1000. Minimum Cost to Merge Stones

Hard | Array | Dynamic Programming | Prefix Sum | Leetcode Link

Problem Description

In this problem, we are given `n` piles of stones arranged in a row. Each pile contains a certain number of stones, given by the array `stones` where `stones[i]` represents the number of stones in the `i-th` pile. We can perform a move that merges `k` consecutive piles into a single pile. The cost of such a move is equal to the sum of stones in the `k` piles that were merged.

The objective is to find the minimum cost to merge all the piles into one. However, there is a constraint: we can only merge exactly `k` consecutive piles at a time. If it is not possible to merge all piles into one, considering this constraint, we should return `-1`. This adds a twist to the problem because we cannot simply merge any number of piles; it has to be exactly `k` piles each time.

Intuition

The solution to this problem makes use of **dynamic programming**. The intuition behind the approach is to break down the problem into smaller subproblems, solve each subproblem, and use the results to build up the solution to the final problem.

Since the costs are dependent on consecutive piles and the number of stones in the piles that are merged, it makes sense to consider:

1. The subproblem of finding the minimal cost to merge a subsequence of piles into a fewer number of piles.
2. Then gradually increase the number of piles being considered until we reach the full list of piles.

However, there are additional points to consider:

- If `(n - 1) % (K - 1)` is not zero, we cannot merge into exactly one pile since at some point the number of piles left cannot be merged by groups of `k`. Therefore, we return `-1`.
- We use a 3D dynamic programming table `f`, where `f[i][j][k]` represents the minimum cost to merge the piles from `i` to `j` into `k` piles.

We perform the following steps:

1. Precompute the prefix sum array to quickly calculate the sum of stones from any pile `i` to any pile `j`.
2. Initialize the table `f[i][i][1]` to `0` because the cost to merge a single pile into a single pile is `0`.
3. Use a bottom-up approach to solve all subproblems starting from the lowest number of piles and expanding to the full list.
4. Update `f[i][j][k]` by considering the cost of merging a certain number of piles and check if adding an additional pile to those already merged is beneficial.
5. To merge piles into one, we look at the result of merging `k` piles and add the cost of all stones from `i` to `j`, stored in the prefix sum array.

The answer to the problem will be stored in `f[1][n][1]`, which represents the minimal cost of merging all piles from 1 to n into a single pile.

Solution Approach

The solution uses a **3D dynamic programming (DP)** table to keep track of the minimum cost for various subproblems, as well as a prefix sum array to efficiently calculate the total number of stones in a range of piles.

Here is a step-by-step explanation of the solution approach:

1. **Initialization**: First, we check if the total number of piles `n` allows them to be merged into one. Since every merge reduces the number of piles by `K - 1`, if `(n - 1) % (K - 1)` is not zero, it's impossible to end with one pile, and we return `-1`.

2. **Prefix Sum Array**: The prefix sum array `s` is prepared, where `s[0]` is set to `0` and `s[i]` is the total number of stones from pile `1` to `i` inclusive. This allows quick calculation of the total number of stones in any range `i` to `j` by simply doing `s[j] - s[i - 1]`.

3. **DP Table**: The 3D DP array `f` is created and filled with infinity (`inf`) to denote that we have not calculated the minimal cost for the given subproblems yet. `f[i][i][1]` is set to `0` for all `i` since there is no cost to "merge" a single pile into itself.

4. **Subproblem Solutions**: Next, the nested loops iterate through all subranges `l` of piles from length `2` to `n` (`n` being the full range). For each subrange, we calculate the cost of merging subranges into `k` piles where `1 <= k <= K`.

5. **Transition**: Inside another nested loop, we calculate `f[i][j][k]` by iterating over all possible middle points `h`. The cost `f[i][j][k]` is updated as the minimum of `f[i][j][k]` or the sum of costs `f[i][h][1]` and `f[h + 1][j][k - 1]`. This represents merging piles from `i` to `h` into one pile and from `h + 1` to `j` into `k - 1` piles.

