Back To Course Home

Grokking the Coding Interview: Patterns for Coding Questions

14% completed

Search Course

Triplet Sum Close to Target (medium)
Triplets with Smaller Sum (medium)
Subarrays with Product Less than a Target (medium)
Dutch National Flag Problem (medium)
Problem Challenge 1
Solution Review: Problem Challenge 1

# Subarrays with Product Less than a Target (medium)

We'll cover the following



## Problem Statement

Given an array with positive numbers and a positive target number, find all of its contiguous subarrays whose **product is less than the target number**.

**Example 1:**

```
Input: [2, 5, 3, 10], target=30
Output: [2], [5], [2, 5], [3], [5, 3], [10]
Explanation: There are six contiguous subarrays whose product is less than the target.
```

**Example 2:**

```
Input: [8, 2, 6, 5], target=50
Output: [8], [2], [8, 2], [6], [2, 6], [5], [6, 5]
Explanation: There are seven contiguous subarrays whose product is less than the target.
```

## Try it yourself

Try solving this question here:

Java | Python3 | JS | C++

```
import java.util.*;

class SubarrayProductLessThanK {

  public static List<List<Integer>> findSubarrays(int[] arr, int target) {
    List<List<Integer>> subarrays = new ArrayList<>();
    // TODO: Write your code here
    return subarrays;
  }
}
```

Test | Save | Reset

## Solution

This problem follows the **Sliding Window** and the **Two Pointers** pattern and shares similarities with Triplets with Smaller Sum with two differences:

1. In this problem, the input array is not sorted.
2. Instead of finding triplets with sum less than a target, we need to find all subarrays having a product less than the target.

The implementation will be quite similar to Triplets with Smaller Sum.

### Code

Here is what our algorithm will look like:

Java | Python3 | C++ | JS

```
import java.util.*;

class SubarrayProductLessThanK {

  public static List<List<Integer>> findSubarrays(int[] arr, int target) {
    List<List<Integer>> result = new ArrayList<>();
    double product = 1;
    int left = 0;
    for (int right = 0; right < arr.length; right++) {
      product *= arr[right];
      while (product >= target && left < arr.length)
        product /= arr[left++];
      // since the product of all numbers from left to right is less than the target therefore,
      // all subarrays from left to right will have a product less than the target too; to avoid
      // duplicates, we will start with a subarray containing only arr[right] and then extend it
      List<Integer> tempList = new LinkedList<>();
      for (int i = right; i >= left; i--) {
        tempList.add(0, arr[i]);
        result.add(new ArrayList<>(tempList));
      }
    }
    return result;
  }

  public static void main(String[] args) {
    System.out.println(SubarrayProductLessThanK.findSubarrays(new int[] { 2, 5, 3, 10 }, 30));
    System.out.println(SubarrayProductLessThanK.findSubarrays(new int[] { 8, 2, 6, 5 }, 50));
  }
}
```

Run | Save | Reset

### Time complexity

The main `for-loop` managing the sliding window takes $O(N)$ but creating subarrays can take up to $O(N^2)$ in the worst case. Therefore overall, our algorithm will take $O(N^3)$.

### Space complexity

Ignoring the space required for the output list, the algorithm runs in $O(N)$ space which is used for the temp list.

Can you try estimating how much space will be required for the output list?

Show Hint

It is not all the Combinations of all elements of the array!

For an array with distinct elements, finding all of its contiguous subarrays is like finding the number of ways to choose two indices, `i` and `j`, in the array such that `i <= j`.

If there are a total of `n` elements in the array, here is how we can count all the contiguous subarrays:

- When `i = 0`, `j` can have any value from `0` to `n-1`, giving a total of `n` choices.
- When `i = 1`, `j` can have any value from `1` to `n-1`, giving a total of `n-1` choices.
- Similarly, when `i = 2`, `j` can have `n-2` choices.

  ...

  ...
- When `i = n-1`, `j` can only have only `1` choice.

Let's combine all the choices:

```
n + (n-1) + (n-2) + ... 3 + 2 + 1
```

Which gives us a total of: $n * (n+1)/2$

So, at most, we need space for $O(n^2)$ output lists. At worst, each subarray can take $O(n)$ space, so overall, our algorithm's space complexity will be $O(n^3)$.