396. Rotate Function

Medium Array Math Dynamic Programming

Problem Description

In this problem, you're given an array of integers called `nums`. This array has `n` elements. The task is to perform rotations on this array and compute a certain value, called the "rotation function `F`", for each rotated version of the array.

A rotation consists of shifting every element of the array to the right by one position, and the last element is moved to the first position. This is a clockwise rotation. If `nums` is rotated by `k` positions clockwise, the resulting array is named `arr_k`.

The rotation function `F` for a rotation `k` is defined as follows:

```
F(k) = 0 * arr_k[0] + 1 * arr_k[1] + ... + (n - 1) * arr_k[n - 1].
```

In other words, each element of the rotated array `arr_k` is multiplied by its index, and the results of these multiplications are summed to give `F(k)`.

The objective is to find out which rotation (from `F(0)` to `F(n-1)`) yields the highest value of `F(k)` and to return this maximum value.

Intuition

The intuition behind the solution comes from observing that the rotation function `F` is closely related to the sum of the array and the previous value of `F`. Specifically, we can derive `F(k)` from `F(k-1)` by adding the sum of the array elements and then subtracting the array size multiplied by the element that just got rotated to the beginning of the array (as this element's coefficient in the rotation function decreases by `n`).

Here's the thinking process:

1. First, compute the initial value of `F(0)` by multiplying each index by its corresponding value in the unrotated array. This gives us the starting point for computing subsequent values of `F(k)`.
2. Keep track of the total sum of the array, as this will be used in computing `F(k)` for `k > 0`.
3. Iterate through the array from `k = 1` to `k = n-1`. In each iteration, calculate `F(k)` based on the previous `F(k-1)` by adding the total sum of the array and subtracting `n` times the element that was at the end of the array in the previous rotation.
4. During each iteration, update the maximum value of `F(k)` found so far.

By the end of the iteration, we have considered all possible rotations and have kept track of the maximum `F(k)` value, which the function returns as the answer.

```
The provided Python solution implements this thinking process:

```python
class Solution:
    def maxRotateFunction(self, nums: List[int]) -> int:
        # Initial calculation of F(0)
        f = sum(i * v for i, v in enumerate(nums))
        # Total sum of the array
        n, s = len(nums), sum(nums)
        # Starting with the maximum as the initial F(0)
        ans = f
        # Looping through the array for subsequent Fs
        for i in range(1, n):
            # Update F(k) based on previous value F(k-1), total sum, and subtracting the last element's contribution
            f = f + s - n * nums[n - i]
            # Update the answer with the max value found
            ans = max(ans, f)
        return ans
```

Learn more about [Math](/problems/math-basics) and [Dynamic Programming](/problems/dynamic_programming_intro) patt

Solution Approach

The solution employs a straightforward approach without any complex algorithms or data structures. It hinges on the mathematical relationship between the values of `F(k)` after each rotation. Let's walk through the steps, aligning them with the provided Python code snippet:

1. **Initial Value of `F(0)`:** Calculating the initial value of `F(0)` involves using a simple loop or in this case, a generator expression, which multiplies each element by its index and sums up these products.

   ```
   f = sum(i * v for i, v in enumerate(nums))
   ```

   The variables `n` and `s` are initialized to store the length of the array and the sum of its elements respectively. This is done to avoid recalculating these values in each iteration of the loop, which follows next.

   ```
   n, s = len(nums), sum(nums)
   ```

2. **Initializing the Maximum Value:** Before beginning the loop, we record the initial value of `F(0)` in the variable `ans` as this might be the maximum value.

   ```
   ans = f
   ```

3. **Iterative Calculation of Subsequent `F(k)`:** We know that the subsequent value `F(k)` can be derived from `F(k-1)` by adding the sum of the array `s` to it and subtracting `n` times the last element of the array before it got rotated to the front, which is `nums[n - i]`.

   The loop runs from `1` to `n - 1` representing all possible `k` rotations (starting from 1 because we have already calculated `F(0)`).

   ```
   for i in range(1, n):
       f = f + s - n * nums[n - i]
       ans = max(ans, f)
   ```

   - We adjust `f` to find the current `F(k)`. The `s` is the total sum of the array, and we subtract the value that would have been added if there had been no rotation multiplied by `n`, which is `f + sum(f - i)`. We're subtracting it because its index in the function `f` effectively decreases by `n` due to the rotation.
   - We update `ans` with the maximum value found so far by comparing it with the newly computed `F(k)`.
4. **Returning the Result**: Once all rotations have been considered, the variable `ans` holds the maximum value found, which is then returned.

In terms of complexity, the time complexity of this solution is `O(n)` since it iterates through the array once after calculating the initial `F(0)`. The space complexity is `O(1)` since it uses a fixed number of variables and doesn't allocate any additional data structures proportionate to the size of the input.

Example Walkthrough

Let's consider a small array `nums = [4, 3, 2, 6]` to illustrate the solution approach.

1. **Initial Value of `F(0)`:** We calculate `F(0)` by multiplying each element by its index and summing them up:

   ```
   F(0) = 0*4 + 1*3 + 2*2 + 3*6 = 0 + 3 + 4 + 18 = 25
   ```

   In the code, this is done using:

   ```
   f = sum(i * v for i, v in enumerate(nums))
   ```

   We also compute the total sum of the array `s = 4 + 3 + 2 + 6 = 15` and store the number of elements `n = 4`. These are calculated once to be used in subsequent rotation calculations:

   ```
   n, s = len(nums), sum(nums)
   ```

2. **Initializing the Maximum Value:** We start by considering `F(0)` as the potential maximum.

   ```
   ans = f  # ans = 25 initially
   ```

3. **Iterative Calculation of Subsequent `F(k)`:** Now we calculate `F(1)` based on `F(0)`:

   ```
   F(1) = F(0) + s - n * nums[n - 1]