6. **Final Calculation**: After considering `k` piles, we calculate `f[i][j][1]` if we are merging into one pile from the current range `i` to `j`. The cost here is the minimal cost of merging into `K` piles plus the sum of all stones in the current range.

The main algorithmic concepts used in this solution are dynamic programming for breaking down the problem into overlapping subproblems and calculating their minimal costs, and the prefix sum technique for fast range sum queries. The use of a 3D array allows keeping track of costs for different numbers of resulting piles in the subranges.

The final answer is found in `f[1][n][1]` which gives the minimum cost to merge all piles from `1` to `n` into one pile.

Example Walkthrough

Let's use a small example to illustrate the solution approach:

Suppose we have `n = 5` piles of stones, arranged in a row and the piles contain `{3, 2, 4, 1, 2}` stones respectively. We can only merge `k = 3` consecutive piles at a time.

Following the solution approach:

1. **Initialization**: We check `(n - 1) % (k - 1) = (5 - 1) % (3 - 1) = 0`, so we can proceed.

2. **Prefix Sum Array**: We prepare the prefix sum array `s`, which will be `{0, 3, 5, 9, 10, 12}` where `s[i] = s[i-1] + stones[i-1]`.

3. **DP Table**: We initialize a 3D DP array `f` with dimensions `[n+1][n+1][k+1]`. We'll set `f[i][i][1]` to `0` for `i` from `1` to `5`.

4. **Subproblem Solutions**: We iterate through subranges of length `l = 2` to `5`. For example:

   - When `l = 2`, our subranges are `(1, 2)`, `(2, 3)`, `(3, 4)`, and `(4, 5)`. We're going to calculate the minimum cost to turn each size-2 subrange into a single pile.
   - We continue with `l = 3`, `l = 4`, and finally `l = 5`, which is our overall problem.
5. **Transition**: We then iterate through all subranges and potential merge points to update our DP table. For example, looking at the subrange `(1, 3)`, we'd calculate the cost of merging pile `1` into one pile and piles `2-3` into one pile and add these costs together.

6. **Final Calculation**: Every time we look at merging into a single pile (`k = 1`), we include the cost of the stones. For the case of our subrange `(1, 3)`, after finding the minimum merge cost into `k = 3` piles, we add the prefix sum of piles `1` to `3`, which is `s[3] - s[0] = 9 - 0 = 9`.

Assuming we fill out the DP table with all the correct values through our iteration, we finally look at `f[1][5][1]` for our answer.

Let's calculate step-by-step for this example:

- For `l = 2`, we just merge the neighboring piles. The costs are `f[1][2][1] = 5`, `f[2][3][1] = 6`, `f[3][4][1] = 5`, and `f[4][5][1] = 3`.
- For `l = 3`, we have three possible merges for piles `(1, 3)`, `(2, 4)`, and `(3, 5)`. We consider the cost for `l = 2` cases we calculated and the single piles. For `(1, 3)`, since we must merge 3 piles at once, the cost is the sum of piles `1` to `3`, so `f[1][3][1] = 9`.
- We continue this way until `l = 5`. Our final merge for `(1, 5)` must consider the cost of merge into `3` piles (`k = 3`) plus the sum of stones from pile `1` to `5`, which is `f[1][3][1] + f[4][5][1] + (s[5] - s[0]) = 9 + 3 + 12 = 24`.

Thus, the result is `f[1][5][1] = 24`, which means the minimal cost to merge all piles from `1` to `5` into one pile is `24`.

