1959. Minimum Total Space Wasted With K Resizing Operations

Medium | Array | Dynamic Programming | Leetcode Link

Problem Description

The problem involves designing a dynamic array that can accommodate a varying number of elements at different times. A dynamic array is an array that can change its size during the execution of a program. We have an array called `nums` where `nums[i]` represents the number of elements in the array at time `i`. Additionally, the array can be resized up to `k` times.

A resize operation can change the array's size to any number, and the size after each resize or at each time needs to be at least as large as the number of elements specified at that time (i.e., `nums[t]`). The efficiency of our dynamic array design is measured by the amount of "space wasted," which is the unused capacity at each point in time. The goal is to minimize the total space wasted after all the insertions, given the constraint that the array can be resized at most `k` times.

Intuition

The intuition behind the solution is to use dynamic programming to explore different ways of resizing the dynamic array while keeping track of the space wasted. Dynamic programming is a method for solving complex problems by breaking them down into simpler sub-problems. It is applicable when the problem can be divided into overlapping sub-problems with the optimal structure.

Firstly, the concept of precomputing the space wasted for all possible subarray lengths is employed. This calculation is stored in a 2D array (matrix `g`), where `g[i][j]` gives the space wasted for a subarray starting at index `i` and ending at index `j`. This is done by iterating over each subarray, tracking the maximum element (`mx`) within that subarray, and calculating the cumulative sum (`s`) of elements. The space wasted is then the maximum element times the length of the subarray minus the sum of elements in the subarray.

After calculating the space wasted for all subarrays, dynamic programming is used to find the minimum total space wasted with exactly `j` resizes (`j <= k+1`). To do this, another 2D array (matrix `f`) is used, where `f[i][j]` represents the minimum space wasted for the first `i` elements with `j` resizes.

The transition function for dynamic programming is the core part that gradually builds up the solution. It iterates over all possible combinations of array sizes and resizes and computes the minimum possible space wasted. In the end, `f[n][k+1]` holds the answer to the problem – the minimum total space wasted after optimizing the resizing operations, given the constraints provided by the `nums` array and the `k` value.

Solution Approach

The implemented solution follows a dynamic programming approach that consists of the following steps:

1. **Pre-compute wasted space for subarrays**: We start by creating a 2D array (matrix `g`) of size `n x n` (`n` being the length of the `nums` array). `g[i][j]` represents the space wasted for a continuous subarray starting at index `i` and ending at index `j`. To fill in the matrix, we iterate through all possible starting and ending indices. For each subarray defined by `(i,j)`, we calculate the accumulated sum `s` of all elements and keep track of the maximum element `mx`. Using these values, we calculate the wasted space as `mx * (j - i + 1) - s`.

2. **Dynamic programming (DP) to minimize total wasted space**: Next, we create a 2D DP array (matrix `f`) with `n + 1` rows (for `n` elements) and `k + 1` columns (for `k` resize operations allowed). Each cell `f[i][j]` represents the minimum total space wasted with `i` elements and `j` resizes. The matrix is initialized with `inf`, representing a large number since we want to minimize the space wasted. The first row of the matrix is initialized to 0 because no space is wasted before any elements are added.

3. **Building the solution from sub-problems**: The solution iterates over each possible number of elements (`i`, from 1 to `n`) and each possible number of resizes (`j`, from 1 to `k + 1`). For each pair `(i, j)`, it explores the possibility of making a resize at every previous position `h` (from 0 to `i - 1`). It then uses the pre-computed `g[h][i - 1]` to find the total wasted space if we did the last resize at position `h`. We determine the minimum space wasted as:

   ```
   f[i][j] = min(f[i][j], f[h][j - 1] + g[h][i - 1])
   ```

4. **Finding the result**: After filling the `f` matrix using the above relation, the minimum space wasted with `n` elements and `k` resizes is found at `f[n][k + 1]`, which is then returned as the result.

In summary, the solution employs dynamic programming with pre-computation and iteration over sub-problems to find the minimum space wasted given a set of constraints. By breaking down the problem in this way, it effectively handles the complexity of determining when and how to resize the dynamic array optimally.

Example Walkthrough

Let's apply the solution approach to a small example to illustrate it better.

Suppose we have the following array `nums`: `[3, 1, 5, 4]`, which indicates the number of elements in the array at each time `i`. We are also given that we can resize the dynamic array up to `k = 2` times.

Step 1: Pre-compute wasted space for subarrays

We begin by creating a matrix `g` with the dimensions `4x4` (`n x n`, where `n` is the length of the `nums` array). The matrix `g` will help us know the wasted space for any subarray `(i, j)`.

- We calculate the wasted space for all subarrays. For example, `g[0][2]` corresponds to the subarray `[3, 1, 5]`. The maximum element `mx` is 5, and the total number of elements `s` sum to 9. The space wasted will be `5 * 3 - 9 = 6`.

This step is repeated to fill out the entire matrix `g`.

