1478. Allocate Mailboxes

Hard Array Math Dynamic Programming Sorting Leetcode Link

Problem Description

The problem entails finding a way to place `k` mailboxes on a street that has houses located at various points, represented by the array `houses`. The goal is to minimize the total distance that each house's occupants must travel to reach their nearest mailbox. The array `houses[i]` denotes the location of the `i`-th house on the street. The key is to allocate these `k` mailboxes in a manner that results in the smallest possible sum of distances from each house to its closest mailbox. The challenge is to accomplish this while ensuring that the computed answer is small enough to be represented by a 32-bit integer.

Intuition

To solve this problem, first, we can think about the scenario with only one mailbox (`k=1`). The best position for a single mailbox that minimizes the total distance would be the median of the house locations since the median minimizes the sum of absolute deviations. This situation becomes more complex as `k` increases.

For the general case with `k` mailboxes, we can sort the house locations for more straightforward analysis and applications of dynamic programming (DP). We can treat it as a DP problem, where the state `dp[i][j]` represents the minimum total distance for the first `i` houses when `j` mailboxes are used.

There are two main components of the solution. The first is to calculate the pairwise distance cost `g[i][j]` which represents the total distance if a single mailbox served the houses from `i` to `j`. This approach leverages the fact that if houses are served by a single mailbox, the optimal location is their median, as stated above, and any additional houses paired to this mailbox will add to this distance the difference between the locations.

The second component involves using DP. We initialize our DP table `f[][]` considering the first element with the cost with one mailbox and then applying the optimal substructure property of DP. We calculate `f[i][j]`, the minimum distance for the first `i` houses with `j` mailboxes, by checking for all possible positions where the `j-1` mailboxes could have been placed before the `i`-th house. This would be `f[p][j-1] + g[p+1][i]`. We choose the minimum of these to find the optimal distance.

The Python code presented defines the function `minDistance` to compute the minimum total distance using this DP approach and returns the value of `f[-1][k]`, which represents the minimum total distance using `k` mailboxes for all houses.

Solution Approach

The solution to this problem utilizes dynamic programming (DP), a method for solving complex problems by breaking them down into simpler subproblems. It involves defining a function or table that keeps track of the results of subproblems to avoid redundant computations. Here's a step-by-step breakdown of the implementation used in the reference solution provided:

1. **Sorting**: The first step in the implementation is to sort the input array `houses`. Sorting the houses helps to easily calculate the distance between any two houses and simplify the problem.

2. **Pre-Calculating Pairwise Distance Costs**: The matrix `g` is populated such that `g[i][j]` holds the total distance from the optimal median mailbox to all houses from `houses[i]` to `houses[j]`. This pre-calculation is pivotal to the efficiency of the algorithm, as it allows quickly accessing the cost of placing a single mailbox for any segment of consecutive houses.

3. **Dynamic Programming Setup**: A 2D array `f` is set up, where each cell `f[i][j]` represents the minimal total distance from the first `i` houses with `j` mailboxes placed optimally. This DP array is initialized with infinity, `inf`, values to represent that, initially, the minimal total distance has not been calculated.

4. **Filling the DP Table**: The table `f` is filled using two nested loops. The outer loop goes over the houses, and the inner loop goes over the possible number of mailboxes. At `f[i][1]`, which represents just 1 mailbox for the first `i` houses, the distance is set as the optimal distance for a single mailbox (`g[0][i]`).

5. **Optimal Substructure**: For states where more than one mailbox is used (`j > 1`), another nested loop is used to go over all potential previous placements of mailboxes (`p`). For each placement, the algorithm checks the result of having the last mailbox serve houses from `p+1` to `i` (`f[p][j - 1] + g[p + 1][i]`) and updates `f[i][j]` with the minimum value found.

6. **Final Result**: Finally, the value `f[n-1][k]` will contain the minimal total distance with `k` mailboxes for all houses, as the last row of the dynamic programming table `f` represents all houses and the `k`th column represents using all `k` mailboxes.

In this way, the algorithm ensures that each subproblem is solved optimally, and the solutions of subproblems are combined to solve larger problems. This DP approach, along with careful pre-calculation of distances and optimal substructure exploitation, enables the solution to find the minimal total distance with `k` mailboxes efficiently.

Example Walkthrough

Let's consider a small example to illustrate the solution approach. Suppose we have an array of house locations `houses = [1, 4, 8, 10, 20]` and we want to place `k = 2` mailboxes.

Following the steps mentioned in the solution approach:

1. **Sorting**: The `houses` array is already sorted.

2. **Pre-Calculating Pairwise Distance Costs**: We will create a matrix `g` to store the total distance with a single mailbox for every segment of houses.

