**Sliding Window Pattern**

#### What is the Sliding Window Pattern?

The sliding window pattern is a technique used to solve problems that involve a contiguous subarray or substring within an array or string. It involves maintaining a window that slides over the array/string to find the optimal solution without reprocessing elements unnecessarily. This helps in reducing the time complexity significantly from O(n^2) to O(n) in many cases.

#### How is it Used?

The sliding window pattern generally works in the following way:

1. Initialize the window with a starting point.

2. Expand or contract the window based on the problem's requirements.

3. Keep track of the best solution while the window slides over the array or string.

#### Where to Use (Conditions)?

The sliding window pattern is useful when:

- You need to find the maximum/minimum/sum/average of a subarray or substring.

- The problem involves contiguous elements.

- The problem can be broken down into overlapping subproblems.

#### Example Problem: Maximum Sum Subarray of Size K

Problem Statement: Given an array of integers and a number `k`, find the maximum sum of a subarray of size `k`.

#### Example to Understand

Consider the array `[2, 1, 5, 1, 3, 2]` and `k = 3`.

1. Start with the first window `[2, 1, 5]` which has a sum of `8`.

2. Slide the window one element to the right to `[1, 5, 1]` and update the sum to `7`.

3. Continue this until the end of the array.

The maximum sum of any subarray of size `k` is `10` from the subarray `[5, 1, 3]`.

def maximum\_sum\_of\_k\_size\_subarray(arr,k)->int:

    max\_sum=0

    window\_sum=0

    start=0

    for i in range(k):

        window\_sum+=arr[i]

    max\_sum=window\_sum

    for i in range(k,len(arr)):

        window\_sum += arr[i] - arr[start]

        max\_sum = max(max\_sum, window\_sum)

        start+=1

    print("maximum sum-->",max\_sum)

# arr = [2,1,5,1,3,2]

# k = 3

# maximum\_sum\_of\_k\_size\_subarray(arr,k)

if \_\_name\_\_ == "\_\_main\_\_":

    t = int(input("Enter number of test cases: "))

    for i in range(t):

        k = int(input("Enter the window size: "))

        arr = list(map(int,input().split()))

        maximum\_sum\_of\_k\_size\_subarray(arr,k)

# Example usage

arr = [2, 1, 5, 1, 3, 2]

