1352. Product of the Last K Numbers

Medium Design Queue Array Math Data Stream

Leetcode Link

Problem Description

The given problem presents us with a scenario where we need to continuously monitor a stream of integers and have the ability to compute the product of the last `k` numbers in the stream at any point. We are required to implement a `ProductOfNumbers` class that can:

1. Be initialized without any numbers in the stream.
2. Support appending numbers to the end of the stream.
3. Calculate the product of the last `k` numbers in the stream.

The constraints ensure that we will not face integer overflow issues since the product will always fit into a 32-bit integer, and the numbers added will within the range 0 to 100. The problem statement specifies that there will be a maximum of 40,000 calls to both `add` and `getProduct` methods which hints at the necessity for an efficient solution to handle large input streams.

Intuition

The straightforward solution would be to multiply the last `k` numbers in the stream every time `getProduct` is called. However, this would not be efficient since the same products could be recalculated many times for different calls to `getProduct`.

A more efficient approach is to maintain a prefix product for the stream. The key insight here is that the product of any contiguous segment of the stream can be quickly calculated using the division of two prefix products.

To realize this solution, we maintain a list initialized with a single element, `1`, which signifies an empty product (since the product of zero numbers is `1`). Whenever a new number is added to the stream that is not zero, we append the product of this number and the last element in the list (which represents the product of all numbers up to that point). When zero is added to the stream, it invalidates all previous products, so we reset the list to its initial state containing just the one element.

For retrieving the product of the last `k` numbers, we simply find the ratio of the last element in the list to the element that is `k` indices before the last element. We need to handle the case where a zero has been added within the last `k` integers, in which case the product should be zero. We check this by comparing the length of the list with `k`; if the list is not long enough after adding zeros, it means there was a zero in the last `k` elements, and we return 0.

Solution Approach

To implement the solution, we utilize the concept of prefix products. Here is a breakdown of this approach and how we use it:

Data Structure:

- We employ a list (`self.s`) as a primary data structure, which starts with one element `[1]` representing the product of zero numbers.

Adding a Number (`add` method):

- When a non-zero number is added to the stream, we multiply the most recently added number (the last element in `self.s`) by the new number and append the result to the list. This maintains the prefix product for the entire stream.
- In case the number `0` is added, we reset `self.s` to `[1]` because the product of any range including zero is zero, and thus all previous products become irrelevant.

Getting Product (`getProduct` method):

- To get the product of the last `k` numbers, we check if our list `self.s` has enough elements to cover `k` numbers without encountering a zero. We do this by checking if `len(self.s) <= k`.
- If the list is too short, this means a zero was added within the last `k` elements, and hence we return `0`.
- If the list is long enough, we use the property that the product of the last `k` numbers can be calculated by taking the `k+1`-th most recent element from the end of the list and dividing the last element of the list by this `k+1`-th element. In code, this is `self.s[-1] // self.s[-k - 1]`.

By maintaining a running product, the algorithm allows for constant-time retrieval of the last `k` products, thereby making the operation efficient. It is assumed that all operations of multiplication and division with respect to the product retrieval will always result in integer values fitting in a 32-bit integer variable, as per the constraints of the problem.

Example:

Consider `self.s = [1, 3, 5, 15]` (implicit products of 1, 3, 3*2, 2*5*1, if we want to get the product of the last 2 numbers:

- We would use `self.s[-1] // self.s[-2 - 1]`, which translates to `15 // 3 = 5`, the product of the last 2 numbers, 2 and 5.

This elegant solution avoids the need for repeated multiplication and directly uses the property of division and prefix products to provide the answers efficiently.

Example Walkthrough

Let's go through a small example to illustrate the solution approach:

1. Initialize the `ProductOfNumbers` class. The internal list `self.s` starts with `[1]`, indicating that the product of zero numbers is 1.

    - `self.s = [1]`

2. Call the add method with the number 2. Since 2 is not zero, multiply the last element in `self.s` by 2 which is `1*2=2` and append it to `self.s`.

