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++

```java
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#

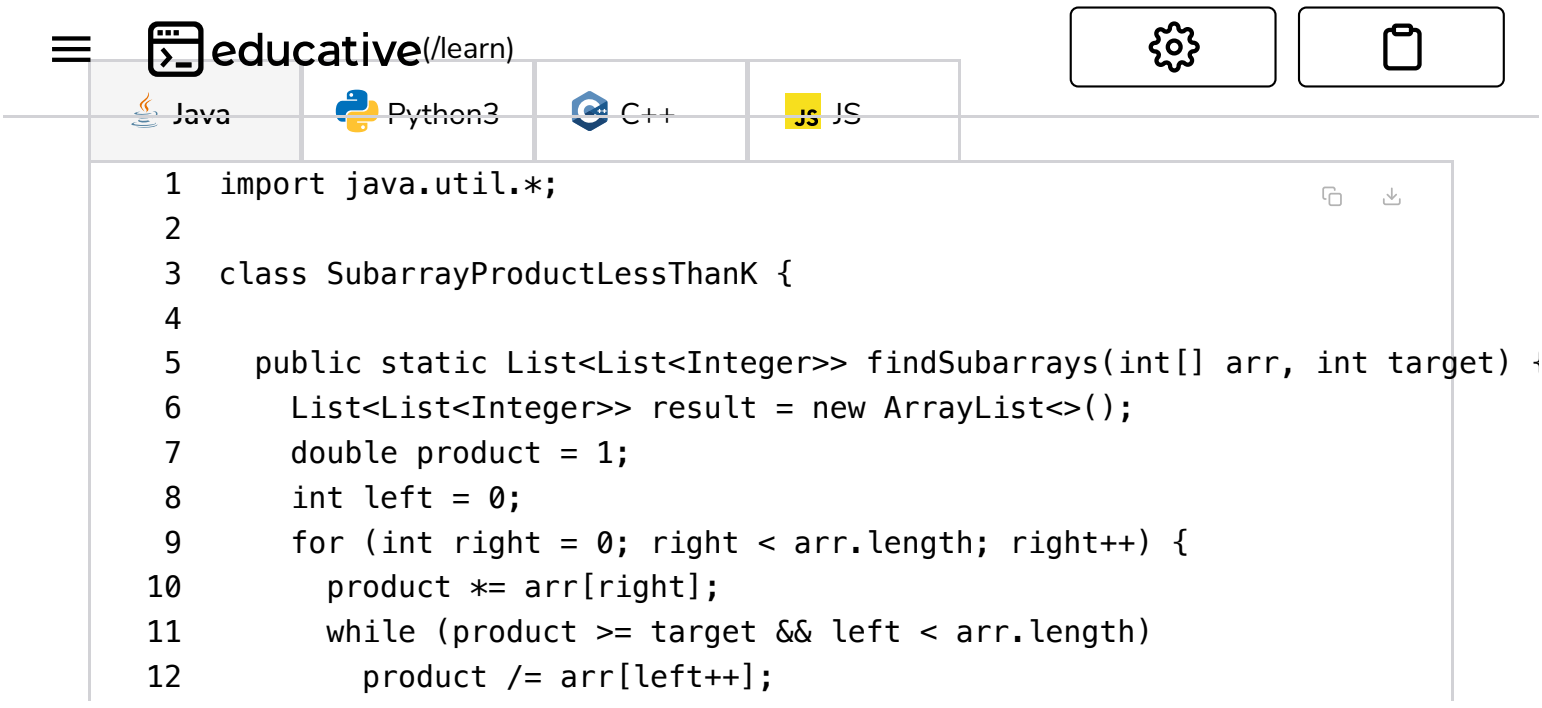
This problem follows the **Sliding Window** and the **Two Pointers** pattern and shares similarities with Triplets with Smaller Sum (https://www.educative.io/collection/page/5668639101419520/5671464854355968/5554621957275648/) with two differences:

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 (https://www.educative.io/collection/page/5668639101419520/5671464854355968/5554621957275648/).

Code#

Here is what our algorithm will look like:




```java
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++];
```

```
13        // since the product of all numbers from left to right is less than
14        // all subarrays from left to right will have a product less than th
15        // duplicates, we will start with a subarray containing only arr[rig
16        List<Integer> tempList = new LinkedList<>();
17        for (int i = right; i >= left; i--) {
18          tempList.add(0, arr[i]);
19          result.add(new ArrayList<>(tempList));
20        }
21      }
22      return result;
23    }
24
25    public static void main(String[] args) {
26      System.out.println(SubarrayProductLessThanK.findSubarrays(new int[] {
27      System.out.println(SubarrayProductLessThanK.findSubarrays(new int[] {
28    }
```

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?

Hide Hint

The worst-case will happen when every subarray has a product less than the target!

So the question will be, how many contiguous subarrays an array can have?

It is definitely not all Permutations of the given array; is it all Combinations of the given array?

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 $O(n^3)$.

← Back

Next →

Triplets with Smaller Sum (medium)

Dutch National Flag Problem (medium)

Mark as Completed