k = 3

print(max\_sum\_subarray(arr, k)) # Output: 10

```

#### C++ Code

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

int maxSumSubarray(vector<int>& arr, int k) {

    int max\_sum = 0;

    int window\_sum = 0;

    int start = 0;

    // Calculate the sum of the first window

    for (int i = 0; i < k; ++i) {

        window\_sum += arr[i];

    }

    max\_sum = window\_sum;

    // Slide the window

    for (int end = k; end < arr.size(); ++end) {

        window\_sum += arr[end] - arr[start];

        max\_sum = max(max\_sum, window\_sum);

        start += 1;

    }

    return max\_sum;

}

int main() {

    vector<int> arr = {2, 1, 5, 1, 3, 2};

    int k = 3;

    cout << maxSumSubarray(arr, k) << endl;  // Output: 10

    return 0;

}

#### Problems on LeetCode

1. [Maximum Sum Subarray of Size K (Easy)](<https://leetcode.com/problems/maximum-average-subarray-i/>)

2. [Longest Substring Without Repeating Characters (Medium)](<https://leetcode.com/problems/longest-substring-without-repeating-characters/>)

3. [Permutation in String (Medium)](<https://leetcode.com/problems/permutation-in-string/>)

4. [Minimum Window Substring (Hard)](<https://leetcode.com/problems/minimum-window-substring>/)

5. [Sliding Window Maximum (Hard)](<https://leetcode.com/problems/sliding-window-maximum/>)

By understanding and practicing the sliding window pattern, you'll be able to efficiently solve a wide range of problems that involve contiguous subarrays or substrings.

**Two Pointers Pattern**

#### What is the Two Pointers Pattern?

The two pointers pattern is a technique used to solve problems involving sorted arrays or lists. It involves using two indices (or pointers) to iterate through the array from different directions, often to find pairs or triplets that meet certain conditions. This approach helps reduce the time complexity significantly compared to brute-force methods.

#### How is it Used?

1. Initialize Two Pointers: Typically, one pointer starts at the beginning (left) and the other at the end (right) of the array.

2. Move Pointers Based on Conditions: Move the pointers towards each other based on the conditions defined by the problem. The pointers may move inward, one of them might move faster, or they might skip certain elements.

3. Check Conditions: At each step, check if the conditions are met, and adjust the pointers accordingly until they meet or cross each other.

#### Where to Use (Conditions)?

The two pointers pattern is useful when:

- The array or list is sorted.

- You need to find pairs or triplets that sum to a specific value.

- The problem involves searching for a combination of elements that satisfy a condition.

#### Example Problem: Two Sum II (Input Array Is Sorted)

Problem Statement: Given an array of integers `numbers` that is already sorted in ascending order, find two numbers such that they add up to a specific target number. Return the indices of the two numbers (1-indexed).

#### Example to Understand

Consider the array `[2, 7, 11, 15]` and the target `9`.

1. Start with two pointers: `left` at the beginning (0) and `right` at the end (3).

2. Check the sum of the elements at the pointers:

- `numbers[left] + numbers[right] = 2 + 15 = 17` (too high, move `right` leftward).

3. Continue until you find the pair:

- `2 + 7 = 9` (correct, return indices `[1, 2]`).

#### Algorithm

1. Initialize `left` to 0 and `right` to `len(numbers) - 1`.

2. While `left < right`:

- Calculate `current\_sum = numbers[left] + numbers[right]`.

- If `current\_sum` is equal to the target, return `[left + 1, right + 1]`.

- If `current\_sum` is less than the target, move `left` pointer to the right.

- If `current\_sum` is greater than the target, move `right` pointer to the left.

3. If no pair is found, return an empty list.

#### Time and Space Complexity

- Time Complexity: O(n), where n is the number of elements in the array.

- Space Complexity: O(1), as we are using only a constant amount of extra space.

#### Python Code

def two\_sum(numbers, target):

    left, right = 0, len(numbers) - 1

    while left < right:

        current\_sum = numbers[left] + numbers[right]

        if current\_sum == target:

            return [left + 1, right + 1]

        elif current\_sum < target:

            left += 1

        else:

            right -= 1

    return []

# Example usage

numbers = [2, 7, 11, 15]

target = 9

print(two\_sum(numbers, target))  # Output: [1, 2]

#### C++ Code

#include <iostream>

#include <vector>

using namespace std;

vector<int> twoSum(vector<int>& numbers, int target) {

    int left = 0, right = numbers.size() - 1;

    while (left < right) {

        int current\_sum = numbers[left] + numbers[right];

        if (current\_sum == target) {

            return {left + 1, right + 1};

        } else if (current\_sum < target) {

            left++;

        } else {

            right--;

        }

    }

    return {};

}

int main() {

    vector<int> numbers = {2, 7, 11, 15};

    int target = 9;

    vector<int> result = twoSum(numbers, target);

    for (int i : result) {

        cout << i << " ";

    }

