1652. Defuse the Bomb

Easy Array Leetcode Link

Problem Description

You are tasked with decrypting a circular array of numbers (`code`) given a specific integer key (`k`). The array is circular, which means that after the last element, it loops back to the first element, and before the first element, it loops back to the last element. To decrypt the array, you need to transform each element following these rules:

- If `k` is greater than 0, replace each element with the sum of the next `k` elements.
- If `k` is less than 0, replace each element with the sum of the previous `k` elements (since `k` is negative, you're summing the `-k` elements that come before).
- If `k` equals 0, replace each element with 0.

The challenge is to perform this decryption and return the new array that represents the defused bomb's code.

Intuition

To solve this problem, we need to simulate the process of replacing each number in the circular array based on the value of `k`. Since the array is circular, we will have to handle the wrap-around when `k` is positive or negative.

The straightforward approach would involve iterating through the array for each element and then summing the next or previous `k` elements depending on the sign of `k`. However, doing this would require careful handling of the indices to avoid going out of bounds and to wrap around when needed.

A more efficient solution is to make use of prefix sums to quickly calculate the sum of any subarray. Prefix sums allow us to sum a range of elements in constant time once the prefix sums array is computed. We can do this by creating a new array that is twice the size of the original array and is a concatenation of two copies of the original array (`code + code`). This way, we can simplify the indexing for the wrap-around without having to mod the index each time.

The prefix sums are calculated using the Python function `accumulate` with an `initial=0` value. This constructs an array `s` where `s[i]` represents the sum of the first `i` elements in the extended array.

When `k` is positive, we take the sum of the subarray that starts just after the current element and extends `k` elements forward. When `k` is negative, we need to take the sum of the subarray that starts `k` elements before the current element and goes up to just before the current element.

Because we are using prefix sums, we calculate these subarray sums by subtracting the appropriate prefix sums:

- If `k > 0`, `ans[i] = s[i + k + 1] - s[i + 1]`.
- If `k < 0`, `ans[i] = s[i + n] - s[i + k + n]`.

Finally, for `k == 0`, we simply return an array filled with zeros.

Using prefix sums and the doubled array trick to handle the circular nature of the problem, we achieve a solution that is efficient and avoids complicated index wrangling.

Solution Approach

To implement the solution, we leverage a well-known algorithmic pattern: the prefix sum array. This array stores the cumulative sums of the elements in an array, allowing fast computation of the sum of any subarray. The solution involves the following steps:

1. **Create an Extended Array**: We initiate by creating an array `s` that's twice as long as the input array `code`, achieved by concatenating `code` with itself (`code + code`). This simplifies our calculations for the circular array since we won't need to manually wrap around the array's ends.

2. **Calculate Prefix Sums**: Using the `accumulate` function with an `initial=0`, we compute the prefix sums for our extended array. The result is that `s[i]` contains the sum of the elements up to, but not including, position `i` in the extended array.

3. **Initialize Answer Array**: We initialize an array `ans` of the same length as `code` with all elements set to 0. This array will store our decrypted code numbers.

4. **Handle k == 0**: If `k` is zero, we know that each element in the array simply becomes 0. Since we've already initialized `ans` to all zeros, we can return it directly without any additional computation.

5. **Compute Decrypted Values**:

   - **If `k > 0`**: For each index `i` in the original array range (from 0 to `n - 1`), we replace the element with the sum of the next `k` elements. To do this without manually calculating the sum each time, we take advantage of our prefix sums array `s`. We calculate `ans[i]` as the difference between `s[i + k + 1]` and `s[i + 1]`. This gives us the sum of the `k` elements following index `i`.
   - **If `k < 0`**: We do something similar but for the previous `k` elements. Since `k` is negative, to get `-k` previous elements, we calculate `ans[i]` as the difference between `s[i + n]` and `s[i + k + n]`. This is equivalent to summing the elements from index `i + k` (inclusive) up to `i` (exclusive) in the circular array.
6. **Return the Answer**: After computing the new values for all `i`, we return the `ans` array as the final decrypted code.

The algorithm's time complexity is `O(n)` where `n` is the length of the `code` array because each element is processed in a constant amount of time due to the precomputed prefix sums.

This solution is not only efficient but also elegant, as it reduces a problem involving potentially complex modular arithmetic and handling of circular array indices to simple array accesses and subtraction operations.

Example Walkthrough

Let's walk through a small example to illustrate the solution mentioned in the content provided.