   F(1) = 25 + 15 - 4*6 = 25 + 15 - 24 = 16
   ```

   And we check if this is greater than the current maximum `ans`:

   ```
   ans = max(ans, f)  # ans remains 25 as 16 < 25
   ```

   Next, we calculate `F(2)`:

   ```
   F(2) = F(1) + s - n * nums[n - 2] = 16 + 15 - 4*2 = 16 + 15 - 8 = 23
   ```

   Again, we update if it's greater than `ans`:

   ```
   ans = max(ans, f)  # ans remains 25 as 23 < 25
   ```

   Finally, calculate `F(3)`:

   ```
   F(3) = F(2) + s - n * nums[n - 3] = 23 + 15 - 4*3 = 23 + 15 - 12 = 26
   ```

   Now `F(3)` is greater than the current maximum `ans`, so we update `ans`:

   ```
   ans = max(ans, f)  # ans is now updated to 26
   ```

4. **Returning the Result**: We have considered all possible rotations (`F(0)` through `F(3)`) and the maximum value of `F(k)` is 26, achieved at `k = 3`. This is returned as the result.

   ```
   return ans  # returns 26
   ```

Putting all this into action with our small example array `nums = [4, 3, 2, 6]`, we find the rotation function that yields the highest value is `F(3)`, and the maximum value returned is 26.

Solution Implementation

Python

```python
class Solution:
    def max_rotate_function(self, nums: List[int]) -> int:
        # Calculate the initial value of the function F
        total_function_value = sum(index * value for index, value in enumerate(nums))

        # Get the number of elements and the sum of all elements in nums
        num_elements = len(nums)
        sum_of_elements = sum(nums)

        # Initialize ans with the initial total function_value
        max_function_value = total_function_value

        # Iterate through the array to find the maximal F value after rotations
        for i in range(1, num_elements):
            # Update the function value by moving the element towards the right and update the function value
            # This is achieved by adding the sum of all elements minus the last element value
            # that is 'rotated' to the 'front' of the array times the number of elements
            total_function_value += sum_of_elements - num_elements * nums[num_elements - i]

            # Update max function value if the new total function value is greater
            max_function_value = max(max_function_value, total_function_value)

        # Return the maximum value found
        return max_function_value
```

Java

```java
class Solution {
    public int maxRotateFunction(int[] nums) {
        int firstComputation = 0;  // This will store the initial computation of the function F(0)
        int sumOfAllNumbers = 0;   // This holds the sum of all the elements in the array
        int n = nums.length;       // Total number of elements in the array
        // Calculate the initial value of F(0) and sum of all numbers in the array
        for (int i = 0; i < n; ++i) {
            firstComputation += i * nums[i];
            sumOfAllNumbers += nums[i];
        }
        int maxResult = firstComputation;  // Initialize maxResult with the first computation of the function