    // Output: 1 2

    return 0;

}

#### Problems on LeetCode

1. [Two Sum II - Input Array Is Sorted (Easy)](<https://leetcode.com/problems/two-sum-ii-input-array-is-sorted/>)

2. [3Sum (Medium)](<https://leetcode.com/problems/3sum/>)

3. [4Sum (Medium)](<https://leetcode.com/problems/4sum>/)

4. [Remove Duplicates from Sorted Array (Easy)](<https://leetcode.com/problems/remove-duplicates-from-sorted-array/>)

5. [Container With Most Water (Medium)](<https://leetcode.com/problems/container-with-most-water/>)

By understanding and practicing the two pointers pattern, you'll be able to efficiently solve a wide range of problems that involve finding pairs or combinations of elements that meet specific conditions in a sorted array or list.

**Fast and Slow Pointers Pattern**

#### What is the Fast and Slow Pointers Pattern?

The fast and slow pointers pattern, also known as the tortoise and hare algorithm, involves using two pointers that move through the array, list, or sequence at different speeds. This pattern is commonly used to detect cycles in linked lists or arrays, find the middle of a list, or solve other problems related to sequences.

#### How is it Used?

1. Initialize Two Pointers: One pointer (slow) moves one step at a time, while the other pointer (fast) moves two steps at a time.

2. Move Pointers: Continue moving the pointers through the sequence.

3. Check Conditions: At each step, check the conditions based on the problem requirements. For cycle detection, check if the fast pointer meets the slow pointer. For finding the middle, check when the fast pointer reaches the end.

#### Where to Use (Conditions)?

The fast and slow pointers pattern is useful when:

- You need to detect cycles in a linked list or array.

- You need to find the middle element of a linked list.

- The problem involves traversing sequences with different speeds.

#### Example Problem: Detect Cycle in a Linked List

Problem Statement: Given a linked list, determine if it has a cycle in it.

#### Example to Understand

Consider a linked list: `1 -> 2 -> 3 -> 4 -> 5 -> 3` (cycle back to node 3).

1. Initialize two pointers, slow and fast, both at the head.

2. Move the slow pointer one step and the fast pointer two steps.

3. If there is a cycle, the fast pointer will eventually meet the slow pointer.

4. If the fast pointer reaches the end (null), there is no cycle.

#### Algorithm

1. Initialize two pointers, `slow` and `fast`, both pointing to the head of the linked list.

2. Move `slow` one step and `fast` two steps in a loop.

3. If `fast` or `fast.next` becomes null, return false (no cycle).

4. If `slow` equals `fast`, return true (cycle detected).

#### Time and Space Complexity

- Time Complexity: O(n), where n is the number of elements in the linked list.

- Space Complexity: O(1), as we are using a constant amount of extra space.

#### Python Code

class ListNode:

    def \_\_init\_\_(self, x):

        self.val = x

        self.next = None

def hasCycle(head):

    if not head or not head.next:

        return False

    slow = head

    fast = head

    while fast and fast.next:

        slow = slow.next

        fast = fast.next.next

        if slow == fast:

            return True

    return False

# Example usage

head = ListNode(1)

head.next = ListNode(2)

head.next.next = ListNode(3)

head.next.next.next = ListNode(4)

head.next.next.next.next = ListNode(5)

head.next.next.next.next.next = head.next.next  # Create a cycle

print(hasCycle(head))  # Output: True

#### C++ Code

#include <iostream>

using namespace std;

class ListNode {

public:

    int val;

    ListNode\* next;

    ListNode(int x) : val(x), next(nullptr) {}

};

bool hasCycle(ListNode\* head) {

    if (!head || !head->next) {

        return false;

    }

    ListNode\* slow = head;

    ListNode\* fast = head;

    while (fast && fast->next) {

        slow = slow->next;

        fast = fast->next->next;

        if (slow == fast) {

            return true;

        }

    }

    return false;

}

int main() {

    ListNode\* head = new ListNode(1);

    head->next = new ListNode(2);

    head->next->next = new ListNode(3);

    head->next->next->next = new ListNode(4);

    head->next->next->next->next = new ListNode(5);

    head->next->next->next->next->next = head->next->next;  // Create a cycle

    cout << (hasCycle(head) ? "True" : "False") << endl;  // Output: True

    return 0;

}

#### Problems on LeetCode

1. [Linked List Cycle (Easy)](<https://leetcode.com/problems/linked-list-cycle/>)

2. [Linked List Cycle II (Medium)](<https://leetcode.com/problems/linked-list-cycle-ii/>)

3. [Find the Duplicate Number (Medium)](<https://leetcode.com/problems/find-the-duplicate-number/>)

4. [Middle of the Linked List (Easy)](<https://leetcode.com/problems/middle-of-the-linked-list>/)

5. [Palindrome Linked List (Easy)](<https://leetcode.com/problems/palindrome-linked-list/>)

By understanding and practicing the fast and slow pointers pattern, you'll be able to efficiently solve a wide range of problems that involve detecting cycles, finding the middle of sequences, and handling other sequence-based challenges.