   To fill `g[i][j]`:

   - The distance when `i == j` is 0 because if there is only one house, no travel is needed.
   - For `i < j`, calculate the median house and the distance to it from each house `i` to `j`.

   In our example:

   - `g[0][0]`, `g[1][1]`, `g[2][2]`, `g[3][3]`, and `g[4][4]` will be 0.
   - For segment `[1, 4]`, the median is at `4` (median of [1, 4]), so `g[0][1]` is `3` (absolute difference |4-1|).
   - Calculate similarly for other segments, for example, `g[0][2]`, which is the segment `[1, 4, 8]`, where the median is `4`, so the total distance is `7` (|4-1| + |8-4|).
3. **Dynamic Programming Setup**: We create a 2D array `f` with `n` rows and `k+1` columns initialized to infinity. `f[i][j]` will eventually represent the minimal total distance for the first `i+1` houses with `j` mailboxes.

4. **Filling the DP Table**: We begin to fill the table `f`.

   For one mailbox and `i` houses, `f[i][1]` will be the value of `g[0][i]`:

   - `f[0][1] = g[0][0] = 0`
   - `f[1][1] = g[0][1] = 3`
   - `f[2][1] = g[0][2] = 7`, and so on.
5. **Optimal Substructure**: Now we update `f` for more than one mailbox.

   We are aiming to fill `f[4][2]` (for all 5 houses with 2 mailboxes). One mailbox will serve houses `0` to `p`, and the second one will serve `p+1` to `4`.

   For `p=0`, the cost would be `f[0][1] + g[1][4]`. For `p=1`, the cost would be `f[1][1] + g[2][4]`. For `p=2`, the cost would be `f[2][1] + g[3][4]`, and so on.

   Select the minimum of these combinations.

6. **Final Result**: The result we are looking for is the value in `f[4][2]`, which will contain the minimal total distance with 2 mailboxes for all 5 houses.

By computing the subproblems carefully and building upon them, we find the minimal total distance for any given `k` and the list of houses.

Python Solution

```python
from typing import List

class Solution:
    def minDistance(self, houses: List[int], k: int) -> int:
        # Sort the houses to ensure they are in increasing order of their locations
        houses.sort()
        n = len(houses)

        # Precompute the cost of putting one mailbox for the houses in range [i:j+1]
        # The cost is the sum of distances from houses[i] to each house in the range.
        costs = [[0] * n for _ in range(n)]
        for i in range(n - 2, -1, -1):
            for j in range(i + 1, n):
                # The cost is built up from the previous smaller range by adding the
                # distance to the new endpoint house[j].
                costs[i][j] = costs[i + 1][j - 1] + houses[j] - houses[i]

        # Initialize the dp array with infinite cost. dp[i][j] represents the minimum
        # total distance from houses[0:i+1] when j mailboxes are used.
        dp = [[float('inf')] * (k + 1) for _ in range(n)]

        # Calculate minimum distances
        for i in range(n):
            # The cost with only one mailbox up to house[i]
            dp[i][1] = costs[0][i]

            # j represents the number of mailboxes to use
            for j in range(2, min(k + 1, i + 2)):
                # Try placing the j-th mailbox after each house[p] and remember minimum total distance
                for p in range(i):
                    dp[i][j] = min(dp[i][j], dp[p][j - 1] + costs[p + 1][i])

        # Return the minimum distance when all houses are covered by k mailboxes
        return dp[-1][k]

# Example usage:
solution = Solution()
print(solution.minDistance([1, 4, 8, 10, 20], 3)) # Should output the minimum distance

```

Java Solution

```java
class Solution {
    public int minDistance(int[] houses, int heaters) {
        // Sort the houses array
        Arrays.sort(houses);
        int numOfHouses = houses.length;

        // Create a distance matrix where g[i][j] is the total distance for grouping houses[i]..houses[j]
        int[][] distanceMatrix = new int[numOfHouses][numOfHouses];
        for (int i = numOfHouses - 2; i >= 0; --i) {
            for (int j = i + 1; j < numOfHouses; ++j) {
                // Use previously computed values to build up the distance
                distanceMatrix[i][j] = distanceMatrix[i + 1][j - 1] + houses[j] - houses[i];
            }
        }

        // Create a dp table where f[i][j] is the minimum distance for the first i houses with j heaters
        int[][] dp = new int[numOfHouses][heaters + 1];
        // Define an 'infinity' value for initial comparison
        final int INF = 1 << 30;

        // Initialize the dp array with infinity
        for (int[] row : dp) {
            Arrays.fill(row, INF);
        }

        for (int i = 0; i < numOfHouses; ++i) {
            // The distance for the first i houses and 1 heater is the total distance of range [0, i]
            dp[i][1] = distanceMatrix[0][i];
            // Check for all feasible heater positions
            for (int j = 2; j <= heaters && j <= i + 1; ++j) {
                for (int p = 0; p < i; ++p) {
                    // Find the best position for the j-th heater, by recursively checking the minimum
                    // distance we found when we placed the last heater at house p
                    dp[i][j] = Math.min(dp[i][j], dp[p][j - 1] + distanceMatrix[p + 1][i]);
                }
            }
        }
        // Answer is the minimum distance to place heaters for all houses
        return dp[numOfHouses - 1][heaters];
    }
}