    - `self.s = [1, 2]`

3. Add another number, say 5, through the add method. Multiply the last element in `self.s` by 5 which is `2*5=10` and append it.

    - `self.s = [1, 2, 10]`

4. Add number 0. Adding 0 resets `self.s` because any product including zero is zero.

    - `self.s = [1]`

5. Now, add the number 4 to the stream. We append `1*4=4` to `self.s`.

    - `self.s = [1, 4]`

6. Next, we add 6 to the stream. We append `4*6=24` to `self.s`.

    - `self.s = [1, 4, 24]`

7. Assume at this point we want to retrieve the product of the last 2 numbers. Since `len(self.s) = 3` is greater than `k = 2`, we proceed with the division: `self.s[-1] // self.s[-2 - 1]` which translates to `24 // 4 = 6`. Thus, `getProduct(2)` returns 6, which is the product of 4 and 6.

    - `getProduct(2) = 6`

8. However, if we want to retrieve the product of the last 4 numbers, since adding zero has reset the list and we only have 2 numbers after that (4 and 6), we return 0 because it's not possible to have a product for the last 4 numbers.

    - `getProduct(4) = 0`

Through this example, you can see how efficient the solution is. It prevents recalculating products with every call to `getProduct`, thanks to the power of prefix products and the efficient handling of zeros in the stream.

Python Solution

```python
class ProductOfNumbers:
    def __init__(self):
        # Initialize a list with one element '1' to represent the prefix product
        self.prefix_products = [1]

    def add(self, num: int) -> None:
        # Add the number to the sequence of numbers
        if num == 0:
            # If the number is zero, reset the list since any subsequent product will be zero
            self.prefix_products = [1]
        else:
            # Calculate the new product by multiplying the last product in the list with the new number
            self.prefix_products.append(self.prefix_products[-1] * num)

    def getProduct(self, k: int) -> int:
        # Returns the product of the last k numbers in the sequence
        if len(self.prefix_products) <= k:
            # If k is greater than or equal to the number of elements, the product is zero
            return 0
        else:
            # Otherwise, return the product of the last k numbers
            # By dividing the last prefix product by the product at (n-k-1)th position
            return self.prefix_products[-1] // self.prefix_products[-k - 1]

# Usage
# Initialize the object
obj = ProductOfNumbers()

# Add numbers with the add method
obj.add(3)  # Sequence is now: [3]
obj.add(0)  # Sequence is now: [1] because a zero resets everything
obj.add(2)  # Sequence is now: [1, 2]
obj.add(5)  # Sequence is now: [1, 2, 10]
obj.add(4)  # Sequence is now: [1, 2, 5, 4]

# Get the product of the last k numbers
print(obj.getProduct(2))  # Output: 20, since the product of last 2 numbers (5, 4) is 20
print(obj.getProduct(3))  # Output: 40, since the product of the last 3 numbers (2, 5, 4) is 40
print(obj.getProduct(4))  # Output: 0, since k>= the number of elements after the last reset (when 0 was added)
```

Java Solution

```java
import java.util.ArrayList;
import java.util.List;

class ProductOfNumbers {
    private List<Integer> prefixProducts;  // List to store the prefix products.

    // Constructor initializes the data structure with a single '1'.
    public ProductOfNumbers() {
        prefixProducts = new ArrayList<>();
        prefixProducts.add(1);
    }

    // Adds a number to the list, handling multiplication and the case when zero is added.
    public void add(int num) {
        if (num == 0) {
            // If the number is zero, the product of all numbers will be zero.
            // Clear the list and start anew with a single '1'.
            prefixProducts.clear();
            prefixProducts.add(1);
        } else {
            // Multiply the last element by 'num' to get the new product and add it to the list.
            int lastProduct = prefixProducts.get(prefixProducts.size() - 1);
            prefixProducts.add(lastProduct * num);
        }
    }

    // Returns the product of the last 'k' numbers added to the list.
    public int getProduct(int k) {
        int n = prefixProducts.size();
        // If 'k' is greater than or equal to the size, all the elements include a zero, return 0.
        if (n <= k) {
            return 0;
        } else {
            // The product of the last 'k' numbers is the last element divided by the (n - k -1)th element.
            return prefixProducts.get(n - 1) / prefixProducts.get(n - k - 1);
        }
    }
}

// This class can be used as follows:
// ProductOfNumbers obj = new ProductOfNumbers();
// obj.add(num);
// int product = obj.getProduct(k);
```

