

Sliding Window

Key terms:

- Fixed Size Subarray
 - Maximum/Minimum Subarray
 - Consecutive/Continuous Elements
 - Longest/Shortest Substring
 - Optimal Window
 - Substring/Window/Range
 - Frequency Count
 - Non-overlapping/Subsequence
 - Sum/Product/Average in a Window
 - Smallest/Largest Window
 - Continuous Increasing/Decreasing
1. **Finding Subarrays or Substrings:** If the problem involves finding a contiguous subarray or substring that meets specific criteria (such as having a certain sum, length, or containing certain elements), the Sliding Window Pattern is likely applicable. Examples include problems like finding the maximum sum subarray, the longest substring with K distinct characters, or the smallest subarray with a sum greater than a target value.
 2. **Optimizing Brute-Force Solutions:** If you have a brute-force solution that involves iterating over all possible subarrays or substrings, the Sliding Window Pattern can often help optimize the solution by reducing unnecessary iterations. By maintaining a window of elements or characters and adjusting its size dynamically, you can avoid redundant computations and achieve better time complexity.
 3. **Tracking Multiple Pointers or Indices:** If the problem involves maintaining multiple pointers or indices within the array or string, the Sliding Window Pattern provides a systematic approach to track and update these pointers efficiently. This is especially useful for problems involving two pointers moving inward from different ends of the array or string.
 4. **Window Size Constraints:** If the problem imposes constraints on the size of the window (e.g., fixed-size window, window with a maximum or minimum size), the Sliding Window Pattern is well-suited for handling such scenarios. You can adjust the window size dynamically while processing the elements or characters within the array or string.
 5. **Time Complexity Optimization:** If the problem requires optimizing time complexity while processing elements or characters in the array or string, the Sliding Window Pattern offers a strategy to achieve linear or near-linear time complexity. By efficiently traversing the array or string with a sliding window, you can often achieve better time complexity compared to naive approaches.

The Sliding Window Pattern typically involves a FOR loop and the following components:

1. **Initialization:** Start by initializing two pointers or indices: one for the start of the window (left pointer) and one for the end of the window (right pointer). These pointers define the current window.
2. **Expanding the Window:** Initially, the window may start with a size of 1 or 0. Move the right pointer to expand the window by including more elements or characters from the array or string. The window grows until it satisfies a specific condition or constraint.
3. **Contracting the Window:** Once the window satisfies the condition or constraint, move the left pointer to contract the window by excluding elements or characters from the beginning of the window. Continue moving the left pointer until the window no longer satisfies the condition or constraint.
4. **Updating Results:** At each step, update the result or perform necessary operations based on the elements or characters within the current window. This may involve calculating the maximum/minimum value, computing a sum, checking for a pattern, or solving a specific subproblem.
5. **Termination:** Continue moving the window until the right pointer reaches the end of the array or string. At this point, the algorithm terminates, and you obtain the final result based on the operations performed during each iteration of the window.

```
def max_sum_subarray(array, k):
    window_sum = 0
    max_sum = float('-inf') #infinitely small number
    window_start = 0

    for window_end in range(len(array)):
        window_sum = window_sum + array[window_end] # Add the next element to the window

        # If the window size exceeds 'k', slide the window by one element
        if window_end >= k - 1:
            max_sum = max(max_sum, window_sum) # Update the maximum sum
            print(max_sum)
            window_sum = window_sum - array[window_start] # Subtract the element going out of the window
            window_start = window_start + 1 # Slide the window to the right

    return max_sum

# Example usage:
array = [1, 3, -1, -3, 5, 3, 6, 7]
k = 3
result = max_sum_subarray(array, k)
print(array)
print("Maximum sum of subarray of size", k, ":", result) # Output: 16 (subarray: [3, 6, 7])
```

Revision #2

Created 20 February 2024 05:47:27 by victor

Updated 26 February 2024 19:54:16 by victor