Two Pointers

Two Pointers:

- Sorted Array/Array is Sorted
- Two Indices/Pointers
- Pointer Manipulation
- Searching/Comparing/Pair Sum
- Closest/Difference
- Intersection/Union of Arrays
- Partitioning/Subarray with Specific Property
- Sliding Window Technique Mentioned Indirectly
- Multiple Pointers Technique
- Removing Duplicates
- Rearranging Array/Reversing Subarray
- Array Traversal

1. **Sorted Arrays or Linked Lists**: If the problem involves a sorted array or linked list, the Two Pointers Pattern is often applicable. By using two pointers that start from different ends of the array or list and move inward, you can efficiently search for a target element, find pairs with a given sum, or remove duplicates.
2. **Window or Range Operations**: If the problem requires performing operations within a specific window or range of elements in the array or list, the Two Pointers Pattern can be useful. By adjusting the positions of two pointers, you can control the size and position of the window and efficiently process the elements within it.
3. **Checking Palindromes or Subsequences**: If the problem involves checking for palindromes or subsequences within the array or list, the Two Pointers Pattern provides a systematic approach. By using two pointers that move toward each other or in opposite directions, you can compare elements symmetrically and determine whether a palindrome or subsequence exists.
4. **Partitioning or Segregation**: If the problem requires partitioning or segregating elements in the array or list based on a specific criterion (e.g., odd and even numbers, positive and negative numbers), the Two Pointers Pattern is often effective. By using two pointers to swap elements or adjust their positions, you can efficiently partition the elements according to the criterion.
5. **Meeting in the Middle**: If the problem involves finding a solution by converging two pointers from different ends of the array or list, the Two Pointers Pattern is well-suited. By moving the pointers toward each other and applying certain conditions or criteria, you can identify a solution or optimize the search process.

The Two Pointers Pattern typically involves a WHILE loop and the following components:

1. **Initialization**: Initialize two pointers (or indices) to start from different positions within the array or list. These pointers may initially point to the beginning, end, or any other suitable positions based on the problem requirements.
2. **Pointer Movement**: Move the pointers simultaneously through the array or list, typically in a specific direction (e.g., toward each other, in the same direction, with a fixed interval). The movement of the pointers may depend on certain conditions or criteria within the problem.
3. **Condition Check**: At each step or iteration, check a condition involving the elements pointed to by the two pointers. This condition may involve comparing values, checking for certain patterns, or performing other operations based on the problem requirements.
4. **Pointer Adjustment**: Based on the condition check, adjust the positions of the pointers as necessary. This adjustment may involve moving both pointers forward, moving only one pointer, or changing the direction or speed of pointer movement based on the problem logic.
5. **Termination**: Continue moving the pointers and performing condition checks until a specific termination condition is met. This condition may involve reaching the end of the array or list, satisfying a specific criterion, or finding a solution to the problem.

```
def two_sum(nums, target):
    left, right = 0, len(nums) - 1  # Initialize two pointers at the beginning and end of the array
    while left < right:
        current_sum = nums[left] + nums[right]
        if current_sum == target:
            return [left, right]  # Return the indices of the two numbers that sum up to the target
        elif current_sum < target:
            left += 1  # Move the left pointer to the right to increase the sum
        else:
            right -= 1  # Move the right pointer to the left to decrease the sum
    return []  # If no such pair is found, return an empty list


# Example usage:
nums = [-2, 1, 2, 4, 7, 11]
target = 13
result = two_sum(nums, target)
print("Indices of two numbers that sum up to", target, ":", result)  # Output: [2, 4] (2 + 11 = 13)
```

Revision #4
Created 19 February 2024 02:39:41 by victor
Updated 26 February 2024 19:54:16 by victor