasks.

Example Code (Java): Kahn's algorithm for topological sort (using indegrees and a queue):

```
int n = numVertices;
int[] indegree = new int[n];
List<List<Integer>> graph = new ArrayList<>();
// ... (build graph and fill indegree for each node) ...
Queue<Integer> q = new LinkedList<>();
for (int i = 0; i < n; i++) {
    if (indegree[i] == 0) q.offer(i);
}
List<Integer> topoOrder = new ArrayList<>();
while (!q.isEmpty()) {
    int u = q.poll();
    topoOrder.add(u);
    for (int v : graph.get(u)) {
        indegree[v]--;
        if (indegree[v] == 0) {
            q.offer(v);
        }
    }
}
// if topoOrder.size() < n, there was a cycle (not all vertices in order)
```

Sample LeetCode Problems: - Course Schedule (detect if it's possible to finish all courses)
- Course Schedule II (find one valid course order)
- Alien Dictionary (derive alphabet order from sorted words)
- Sequence Reconstruction (check if a sequence can be uniquely reconstructed from subsequences)
- Minimum Height Trees (find all roots that produce minimum tree height)
- Parallel Courses (minimum semesters to finish all courses given prerequisites)
- Alien Dictionary (repeated here for emphasis, classic topo sort problem)
- (Many scheduling problems can be mapped to topological sort of a DAG)

Backtracking (DFS with State)

When to Use It: - Use backtracking for problems where you need to explore all possible combinations or permutations and make decisions that lead to partial solutions, then undo (backtrack) and try other possibilities. This includes combinatorial problems like generating subsets, permutations, combinations, solving Sudoku, N-Queens, etc. - Backtracking is appropriate when a brute-force search is needed but with pruning of invalid paths as early as possible. It is essentially a DFS on a state-space tree of choices.

Why to Use It: - Backtracking systematically tries all paths, but cuts off those that violate constraints, which dramatically reduces the search space for many problems. It's a clear way to build solutions incrementally and backtrack when a partial solution cannot lead to a valid complete solution. - Although backtracking can be exponential in the worst case, it's often the only straightforward approach for exhaustive search problems. With careful pruning and heuristics, it can solve many practical problems reasonably.

Example Code (Java): Backtracking to generate all subsets of an array:

```
void backtrack(List<Integer> tempList, int start, int[] nums,
List<List<Integer>> result) {
    result.add(new ArrayList<>(tempList));          // record current subset
    for (int i = start; i < nums.length; i++) {
        tempList.add(nums[i]);                      // choose element nums[i]
        backtrack(tempList, i + 1, nums,
result);   // explore further with this choice
        tempList.remove(tempList.size() - 1);       // backtrack: remove last
chosen element
    }
}
```

Sample LeetCode Problems: - Subsets (all subsets of a set)
- Permutations (all permutations of a list)
- Permutations II (permutations with duplicate elements)
- Combinations (n choose k combinations)
- Combination Sum (find combinations that sum to target)
- Combination Sum II (combinations with no duplicate sets)
- N-Queens (place queens on a chessboard)
- Sudoku Solver

- Generate Parentheses (generate all valid parentheses strings)
- Palindrome Partitioning (partition string into palindromic substrings)

Dynamic Programming (DP)

When to Use It: - Use DP when a problem can be broken into overlapping subproblems that depend on smaller subproblems (optimal substructure). Classic signs include recursive formulas, decisions that depend on previous results (e.g., Fibonacci sequence, knapsack, edit distance, etc.), or asking for the optimum (min/max) value under constraints. - If you can formulate a problem in terms of smaller states (like using indices, or remaining capacity, or substrings) and it has repeated computations, DP is likely applicable. It's common in path-finding on grids, sequence alignment, partitioning problems, and optimization problems.

Why to Use It: - DP saves computation by storing results of subproblems (memoization or tabulation) so they are not recomputed. This often turns exponential brute-force recursion into polynomial time solutions. - By carefully defining state and recurrence relations, DP ensures that each state is solved once, and often allows straightforward calculation of the final answer from those states. It can be done top-down (recursive with memo) or bottom-up (iterative filling of a DP table).

