Topic Introduction

Today's PrepLetter is all about *subarray* problems, a favorite theme in coding interviews. You'll often see questions that ask for the count, length, or properties of contiguous subarrays meeting a certain condition. The key techniques at play are **prefix sums** and the **sliding window** pattern.

Let's break down these concepts:

• **Prefix Sum**: This is a running total of the elements in an array. By storing the sum up to each index, you can compute the sum of any subarray in constant time. It's especially handy when you need to check sums for *all* possible subarrays efficiently, without recomputing each time.

• **Sliding Window**: Imagine a window moving across the array, expanding or shrinking based on your condition. This is useful when dealing with contiguous sequences where you don't need to look back at older elements once you've moved past them.

When are these useful?

• Whenever you have to find subarrays (not subsequences!) with certain sums, products, or lengths.
• They're very common in interviews, testing both algorithmic thinking and code efficiency.

Quick Example (not from today's problems):

Suppose you're asked: "Find the length of the longest subarray with sum less than or equal to a given value S."
You could use sliding window: start with two pointers, expand right until you exceed S, then move left pointer to shrink the window.

Today, we'll tackle three classic problems:

• Subarray Sum Equals K
• Subarray Product Less Than K
• Maximum Size Subarray Sum Equals k

Why are these grouped?
All three deal with properties of contiguous subarrays, and the optimal solutions cleverly use prefix sum or sliding window techniques. The first and third focus on subarray sums and use prefix sum with a hash map, while the second uses a sliding window for products, showing how a similar pattern adapts to different operations.

Let's dive in!

Problem 1: Subarray Sum Equals K

Leetcode #560: Subarray Sum Equals K

Problem Statement (in my words):
Given an array of integers and an integer k, count the number of contiguous subarrays whose sum equals k.

Example:

Input: `nums = [1, 2, 3], k = 3`
Output: `2`

Explanation: The subarrays are `[1,2]` and `[3]` (both sum to 3).

Thought Process:

- Brute-force would mean checking every possible subarray and summing their values. That's O(n^2) time.
- Instead, can we use running sums (prefix sums) to speed this up?

Try This:

For `nums = [1, 1, 1], k = 2`

What are the subarrays that sum to 2?

Brute-force Approach:

- For every starting index, try every ending index, sum the subarray, and check if it equals k.
- **Time Complexity:** O(n^2)

Optimal Approach: Prefix Sum + Hash Map

- Keep a running sum as you go through the array.
- At each step, check: have I seen a sum of (current sum - k) before?

If so, there are as many subarrays ending here summing to k as the number of times that sum occurred.

- Use a hash map to track how many times each prefix sum has been seen.

Python Solution

```
def subarraySum(nums, k):
    from collections import defaultdict

    count = 0
    curr_sum = 0
    prefix_sums = defaultdict(int)
    prefix_sums[0] = 1  # Needed for subarrays starting at index 0

    for num in nums:
        curr_sum += num
        count += prefix_sums[curr_sum - k]
        prefix_sums[curr_sum] += 1

    return count
```

Time Complexity: O(n)

Space Complexity: O(n) (in the worst case, all prefix sums are different)

Code Explanation

- `curr_sum` keeps the running total (prefix sum).

- `prefix_sums` counts how often each prefix sum has occurred.
- For each number:
  - Update the running sum.
  - If `(curr_sum - k)` was seen before, it means there exists a subarray ending here with sum k.
  - Increment the count accordingly.
  - Record the current running sum in the map.

Trace Example

For `nums = [1, 2, 3], k = 3`:

- Start: curr_sum = 0, prefix_sums = {0:1}
- num=1: curr_sum=1; prefix_sums[1-3]=prefix_sums[-2]=0; count=0; prefix_sums[1]=1
- num=2: curr_sum=3; prefix_sums[3-3]=prefix_sums[0]=1; count=1; prefix_sums[3]=1
- num=3: curr_sum=6; prefix_sums[6-3]=prefix_sums[3]=1; count=2; prefix_sums[6]=1

Final answer: 2

Try This Test Case:

`nums = [1, -1, 0], k = 0`

What should the answer be?

Take a moment to solve this on your own before jumping into the code!

Problem 2: Subarray Product Less Than K

Leetcode #713: Subarray Product Less Than K

Problem Statement (my words):

Given an array of positive integers and an integer k, count the number of contiguous subarrays where the product of all elements is less than k.