```

C++ Solution

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

class Solution {
public:
    int minDistance(vector<int>& houses, int k) {
        // The size of the houses array.
        int n = houses.size();
        // Sorting houses in non-descending order.
        sort(houses.begin(), houses.end());
        // cost[i][j] will hold the cost of having one mailbox for the houses[i...j].
        vector<vector<int>> cost(n, vector<int>(n, 0));

        // Pre-calculate the cost of each range of houses.
        for (int i = n - 2; i >= 0; --i) {
            for (int j = i + 1; j < n; ++j) {
                cost[i][j] = cost[i + 1][j - 1] + houses[j] - houses[i];
            }
        }

        // dp[i][j] will hold the minimum total distance for houses[0...i] with j mailboxes.
        vector<vector<int>> dp(n, vector<int>(k + 1, INT_MAX));

        // Initialize the dp array for the case with 1 mailbox.
        for (int i = 0; i < n; ++i) {
            dp[i][1] = cost[0][i];
            // Check for the minimum distance using 1 to k mailboxes.
            for (int j = 1; j <= k && j <= i + 1; ++j) {
                // Calculate the minimum total distance by comparing all possible partitions.
                for (int p = 0; p < i; ++p) {
                    dp[i][j] = min(dp[i][j], dp[p][j - 1] + cost[p + 1][i]);
                }
            }
        }
        // The answer is the min total distance for all houses with k mailboxes.
        return dp[n - 1][k];
    }
};

```

Typescript Solution

```typescript
function minDistance(houses: number[], k: number): number {
    // The size of the houses array.
    let n: number = houses.length;
    // Sorting houses in non-descending order.
    houses.sort((a, b) => a - b);
    // cost[i][j] will hold the cost of having one mailbox for the houses[i...j].
    let cost: number[][] = Array.from(Array(n), () => Array(n).fill(0));

    // Pre-calculate the cost of each range of houses.
    for (let i = n - 2; i >= 0; --i) {
        for (let j = i + 1; j < n; ++j) {
            cost[i][j] = cost[i + 1][j - 1] + houses[j] - houses[i];
        }
    }

    // dp[i][j] will hold the minimum total distance for houses[0...i] with j mailboxes.
    let dp: number[][] = Array.from(Array(n), () => Array(k + 1).fill(Number.MAX_SAFE_INTEGER));

    // Initialize the dp array for the case with 1 mailbox.
    for (let i = 0; i < n; ++i) {
        dp[i][1] = cost[0][i];
        // Check for the minimum distance using 1 to k mailboxes.
        for (let j = 1; j <= k && j <= i + 1; ++j) {
            // Calculate the minimum total distance by comparing all possible partitions.
            for (let p = 0; p < i; ++p) {
                dp[i][j] = Math.min(dp[i][j], dp[p][j - 1] + cost[p + 1][i]);
            }
        }
    }
    // The answer is the minimum total distance for all houses with k mailboxes.
    return dp[n - 1][k];
}

```

Time and Space Complexity

Time Complexity

The time complexity of the code can be analyzed by looking at the nested loops and the operations performed within each:

1. The `houses.sort()` call has a time complexity of `O(n log n)`, where `n` is the number of houses, since sorting is typically implemented as a comparison sort.

2. The next double loop constructs the `g` matrix, where for each element `g[i][j]`, it calculates the cost of placing one mailbox for the houses ranging from index `i` to `j`. This part of the code has two nested loops ranging from `i` to `n` and `j` from `i+1` to `n`, contributing a time complexity of `O(n²)`.

3. The last double loop populates the `f` matrix, which represents the minimum cost of placing `j` mailboxes for the first `i` houses. The innermost loop is also ranging up to `n`, making the time complexity of these loops `O(n² * k)`, where `k` is the number of mailboxes.

Overall, the dominant part of the time complexity comes from the last nested loop, and therefore, the overall time complexity is `O(n² * k)`.

Space Complexity

The space complexity can be determined from the space allocated for the dynamic programming matrices:

1. The `g` matrix is an `n x n` matrix, which results in a space complexity of `O(n²)`.

2. Similarly, the `f` matrix is an `n x (k + 1)` matrix, but since `k` could be at most `n`, the space complexity in the worst case would also be `O(n²)`.

Taking both matrices into account, the overall space complexity of the algorithm is `O(n²)`.