Step 2: Dynamic programming to minimize total wasted space

Now, we create a matrix `f` with dimensions `5x3` (`n+1 x k+1`) to store the minimum total space wasted for `i` elements with `j` resizes. Initially, `f` is filled with a large value (e.g., `inf`), except `f[0][*]`, which is filled with 0 since no space is wasted before any elements are added.

Step 3: Building the solution from sub-problems

We populate the `f` matrix based on the previously calculated values in `g`. For each `i` and `j`, we look back at each possible 'last resize' point `h`:

- Consider `f[4][2]`: we explore `h = 0`, `h = 1`, `h = 2`, and `h = 3`.

Using the previous state `f[h][j - 1]` and adding the wasted space of the subarray from `h` to `i-1` (obtained from `g[h][i - 1]`), we update `f[i][j]` to the minimum of these values.

As an example, let's look at the case when `j = 2` (meaning one resize has already been done) and `i = 4`:

- Without any previous resizes, `f[4][2]` would initially be `inf`.
- Consider the resize at `h = 2`, we take `f[2][1]` (the best case of no resizes up to 2 elements) and add the waste from resizing at that point for the rest of the elements: `g[2][3]`.
- We do this for all `h` and keep the minimum value in `f[4][2]`.

Step 4: Finding the result

After completing the `f` matrix, we find the minimum space wasted with `n = 4` elements and `k = 2` resizes at `f[4][3]` since we indexed our resizes starting from 1 up to `k+1`.

By iterating over the sub-problems and combining their results, we are able to establish the entire space wasted and then select the minimum. The result in `f[4][3]` gives us the minimum total space wasted for our `nums` array with `k = 2` resizes. This demonstrates the application of dynamic programming to efficiently solve the problem.

Python Solution

```python
from typing import List
from math import inf

class Solution:
    def min_space_wasted_k_resizing(self, nums: List[int], k: int) -> int:
        # Allow k resizings (plus one to account for zero indexing)
        k += 1
        n = len(nums)
        # Initialize a grid to store the wasted space between every pair of indices
        wasted_space_grid = [[0] * n for _ in range(n)]

        # Pre-calculate the wasted space for all possible segments
        for start_index in range(n):
            total_segment_sum = 0  # Sum of elements in the current segment
            max_element = 0        # Max element in the current segment

            for end_index in range(start_index, n):
                total_segment_sum += nums[end_index]
                max_element = max(max_element, nums[end_index])
                # Calculate the wasted space for the current segment
                segment_length = end_index - start_index + 1
                wasted_space = max_element * segment_length - total_segment_sum
                wasted_space_grid[start_index][end_index] = wasted_space

        # Initialize a DP table for the minimum wasted space
        min_waste_dp = [[inf] * (k + 1) for _ in range(n + 1)]
        min_waste_dp[0][0] = 0

        # DP to find the minimum wasted space with up to k resizings
        for i in range(1, n + 1):
            for resizing_count in range(1, k + 1):
                for prev_partition_end in range(i):
                    # Update the DP table entry with the minimum wasted space
                    waste_with_prev_partition = min_waste_dp[prev_partition_end][resizing_count - 1]
                    current_waste = waste_with_prev_partition + wasted_space_grid[prev_partition_end][i - 1]
                    min_waste_dp[i][resizing_count] = min(min_waste_dp[i][resizing_count], current_waste)

        # Return the minimum wasted space after n elements and k resizings
        return min_waste_dp[-1][-1]

# Example usage:
# solution = Solution()
# print(solution.min_space_wasted_k_resizing(nums=[10, 20], k=0))
```

