1652. Defuse the Bomb

Problem Description

Easy

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.

You are tasked with decrypting a circular array of numbers (code) given a specific integer key (k). The array is circular, which

- 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.

bounds and to wrap around when needed.

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

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 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.

When k is positive, we take the sum of the subarray that starts just after the current element and extends k elements forward.

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

Solution Approach

```
To implement the solution, we leverage a well-known algorithmic pattern: the prefix sum array. This array stores the cumulative
```

(exclusive) in the circular array.

amount of time due to the precomputed prefix sums.

decrypt the array following the solution approach:

manually wrap around the array's ends.

• 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.

zeros, we can return it directly without any additional computation.

between s[i + k + 1] and s[i + 1]. This gives us the sum of the k elements following index i.

handling of circular array indices to simple array accesses and subtraction operations.

sums of the elements in an array, allowing fast computation of the sum of any subarray. The solution involves the following steps: Create an Extended Array: We initiate by creating an array s that's twice as long as the input array code, achieved by

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.

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. **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

concatenating code with itself (code + code). This simplifies our calculations for the circular array since we won't need to

- **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
- **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

This solution is not only efficient but also elegant, as it reduces a problem involving potentially complex modular arithmetic and

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

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

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

∘ 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

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

Calculate Prefix Sums: Using the prefix sums concept, we calculate the sums as follows: [0, 5, 12, 13, 17, 22, 29, 30, 34]

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

• 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 = 10

 \circ 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.

0].

12. So, ans [0] becomes 12.

from typing import List

if k == 0:

if k > 0:

else:

if (k == 0) {

return answer;

class Solution:

from itertools import accumulate

7, 1, 4].

Example Walkthrough

 \circ 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. \circ 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.

Return the Answer: The resulting decrypted array is [12, 10, 16, 13].

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

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

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

wrap-around through modulus operations, enabling us to calculate each element's sum in constant time. Solution Implementation

The time complexity of this approach is O(n). The prefix sum array and extending the s array let us avoid dealing with circular

Python

If k is 0, the task is to return a list of the same length filled with zeros

If k is positive, sum the next k values from the element's position

If k is negative, sum the k values preceding the element's position

decrypted_code[i] = prefix_sum[i + k + 1] - prefix_sum[i + 1]

Create a prefix sum array with the code list repeated twice

def decrypt(self, code: List[int], k: int) -> List[int]:

Get the length of the code list

Initial answer list filled with zeros

decrypted_code = [0] * length_of_code

length_of_code = len(code)

return decrypted_code

Return the decrypted code

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.

// Create a prefix sum array with double the size of the code array plus one

// This allows easy calculation over circular arrays

// Calculate prefix sums for `code` replicated twice

prefixSum[i + 1] = prefixSum[i] + code[i % n];

// If k is positive, sum next k elements

ans[i] = prefixSum[i + k + 1] - prefixSum[i + 1];

ans[i] = prefixSum[i + n] - prefixSum[i + k + n];

// If k is positive, replace every element with the sum of the next k elements

// If k is 0, fill the entire code array with 0s and return it

// Determine the direction of the sum based on the sign of k

// If k is negative, replace every element with the sum of the previous -k elements

// If k is negative, sum previous k elements

vector<int> prefixSum(2 * n + 1, 0);

for (int i = 0; i < 2 * n; ++i) {

for (int i = 0; i < n; ++i) {

// Return the decrypted code

// If k is zero, replace every element with 0

const codeLength = code.length;

if (k > 0) {

} else {

return ans;

if (k === 0) {

// Example usage

from typing import List

if k == 0:

from itertools import accumulate

const k = 3;

class Solution:

const encryptedCode = [5, 7, 1, 4];

};

TypeScript

// Decrypt the code based on the value of k

// Function to decrypt a code array based on an integer k

function decrypt(code: number[], k: number): number[] {

// Get the number of elements in the code array

return new Array(codeLength).fill(0);

for (let i = 1; i < codeLength; i++) {</pre>

// If k is 0, then the decryption is a zero array of length n.

return decrypted_code

'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):

decrypted_code[i] = prefix_sum[i + length_of_code] - prefix_sum[i + k + length_of_code]

```
Java
```

class Solution {

```
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++
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 sum array with size double the code (to handle wrap around) plus one (for easing the prefix sum calculation).

```
const isNegativeK = k < 0;</pre>
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
```

// Update prefix by subtracting and adding elements at the boundaries

// Update suffix by subtracting and adding elements at the boundaries

let [previousPrefix, previousSuffix] = sumMap.get(i - 1);

previousPrefix += code[(i - 1) % codeLength];

previousSuffix += code[(i + k) % codeLength];

sumMap.set(i, [previousPrefix, previousSuffix]);

previousSuffix -= code[i % codeLength];

// Construct and return the decrypted code array

def decrypt(self, code: List[int], k: int) -> List[int]:

// Store the updated sums in the map

const decryptedCode = decrypt(encryptedCode, k);

Get the length of the code list

Initial answer list filled with zeros

decrypted_code = [0] * length_of_code

length_of_code = len(code)

Return the decrypted code

return decrypted_code

Time and Space Complexity

Time Complexity

(n).

previousPrefix -= code[(codeLength - k - 1 + i) % codeLength];

return code.map((_, index) => sumMap.get(index)[isNegativeK ? 0 : 1]);

console.log(decryptedCode); // Output will be the decrypted code based on the value of k

If k is 0, the task is to return a list of the same length filled with zeros

```
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
```

• 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.

The main operations to consider for time complexity are the accumulation of the list and the for loop that iterates over the range

decrypted_code[i] = prefix_sum[i + length_of_code] - prefix_sum[i + k + length_of_code]

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

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

• 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.

since both are linear terms and we do not multiply complexities when they are of the same order.

• 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 0(2n+1) which simplifies to O(n) since constant factors and lower-order terms are ignored.

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)