Topic Introduction

Today's PrepLetter dives into the world of **subarray optimization with Dynamic Programming**. If you've ever been asked to "find the maximum sum subarray" or "get the largest product of a contiguous slice," you've touched this concept! These problems often show up in interviews because they test your ability to recognize patterns, optimize brute-force solutions, and handle subtle edge cases.

What is a subarray?
A subarray is a contiguous part of an array. For example, in `[3, -1, 2, 5]`, `[3, -1]` or `[2, 5]` are subarrays, but `[3, 2]` is not.

Dynamic Programming for Subarrays:
Dynamic Programming (DP) is a technique used to break down complex problems into simpler subproblems and store their solutions to avoid redundant work. When we talk about DP with subarrays, we're often trying to solve "maximum/minimum sum/product of subarrays" efficiently without checking every possible subarray (which can be very slow).

Kadane's Algorithm
This is the classic DP approach for finding the maximum sum subarray in O(n) time. The key idea: as you move through the array, at each position, decide whether to extend the previous subarray or start fresh at the current element.

Example (not one of today's problems):
Given `[2, -3, 4, -1, 2, 1, -5, 4]`, Kadane's algorithm helps you find that `[4, -1, 2, 1]` gives the max sum, which is 6.

Why is this useful for interviews?

- Efficiently solves problems that look brute-force at first glance.
- Tests your understanding of problem decomposition, state maintenance, and edge-case handling.
- Many variations build on this theme, so mastering it opens doors to handle more advanced problems.

Why These Problems Go Together

Today, we'll look at:

- Maximum Subarray
- Maximum Product Subarray
- Maximum Sum Circular Subarray

What connects them?
All three ask for an optimal subarray (sum or product), and all can be solved using Dynamic Programming, specifically variants of Kadane's Algorithm. The first is the classic sum, the second tweaks the approach to handle products (and negative numbers), and the third adds a twist by allowing subarrays to wrap around in a circular array.

Problem 1: Maximum Subarray

Problem Statement

Given an integer array `nums`, find the contiguous subarray (containing at least one number) that has the largest sum, and return its sum.

Leetcode Link

Example:

Input: `[-2,1,-3,4,-1,2,1,-5,4]`

Output: `6`

Explanation: The subarray `[4,-1,2,1]` has the largest sum = 6.

Try this input by hand:

Input: `[5, 4, -1, 7, 8]`

(What's the largest sum subarray?)

Brute-force Approach

Check all possible subarrays, compute their sums, and track the maximum.

Time Complexity: O(n^2) (or O(n^3) if you're not careful)

Why is this slow? There are O(n^2) subarrays, and summing each can take O(n) time.

Optimal Approach: Kadane's Algorithm

Core idea:

As you iterate, at each index, decide:

- Should I extend the current subarray, or
- Start a new subarray at this element?

Logic:

- Keep two variables:
  - `current_sum`: the max sum ending at the current position
  - `max_sum`: the overall max found so far
- At each number, `current_sum = max(num, current_sum + num)`
  - If adding the current number to the previous sum is worse than starting fresh, start fresh.
- Update `max_sum` if `current_sum` is greater.

Step-by-step Example:

Input: `[-2,1,-3,4,-1,2,1,-5,4]`

Init: `current_sum = max_sum = -2`

- At 1: `current_sum = max(1, -2+1) = 1` → `max_sum = 1`
- At -3: `current_sum = max(-3, 1-3) = -2`
- At 4: `current_sum = max(4, -2+4) = 4` → `max_sum = 4`
- At -1: `current_sum = max(-1, 4-1) = 3`
- At 2: `current_sum = max(2, 3+2) = 5` → `max_sum = 5`
- At 1: `current_sum = max(1, 5+1) = 6` → `max_sum = 6`

- (Continue...)

Try this input:

Input: `[1, -2, 3, 5, -1, 2]`

(What's the answer? Try with pen and paper!)

Python Solution

```
def maxSubArray(nums):
    # Initialize both to the first element
    current_sum = max_sum = nums[0]

    for num in nums[1:]:
        # Decide: extend or start new subarray
        current_sum = max(num, current_sum + num)
        # Update the overall maximum
        max_sum = max(max_sum, current_sum)

    return max_sum
```

Time Complexity: O(n)

Space Complexity: O(1) (no extra space except variables)

Code Explanation

- `current_sum` tracks the best sum ending at the current position.
- `max_sum` keeps your best answer so far.
- For each number, decide whether to start a new subarray or continue the previous one.
- Update `max_sum` if you found a new maximum.

Trace with `[−2,1,−3,4,−1,2,1,−5,4]`:

- Start: `current_sum = max_sum = -2`
- At 1: `current_sum = 1` (start new), `max_sum = 1`
- At -3: `current_sum = -2`, `max_sum = 1`
- At 4: `current_sum = 4` (start new), `max_sum = 4`
- At -1: `current_sum = 3`, `max_sum = 4`
- At 2: `current_sum = 5`, `max_sum = 5`
- At 1: `current_sum = 6`, `max_sum = 6`
- At -5: `current_sum = 1`, `max_sum = 6`
- At 4: `current_sum = 5`, `max_sum = 6`

Try this test case:

Input: `[8, -19, 5, -4, 20]`

What's the maximum subarray sum?

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

Problem 2: Maximum Product Subarray

Problem Statement

Given an integer array `nums`, find the contiguous subarray (containing at least one number) which has the largest product, and return the product.