Example Code (Java): Bottom-up DP for Fibonacci (or Climbing Stairs) as a simple example:

```
int fib(int n) {
    if (n < 2) return n;
    int[] dp = new int[n + 1];
    dp[0] = 0;
    dp[1] = 1;
    for (int i = 2; i <= n; i++) {
        dp[i] = dp[i-1] + dp[i-2];
    }
    return dp[n];
}
```

(This uses a DP array; one could optimize it to use two variables since only the last two results are needed.)

Sample LeetCode Problems: - Climbing Stairs (Fibonacci-style step count)
- House Robber (max sum with non-adjacent constraint)
- Coin Change (fewest coins to make an amount)
- Longest Increasing Subsequence
- Longest Common Subsequence
- Edit Distance (Levenshtein distance)
- Partition Equal Subset Sum (subset-sum problem)
- Unique Paths (grid path count)
- Longest Palindromic Substring
- Word Break

Greedy

When to Use It: - Use greedy algorithms when a problem can be solved by making a sequence of local optimal choices that lead to a global optimal solution. Typical scenarios include interval scheduling (choose earliest finishing interval first), optimization problems like minimizing costs or maximizing profits where choosing the best immediate option works (e.g., coin change for certain coin systems, Huffman coding, etc.). - Greedy is effective when selecting a local optimum at each step can be proven (or observed) to yield a global optimum. These problems often involve sorting by some criterion and then iterating (e.g., scheduling by earliest deadline, choosing largest value first, etc.).

Why to Use It: - Greedy algorithms are usually simple and efficient (often O(n log n) due to sorting, or O(n) if already sorted) because they don't explore all possibilities — they commit to choices as they go. When applicable, they drastically reduce complexity compared to brute force or DP. - Many real-world like problems (like interval scheduling, activity selection, making change with standard coin denominations) have optimal greedy solutions, which are easier to implement and understand. The key is identifying the greedy strategy (sorting or selecting based on a heuristic) that is optimal.

Example Code (Java): Selecting maximum number of non-overlapping intervals (Activity Selection):

```
// intervals sorted by end time
Arrays.sort(intervals, (a,b) -> Integer.compare(a[1], b[1]));
int count = 0;
int lastEnd = Integer.MIN_VALUE;
for (int[] interval : intervals) {
    if (interval[0] >= lastEnd) {
        count++;
        lastEnd = interval[1];  // take this interval and update the end time
threshold
    }
}
```

Sample LeetCode Problems: - Jump Game (Greedy reachable check)
- Jump Game II (minimum jumps to reach end)
- Gas Station (circular tour problem)
- Candy (assign ratings greedily from two sides)
- Partition Labels (greedily cut string into largest possible chunks)
- Assign Cookies (greedily satisfy children with smallest sufficient cookie)
- Minimum Number of Arrows to Burst Balloons (interval greedy by end points)
- Non-overlapping Intervals (remove least intervals to avoid overlaps)
- Queue Reconstruction by Height (greedy insertion by height order)
- Lemonade Change (greedy check for giving change)

Binary Search

When to Use It: - Use binary search when you have a **sorted array** (or can impose a sorted order or monotonic condition on the search space) and need to find an element, insertion point, or decision boundary efficiently. - This pattern also applies to search in an implicit range or answer space — for example, finding a minimum feasible value or optimizing an answer (like minimum speed, capacity, or time) when a condition function is monotonic (if `x` is possible, then all > `x` are possible, etc.).

Why to Use It: - Binary search reduces the search space by half at each step, giving O(log n) time complexity for finding an element in a sorted list or for deciding on an answer value in a monotonic space. This is significantly faster than linear scans for large inputs. - It's a fundamental pattern for divide-and-conquer approaches. Beyond searching for exact values, it's used for finding boundaries (first/last occurrence, lower/upper bounds) and for problems where you "guess" an answer and check feasibility (often called binary search on the answer).