Java Solution

```java
class Solution {
    public int minSpaceWastedKResizing(int[] nums, int k) {
        k++; // Increment k because we can make "k+1" partitions
        int n = nums.length; // Length of the nums array
        int[][] wastedSpaceGrid = new int[n][n]; // 2D array to record wasted space for segments

        // Pre-compute wasted space for each segment [i, j]
        for (int i = 0; i < n; ++i) {
            int sum = 0, maxNum = 0;
            for (int j = i; j < n; ++j) {
                sum += nums[j];
                maxNum = Math.max(maxNum, nums[j]);
                wastedSpaceGrid[i][j] = maxNum * (j - i + 1) - sum;
            }
        }

        // f[i][j] represents the minimum wasted space using j resizings up to the i-th element
        int[][] dp = new int[n + 1][k + 1];
        int infinity = Integer.MAX_VALUE; // Using a large number to represent infinity

        // Initialize the dp array with infinity
        for (int i = 0; i < dp.length; ++i) {
            Arrays.fill(dp[i], infinity);
        }
        dp[0][0] = 0; // Base case: 0 wasted space with 0 elements and 0 resizings

        // Fill up the DP table
        for (int i = 1; i <= n; ++i) {
            for (int j = 1; j <= k; ++j) {
                for (int h = 0; h < i; ++h) {
                    // Calculate minimum wasted space for dp[i][j]
                    dp[i][j] = Math.min(dp[i][j], dp[h][j - 1] + wastedSpaceGrid[h][i - 1]);
                }
            }
        }

        // Return minimum wasted space with n elements and k resizings
        return dp[n][k];
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    int minSpaceWastedKResizing(vector<int>& nums, int k) {
        ++k; // Incrementing k because we can perform k+1 operations
        int n = nums.size(); // n is the size of the input array nums

        // g will store the extra space wasted for each subarray
        vector<vector<int>> wastedSpace(n, vector<int>(n));

        // Calculating the amount of space wasted if we resize from i to j
        for (int start = 0; start < n; ++start) {
            int sum = 0, maxElement = 0;
            for (int end = start; end < n; ++end) {
                maxElement = max(maxElement, nums[end]);
                sum += nums[end];
                wastedSpace[start][end] = maxElement * (end - start + 1) - sum;
            }
        }

        int infinity = 0x3f3f3f3f; // Using a large number to represent infinity
        // f will store the minimum wasted space for subarrays with different numbers of resizing operations
        vector<vector<int>> minWastedSpace(n + 1, vector<int>(k + 1, infinity));
        minWastedSpace[0][0] = 0; // Base case: no numbers and no operations equals zero wasted space

        // Calculating the minimum wasted space for each subarray with j operations
        for (int i = 1; i <= n; ++i) {
            for (int operations = 1; operations <= k; ++operations) {
                for (int previousIndex = 0; previousIndex < i; ++previousIndex) {
                    minWastedSpace[i][operations] = min(
                        minWastedSpace[i][operations],
                        minWastedSpace[previousIndex][operations - 1] + wastedSpace[previousIndex][i - 1]
                    );
                }
            }
        }

        // The answer is the minimum wasted space for the whole array with k operations
        return minWastedSpace[n][k];
    }
};
```

Typescript Solution

```typescript
function minSpaceWastedKResizing(nums: number[], k: number): number {
    k++; // Incrementing k because we can perform k+1 operations
    const n: number = nums.length; // n is the size of the input array nums

    // wastedSpace will store the extra space wasted for each subarray
    let wastedSpace: number[][] = Array.from({ length: n }, () => Array(n).fill(0));

    // Calculating the amount of space wasted if we resize from i to j
    for (let start = 0; start < n; ++start) {
        let sum: number = 0, maxElement: number = 0;
        for (let end = start; end < n; ++end) {
            maxElement = Math.max(maxElement, nums[end]);
            sum += nums[end];
            wastedSpace[start][end] = maxElement * (end - start + 1) - sum;
        }
    }

    const infinity: number = Infinity; // Using Infinity to represent a very large number
    // minWastedSpace will store the minimum wasted space for subarrays with different numbers of resizing operations
    let minWastedSpace: number[][] = Array.from({ length: n + 1 }, () => Array(k + 1).fill(infinity));
    minWastedSpace[0][0] = 0; // Base case: no numbers and no operations equals zero wasted space

    // Calculating the minimum wasted space for each subarray with j operations
    for (let i = 1; i <= n; ++i) {
        for (let operations = 1; operations <= k; ++operations) {
            for (let previousIndex = 0; previousIndex < i; ++previousIndex) {
                minWastedSpace[i][operations] = Math.min(
                    minWastedSpace[i][operations],
                    minWastedSpace[previousIndex][operations - 1] + wastedSpace[previousIndex][i - 1]
                );
            }
        }
    }

    // The answer is the minimum wasted space for the whole array with k operations
    return minWastedSpace[n][k];
}
```

Time and Space Complexity

Time Complexity

The time complexity of the code is determined by several nested loops:

- There are two loops used to fill in the `g` matrix, which stores the space wasted if we resize the array from `i` to `j`. The outer loop runs `n` times, and the inner loop runs at most `n` times in the worst case, resulting in a time complexity of `O(n^2)` for this part.
- There are three loops used to calculate the minimum space wasted with `k` resizings in matrix `f`:
  - The outermost loop runs `n + 1` times (accounting for `i` from `1` to `n`).
  - The middle loop runs `k + 1` times, since `k` is incremented at the beginning (`k += 1`).
  - The innermost loop runs up to `i` times, which in the worst case would be `n` times.

  Combining these loops, we get a time complexity of `O(n^2 * (k + 1))` for this second part.

Combining both parts, the total time complexity is `O(n^2 + n^2 * (k + 1))`, which simplifies to `O(n^2 * k)`.

Space Complexity

The space complexity is determined by the space needed to store the matrices `g` and `f`:

- The `g` matrix is `n` by `n`, so it requires `O(n^2)` space.
- The `f` matrix is `(n + 1)` by `(k + 1)`, so it requires `O(n * k)` space.

The total space complexity is `O(n^2 + n * k)`. Since in most cases `k` would be less than `n`, this simplifies to `O(n^2)`.