**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.