2892. Minimizing Array After Replacing Pairs With Their Product

Medium Greedy Array Dynamic Programming

Problem Description

In this problem, we have an integer array `nums` and an integer `k`. We have the option to perform an operation on the array multiple times, where each operation involves selecting two adjacent elements, `x` and `y`, and if their product `x * y` is less than or equal to `k`, we can merge them into a single element with the value `x * y`. Our goal is to find the minimum possible length of the array after performing any number of these operations.

To illustrate, consider the array `[1, 2, 3, 4]` with `k = 5`. We could merge `1` and `2` to get `[2, 3, 4]` because `1 * 2 = 2` which is less than `k`. However, we could not merge `2` and `3` in the resulting array because `2 * 3 = 6` which is greater than `k`. Thus, the operation can only be performed when the product of the two chosen adjacent numbers is less than or equal to `k`.

The problem asks us to compute the minimum length of the array possible by applying this operation optimally, which means merging whenever we can under the given constraint, thus potentially reducing the length of the array as much as possible.

Intuition

The intuition behind the solution is to apply a greedy strategy. We start from the beginning of the array and try to merge elements wherever possible. This greedy approach works because merging earlier in the array can only provide more possibilities for further merging later - there is no scenario where not merging at the earliest opportunity can lead to a shorter array at the end.

Here's the step-by-step thinking process:

1. We keep track of the current product of merged elements, starting with the first element.
2. We then iterate over the remaining elements of the array one by one.
3. If the current element is `0`, we know that we can merge the entire array into one element with value `0`, thus directly returning `1` as the smallest possible length.
4. If the current element `x` and the tracked product `y` have a product `x * y` that is less than or equal to `k`, we merge `x` with `y`, and the product becomes `y = x * y`.
5. If `x * y` is greater than `k`, we can't merge `x` with `y`. So, we must start a new product sequence beginning with `x`, and we increment the answer count `ans` by `1`, signaling an increase in the final array length.
6. We finalize the overall smallest possible length with the accumulated count `ans` at the end of the array iteration.

This approach guarantees that we consider each element for merging without skipping any potential opportunity and ensure the minimum length of the array.

Solution Approach

The algorithm follows a simple yet efficient approach that works well due to the nature of the problem which allows for a greedy strategy. There is no need for advanced data structures or complex patterns. The implementation uses basic variables to keep track of the state as we iterate through the `nums` array.

Here is a breakdown of the solution code:

1. **Initialization of State Variables**: We initialize two variables - `ans` to `1` and `y` to the first element in `nums`. Variable `ans` represents the eventual length of the array after merging, and `y` keeps track of the product of the currently merged sequence of elements.
2. **Loop Through Elements**: The solution uses a `for` loop to traverse the elements of the array starting from the second element. This is because the first element has already been taken into account as the initial product in `y`.
3. **Special Case for Zero**: If an element `x` is encountered such that `x == 0`, the result is immediate. Every element in the array can be merged into one element with value `0`, and the function returns `1`.
4. **Greedy Merging**: If the product of `x` (the current element) and `y` (the product of the already merged sequence) is less than or equal to `k` (`x * y <= k`), then we merge `x` with the already merged sequence by multiplying `x` with `y` (`y *= x`).
5. **Inability to Merge**: When the product `x * y` exceeds `k`, merging is not possible according to the problem's rules. In this case, `x` is set to be the new product (`y = x`) as we start a new merge sequence. We also increment `ans` by `1` to account for the new element added to the final array.
6. **Return the Final Answer**: After the loop has terminated, `ans` holds the minimum length of the array possible after making all the mergings, and it is returned as the solution.

The entire approach can be considered as an application of the greedy paradigm because at each step, it makes the local optimum choice to merge if it's possible, which ultimately leads to a globally optimal solution of the smallest possible length of the final array.

Here's the python code using this approach:

```
class Solution:
    def minArrayLength(self, nums: List[int], k: int) -> int:
        ans, y = 1, nums[0]  # step 1
        for x in nums[1:]:   # step 2
            if x == 0:       # step 3
                return 1
            if x * y <= k:   # step 4
                y *= x
            else:            # step 5
                y = x
                ans += 1
        return ans           # step 6
```

This solution is linear, as it passes through the array exactly once, thus making the time complexity `O(n)`, where `n` is the number of elements in the array. The space complexity is `O(1)` since only a constant amount of extra space is used beyond the input array.

Example Walkthrough

Let's consider a small example to illustrate the solution approach.

Suppose we have the array `nums = [1, 2, 3, 2]` and `k = 3`.

We initialize our answer `ans` to `1` because we always have at least one element, and `y` (the product of merged elements) to `nums[0]`, in this case `1`.

Now, let's walk through the array:

- We consider the next element `2`. The product of `y` (currently `1`) and `2` is `2`, which is less than or equal to `k`. Therefore, we can merge these two elements. After merging, `y` becomes `2`.
- Moving to the next element, which is `3`, the product of `y` (now `2`) and `3` is `6`, which is greater than `k`. We cannot merge them, so we designate `3` as the start of a new product sequence. `y` is updated to `3` and we increment `ans` to `2`.
- Finally, we see another `2`. The product of `y` (`3`) and `2` is `6` again, exceeding `k`. This means we start another new sequence with this `2`, update `y` to `2`, and increment `ans` to `3`.