Example Code (Java): Classic binary search on a sorted array:

```
int binarySearch(int[] arr, int target) {
    int low = 0, high = arr.length - 1;
    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (arr[mid] == target) {
            return mid;              // target found
        } else if (arr[mid] < target) {
            low = mid + 1;           // target is in the right half
        } else {
            high = mid - 1;          // target is in the left half
        }
    }
    return -1;  // not found
}
```

Sample LeetCode Problems: - Binary Search (basic implementation)
- Search Insert Position
- First Bad Version
- Search in Rotated Sorted Array
- Find Minimum in Rotated Sorted Array
- Find Peak Element
- Find First and Last Position of Element in Sorted Array
- Find Smallest Letter Greater Than Target
- Koko Eating Bananas (binary search on eating speed)
- Capacity To Ship Packages Within D Days

Trie-Based (Prefix Tree)

When to Use It: - Use a Trie (prefix tree) for problems involving a large collection of strings where prefix querying is required. Typical scenarios include autocomplete systems, dictionary word searches (like finding words with a given prefix), word games (Boggle, Word Search II), and implementing word dictionaries with prefix/suffix constraints. - Tries are ideal when you need to repeatedly query or insert strings by prefix or need to store a dictionary of words for quick prefix-based lookups (each letter is one level in the tree). They trade memory for speed, enabling character-by-character branching.

Why to Use It: - A trie provides O(m) time for inserting and searching a word (where m is the word length), which can be much faster than comparing against all words especially when the number of words is large. All words sharing a prefix share the path in the trie, which is memory efficient for common prefixes. - Tries also support advanced operations like prefix search, lexicographic traversal, and even wildcard matching (with modifications). They are a clear choice whenever hashing is not sufficient (like needing longest prefix match or enumerating all words with a prefix).

Example Code (Java): Basic Trie implementation (insert and search):

```
class TrieNode {
    TrieNode[] children = new TrieNode[26];
    boolean isWord = false;
}
class Trie {
    TrieNode root = new TrieNode();
    void insert(String word) {
        TrieNode node = root;
        for (char c : word.toCharArray()) {
            int idx = c - 'a';
            if (node.children[idx] == null) {
                node.children[idx] = new TrieNode();
            }
            node = node.children[idx];
        }
        node.isWord = true;
    }
    boolean search(String word) {
        TrieNode node = root;
        for (char c : word.toCharArray()) {
            int idx = c - 'a';
            if (node.children[idx] == null) return false;
            node = node.children[idx];
        }
        return node.isWord;
    }
    boolean startsWith(String prefix) {
        TrieNode node = root;
```

```
        for (char c : prefix.toCharArray()) {
            int idx = c - 'a';
            if (node.children[idx] == null) return false;
            node = node.children[idx];
        }
        return true;
    }
}
```

Sample LeetCode Problems: - Implement Trie (Prefix Tree)
- Add and Search Word (Data structure with `.` wildcard)
- Word Search II (find all words in a grid)
- Design Search Autocomplete System
- Replace Words (replace substrings in sentence using dictionary prefixes)
- Word Break II (construct sentences from dictionary, a Trie can optimize prefix checks)
- Word Squares (build word square using prefix pruning)
- Longest Word in Dictionary (find longest word buildable letter by letter)
- Map Sum Pairs (Trie for prefix sum of weights)
- Prefix and Suffix Search (combined Trie solution for prefix & suffix queries)

Union-Find (Disjoint Set Union)

When to Use It: - Use Union-Find for problems about connectivity in graphs (especially undirected) — e.g., finding connected components, checking if adding an edge creates a cycle, connectivity queries, or grouping elements by some relation. It's common in network connectivity, Kruskal's algorithm for MST, and puzzles where "island" or grouping logic appears. - It's ideal when the problem can be modeled as incremental union operations and find queries, such as merging accounts, finding if people are in the same friend circle, or analyzing connections like social networks or computer networks.

