2735. Collecting Chocolates

Medium Array Enumeration Leetcode Link

Problem Description

In this problem, you have an array `nums` of size `n` that indicates the cost of collecting chocolates of different types. The index `i` corresponds to the chocolate of the `i^th` type, and its cost is given by `nums[i]`. The goal is to collect one chocolate of each type at the minimum total cost.

To do this, you are allowed to perform operations. Each operation incurs a fixed cost `x`, and as a result, it changes the type of all chocolates simultaneously in a cyclic manner: the chocolate of `i^th` type becomes the chocolate of `((i + 1) mod n)^th` type. You need to calculate the minimum cost to collect all different types of chocolates assuming you can do as many operations as needed.

Intuition

To find the minimum cost to collect all chocolate types, we need to consider two types of costs: the cost of each chocolate collected and the cost of the operations performed to change the chocolates' types.

A common strategy in such problems with cyclic operations and transformations is to simulate each possible scenario and keep track of the minimum cost. Specifically, we can simulate the cost of collecting chocolates without any operations and then after one operation, after two operations, and so on. We continue doing this until we have simulated the scenario after `n-1` operations, as doing `n` or more operations would just cycle back to a previous state.

At each step, while simulating for `j` operations, we need to consider the cheapest chocolate we can collect at each type, given that we have performed `j` operations. This can be precomputed in a 2D array `f`, where `f[i][j]` represents the minimum cost to collect a chocolate of the `i^th` type after `j` operations. This array is filled by finding the cheaper option for each step: purchasing the chocolate of that type without any operation, or the cost of the chocolate of the next type (considering it as the current type due to the operation). Once we have this precomputed array, we can simply iterate over it to calculate the total cost of collecting all chocolate types for each number of operations, adding the cost of operations themselves (`x * j`), and choose the minimum total cost.

The idea is that by considering all the possibilities, we can pinpoint the optimal combination of purchases and operations to achieve the minimum cost.

Solution Approach

The implementation of the solution involves the following concepts:

1. **Precomputation for each chocolate type and operation**: The solution uses a 2D list, `f`, of size `n x n`, where `n` is the number of chocolate types. `f[i][j]` represents the minimum cost to collect a chocolate of the `i^th` type after `j` operations. To fill this array, we iterate over each chocolate type `i` and for each type, we go through each possible number of operations `j`. The cost without any operation is just the cost of the chocolate `nums[i]`, thus `f[i][0] = nums[i]`. For subsequent operations, we take the minimum of the cost `f[i][j - 1]` and the cost of chocolate of the `((i + j) mod n)^th` type (`nums[(i + j) % n]`), because each operation shifts the chocolate types by one.

2. **Calculating the minimum cost for collecting all chocolate types**: We iterate over the number of operations `j` from `0` to `n-1`, compute the cost of collecting all types of chocolates using our precomputed values in `f`, then add the cost of performing `j` operations `(x * j)`. The variable `ans` is used to keep track of the minimum total cost.

3. **Minimizing total cost**: For `j` operations, we sum the precomputed minimum costs of collecting each chocolate `f[i][j]` for all `i` from `0` to `n-1`, add the cost of the operations, and compare it with the current `ans` to check if we've found a cheaper total cost `min(ans, cost)`.

Here's a closer look at the nested loops:

- The outer loop runs through each type index `i` and initializes `f[i][0]`.
- The inner nested loop runs for number of operations `j` and computes `f[i][j]`.
- The final loop, outside of the nested loops, runs through all possible operation counts `j`, sums up the minimum costs for all types and adds the operation cost, checking for a new minimum.

By considering all possible scenarios and picking the minimum, this approach ensures we find the optimal solution.

The important data structure used here is a 2D list (list of lists in Python) to store the minimum possible costs, which is a common approach for dynamic problems where a value depends on previously computed values. Additionally, the modulus operator `%` plays a crucial role for cyclic arithmetic within the array bounds. The solution leverages complete enumeration to ensure that all options are considered, which is often a useful pattern when dealing with small enough problem spaces where considering all possibilities is feasible.

Example Walkthrough

Let's consider a small example to illustrate the solution approach. Suppose we have an array of chocolate costs `nums` given as `[5, 3, 4]` and the fixed cost of each operation `x` as `1`. There are three types of chocolates, and we want to collect one of each type at the minimum cost.