Suppose we have a circular array `code` with elements `[5, 7, 1, 4]` and we are given an integer key `k = 3`. Here's how we would decrypt the array following the solution approach:

1. **Create an Extended Array**: We first concatenate the array `code` with itself. So the extended array `s` becomes `[5, 7, 1, 4, 5, 7, 1, 4]`.

2. **Calculate Prefix Sums**: Using the prefix sums concept, we calculate the sums as follows:

   ```
   [0, 5, 12, 13, 17, 22, 29, 30, 34]
   ```

   The array starts with `0` because initially, there's nothing to sum.

3. **Initialize Answer Array**: We prepare an answer array `ans` with the same length as `code` and initialize it with zeros: `[0, 0, 0, 0]`.

4. **Handle k == 0**: In this case, `k` is not 0, so we proceed to the next step.

5. **Compute Decrypted Values**: Since `k > 0`, we update each element of `ans` as follows:

   - For `i = 0` (corresponding to element `5`), the sum of the next `k = 3` elements is `s[0 + 3 + 1] - s[0 + 1]`, which is `s[4] - s[1]` = `17 - 5` = `12`. So, `ans[0]` becomes `12`.
   - For `i = 1` (corresponding to element `7`), the sum is `s[1 + 3 + 1] - s[1 + 1]`, which is `s[5] - s[2]` = `22 - 12` = `10`. So, `ans[1]` becomes `10`.
   - For `i = 2` (corresponding to element `1`), the sum is `s[2 + 3 + 1] - s[2 + 1]`, which is `s[6] - s[3]` = `29 - 13` = `16`. So, `ans[2]` becomes `16`.
   - For `i = 3` (corresponding to element `4`), the sum is `s[3 + 3 + 1] - s[3 + 1]`, which is `s[7] - s[4]` = `30 - 17` = `13`. So, `ans[3]` becomes `13`.
6. **Return the Answer**: The resulting decrypted array is `[12, 10, 16, 13]`.

The time complexity of this approach is `O(n)`. The prefix sum array and extending the `s` array let us avoid dealing with circular wrap-around through modulus operations, enabling us to calculate each element's sum in constant time.

Python Solution

```python
from typing import List
from itertools import accumulate

class Solution:
    def decrypt(self, code: List[int], k: int) -> List[int]:
        # Get the length of the code list
        length_of_code = len(code)
        # Initial answer list filled with zeros
        decrypted_code = [0] * length_of_code

        # If k is 0, the task is to return a list of the same length filled with zeros
        if k == 0:
            return decrypted_code

        # Create a prefix sum array with the code list repeated twice
        # 'initial=0' is to set the starting value of the accumulation to zero
        prefix_sum = list(accumulate(code + code, initial=0))

        # Iterate through each element in the code list to compute its decrypted value
        for i in range(length_of_code):
            if k > 0:
                # If k is positive, sum the next k values from the element's position
                decrypted_code[i] = prefix_sum[i + k + 1] - prefix_sum[i + 1]
            else:
                # If k is negative, sum the k values preceding the element's position
                decrypted_code[i] = prefix_sum[i + length_of_code] - prefix_sum[i + k + length_of_code]

        # Return the decrypted code
        return decrypted_code

```

Java Solution

```java
class Solution {
    public int[] decrypt(int[] code, int k) {
        int n = code.length; // Length of the code array.
        int[] answer = new int[n]; // The resultant array after decryption.

        // If k is 0, then the decryption is a zero array of length n.
        if (k == 0) {
            return answer;
        }

        // Create a sum array with size double the code (to handle wrap around) plus one (for easing the prefix sum calculation).
        int[] sum = new int[n * 2 + 1];

        // Compute the prefix sums for the array.
        for (int i = 0; i < n * 2; ++i) {
            sum[i + 1] = sum[i] + code[i % n];
        }

        // Process each element in the code array.
        for (int i = 0; i < n; ++i) {
            // If k is positive, sum the next k elements including the current one.
            if (k > 0) {
                answer[i] = sum[i + k + 1] - sum[i + 1];
            } else {
                // If k is negative, sum the k elements before the current one.
                answer[i] = sum[i + n] - sum[i + k + n];
            }
        }

        // Return the decrypted code.
        return answer;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    vector<int> decrypt(vector<int>& code, int k) {
        int n = code.size(); // Get the size of the code vector
        vector<int> ans(n, 0); // Initialize the answer vector with `n` zeros

        // If k is zero, no decryption is performed; return the initialized answer
        if (k == 0) {
            return ans;
        }

        // Create a prefix sum array with double the size of the code array plus one
        // This allows easy calculation over circular arrays
        vector<int> prefixSum(2 * n + 1, 0);

        // Calculate prefix sums for `code` replicated twice
        for (int i = 0; i < 2 * n; ++i) {
            prefixSum[i + 1] = prefixSum[i] + code[i % n];
        }

        // Decrypt the code based on the value of k
        for (int i = 0; i < n; ++i) {
            if (k > 0) {
                // If k is positive, sum next k elements
                ans[i] = prefixSum[i + k + 1] - prefixSum[i + 1];
            } else {
                // If k is negative, sum previous k elements
                ans[i] = prefixSum[i + n] - prefixSum[i + k + n];
            }
        }

        // Return the decrypted code
        return ans;
    }
};