Why to Use It: - Union-Find provides near O(1) (amortized inverse-Ackermann) time for union and find operations when optimized with path compression and union by rank/size. This efficiency allows handling large numbers of connectivity operations quickly. - It simplifies logic for connectivity: instead of doing DFS/BFS repeatedly for connectivity queries, we can just maintain and query set membership. It's also easy to detect cycles in an undirected graph — if two vertices are already connected (find returns same root) and we try to union them, that edge is a cycle.

Example Code (Java): Union-Find structure with path compression:

```
class UnionFind {
    int[] parent;
    int[] rank;
    UnionFind(int n) {
        parent = new int[n];
        rank = new int[n];
        for (int i = 0; i < n; i++) {
```

```
            parent[i] = i;
            rank[i] = 0;
        }
    }
    int find(int x) {
        if (parent[x] != x) {
            parent[x] = find(parent[x]);  // path compression
        }
        return parent[x];
    }
    boolean union(int x, int y) {
        int rootX = find(x), rootY = find(y);
        if (rootX == rootY) return false;
        if (rank[rootX] < rank[rootY]) {
            parent[rootX] = rootY;
        } else if (rank[rootX] > rank[rootY]) {
            parent[rootY] = rootX;
        } else {
            parent[rootY] = rootX;
            rank[rootX]++;
        }
        return true;
    }
}
```

Sample LeetCode Problems: - Number of Connected Components in an Undirected Graph
- Graph Valid Tree (check if graph is a single connected component without cycle)
- Number of Provinces (Friend Circles)
- Accounts Merge
- Redundant Connection (detect extra edge creating a cycle)
- Satisfiability of Equality Equations (union-find on variables)
- Most Stones Removed with Same Row or Column (connect stones by row/col)
- Regions Cut By Slashes (grid connectivity problem)
- Making a Large Island (connect components by flipping one 0 to 1)
- Min Cost to Connect All Points (use union-find in Kruskal's MST)

Heap / Priority Queue

When to Use It: - Use a Heap/Priority Queue when you need to repeatedly retrieve and/or remove the smallest or largest element in a dynamic set. Common scenarios: "Top K" or "Kth smallest/largest" problems, scheduling problems, merging sorted sequences, or anytime you maintain a running set of elements and need fast access to min or max. - Applicable in streaming or online algorithms where elements arrive and you need to maintain a structure (like median finding, or processing tasks by priority). Also used in Dijkstra's algorithm for shortest paths (min-heap on distances).

Why to Use It: - A min-heap or max-heap provides O(log n) insertion and removal of the extreme element, which is very efficient compared to O(n) for lists. This makes it ideal for handling large data where sorting all elements upfront isn't feasible or when the data is being produced incrementally. - Heaps simplify "keep K largest/smallest" logic by naturally discarding out-of-range elements. For example, maintaining a min-heap of size K for largest K elements (pop when size exceeds K) or a max-heap of size K for smallest K elements. They are also flexible for custom comparison (e.g., by frequency, by value, etc.).

Example Code (Java): Using a min-heap to find the Kth largest element:

```
PriorityQueue<Integer> minHeap = new PriorityQueue<>();
for (int num : nums) {
    minHeap.offer(num);
    if (minHeap.size() > k) {
        minHeap.poll();  // remove smallest element to keep size k
    }
}
// The root of the minHeap is the k-th largest element
```

Sample LeetCode Problems: - Kth Largest Element in an Array
- Top K Frequent Elements
- Merge k Sorted Lists
- K Closest Points to Origin
- Kth Smallest Element in a Sorted Matrix
- Find Median from Data Stream (two heaps: max-heap + min-heap)
- Sliding Window Median (two heaps or multiset to maintain window median)
- Task Scheduler (greedy scheduling using max-heap for tasks frequency)
- Smallest Range Covering Elements from K Lists (min-heap to track current min among iterators)
- Last Stone Weight (max-heap to simulate stone smashing)

Monotonic Stack / Monotonic Queue

When to Use It: - Use a monotonic stack or queue for problems that involve comparing elements across the array where you need to know the next greater or smaller element, or maintain a window with certain min/max properties. Common examples: Next Greater Element, Daily Temperatures, stock span, and computing spans or ranges in histograms. - A monotonic **stack** is useful when you want to efficiently find the next greater/smaller element for each element. A monotonic **queue** (deque) is useful for maintaining the max or min in a sliding window (e.g., Sliding Window Maximum).

