2361. Minimum Costs Using the Train Line

Hard Array Dynamic Programming Leetcode Link

Problem Description

Imagine you're in a city with a train system that has two different types of routes, the regular and express routes, covering the same series of stops from stop 0 to stop n. Each segment between two consecutive stops has a cost associated with it, depending on which route you take: the cost is outlined in two arrays, `regular` for the regular route and `express` for the express route.

While on the regular route, if you decide to switch to the express route, it incurs an additional cost, specified by the `expressCost`. However, transferring back to the regular route from the express route doesn't cost anything. Also, whenever you decide to switch to the express route, you have to pay the `expressCost` every single time.

The goal is to calculate the minimum cost to reach each stop from stop 0 by possibly switching between routes strategically to minimize your total cost. The output should be an array representing the minimum cost to reach each stop, starting from stop 1 up to stop n (1-indexed).

Intuition

To find the minimum cost to reach each stop, it's necessary to track the cost of two scenarios at any stop: staying on the regular route, and being on the express route. At each stop, you can stay on the current route or transfer to the other one (with the possibility of an additional `expressCost` if switching to the express route).

We begin by initializing two variables, `f` and `g`:

- `f` represents the minimum cost of reaching the current stop via the regular route.
- `g` represents the minimum cost of reaching the current stop via the express route.

Starting from stop 0, we iterate over each stop. For each stop, we calculate the new costs `ff` and `gg`.

- `ff` represents the new cost of reaching the next stop on the regular route. This cost is the minimum between staying on the regular route (`f + a`, where `a` is the regular cost of the next segment) or switching from the express route to the regular route (`g + a`, since switching back to regular is free).
- `gg` represents the new cost of reaching the next stop on the express route. This is the minimum between switching from the regular route (`f + expressCost + b`, where `b` is the express cost of the next segment) or staying on the express route (`g + b`).

After calculating `ff` and `gg`, we update `f` and `g` respectively. The minimum of `f` and `g` is then stored in the cost array as the minimum cost to reach the current stop. This way, the process accounts for the possibility that it might be cheaper to switch to the express route or to stay on the regular route at different points along the journey.

The final array represents the least amount of money you need to spend to reach each stop, and it's built progressively as we loop through all stops.

Solution Approach

The solution to this problem adopts a dynamic programming approach, which is a method for efficiently solving problems that have overlapping subproblems and optimal substructure properties by breaking them down into simpler subproblems.

In this case, the optimal cost to reach a certain stop can be calculated based on the optimal costs to reach previous stops. The solution uses two variables, `f` and `g`, to keep track of the accumulated costs of the regular and express routes up until the current stop. These accumulate the total cost of reaching the current stop on their respective routes, considering all the previous decisions made (to switch routes or not).

The solution involves a loop that iterates through each pair of costs `(a, b)` from the `regular` and `express` lists. Here is the breakdown of the loop's execution:

- For each index `i` (1-indexed), we calculate `ff` and `gg` which are the tentative costs for the next stop on the regular and express routes respectively.
  - `ff` is calculated as `min(f + a, g + a)`. Here, `f + a` is the cost if we continue on the regular route, and `g + a` is the cost if we transfer from the express to the regular route at this stop, which comes free of charge.
  - `gg` is calculated as `min(f + expressCost + b, g + b)`. The `f + expressCost + b` part computes the cost if we switch to the express route from the regular route, which includes the `expressCost`, and `g + b` computes the cost if we keep going on the express route.
- After calculating `ff` and `gg`, the variables `f` and `g` are updated with the new values `ff` and `gg` respectively. This is done because we've now accumulated the cost to reach the next stop (i), and we need to keep our accumulation up to date.
- The minimum of `f` and `g` at stop `i` (which has become our new stop `i - 1` for the next iteration of the loop) is saved into the cost array, as it represents the minimum cost to reach this stop from the start.

By using this approach, the solution iteratively builds up the minimum cost to reach each stop, considering both routes and the potential to switch between them along the way. The final cost array captures the minimum accumulated costs to reach every stop using the optimal strategy.

The algorithm assumes the `inf` (infinity) value is a large enough number to represent an impossible high cost that would not be considered a minimum in any practical scenario, which is used to initialize the cost of using the express route before any stops have been reached via the express route. Moreover, the solution benefits from the use of the `enumerate` function in Python, which allows iterating over both the indices and the elements of the costs lists simultaneously.

Example Walkthrough

Let's say we have a city with 4 stops (stop 0 to stop 3), and the costs for the regular and express routes are given by `regular = [1, 3, 2]` and `express = [4, 1, 2]` respectively. Assume the additional cost to switch to the express route is represented by the variable `expressCost = 2`. According to the problem, switching back to the regular route is free. We want to calculate the minimum cost to reach each stop from stop 0.