```

Typescript Solution

```typescript
// Function to decrypt a code array based on an integer k
// If k is positive, replace every element with the sum of the next k elements
// If k is negative, replace every element with the sum of the previous -k elements
// If k is zero, replace every element with 0
function decrypt(code: number[], k: number): number[] {
    // Get the number of elements in the code array
    const codeLength = code.length;

    // If k is 0, fill the entire code array with 0s and return it
    if (k === 0) {
        return new Array(codeLength).fill(0);
    }

    // Determine the direction of the sum based on the sign of k
    const isNegativeK = k < 0;
    if (isNegativeK) {
        k = -k; // Make k positive for easier calculations
    }

    // Initialize a map to store the sum of elements at each index for prefix and suffix
    const sumMap = new Map<number, [number, number]>();

    // Calculate the initial prefix and suffix sums
    let prefixSum = 0;
    let suffixSum = 0;
    for (let i = 1; i <= k; i++) {
        prefixSum += code[codeLength - i];
        suffixSum += code[i % codeLength];
    }

    // Store the initial sums in the map
    sumMap.set(0, [prefixSum, suffixSum]);

    // Calculate and store the prefix and suffix sums for each element
    for (let i = 1; i < codeLength; i++) {
        let [previousPrefix, previousSuffix] = sumMap.get(i - 1);

        // Update prefix by subtracting and adding elements at the boundaries
        previousPrefix -= code[(codeLength - k - 1 + i) % codeLength];
        previousPrefix += code[(i - 1) % codeLength];

        // Update suffix by subtracting and adding elements at the boundaries
        previousSuffix -= code[i % codeLength];
        previousSuffix += code[(i + k) % codeLength];

        // Store the updated sums in the map
        sumMap.set(i, [previousPrefix, previousSuffix]);
    }

    // Construct and return the decrypted code array
    return code.map((_, index) => sumMap.get(index)[isNegativeK ? 0 : 1]);
}

// Example usage
const encryptedCode = [5, 7, 1, 4];
const k = 3;
const decryptedCode = decrypt(encryptedCode, k);
console.log(decryptedCode); // Output will be the decrypted code based on the value of k

```

Time and Space Complexity

Time Complexity

The main operations to consider for time complexity are the accumulation of the list and the for loop that iterates over the range `(n)`.

- `accumulate(code + code, initial=0)`: Doubling the list `code` has a time complexity of `O(n)` since it involves copying the elements. The `accumulate()` function performs a prefix sum operation, which takes `O(2n)` time over the doubled list (since the list `code` is appended to itself).
- The for loop: This iterates over the list of length `n` and performs constant-time operations inside the loop. No nested loops are present, therefore, this part contributes `O(n)` to the time complexity.

Hence, the total time complexity of the code is `O(n) + O(2n) + O(n)` which simplifies to `O(n)`.

Space Complexity

For space complexity analysis, the additional space used by the algorithm aside from the input and output is considered.

- `s = list(accumulate(code + code, initial=0))`: It creates a new list from the accumulated sums of the doubled original list. Since the list is doubled in size before accumulating, and an extra element is added due to `initial=0`, the space complexity for this part is `O(2n+1)` which simplifies to `O(n)` since constant factors and lower-order terms are ignored.
- `ans = [0] * n`: Allocates a list of the same length as the input list `code`. This contributes `O(n)` to the space complexity.

Other variables (`n`, `i`, `k`) use constant space and do not scale with the size of the input.

The total space complexity of the code is `O(n)` for the accumulated sums list plus `O(n)` for the answer list, which simplifies to `O(n)` since both are linear terms and we do not multiply complexities when they are of the same order.