**Merge Intervals Pattern**

#### What is the Merge Intervals Pattern?

The merge intervals pattern is a technique used to solve problems involving overlapping intervals. It involves merging overlapping intervals to produce a set of non-overlapping intervals. This pattern is commonly used in scheduling, meeting room allocation, and other scenarios where intervals need to be combined or adjusted.

#### How is it Used?

1. Sort Intervals: Start by sorting the intervals based on the start time.

2. Initialize Merged List: Create an empty list to hold merged intervals.

3. Iterate and Merge: Iterate through the sorted intervals, and for each interval:

- If the merged list is empty or the current interval does not overlap with the last merged interval, add it to the merged list.

- If it does overlap, merge the current interval with the last merged interval by updating the end time.

#### Where to Use (Conditions)?

The merge intervals pattern is useful when:

- You need to combine overlapping intervals.

- You need to handle intervals in scheduling problems.

- The problem involves adjusting or merging time ranges or numerical ranges.

#### Example Problem: Merge Intervals

Problem Statement: Given a collection of intervals, merge all overlapping intervals.

#### Example to Understand

Consider the intervals `[[1, 3], [2, 6], [8, 10], [15, 18]]`.

1. Sort the intervals: `[[1, 3], [2, 6], [8, 10], [15, 18]]`.

2. Initialize the merged list with the first interval: `[[1, 3]]`.

3. Iterate and merge:

- Merge `[1, 3]` and `[2, 6]` to get `[1, 6]`.

- Add `[8, 10]` as it doesn't overlap with `[1, 6]`.

- Add `[15, 18]` as it doesn't overlap with `[8, 10]`.

The merged intervals are `[[1, 6], [8, 10], [15, 18]]`.

#### Algorithm

1. Sort the intervals by their start time.

2. Initialize a list `merged` with the first interval.

3. Iterate through each interval:

- If the current interval's start time is less than or equal to the last merged interval's end time, merge them by updating the end time of the last merged interval.

- Otherwise, add the current interval to the merged list.

4. Return the merged list.

#### Time and Space Complexity

- Time Complexity: O(n log n), where n is the number of intervals (due to sorting).

- Space Complexity: O(n), as we are using a list to store the merged intervals.

#### Python Code

def merge\_intervals(intervals):

    if not intervals:

        return []

    # Sort intervals by the start time

    intervals.sort(key=lambda x: x[0])

    merged = [intervals[0]]

    for current in intervals[1:]:

        last = merged[-1]

        if current[0] <= last[1]:  # Overlapping intervals

            last[1] = max(last[1], current[1])  # Merge intervals

        else:

            merged.append(current)

    return merged

# Example usage

intervals = [[1, 3], [2, 6], [8, 10], [15, 18]]

print(merge\_intervals(intervals))  # Output: [[1, 6], [8, 10], [15, 18]]

#### C++ Code

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

vector<vector<int>> mergeIntervals(vector<vector<int>>& intervals) {

    if (intervals.empty()) {

        return {};

    }

    // Sort intervals by the start time

    sort(intervals.begin(), intervals.end(), [](const vector<int>& a, const vector<int>& b) {

        return a[0] < b[0];

    });

    vector<vector<int>> merged;

    merged.push\_back(intervals[0]);

    for (int i = 1; i < intervals.size(); ++i) {

        vector<int>& last = merged.back();

        vector<int>& current = intervals[i];

        if (current[0] <= last[1]) {  // Overlapping intervals

            last[1] = max(last[1], current[1]);  // Merge intervals

        } else {

            merged.push\_back(current);

        }

    }

    return merged;

}

int main() {

    vector<vector<int>> intervals = {{1, 3}, {2, 6}, {8, 10}, {15, 18}};

    vector<vector<int>> result = mergeIntervals(intervals);

    for (const auto& interval : result) {

        cout << "[" << interval[0] << ", " << interval[1] << "] ";

    }

    // Output: [1, 6] [8, 10] [15, 18]

    return 0;

}

#### Problems on LeetCode

1. [Merge Intervals (Medium)](<https://leetcode.com/problems/merge-intervals>/)

2. [Insert Interval (Medium)](<https://leetcode.com/problems/insert-interval/>)

3. [Employee Free Time (Hard)](<https://leetcode.com/problems/employee-free-time/>)

4. [Non-overlapping Intervals (Medium)](<https://leetcode.com/problems/non-overlapping-intervals>/)

5. [Meeting Rooms II (Medium)](<https://leetcode.com/problems/meeting-rooms-ii>/)

By understanding and practicing the merge intervals pattern, you'll be able to efficiently solve a wide range of problems that involve combining or adjusting overlapping intervals.

**Cyclic Sort Pattern**

#### What is the Cyclic Sort Pattern?

The cyclic sort pattern is a technique used to solve problems involving arrays where the elements are in a known range, typically from 1 to n. This pattern leverages the fact that the elements are supposed to be in a specific order and repositions them to their correct positions in the array. It is particularly useful for solving problems related to finding missing numbers, duplicates, or any problem that involves rearranging elements.