Python Solution

```python
from itertools import accumulate
from math import inf

class Solution:
    def mergeStones(self, stones: List[int], K: int) -> int:
        # Calculate the total number of stones.
        num_stones = len(stones)

        # If we cannot converge to a single pile with given K, return -1.
        if (num_stones - 1) % (K - 1):
            return -1

        # Calculate the prefix sum of the stones array for quick range sum calculation.
        prefix_sum = list(accumulate(stones, initial=0))

        # Initialize the dynamic programming table with infinity.
        dp = [[[inf] * (K + 1) for _ in range(num_stones + 1)] for _ in range(num_stones + 1)]

        # A single stone pile has 0 cost to merge.
        for i in range(1, num_stones + 1):
            dp[i][i][1] = 0

        # Iterate over all possible lengths of the subarray of stones (from 2 to num_stones).
        for length in range(2, num_stones + 1):
            for i in range(1, num_stones - length + 2):
                j = i + length - 1  # The end index of the subarray

                # Iterate over all possible numbers of piles K.
                for k in range(1, K + 1):

                    # Split the subarray into two parts and add the cost of merging them.
                    for h in range(i, j):
                        dp[i][j][k] = min(dp[i][j][k], dp[i][h][1] + dp[h + 1][j][k - 1])

                # If we can form a single pile, calculate the cost including merging the last pile.
                if k == 1:
                    dp[i][j][1] = dp[i][j][K] + prefix_sum[j] - prefix_sum[i - 1]

        # Return the minimum cost to merge the entire array into one pile.
        return dp[1][num_stones][1]
```

Java Solution

```java
class Solution {
    public int mergeStones(int[] stones, int K) {
        int stoneCount = stones.length;

        // If it's not possible to merge to one pile, return -1
        if ((stoneCount - 1) % (K - 1) != 0) {
            return -1;
        }

        // Prefix sums of stones array for calculating subarray sums
        int[] prefixSums = new int[stoneCount + 1];
        for (int i = 1; i <= stoneCount; ++i) {
            prefixSums[i] = prefixSums[i - 1] + stones[i - 1];
        }

        // Initialize DP table with infinity values except for the base case: f[i][i][1] = 0
        final int infinity = 1 << 20;
        int[][][] dpTable = new int[stoneCount + 1][stoneCount + 1][K + 1];
        for (int[][] secondDimension : dpTable) {
            for (int[] thirdDimension : secondDimension) {
                Arrays.fill(thirdDimension, infinity);
            }
        }
        for (int i = 1; i <= stoneCount; ++i) {
            dpTable[i][i][1] = 0;
        }

        // Dynamic programming to find minimal cost
        for (int length = 2; length <= stoneCount; ++length) {
            for (int i = 1; i + length - 1 <= stoneCount; ++i) {
                int j = i + length - 1;
                for (int piles = 1; piles <= K; ++piles) {
                    // Calculate minimum cost for merging stones into 'piles' piles
                    for (int mid = i; mid < j; ++mid) {
                        dpTable[i][j][piles] = Math.min(dpTable[i][j][piles],
                                                        dpTable[i][mid][1] + dpTable[mid + 1][j][piles - 1]);
                    }
                }
                // Merge K piles into 1 pile incurs additional cost which is the sum of stones[i...j]
                dpTable[i][j][1] = dpTable[i][j][K] + prefixSums[j] - prefixSums[i - 1];
            }
        }
        // Return minimal cost to merge all stones into 1 pile
        return dpTable[1][stoneCount][1];
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    int mergeStones(vector<int>& stones, int K) {
        int n = stones.size();

        // Check if it's possible to merge the stones into a single pile
        if ((n - 1) % (K - 1) != 0) {
            return -1;
        }

        // Prefix sums array to calculate the sum of stones between two indexes efficiently
        vector<int> prefixSums(n + 1, 0);
        for (int i = 1; i <= n; ++i) {
            prefixSums[i] = prefixSums[i - 1] + stones[i - 1];
        }

        // Dynamic programming table where f[i][j][k] represents the minimum cost to merge
        // stones[i] to stones[j] into k piles
        vector<vector<vector<int>>> dp(n + 1, vector<vector<int>>(n + 1, vector<int>(K + 1, INT_MAX)));

        // Base case: cost of a single stone (itself) as a pile is 0
        for (int i = 1; i <= n; ++i) {
            dp[i][i][1] = 0;
        }

        // Fill in dp table using bottom-up dynamic programming approach
        for (int l = 2; l <= n; ++l) { // length of the range
            for (int i = 1; i + l - 1 <= n; ++i) {
                int j = i + l - 1;
                for (int k = 2; k <= K; ++k) { // number of piles
                    for (int mid = i; mid < j; mid += K - 1) { // possible split point, ensuring mergeablility
                        dp[i][j][k] = min(dp[i][j][k], dp[i][mid][1] + dp[mid + 1][j][k - 1]);
                    }
                }
                // Cost to merge into 1 pile is the cost to make K piles plus the total sum of the stones
                if (dp[i][j][K] < INT_MAX) {
                    dp[i][j][1] = dp[i][j][K] + prefixSums[j] - prefixSums[i - 1];
                }
            }
        }

        // Final answer: the cost to merge the whole array into one pile
        return dp[1][n][1];
    }
};
```