Leetcode Link

Example:

Input: `[2,3,-2,4]`

Output: `6`

Explanation: The subarray `[2,3]` has the largest product = 6.

Another test case to try:

Input: `[-2,0,-1]`

(What do you expect? Try it on paper!)

How is this different from Problem 1?

- Now, we care about the product, not the sum.
- Negatives can flip the product from small to large (and vice versa).
- Zeroes can break subarrays.
- We need to track both the maximum and minimum products at each position.

Brute-force Approach

- Try all subarrays, calculate the product of each, and keep the maximum.
- **Time Complexity:** O(n^2) or worse.

Optimal Approach: Modified Kadane's Algorithm

Key insight:

At each position, the max product could be from:

- The current number alone
- Current number times previous max product
- Current number times previous min product (since multiplying two negatives can make a big positive)

So, at each step, keep track of:

- `max_prod`: the maximum product ending here

- `min_prod`: the minimum product ending here (could become max if next number is negative)

Steps:

- Initialize `max_prod`, `min_prod`, and `result` to `nums[0]`
- For each number from the second one onward:
  - If the number is negative, swap `max_prod` and `min_prod` (since the sign flips)
  - Update `max_prod = max(num, max_prod * num)`
  - Update `min_prod = min(num, min_prod * num)`
  - Update `result = max(result, max_prod)`

Pseudocode:

```
set max_prod, min_prod, result to nums[0]
for each num in nums[1:]:
    if num < 0:
        swap max_prod and min_prod
    max_prod = max(num, max_prod * num)
    min_prod = min(num, min_prod * num)
    result = max(result, max_prod)
return result
```

Example:

Input: `[2,3,-2,4]`

- Start: max_prod = min_prod = result = 2
- At 3:
  - num is positive, no swap
  - max_prod = max(3, 2x3) = 6
  - min_prod = min(3, 2x3) = 3
  - result = max(2, 6) = 6
- At -2:
  - num is negative, swap max_prod and min_prod
  - max_prod = max(-2, 3x-2) = max(-2, -6) = -2
  - min_prod = min(-2, 6x-2) = min(-2, -12) = -12
  - result = max(6, -2) = 6
- At 4:
  - num is positive, no swap
  - max_prod = max(4, -2x4) = max(4, -8) = 4
  - min_prod = min(4, -12x4) = min(4, -48) = -48
  - result = max(6, 4) = 6

So the answer is 6.

Try this test case:

Input: `[2, -5, -2, -4, 3]`

What’s the maximum product subarray?

Time Complexity: O(n)
Space Complexity: O(1)

Problem 3: Maximum Sum Circular Subarray

Problem Statement
Given a circular integer array `nums` (the next element of the last is the first), find the maximum possible sum of a non-empty subarray (contiguous elements, possibly wrapping around), and return it.
Leetcode Link

Example:
Input: `[1, -2, 3, -2]`
Output: `3`
Explanation: The subarray `[3]` has the maximum sum.

What’s the twist?

- The subarray can wrap around the end to the front.
- There are two possible types of max subarrays:
    - **Non-wrapping:** Standard max subarray (Kadane’s Algorithm)
    - **Wrapping:** The sum of the whole array minus the minimum subarray (i.e., removing the worst dip in the array)

Brute-force Approach

- Try all possible subarrays, including those that wrap around.
- **Time Complexity:** O(n^2)

Optimal Approach

Key insight:
- The maximum sum subarray is either:
    - The standard (non-circular) maximum subarray
    - The sum of the array minus the minimum subarray (which leaves you with the "wrap-around" part)
- Exception: if all numbers are negative, the "wrap-around" case gives 0, which is not valid (must return the largest single element).

Pseudocode:

```
total_sum = sum(nums)
max_sum = standard Kadane's result
```

```
min_sum = min subarray sum (Kadane's, but with min)
if max_sum < 0:
    return max_sum
else:
    return max(max_sum, total_sum - min_sum)
```

Example:

Input: `[5, -3, 5]`

- total_sum = 7
- max_sum = 10 (from [5, -3, 5])
- min_sum = -3 (from [-3])
- total_sum - min_sum = 10
- So, return max(10, 10) = 10

Try this test case:

Input: `[3, -1, 2, -1]`

What's the answer if the subarray can wrap?

Time Complexity: O(n)

Space Complexity: O(1)

Summary and Next Steps

What did we learn?

- All three problems use DP to optimize subarray computations.
- **Kadane's Algorithm** is the core pattern: at each step, decide to extend or restart.
- For products, track both max and min due to negative numbers.
- For circular arrays, combine max subarray logic with min subarray logic to consider wrap-arounds.

Common traps:

- Not handling negatives or zeros in the product problem.
- Forgetting to check for the "all-negative" case in the circular sum problem.
- Not resetting subarray computations correctly.

Action List:

- Solve all 3 problems on your own — even the one with code provided here!
- For Problem 2 and 3, try writing full code, then compare with the Leetcode discuss section or other solutions.
- Challenge yourself to come up with edge cases (e.g., all negatives, all positives, zeros in the array).
- Try to implement the circular subarray sum using a brute-force approach, then optimize it.
- Practice explaining the logic out loud or to a friend — it helps!
- If you get stuck, pause, review the patterns, and try again. That's how you build real understanding!

Happy coding!