#### How is it Used?

1. Iterate through the Array: Use a loop to iterate through the array.

2. Place Elements in Correct Positions: For each element, if it is not in the correct position, swap it with the element at its correct position.

3. Continue Until Sorted: Repeat the process until all elements are in their correct positions.

#### Where to Use (Conditions)?

The cyclic sort pattern is useful when:

- The array elements are in a known range (e.g., 1 to n).

- You need to find missing or duplicate numbers.

- The problem involves rearranging elements in a specific order.

#### Example Problem: Find All Missing Numbers

Problem Statement: Given an array of integers where each integer is between 1 and n (inclusive), find all the integers between 1 and n that do not appear in the array.

#### Example to Understand

Consider the array `[4, 3, 2, 7, 8, 2, 3, 1]`.

1. Place Elements in Correct Positions:

- Swap elements to place each number at its correct index (i.e., value 1 at index 0, value 2 at index 1, etc.).

2. After Sorting: The array will look like `[1, 2, 3, 4, 3, 2, 7, 8]`.

3. Identify Missing Numbers:

- Iterate through the array and find indices that do not have the correct values.

Missing numbers are `5` and `6`.

#### Algorithm

1. Iterate through the array using a while loop.

2. For each element, if it is not in its correct position, swap it with the element at its correct position.

3. Continue the process until all elements are in their correct positions.

4. After sorting, iterate through the array again to identify missing numbers (indices where the value does not match the index + 1).

#### Time and Space Complexity

- Time Complexity: O(n), where n is the number of elements in the array.

- Space Complexity: O(1), as we are not using any extra space.

#### Python Code

def find\_disappeared\_numbers(nums):

    i = 0

    while i < len(nums):

        j = nums[i] - 1

        if nums[i] != nums[j]:

            nums[i], nums[j] = nums[j], nums[i]

        else:

            i += 1

    missing\_numbers = []

    for i in range(len(nums)):

        if nums[i] != i + 1:

            missing\_numbers.append(i + 1)

    return missing\_numbers

# Example usage

nums = [4, 3, 2, 7, 8, 2, 3, 1]

print(find\_disappeared\_numbers(nums))  # Output: [5, 6]

#### C++ Code

#include <iostream>

#include <vector>

using namespace std;

vector<int> findDisappearedNumbers(vector<int>& nums) {

    int i = 0;

    while (i < nums.size()) {

        int j = nums[i] - 1;

        if (nums[i] != nums[j]) {

            swap(nums[i], nums[j]);

        } else {

            ++i;

        }

    }

    vector<int> missing\_numbers;

    for (int i = 0; i < nums.size(); ++i) {

        if (nums[i] != i + 1) {

            missing\_numbers.push\_back(i + 1);

        }

    }

    return missing\_numbers;

}

int main() {

    vector<int> nums = {4, 3, 2, 7, 8, 2, 3, 1};

    vector<int> result = findDisappearedNumbers(nums);

    for (int num : result) {

        cout << num << " ";

    }

    // Output: 5 6

    return 0;

}

#### Problems on LeetCode

1. [Find All Numbers Disappeared in an Array (Easy)](<https://leetcode.com/problems/find-all-numbers-disappeared-in-an-array/>)

2. [Find the Duplicate Number (Medium)]([https://leetcode.com/problems/find-the-duplicate-number](https://leetcode.com/problems/find-the-duplicate-number/)/)

3. [Find All Duplicates in an Array (Medium)](<https://leetcode.com/problems/find-all-duplicates-in-an-array>/)

4. [Missing Number (Easy)](<https://leetcode.com/problems/missing-number/>)

5. [First Missing Positive (Hard)](<https://leetcode.com/problems/first-missing-positive/>)

By understanding and practicing the cyclic sort pattern, you'll be able to efficiently solve a wide range of problems that involve rearranging elements and finding missing or duplicate numbers in arrays with a known range.

**Depth-First Search (DFS) Pattern**

#### What is the DFS Pattern?

Depth-First Search (DFS) is a traversal technique used in graphs and trees. It explores as far as possible along each branch before backtracking. DFS can be implemented using a stack (either explicitly with a data structure or implicitly via recursion).

#### How is it Used?

1. Start at the Root: Begin at the root (or any arbitrary node in the case of a graph).

2. Explore Deeply: Traverse to the deepest node in the current path.

3. Backtrack: Once the deepest node is reached, backtrack and explore unvisited nodes.

4. Continue Until All Nodes are Visited: Repeat the process until all nodes are visited.

#### Where to Use (Conditions)?

DFS is useful when:

- You need to explore all nodes and paths in a graph or tree.

- The problem involves connectivity, paths, or cycles in graphs.

- You need to perform operations on all nodes (like topological sorting, finding connected components).

#### Example Problem: Depth-First Search in a Binary Tree

Problem Statement: Implement DFS to traverse a binary tree and print all nodes.

#### Example to Understand

Consider the binary tree:

```

1

/ \

2 3

/ \ \

4 5 6

```

Using DFS (preorder traversal):

1. Visit `1`

2. Visit `2`

3. Visit `4`

4. Visit `5`

5. Visit `3`

6. Visit `6`

#### Algorithm

1. Recursive Approach:

- Start at the root node.

- Visit the node (process it).

- Recursively visit the left subtree.

- Recursively visit the right subtree.

2. Iterative Approach:

- Use a stack to manage the nodes.

- Push the root node to the stack.

- While the stack is not empty:

- Pop a node from the stack.

- Visit the node.

- Push its right child to the stack (if exists).

- Push its left child to the stack (if exists).

#### Time and Space Complexity

- Time Complexity: O(V + E), where V is the number of vertices and E is the number of edges.

- Space Complexity: O(V) in the worst case for both recursive and iterative implementations (due to the call stack in recursion or the stack in iteration).

#### Python Code

class TreeNode:

    def \_\_init\_\_(self, val=0, left=None, right=None):

        self.val = val

        self.left = left

        self.right = right

# Recursive DFS (Preorder)

def dfs\_recursive(root):

    if root:

        print(root.val, end=' ')

        dfs\_recursive(root.left)

        dfs\_recursive(root.right)

# Iterative DFS (Preorder)

def dfs\_iterative(root):

    if not root:

        return

    stack = [root]

    while stack:

        node = stack.pop()

        print(node.val, end=' ')

        if node.right:

            stack.append(node.right)

        if node.left:

            stack.append(node.left)

# Example usage

root = TreeNode(1)

root.left = TreeNode(2)

root.right = TreeNode(3)

root.left.left = TreeNode(4)

root.left.right = TreeNode(5)

root.right.right = TreeNode(6)

print("Recursive DFS: ", end='')

dfs\_recursive(root)  # Output: 1 2 4 5 3 6

print("\nIterative DFS: ", end='')

dfs\_iterative(root)  # Output: 1 2 4 5 3 6

#### C++ Code

#include <iostream>

#include <stack>

using namespace std;

struct TreeNode {

    int val;

    TreeNode\* left;

    TreeNode\* right;

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

};

// Recursive DFS (Preorder)

void dfsRecursive(TreeNode\* root) {

    if (root) {

        cout << root->val << " ";

        dfsRecursive(root->left);

        dfsRecursive(root->right);

    }

}

// Iterative DFS (Preorder)

void dfsIterative(TreeNode\* root) {

    if (!root) return;

    stack<TreeNode\*> s;

    s.push(root);

    while (!s.empty()) {

        TreeNode\* node = s.top();

        s.pop();

        cout << node->val << " ";

        if (node->right) s.push(node->right);

        if (node->left) s.push(node->left);

    }

}

int main() {

    TreeNode\* root = new TreeNode(1);

    root->left = new TreeNode(2);

    root->right = new TreeNode(3);

    root->left->left = new TreeNode(4);

    root->left->right = new TreeNode(5);

    root->right->right = new TreeNode(6);

    cout << "Recursive DFS: ";

    dfsRecursive(root);  // Output: 1 2 4 5 3 6

    cout << "\nIterative DFS: ";

    dfsIterative(root);  // Output: 1 2 4 5 3 6

    return 0;

}

#### Problems on LeetCode

1. [Binary Tree Inorder Traversal (Easy)](<https://leetcode.com/problems/binary-tree-inorder-traversal/>)

2. [Binary Tree Preorder Traversal (Easy)](<https://leetcode.com/problems/binary-tree-preorder-traversal/>)

3. [Binary Tree Postorder Traversal (Easy)](<https://leetcode.com/problems/binary-tree-postorder-traversal>/)

4. [Maximum Depth of Binary Tree (Easy)](<https://leetcode.com/problems/maximum-depth-of-binary-tree/>)

5. [Path Sum (Easy)](<https://leetcode.com/problems/path-sum/>)

By understanding and practicing the DFS pattern, you'll be able to efficiently solve a wide range of problems that involve traversing trees and graphs, finding paths, and handling connectivity issues.

**Subsets Pattern**

#### What is the Subsets Pattern?

The subsets pattern involves generating all possible subsets (or combinations) of a given set of elements. This pattern is useful in solving problems related to power sets, combinations, and various scenarios where all possible groups or selections need to be considered.

#### How is it Used?

1. Start with an Empty Set: Begin with an empty subset.

2. Iteratively Build Subsets: For each element in the input set, create new subsets by adding the element to the existing subsets.

3. Include All Subsets: Continue this process until all elements have been processed, resulting in all possible subsets.

#### Where to Use (Conditions)?

The subsets pattern is useful when:

- You need to find all possible combinations of a set.

- The problem involves generating power sets.

- You need to explore all groupings or selections from a given set.

#### Example Problem: Generate All Subsets

Problem Statement: Given a set of distinct integers, return all possible subsets.

#### Example to Understand

Consider the input set `[1, 2, 3]`.

1. Start with the empty subset: `[[]]`.

2. Add `1` to all existing subsets: `[[], [1]]`.

3. Add `2` to all existing subsets: `[[], [1], [2], [1, 2]]`.

4. Add `3` to all existing subsets: `[[], [1], [2], [1, 2], [3], [1, 3], [2, 3], [1, 2, 3]]`.

#### Algorithm

1. Initialize the result with an empty subset.