Typescript Solution

```typescript
// Given an array of stones, returns the minimum cost to merge all stones into one pile
function mergeStones(stones: number[], K: number): number {
    let n: number = stones.length;

    // Check if it's possible to merge the stones into a single pile
    if ((n - 1) % (K - 1) !== 0) {
        return -1;
    }

    // Prefix sums array to calculate the sum of stones between two indices efficiently
    let prefixSums: number[] = new Array(n + 1).fill(0);
    for (let i = 1; i <= n; ++i) {
        prefixSums[i] = prefixSums[i - 1] + stones[i - 1];
    }

    // Dynamic programming table where dp[i][j][k] represents the minimum cost to merge
    // stones from index i to j into k piles
    let dp: number[][][] = Array.from({ length: n + 1 }, () =>
        Array.from({ length: n + 1 }, () => new Array(K + 1).fill(Number.MAX_SAFE_INTEGER))
    );

    // Base case: cost of a single stone (itself) as one pile is 0
    for (let i = 1; i <= n; ++i) {
        dp[i][i][1] = 0;
    }

    // Fill in the dp table using a bottom-up dynamic programming approach
    for (let length = 2; length <= n; ++length) {
        for (let i = 1; i + length - 1 <= n; ++i) {
            let j = i + length - 1;
            for (let piles = 2; piles <= K; ++piles) {
                for (let mid = i; mid < j; mid += K - 1) {
                    dp[i][j][piles] = Math.min(dp[i][j][piles], dp[i][mid][1] + dp[mid + 1][j][piles - 1]);
                }
            }
            // Cost to merge into one pile is the cost to make K piles plus the total sum of the stones
            if (dp[i][j][K] < Number.MAX_SAFE_INTEGER) {
                dp[i][j][1] = dp[i][j][K] + prefixSums[j] - prefixSums[i - 1];
            }
        }
    }

    // The final answer is the cost to merge the entire array into one pile
    return dp[1][n][1];
}
```

Time and Space Complexity

The given Python code is designed to solve a problem on merging stones with certain constraints. Here's the analysis of the time complexity and space complexity of the code:

Time Complexity

The time complexity is influenced by several nested loops and the independent computation performed within them. Here's a breakdown:

1. The first loop initializes the `f` list with a single loop running for `n`, resulting in `O(n)` complexity.

2. Then, there's a triple-nested loop structure where:

   - The outermost loop runs for `n` times, indicating the different lengths `l` that the subproblems can take.
   - The second loop also runs up to `n` times, designating the starting point `i` of a subproblem.
   - The third loop iterates `K` times, which corresponds to the number of piles to merge.
   - Inside the third loop, there is an additional loop running for `j-i` times, which in the worst case is `O(n)`.

The innermost computation `f[i][j][k] = min(f[i][j][k], f[i][h][1] + f[h + 1][j][k - 1])` happens in constant time `O(1)`, since it's just a minimization of pre-calculated values and arithmetic operations.

Combining all these factors, the resulting time complexity is `O(n^3 * K * n) = O(n^4 * K)`.

Space Complexity

The space complexity is influenced by the storage requirements of the `f` list, which is a 3D list of dimensions `(n + 1) x (n + 1) x (K + 1)`. This results in a space complexity of `O(n^2 * K)`.

There are additional variables and a prefix sum array `s`, but these have significantly lower space costs `O(n)` compared to the `f` list, so they do not affect the overall space complexity. Thus, the dominating term remains `O(n^2 * K)`.