C++ Solution

```cpp
#include <vector>

class ProductOfNumbers {
public:
    ProductOfNumbers() {
        // Initialize with 1, to handle the product calculation
        prefix_products_.push_back(1);
    }

    void add(int num) {
        // If number is zero, reset the products list as any subsequent product will be 0
        if (num == 0) {
            prefix_products_.clear();
            prefix_products_.push_back(1);
        } else {
            // Otherwise, calculate the new product and add to the list
            int new_product = prefix_products_.back() * num;
            prefix_products_.push_back(new_product);
        }
    }

    int getProduct(int k) {
        int size = prefix_products_.size();
        // If k is greater or equal to the size of the list, return 0 since there are not enough elements
        if (size <= k) {
            return 0;
        } else {
            // Otherwise, return the product of the last k numbers
            int product = prefix_products_.back() / prefix_products_[size - k - 1];
            return product;
        }
    }

private:
    // Stores the prefix products; prefix_products_[i] is the product of all numbers up to index i
    std::vector<int> prefix_products_;
};

/**
 * Your ProductOfNumbers object will be instantiated and called as such:
 * ProductOfNumbers* obj = new ProductOfNumbers();
 * obj->add(num);
 * int param_2 = obj->getProduct(k);
 */
```

Typescript Solution

```typescript
// A global variable to store the prefix products,
// where prefixProducts[i] is the product of all numbers up to index i.
let prefixProducts: number[] = [1];

// Adds a number to the list of products.
function add(num: number): void {
    if (num === 0) {
        // If number is zero, reset the prefixProducts array as any subsequent product will be 0.
        prefixProducts = [1];
    } else {
        // Calculate the new product based on the last element and add it to the prefixProducts array.
        const newProduct = prefixProducts[prefixProducts.length - 1] * num;
        prefixProducts.push(newProduct);
    }
}

// Returns the product of the last k numbers added.
function getProduct(k: number): number {
    const size = prefixProducts.length;
    if (size <= k) {
        // If k is greater or equal to the size of the array, return 0 since there are not enough elements.
        return 0;
    } else {
        // Calculate the product of the last k numbers by dividing the last product
        // by the product at the index 'size - k - 1'.
        const product = prefixProducts[size - 1] / prefixProducts[size - k - 1];
        return product;
    }
}
```

Time and Space Complexity

Time Complexity

- **Initialization (`__init__` method):** The time complexity is $O(1)$ as it only sets the initial list with one element.
- **Adding a number (`add` method):**
    - If the number is not zero, we multiply the last element in the list by the new number and append the result. This has a time complexity of $O(1)$.
    - If the number is zero, we reset the list which also has a time complexity of $O(1)$.
- **Getting the product (`getProduct` method):**
    - If we need to retrieve the product of the last k numbers, we do so by dividing the last number in the list by the `len(s) - k - 1`th number, given that `len(s)` is greater than k. The division itself is $O(1)$, but accessing the list elements is also $O(1)$ since list accesses by index in Python are constant time.
    - Returning 0 if `len(s) <= k` is also $O(1)$.

Overall, for each method, the time complexity is $O(1)$.

Space Complexity

- The space complexity is primarily due to the list `s` that stores the product of all numbers up to the current position.
- **Space Complexity for the `s` list:** This grows linearly with the number of elements added, barring when zero resets the list 's'. Therefore, the space complexity is $O(n)$, where n is the number of add operations before the last reset (adding 0).
- **Space Complexity for `add` and `getProduct` methods:** Aside from the storage in the list `s`, no additional space is used, so the space complexity for each operation is $O(1)$.

Overall, the space complexity of the `ProductOfNumbers` class is $O(n)$, where n is the size of the list `s` at a given time, which equals the number of elements added since the last addition of zero.