Firstly, we initialize our 2D list `f` to precompute the costs, which will look like this after initialization (assuming `n = 3` for our example):

```
f = [[5, 0, 0],
     [3, 0, 0],
     [4, 0, 0]]
```

Here, `f[i][0] = nums[i]` because the cost of collecting the chocolate of the `i^th` type without any operations is just the cost itself (`nums[i]`).

Now, we start filling out `f` for each chocolate after each operation:

- For 0 operations, no change is made; we leave the initial values.
- For 1 operation, `f[i][1]` would be the minimum of `f[i][0]` and `nums[(i + 1) % n]`.
- For 2 operations, `f[i][2]` would be the minimum of `f[i][1]` and `nums[(i + 2) % n]`.

After precomputation, the `f` matrix will be filled out:

```
f = [[5, 3, 3],
     [3, 4, 3],
     [4, 5, 3]]
```

In above `f` matrix:

- `f[0][1] = min(f[0][0], nums[(0 + 1) % 3]) = min(5, 3) = 3`
- `f[1][1] = min(f[1][0], nums[(1 + 1) % 3]) = min(3, 4) = 3`
- And so forth for other elements...

Now we calculate the minimum total cost of collecting all types of chocolates for each number of operations:

- With 0 operations, the cost is `5 + 3 + 4 = 12`.
- With 1 operation, the cost is `3 + 4 + 5 (cost from f) + 1 (cost of operation) = 13`.
- With 2 operations, the cost is `3 + 3 + 3 (cost from f) + 2 * 1 (cost of operations) = 11`.

Comparing the total costs, we see that the minimum cost to collect all chocolate types is `11` after performing 2 operations. Hence, the answer for this example is `11`.

Python Solution

```python
class Solution:
    def min_cost(self, nums: List[int], x: int) -> int:
        # Get the length of the input array.
        num_count = len(nums)

        # Initialize an NxN matrix to hold the minimum values
        # for subarrays starting from each index with varying lengths.
        min_values = [[0] * num_count for _ in range(num_count)]

        # Start by populating the matrix with the individual numbers
        # And then fill in the rest by comparing with previous minimum values.
        for i, value in enumerate(nums):
            min_values[i][0] = value
            for j in range(1, num_count):
                min_values[i][j] = min(min_values[i][j - 1], nums[(i + j) % num_count])

        # The initial minimum cost is set to be infinity for comparison purposes.
        min_cost = float('inf')

        # Analyze each subarray's minimum value and calculate the total cost
        # considering the cost multiplier 'x' and find the minimum cost.
        for j in range(num_count):
            cost = sum(min_values[i][j] for i in range(num_count)) + x * j
            min_cost = min(min_cost, cost)

        # Return the computed minimum cost.
        return min_cost

```

Java Solution

```java
class Solution {
    public long minCost(int[] nums, int x) {
        int arrayLength = nums.length;
        // Dynamic programming table where f[i][j] will hold the minimum value of nums[i]
        // when considered up to i+j shifted circularly by the array length
        int[][] dpTable = new int[arrayLength][arrayLength];

        // Initialize the dynamic programming table
        for (int i = 0; i < arrayLength; ++i) {
            dpTable[i][0] = nums[i];
            for (int j = 1; j < arrayLength; ++j) {
                // Calculate the minimum value within the shifting window
                dpTable[i][j] = Math.min(dpTable[i][j - 1], nums[(i + j) % arrayLength]);
            }
        }

        // Initialize answer to a very large value
        long minCost = Long.MAX_VALUE;

        // Calculate the minimum cost by iterating over possible shift steps
        for (int shiftSteps = 0; shiftSteps < arrayLength; ++shiftSteps) {
            // Cost of making the shifts
            long cost = 1L * shiftSteps * x;
            // Calculate the total cost including the minimum values at each shift step
            for (int i = 0; i < arrayLength; ++i) {
                cost += dpTable[i][shiftSteps];
            }
            // Update the minimum cost if a lower one is found
            minCost = Math.min(minCost, cost);
        }

        // Return the minimum cost found
        return minCost;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <algorithm>
using namespace std;

class Solution {
public:
    long long minCost(vector<int>& nums, int x) {
        int n = nums.size(); // Get the size of nums vector

        // Initialize a 2D vector with 'n' rows and 'n' values in each row to store minimum values
        vector<vector<int>> minValues(n, vector<int>(n, 0));

        // Calculate the minimum values starting from each index i going up to j elements
        for (int i = 0; i < n; ++i) {
            minValues[i][0] = nums[i]; // Base case: minimum of one element is the element itself
            for (int j = 1; j < n; ++j) {
                // Calculate min value in nums starting from i, considering j elements, wrapping around using modulus
                minValues[i][j] = min(minValues[i][j - 1], nums[(i + j) % n]);
            }
        }

        long long ans = 1LL << 60; // Initialize answer to a very large value

        // Check each possibility of j rotations and the corresponding cost
        for (int j = 0; j < n; ++j) {
            long long cost = 1LL * j * x; // Cost for rotating j times
            for (int i = 0; i < n; ++i) {
                // Add the minimum value found starting from i, with j elements considered
                cost += minValues[i][j];
            }
            // If the cost for this number of rotations is less than the current answer, update the answer
            ans = min(ans, cost);
        }

        return ans; // Return the minimum possible cost
    }
};