How is this similar or different from Problem 1?

Instead of sum, we're dealing with product. The sum trick (prefix sum and hash map) doesn't work for products, but the sliding window pattern does because all numbers are positive.

Brute-force Approach:

- Try every subarray, multiply to get the product.
- **Time Complexity:** O(n^2)

Optimal Approach: Sliding Window

- Because all numbers are positive, if the product is too high, moving the left pointer right will only decrease the product.
- For each right pointer, expand the window until the product is too high, then shrink from the left as needed.

Step-by-Step Logic:

- Initialize two pointers: left = 0, product = 1
- Iterate right from 0 to n-1:
  - Multiply product by nums[right]
  - While product >= k and left <= right: divide product by nums[left], move left pointer right
  - The number of subarrays ending at right and starting anywhere between left and right is (right - left + 1)
- Add that count to answer

Pseudocode:

```
initialize left = 0, product = 1, count = 0
for right from 0 to n-1:
    product *= nums[right]
    while product >= k and left <= right:
        product /= nums[left]
        left += 1
    count += right - left + 1
return count
```

Example:

Input: nums = [10, 5, 2, 6], k = 100

Output: 8

Step-by-step:

- right=0: product=10 < 100, count += 1
- right=1: product=50 < 100, count += 2
- right=2: product=100 >= 100; shrink left (product=10); count += 2
- right=3: product=60 < 100, count += 3

Total: 1+2+2+3 = 8

Try This Test Case:

`nums = [1, 2, 3], k = 0`

What should the answer be?

Time Complexity: O(n)

Space Complexity: O(1)

Problem 3: Maximum Size Subarray Sum Equals k

Leetcode #325: Maximum Size Subarray Sum Equals k

Problem Statement:

Given an integer array and an integer k, find the length of the longest contiguous subarray that sums to k.

How is it different?

Now, instead of *counting* subarrays, we want the *maximum length* of such a subarray. The sum can be positive, negative, or zero.

The pattern is similar to the first problem, but this time we care about the *first* occurrence of a prefix sum (to get the longest window).

Brute-force Approach:

- For every start index, try every end index, sum subarray, check if it equals k, and track max length.
- **Time Complexity:** O(n^2)

Optimal Approach: Prefix Sum + Hash Map (First Occurrence)

- Use a map to record the *first* occurrence of each prefix sum.
- For each index, check if (curr_sum - k) has been seen before.
- If so, the subarray between that index and current index sums to k. Update max length.
- If curr_sum hasn't been seen before, add it to the map (since we want the *longest*).

Pseudocode:

```
initialize prefix_map = {0: -1}
curr_sum = 0
max_len = 0
for i from 0 to n-1:
    curr_sum += nums[i]
    if (curr_sum - k) in prefix_map:
        max_len = max(max_len, i - prefix_map[curr_sum - k])
    if curr_sum not in prefix_map:
        prefix_map[curr_sum] = i
return max_len
```

Example:

Input: nums = [1, -1, 5, -2, 3], k = 3
Output: 4
Explanation: The subarray [-1, 5, -2, 3] sums to 3.

Try This Test Case:

```
nums = [-2, -1, 2, 1], k = 1
```

Time Complexity: O(n)
Space Complexity: O(n)

Summary and Next Steps

Today you tackled three subarray problems that showcase the power of prefix sums, hash maps, and sliding windows:

- **Prefix sums** with hash maps are perfect for finding subarrays with a target sum or counting their occurrences.
- **Sliding window** shines for contiguous subarrays with multiplicative or monotonic constraints (especially when all numbers are positive).

Key Patterns:

- When dealing with *sum*, try prefix sums and hash maps.
- When dealing with *product* and all numbers are positive, sliding window is often optimal.
- For longest subarray, track the earliest index for each prefix sum.

Common Pitfalls:

- Forgetting to initialize the prefix sum map with {0: 1} or {0: -1} as needed.
- Forgetting to handle zero or negative numbers (especially for products).
- Not updating the hash map only on first occurrence when finding the *longest* subarray.

Action List

- Solve all three problems yourself, even the one with code provided.
- For Problem 2, try thinking about how you'd handle negative numbers (hint: sliding window may not work).
- For Problem 3, can you extend the logic to find all such subarrays, not just the longest?
- Compare your solution to others on Leetcode — see how they handle edge cases.
- Practice explaining your approach out loud, as you would in an interview.

Keep practicing—these patterns will show up again and again!