        // Compute the maximum value of F(i) by iterating through the possible rotations
        for (int i = 1; i < n; ++i) {
            // Compute the next value of F based on the previous value (F = F + sum - n * nums[n - i])
            firstComputation = firstComputation + sumOfAllNumbers - n * nums[n - i];
            // Update maxResult if the new computed value is greater than the current maxResult
            maxResult = Math.max(maxResult, firstComputation);
        }
        // Return the maximum result found
        return maxResult;
    }
}
```

C++

```cpp
#include <vector>
#include <algorithm> // For std::max

class Solution {
public:
    int maxRotateFunction(std::vector<int>& nums) {
        int currentFunctionValue = 0; // Initialize sum of i*nums[i]
        int sumOfElements = 0; // Initialize sum of nums[i] for all i
        int numberOfElements = nums.size(); // Number of elements in the array

        // Calculate initial configuration values for currentFunctionValue and sumOfElements
        for (int i = 0; i < numberOfElements; ++i) {
            currentFunctionValue += i * nums[i];
            sumOfElements += nums[i];
        }

        int maxFunctionValue = currentFunctionValue; // Initialize the maximal value of F with current configuration

        // Iterate over the array to find the maximal rotation function value
        for (int i = 1; i < numberOfElements; ++i) {
            // Compute the next value of F by adding the sumOfElements and subtracting
            // the last element multiplied by the number of elements
            currentFunctionValue = currentFunctionValue + sumOfElements - numberOfElements * nums[numberOfElements - i];

            // Update the maxFunctionValue with the maximum of current and the newly computed value
            maxFunctionValue = std::max(maxFunctionValue, currentFunctionValue);
        }

        // Return the maximum value found for the rotation function
        return maxFunctionValue;
    }
};
```

TypeScript

```typescript
function maxRotateFunction(nums: number[]): number {
    const numElements = nums.length; // The number of elements in the input array

    // Calculate the sum of all numbers in the array
    const totalSum = nums.reduce((accumulator, value) => accumulator + value, 0);

    // Initialization the function result using the formula F(0) = 0 * nums[0] + 1 * nums[1] + ... + (n-1) * nums[n-1]
    let maxFunctionValue = nums.reduce((accumulator, value, index) => accumulator + value * index, 0);

    // The previous state's function value, starting with F(0)
    let previousFunctionValue = maxFunctionValue;

    // Iterate through the array to find the maximum function value after each rotation
    for (let i = 1; i < numElements; ++i) {
        // Calculate the function value F(i) for the current rotation based on F(i-1), the previous rotation
        previousFunctionValue = previousFunctionValue - totalSum - nums[i - 1] * numElements + nums[i - 1];
        // Update the maximum function value found so far
        maxFunctionValue = Math.max(maxFunctionValue, previousFunctionValue);
    }

    // Return the maximum found function value
    return maxFunctionValue;
}
```

```python
class Solution:
    def max_rotate_function(self, nums: List[int]) -> int:
        # Calculate the initial value of the function F
        total_function_value = sum(index * value for index, value in enumerate(nums))

        # Get the number of elements and the sum of all elements in nums
        num_elements = len(nums)
        sum_of_elements = sum(nums)

        # Initialize ans with the initial total function_value
        max_function_value = total_function_value

        # Iterate through the array to find the maximal F value after rotations
        for i in range(1, num_elements):
            # Update the function value by moving the element towards the right and update the function value
            # This is achieved by adding the sum of all elements minus the last element value
            # that is 'rotated' to the 'front' of the array times the number of elements
            total_function_value += sum_of_elements - num_elements * nums[num_elements - i]

            # Update max function value if the new total function value is greater
            max_function_value = max(max_function_value, total_function_value)

        # Return the maximum value found
        return max_function_value
```

Time and Space Complexity

The time complexity of the code is `O(n)`, where `n` is the length of the `nums` list. This is because there is one initial loop that goes through the numbers in `nums` to calculate the initial value of `f`, which will take `O(n)` time. After that, there is a for-loop that iterates `n-1` times, and in each iteration, it performs a constant amount of work which does not depend on `n`. Hence, the loop contributes `O(n)` to the time complexity as well.

The space complexity of the code is `O(1)`. This is because only a constant amount of extra space is used for variables `f`, `n`, `s`, and `ans`, and the input `nums` is not being copied or expanded, thus the space used does not grow with the size of the input.
```