2. Iterate through each element in the input set.

3. For each element, create new subsets by adding it to each existing subset in the result.

4. Add these new subsets to the result.

5. Return the result.

#### Time and Space Complexity

- Time Complexity: O(2^n), where n is the number of elements in the input set (since there are 2^n subsets).

- Space Complexity: O(2^n), as the result contains 2^n subsets.

#### Python Code

def subsets(nums):

    result = [[]]  # Start with the empty subset

    for num in nums:

        # For each element, add it to all existing subsets

        new\_subsets = [curr + [num] for curr in result]

        result.extend(new\_subsets)

    return result

# Example usage

nums = [1, 2, 3]

print(subsets(nums))  # Output: [[], [1], [2], [1, 2], [3], [1, 3], [2, 3], [1, 2, 3]]

#### C++ Code

#include <iostream>

#include <vector>

using namespace std;

vector<vector<int>> subsets(vector<int>& nums) {

    vector<vector<int>> result = {{}};

    for (int num : nums) {

        vector<vector<int>> new\_subsets;

        for (const auto& curr : result) {

            vector<int> new\_subset = curr;

            new\_subset.push\_back(num);

            new\_subsets.push\_back(new\_subset);

        }

        result.insert(result.end(), new\_subsets.begin(), new\_subsets.end());

    }

    return result;

}

int main() {

    vector<int> nums = {1, 2, 3};

    vector<vector<int>> result = subsets(nums);

    for (const auto& subset : result) {

        cout << "[";

        for (int num : subset) {

            cout << num << " ";

        }

        cout << "] ";

    }

    // Output: [] [1] [2] [1 2] [3] [1 3] [2 3] [1 2 3]

    return 0;

}

#### Problems on LeetCode

1. [Subsets (Medium)](<https://leetcode.com/problems/subsets>/)

2. [Subsets II (Medium)](<https://leetcode.com/problems/subsets-ii/>)

3. [Combination Sum (Medium)](<https://leetcode.com/problems/combination-sum>/)

4. [Combination Sum II (Medium)](<https://leetcode.com/problems/combination-sum-ii>/)

5. [Permutations (Medium)](<https://leetcode.com/problems/permutations>/)

By understanding and practicing the subsets pattern, you'll be able to efficiently solve a wide range of problems that involve generating all possible groupings, combinations, or selections from a given set.

**Bitwise XOR Pattern**

#### What is the Bitwise XOR Pattern?

The bitwise XOR (exclusive OR) pattern leverages the properties of the XOR operation to solve problems involving pairs, finding missing numbers, and other scenarios where elements need to be identified or manipulated based on their binary representations. XOR is a binary operation that takes two bits and returns `1` if they are different and `0` if they are the same.

#### How is it Used?

1. Identifying Unique Elements: XOR of a number with itself is 0, and XOR of a number with 0 is the number itself. This property is useful for finding unique elements in arrays where elements appear in pairs except for one or more unique elements.

2. Swapping Values: XOR can be used to swap two values without using a temporary variable.

3. Finding Missing Numbers: XOR can help identify a missing number in a sequence where elements are supposed to appear exactly once.

#### Where to Use (Conditions)?

The bitwise XOR pattern is useful when:

- Elements in the array are supposed to appear in pairs except for one or more unique elements.

- The problem involves finding missing numbers or unique elements.

- Efficient bitwise operations are preferred over more complex algorithms.

#### Example Problem: Find the Single Number

Problem Statement: Given a non-empty array of integers, every element appears twice except for one. Find that single one.

#### Example to Understand

Consider the array `[2, 2, 1]`.

Using XOR:

1. `2 ^ 2 = 0`

2. `0 ^ 1 = 1`

So, the unique number is `1`.

#### Algorithm

1. Initialize a variable `result` to 0.

2. Iterate through each element in the array.

3. XOR each element with `result`.

4. After completing the iteration, `result` will hold the unique element.

#### Time and Space Complexity

- Time Complexity: O(n), where n is the number of elements in the array.

- Space Complexity: O(1), as no extra space is used.

#### Python Code

def single\_number(nums):

    result = 0

    for num in nums:

        result ^= num  # XOR each element with result

    return result

# Example usage

nums = [4, 1, 2, 1, 2]

print(single\_number(nums))  # Output: 4

#### C++ Code

#include <iostream>

#include <vector>

using namespace std;

int singleNumber(vector<int>& nums) {

    int result = 0;

    for (int num : nums) {

        result ^= num;  // XOR each element with result

    }

    return result;

}

int main() {

    vector<int> nums = {4, 1, 2, 1, 2};

    cout << singleNumber(nums) << endl;  // Output: 4

    return 0;

}

#### Problems on LeetCode

1. [Single Number (Easy)](<https://leetcode.com/problems/single-number>/)

2. [Single Number II (Medium)](<https://leetcode.com/problems/single-number-ii/>)

3. [Single Number III (Medium)](<https://leetcode.com/problems/single-number-iii>/)