```

Typescript Solution

```typescript
function minCost(nums: number[], x: number): number {
    const n = nums.length; // Get the size of nums array

    // Initialize a 2D array with 'n' rows and 'n' values in each row to store minimum values
    const minValues: number[][] = Array.from({ length: n }, () => Array(n).fill(0));

    // Calculate the minimum values starting from each index i up to j elements
    for (let i = 0; i < n; ++i) {
        minValues[i][0] = nums[i]; // Base case: minimum of one element is the element itself
        for (let j = 1; j < n; ++j) {
            // Calculate min value in nums starting from i, considering j elements, wrapping around using modulus
            minValues[i][j] = Math.min(minValues[i][j - 1], nums[(i + j) % n]);
        }
    }

    let ans = Number.MAX_SAFE_INTEGER; // Initialize answer to a very large value

    // Check each possibility of j rotations and the corresponding cost
    for (let j = 0; j < n; ++j) {
        let cost = 1 * j * x; // Cost for rotating j times
        for (let i = 0; i < n; ++i) {
            // Add the minimum value found starting from i, with j elements considered
            cost += minValues[i][j];
        }
        // If the cost for this number of rotations is less than the current answer, update the answer
        ans = Math.min(ans, cost);
    }

    return ans; // Return the minimum possible cost
}

```

Time and Space Complexity

Time Complexity

The given Python code consists of multiple nested loops. Here is the breakdown of its time complexity:

1. The first `for` loop with variable `i` is iterating through the list `nums`. It executes `n` times, where `n` is the length of the list.
2. Inside this loop, another `for` loop with variable `j` is running from `0` to `n`. This loop is executed `n*(n-1)/2` times in total because it starts at 1 and iterates to `n - 1` for each value of `i`.
3. Inside the inner loop, there's the expression `nums[(i + j) % n]`, which has a constant time operation.
4. The computation of `min(f[i][j - 1], nums[(i + j) % n])` is also a constant time operation.

Combining these, we have the `i` loop running `n` times and the `j` loop running at most `n` times for each `i`. Therefore, the time complexity for these loops is `O(n^2)`.

The second part of the code includes another `for` loop with variable `j` iterating `n` times.

1. Inside this loop, the `sum()` function iterates over all `n` elements of `f` for each `j`, which takes `O(n)` time.
2. The calculation `x * j` is a constant time operation executed `n` times.

Hence, the additional time complexity contributed by the second part is `O(n^2)`.

Adding both parts, the **overall time complexity** of the code is `O(n^2)` + `O(n^2)` = `O(n^2)`.

Space Complexity

The space complexity of the code primarily involves the storage used by the two-dimensional list `f` which has `n` lists of `n` integers each. Apart from this, there are constant space usages for variables such as `n`, `i`, `v`, `cost`, and `ans`.

The two-dimensional list `f` requires `O(n^2)` space.

Thus, the **overall space complexity** of the code is `O(n^2)`.