After iterating through the array, we find that the minimum possible length of the array after performing the operations is `3`.

In this example, our step-by-step process helped us to approach the problem systematically and apply the operation optimally to obtain the smallest possible length of the array.

Solution Implementation

Python

```
from typing import List

class Solution:
    def minArrayLength(self, nums: List[int], threshold: int) -> int:
        # Initialize the minimum length to 1 and the running product to the first element
        min_length, running_product = 1, nums[0]

        # Loop through the elements of the array, starting from the second element
        for num in nums[1:]:
            # If an element is 0, the minimum length is always 1, as 0 multiplied by any number is 0
            # which is always less than or equal to threshold
            if num == 0:
                return 1
            # If the current number multiplied by running product
            # is less than or equal to the threshold, multiply the running product by the current number
            if num * running_product <= threshold:
                running_product *= num
            # If the current running product exceeds the threshold,
            # reset the running product to the current number and increment the minimum length
            else:
                running_product = num
                min_length += 1

        # Return the computed minimum length
        return min_length
```

Java

```
class Solution {

    // Method to find the minimum array length of continuous elements that when multiplied do not exceed the given threshold 'k'.
    public int minArrayLength(int[] nums, int k) {
        int minLength = 1; // Initialize the minimum length to 1.
        long product = nums[0]; // Initialize the product with the first element.

        // Loop through the array starting from the second element.
        for (int i = 1; i < nums.length; ++i) {
            int currentElement = nums[i]; // Store the current array element.

            // If the current element is 0, we can always return 1 as zero times any number is always less than or equal to k.
            if (currentElement == 0) {
                return 1;
            }

            // If multiplying the current product with the current element is still less than or equal to 'k',
            // we can increase the product by multiplying it with the current element.
            if (product * currentElement <= k) {
                product *= currentElement;
            } else {
                // If the product exceeds 'k', we start a new subsequence of numbers starting with the current element.
                product = currentElement;
                ++minLength; // Increment the counter for the minimum length as we begin a new subsequence.
            }
        }

        // Return the minimum length of the subsequence of continuous elements.
        return minLength;
    }
}
```

C++

```
#include <vector> // Include vector header for using std::vector
using std::vector;

class Solution {
public:
    // Determines the minimum contiguous subarray length
    // where the product of elements is less than or equal to k.
    int minArrayLength(vector<int>& nums, int k) {
        int minLength = 1; // The minimum length starts at 1.
        long long currentProd = nums[0]; // Current product starts with the first element.

        // Iterate over the array starting from the second element.
        for (int i = 1; i < nums.size(); ++i) {
            int currentNum = nums[i]; // Hold the current number in the array.

            // If the current number is 0, the answer must be 1,
            // since the problem likely would want to find a non-empty product
            // that is less than or equal to k (0 is trivially less than or equal to any k).
            if (currentNum == 0) {
                return 1;
            }

            // If the product of the current number and current product is less than or equal to k,
            // multiply the current product with the current number.
            if (currentNum * currentProd <= k) {
                currentProd *= currentNum;
            } else {
                // If the current product exceeds k, start a new subarray from this number,
                // and increment the minimum length.
                currentProd = currentNum;
                ++minLength;
            }
        }

        return minLength; // Return the minimum subarray length.
    }
};
```

TypeScript

```
function minArrayLength(nums: number[], k: number): number {
    // Initialize the answer to 1, assuming at minimum we'll have one subarray
    // Start the product with the first element of nums array
    let [subArrayCount, product] = [1, nums[0]];

    // Iterate through the nums array starting from the second element
    for (const currentNumber of nums.slice(1)) {
        // If the current number is 0, the minimum array length is immediately 1
        if (currentNumber === 0) {
            return 1;
        }
        // If multiplying the current product with the current number is less than or equal to k,
        // update the product by multiplying it with the current number
        if (currentNumber * product <= k) {
            product *= currentNumber;
        } else {
            // If the product exceeds k, reset the product to the current number and increment subArrayCount
            product = currentNumber;
            ++subArrayCount;
        }
    }
    // Return the minimum number of subarrays found that satisfies the condition
    return subArrayCount;
}
```

```
from typing import List

class Solution:
    def minArrayLength(self, nums: List[int], threshold: int) -> int:
        # Initialize the minimum length to 1 and the running product to the first element
        min_length, running_product = 1, nums[0]

        # Loop through the elements of the array, starting from the second element
        for num in nums[1:]:
            # If an element is 0, the minimum length is always 1, as 0 multiplied by any number is 0
            # which is always less than or equal to threshold
            if num == 0:
                return 1
            # If the current number multiplied by running product
            # is less than or equal to the threshold, multiply the running product by the current number
            if num * running_product <= threshold:
                running_product *= num
            # If the current running product exceeds the threshold,
            # reset the running product to the current number and increment the minimum length
            else:
                running_product = num
                min_length += 1

        # Return the computed minimum length
        return min_length
```

Time and Space Complexity

The time complexity of the given code is `O(n)`, where `n` is the length of the array `nums`. This is because the code iterates through the `nums` array exactly once, with each iteration involving a constant amount of work, such as arithmetic operations and conditional checks.

The space complexity of the code is `O(1)`. The code only uses a fixed amount of extra space for the variables `ans`, `y`, and `x`, regardless of the size of the input array. Thus, the amount of extra memory used does not scale with the size of the input, making the space complexity constant.