4. [Missing Number (Easy)]([https://leetcode.com/problems/missing-number](https://leetcode.com/problems/missing-number/)/)

5. [Find the Duplicate Number (Medium)](<https://leetcode.com/problems/find-the-duplicate-number/>)

### Properties of XOR

- Self-Inverse: `a ^ a = 0`

- Identity: `a ^ 0 = a`

- Commutative: `a ^ b = b ^ a`

- Associative: `(a ^ b) ^ c = a ^ (b ^ c)`

By understanding and practicing the bitwise XOR pattern, you'll be able to efficiently solve a wide range of problems that involve finding unique elements, missing numbers, and performing efficient bitwise operations.

**Top K Elements Pattern**

#### What is the Top K Elements Pattern?

The Top K Elements pattern is used to solve problems that require finding the largest, smallest, or most frequent K elements in a dataset. This pattern typically involves the use of data structures like heaps (priority queues) to efficiently manage and retrieve the top K elements.

#### How is it Used?

1. \*\*Using Min-Heap (for Largest K Elements):\*\* Maintain a min-heap of size K. Iterate through the elements and ensure that the heap contains the K largest elements by keeping it at size K and replacing the smallest element when a larger element is encountered.

2. \*\*Using Max-Heap (for Smallest K Elements):\*\* Similar to the min-heap, but maintain a max-heap of size K for finding the smallest K elements.

3. \*\*Using Frequency Map and Heap (for Most Frequent K Elements):\*\* Use a hash map to count the frequency of elements, then use a heap to keep track of the top K frequent elements.

#### Where to Use (Conditions)?

The Top K Elements pattern is useful when:

- You need to find the largest or smallest K elements in a dataset.

- The problem involves frequent elements, like finding the K most frequent numbers or words.

- Efficient retrieval of top K elements is required.

#### Example Problem: Top K Frequent Elements

\*\*Problem Statement:\*\* Given an integer array, return the K most frequent elements.

#### Example to Understand

Consider the array `[1, 1, 1, 2, 2, 3]` and `K = 2`.

1. Count the frequency of each element:

- `1: 3`

- `2: 2`

- `3: 1`

2. Use a min-heap to keep track of the top 2 frequent elements.

The result is `[1, 2]`.

#### Algorithm

1. Create a frequency map using a hash map.

2. Use a min-heap to keep track of the top K frequent elements.

3. Iterate through the frequency map, adding elements to the heap and maintaining its size at K.

4. Extract elements from the heap to get the result.

#### Time and Space Complexity

- \*\*Time Complexity:\*\* O(N log K), where N is the number of elements in the array.

- \*\*Space Complexity:\*\* O(N + K), for the frequency map and the heap.

#### Python Code

import heapq

from collections import Counter

def top\_k\_frequent(nums, k):

    # Step 1: Build the frequency map

    freq\_map = Counter(nums)

    # Step 2: Use a min-heap to keep track of top K frequent elements

    min\_heap = []

    for num, freq in freq\_map.items():

        heapq.heappush(min\_heap, (freq, num))

        if len(min\_heap) > k:

            heapq.heappop(min\_heap)

    # Step 3: Extract elements from the heap

    result = [num for freq, num in min\_heap]

    return result

# Example usage

nums = [1, 1, 1, 2, 2, 3]

k = 2

print(top\_k\_frequent(nums, k))  # Output: [1, 2]

#### C++ Code

#include <iostream>

#include <vector>

#include <unordered\_map>

#include <queue>

using namespace std;

vector<int> topKFrequent(vector<int>& nums, int k) {

    // Step 1: Build the frequency map

    unordered\_map<int, int> freq\_map;

    for (int num : nums) {

        freq\_map[num]++;

    }

    // Step 2: Use a min-heap to keep track of top K frequent elements

    priority\_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> min\_heap;

    for (auto& pair : freq\_map) {

        min\_heap.push({pair.second, pair.first});

        if (min\_heap.size() > k) {

            min\_heap.pop();

        }

    }

    // Step 3: Extract elements from the heap

    vector<int> result;

    while (!min\_heap.empty()) {

        result.push\_back(min\_heap.top().second);

        min\_heap.pop();

    }

    return result;

}

int main() {

    vector<int> nums = {1, 1, 1, 2, 2, 3};

    int k = 2;

    vector<int> result = topKFrequent(nums, k);

    for (int num : result) {

        cout << num << " ";

    }

    // Output: 2 1

    return 0;

}

#### Problems on LeetCode

1. [Top K Frequent Elements (Medium)](<https://leetcode.com/problems/top-k-frequent-elements/>)

2. [Kth Largest Element in an Array (Medium)](<https://leetcode.com/problems/kth-largest-element-in-an-array/>)

3. [Top K Frequent Words (Medium)](<https://leetcode.com/problems/top-k-frequent-words/>)

4. [K Closest Points to Origin (Medium)](<https://leetcode.com/problems/k-closest-points-to-origin/>)

5. [Find K Pairs with Smallest Sums (Medium)](<https://leetcode.com/problems/find-k-pairs-with-smallest-sums/>)

By understanding and practicing the Top K Elements pattern, you'll be able to efficiently solve problems that require finding the largest, smallest, or most frequent K elements in a dataset using heaps and other efficient data structures.