Why to Use It: - Monotonic structures ensure that each element is pushed and popped at most once, yielding linear time solutions (O(n)) for problems that naive solutions would handle in O(n^2). By keeping the stack/queue ordered (monotonically increasing or decreasing), you discard elements that are no longer useful, thus avoiding redundant comparisons. - These patterns are particularly powerful for range queries and online processing of sequences where maintaining an invariant order in a stack/deque gives immediate access to the next needed element (like the next greater element to the right, or the maximum in the current window).

Example Code (Java): Next Greater Element to the right using a decreasing stack:

```java
int n = arr.length;
int[] result = new int[n];
Stack<Integer> stack = new Stack<>();  // will store values (or indices) in
decreasing order
for (int i = n - 1; i >= 0; i--) {
    while (!stack.isEmpty() && stack.peek() <= arr[i]) {
        stack.pop();    // pop all smaller or equal elements, they are not next
greater for arr[i]
    }
    result[i] = stack.isEmpty() ? -1 : stack.peek();
    stack.push(arr[i]);
}
```

Sample LeetCode Problems: - Next Greater Element I
- Next Greater Element II (circular array variant)
- Daily Temperatures
- Asteroid Collision (use stack to handle collisions)
- Largest Rectangle in Histogram (monotonic stack for heights span)
- Trapping Rain Water (compute water trapped using stack or two-pointer)
- Sliding Window Maximum (monotonic deque to maintain max in window)
- Min Stack (stack that can get min in O(1) — can use two stacks in monotonic fashion)
- Online Stock Span (monotonic stack to compute span of stock prices)
- Sum of Subarray Minimums (monotonic stack for next smaller elements)

Bit Manipulation

When to Use It: - Use bit manipulation for problems that involve binary representations of numbers, where operations like AND, OR, XOR, shifts, etc., can be leveraged. Typical scenarios include toggling bits, checking for even/odd, power of two, finding subsets (via bit masks), or optimizing space by storing booleans as bits. - Also useful in problems where you have a set of flags or states (bits in an integer can represent a subset or a set of boolean conditions), or when you need to perform arithmetic tricks (like multiplying/dividing by 2 via shifts, or checking common bits between numbers).

Why to Use It: - Bit operations are extremely fast (constant time) and can pack a lot of information in one integer. They often enable solutions that avoid loops (for example, checking if `n` is a power of two with `n & (n-1)` trick, or using XOR to find a unique number). - Using bit masks is a common technique to represent subsets or states in DP, enabling efficient transition and storage. Bit tricks can also simplify certain arithmetic or set operations that would otherwise require more complex logic.

Example Code (Java): Using a bit mask to generate all subsets of a set of size `n`:

```
int n = nums.length;
List<List<Integer>> allSubsets = new ArrayList<>();
for (int mask = 0; mask < (1 << n); mask++) {
    List<Integer> subset = new ArrayList<>();
    for (int j = 0; j < n; j++) {
        if ((mask & (1 << j)) != 0) {      // check if j-th bit of mask is 1
            subset.add(nums[j]);
        }
    }
    allSubsets.add(subset);
}
```

(Another common bit trick: `if ((x & (x - 1)) == 0)`*, then x is a power of two.)*

Sample LeetCode Problems: - Single Number (find unique number using XOR)
- Single Number II (use bit counts to find unique number appearing once when others appear thrice)
- Counting Bits (count set bits in all numbers 0..n)
- Power of Two (check if a number is a power of 2)
- Reverse Bits
- Sum of Two Integers (bitwise addition without `+`)
- Subsets (using bit masks as shown above)
- Missing Number (can use XOR of all indices and numbers)
- Hamming Distance (count differing bits between two numbers)
- Maximum XOR of Two Numbers in an Array

Graph – Shortest Path (Dijkstra & Bellman-Ford)

When to Use It: - Use shortest path algorithms for weighted graph problems asking for the minimum cost or distance to get from one node to another (or to all others). **Dijkstra's algorithm** is ideal for graphs with non-negative edge weights (like road networks, networks latency, etc.), to find the shortest path distances. **Bellman-Ford** is used when edge weights can be negative or when you need to detect negative cycles (albeit with higher time complexity). - Common scenarios: network delay, cheapest flight with some constraints, routing problems, etc. If the problem is an unweighted graph, BFS suffices for shortest path; for weighted, these algorithms come into play.

Why to Use It: - Dijkstra's algorithm efficiently finds shortest paths in O(E log V) using a min-heap, which is much faster than brute forcing all paths. It greedily picks the nearest unvisited node and relaxes its edges, guaranteeing the optimal distances (for non-negative weights). - Bellman-Ford runs in O(V * E) and is useful when edges can have negative weights (but no negative cycle reachable from source). It's simpler (repeatedly relax all edges) and can handle scenarios Dijkstra cannot (like detecting a profit cycle in currency exchange, which is a negative cycle problem). - Both are fundamental for pathfinding in weighted graphs and form the basis for more complex algorithms like Floyd-Warshall (all-pairs shortest paths) or Johnson's algorithm.

Example Code (Java): Dijkstra's algorithm for shortest paths from a source:

```
int n = graph.length;
int[] dist = new int[n];
Arrays.fill(dist, Integer.MAX_VALUE);
dist[start] = 0;
PriorityQueue<int[]> pq = new PriorityQueue<>(Comparator.comparingInt(a ->
a[0]));
pq.offer(new int[]{0, start});  // {distance, node}
while (!pq.isEmpty()) {
    int[] top = pq.poll();
    int d = top[0], u = top[1];
    if (d != dist[u]) continue;           // ignore outdated pair
    for (int[] edge : graph[u]) {         // for each neighbor (v, weight)
        int v = edge[0], w = edge[1];
        if (dist[u] + w < dist[v]) {
            dist[v] = dist[u] + w;
            pq.offer(new int[]{dist[v], v});
        }
    }
}
```

(For Bellman-Ford, you would relax all edges V-1 times in loops. If you can still relax on the V-th iteration, there's a negative cycle.)

Sample LeetCode Problems: - Network Delay Time (shortest time for signal to reach all nodes)
- Cheapest Flights Within K Stops (constrained shortest path, can use BFS/Bellman-Ford style)
- Path with Maximum Probability (maximize probability, analogous to shortest path by using -log weights or modifying comparison)
- The Maze II (shortest distance in a maze with rolling ball)
- Swim in Rising Water (minimum time to cross grid, solvable via BFS/Dijkstra on grid)
- Minimum Cost to Reach Destination in Time (graph with time and cost constraints)
- Alien Dictionary (topological sort for order, not exactly shortest path but DAG longest path concept)
- Bellman-Ford can be applied to detect negative cycles (e.g., Arbitrage problem if it existed on LeetCode)

Tree Traversals (Inorder, Preorder, Postorder, Level Order)

When to Use It: - Use tree traversal patterns to visit all nodes of a tree in a particular order. **Preorder** (root-left-right) is useful for copying a tree, prefix expression evaluation, or serialization. **Inorder** (left-root-right) is mainly used for binary search trees when you need values in sorted order. **Postorder** (left-right-root) is useful for deleting trees or postfix evaluations, and processing children before the parent. **Level Order** (BFS on tree) is used when you need to process nodes level by level (e.g., level order printing, connecting level siblings). - Essentially, whenever you have a tree and need to examine or output all nodes in a controlled order, one of these traversals will be applied. Many tree problems boil down to doing a DFS (pre/in/post) or BFS (level order) and handling the nodes in the required sequence.

Why to Use It: - Traversals ensure every node is visited exactly once, covering the entire tree systematically. Different traversal orders are chosen based on the problem's needs (e.g., inorder gives sorted sequence for

BST, level order is needed for problems like populating next right pointers or level-order averages). - Recognizing which traversal to use can simplify a tree problem greatly. For example, postorder can be used to compute properties bottom-up (like depths, balances), preorder can be used to create copies or flatten structures, and level order is key for breadth-wise computations (like breadth-first search applications on trees).

Example Code (Java): Recursive DFS traversals:

```
// Preorder traversal (Root -> Left -> Right)
void preorder(TreeNode node, List<Integer> result) {
    if (node == null) return;
    result.add(node.val);
    preorder(node.left, result);
    preorder(node.right, result);
}

// Inorder traversal (Left -> Root -> Right)
void inorder(TreeNode node, List<Integer> result) {
    if (node == null) return;
    inorder(node.left, result);
    result.add(node.val);
    inorder(node.right, result);
}

// Postorder traversal (Left -> Right -> Root)
void postorder(TreeNode node, List<Integer> result) {
    if (node == null) return;
    postorder(node.left, result);
    postorder(node.right, result);
    result.add(node.val);
}
```

(For level order, see the BFS pattern above using a queue.)

Sample LeetCode Problems: - Binary Tree Preorder Traversal
- Binary Tree Inorder Traversal
- Binary Tree Postorder Traversal
- Binary Tree Level Order Traversal
- Binary Tree Zigzag Level Order Traversal
- Binary Tree Level Order Traversal II (bottom-up level order)
- Binary Tree Right Side View (view requires level order or DFS with tracking depth)
- Binary Tree Paths (uses DFS preorder to record paths)
- Kth Smallest Element in a BST (inorder traversal of BST)
- Balanced Binary Tree (postorder used to compute heights bottom-up)

Prefix Sum (Cumulative Sum)

When to Use It: - Use prefix sums for array problems where you need to quickly calculate the sum of any subarray or when you need to transform a problem into cumulative frequencies. Common scenarios: range sum queries, finding subarrays with a target sum, or any problem where differences of cumulative totals simplify the computation (like finding equal subarray splits or handling incremental updates). - If you find yourself wanting to sum up subarrays repeatedly, a prefix sum (or difference array) can likely reduce the computation. It's also useful in 2D for submatrix sum queries or for converting a problem into a sum-over-range formulation.

Why to Use It: - A prefix sum array `P` of an array `A` (where `P[i]` is sum of `A[0..i-1]`) allows computation of sum of any subarray `A[i..j]` in O(1) as `P[j+1] - P[i]`. This optimization turns many O(n^2) sum computations into O(n). - In addition, prefix sums can be combined with hashing to find subarrays with certain sums (e.g., using a map to store seen prefix sums to detect equal sum segments). They can also help in finding patterns like subarrays with equal 0s and 1s (by treating one category as -1 and looking for equal prefix sum occurrences).

Example Code (Java): Building a prefix sum and answering range sum queries:

```
int n = arr.length;
int[] prefix = new int[n + 1];
prefix[0] = 0;
for (int i = 1; i <= n; i++) {
    prefix[i] = prefix[i-1] + arr[i-1];
}
// Now sum of subarray arr[i..j] can be obtained by:
int subarraySum(int i, int j) {
    return prefix[j+1] - prefix[i];
}
```

(For 2D arrays, you can build a 2D prefix matrix similarly to get submatrix sums in O(1).)

Sample LeetCode Problems: - Range Sum Query - Immutable (prefix sum for quick range queries)
- Range Sum Query 2D - Immutable
- Subarray Sum Equals K (using prefix sum and hash map for count)
- Contiguous Array (find longest subarray with equal 0s and 1s via prefix sum trick)
- Find Pivot Index (calculate total sum and prefix to find balance point)
- Maximum Subarray (Kadane's algorithm is essentially on prefix sum differences)
- Product of Array Except Self (prefix and suffix products)
- Subarray Sums Divisible by K (use prefix sum mod K to find congruent pairs)
- Continuous Subarray Sum (prefix sum mod pattern to detect multiples)
- Pivot Index (uses prefix sums to compare left/right sums efficiently)