Initialization:

- f = 0 (cost of reaching the first stop on the regular route is always 0 since we start here)
- g = inf (we haven't reached any stops via the express route yet)

Step-by-step explanation:

- To reach stop 1:
  - Regular route: `ff = min(f + regular[0], g + regular[0]) = min(0 + 1, inf + 1) = 1`
  - Express route: `gg = min(f + expressCost + express[0], g + express[0]) = min(0 + 2 + 4, inf + 4) = 6`
  - Update f and g: `f = 1`, `g = 6`. The minimum cost to reach stop 1 is `min(f, g) = 1`.
- To reach stop 2:
  - Regular route: `ff = min(f + regular[1], g + regular[1]) = min(1 + 3, 6 + 3) = 4`
  - Express route: `gg = min(f + expressCost + express[1], g + express[1]) = min(1 + 2 + 1, 6 + 1) = 4`
  - Update f and g: `f = 4`, `g = 4`. The minimum cost to reach stop 2 is `min(f, g) = 4`.
- To reach stop 3:
  - Regular route: `ff = min(f + regular[2], g + regular[2]) = min(4 + 2, 4 + 2) = 6`
  - Express route: `gg = min(f + expressCost + express[2], g + express[2]) = min(4 + 2 + 2, 4 + 2) = 6`
  - Update f and g: `f = 6`, `g = 6`. The minimum cost to reach stop 3 is `min(f, g) = 6`.

Conclusion:

The final array containing the minimum costs to reach stops 1 to 3 is `[1, 4, 6]`. This represents the least amount of money needed to reach each stop when choosing the optimal route at each step of the journey.

Python Solution

```python
from typing import List

class Solution:
    def minimum_costs(
        self, regular: List[int], express: List[int], express_cost: int
    ) -> List[int]:
        n = len(regular)  # Total number of days
        # Initialize cost for regular and express as zero for day 0
        regular_cost = 0
        express_cost_total = float('inf')  # Set initial express cost to infinity
        costs = [0] * n  # Initialize the list to store minimum costs for each day

        # Iterate through each day's regular and express costs
        for i, (regular_day_cost, express_day_cost) in enumerate(zip(regular, express), 1):
            # Calculate the minimum cost to take regular path on the current day
            min_cost_regular = min(regular_cost + regular_day_cost, express_cost_total + regular_day_cost)
            # Calculate the minimum cost to take express path on the current day, including the express_cost
            min_cost_express = min(regular_cost + express_cost + express_day_cost, express_cost_total + express_day_cost)

            # Update the total regular and express costs to reflect today's costs
            regular_cost, express_cost_total = min_cost_regular, min_cost_express

            # Store the minimum of the two costs in the costs array
            costs[i - 1] = min(regular_cost, express_cost_total)

        return costs  # Return the minimum costs for each day

```

Java Solution

```java
class Solution {
    public long[] minimumCosts(int[] regular, int[] express, int expressCost) {
        // Determine the number of days based on the regular array length
        int numberOfDays = regular.length;

        // f represents the minimum cost using regular routes up to day i
        long minCostRegular = 0;
        // g represents the minimum cost using express routes up to day i (initially set to a large number)
        long minCostExpress = Long.MAX_VALUE / 2; // Long.MAX_VALUE / 2 to avoid overflow in future calculations

        // Array to store the minimum cost for each day
        long[] cost = new long[numberOfDays];

        // Iterate through each day to find minimum costs
        for (int i = 0; i < numberOfDays; ++i) {
            // Cost of regular and express route for the current day i
            int costRegular = regular[i];
            int costExpress = express[i];

            // Calculating the minimum cost if using the regular route on day i
            long newMinCostRegular = Math.min(minCostRegular + costRegular, minCostExpress + costRegular);

            // Calculating the minimum cost if using the express route on day i, with expressCost included
            long newMinCostExpress = Math.min(minCostRegular + expressCost + costExpress, minCostExpress + costExpress);

            // Update the minimum costs for regular and express
            minCostRegular = newMinCostRegular;
            minCostExpress = newMinCostExpress;

            // Store the minimum of the two in the cost array for the day i
            cost[i] = Math.min(minCostRegular, minCostExpress);
        }

        return cost; // Return the array containing minimum costs for each day
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <algorithm>

class Solution {
public:
    // Calculates the minimum costs for each station using either regular or express service.
    vector<long long> minimumCosts(vector<int>& regular, vector<int>& express, int expressCost) {
        int n = regular.size(); // Number of stations
        long long costRegular = 0; // Minimum cost using regular service up to current station
        long long costExpress = LLONG_MAX; // Minimum cost using express service up to current station, initialized with max value
        vector<long long> minCosts(n); // Stores the minimum cost for each station

        // Iterate through each station
        for (int i = 0; i < n; ++i) {
            int regularCost = regular[i]; // Cost of regular service at current station
            int expressCostAtStation = express[i]; // Cost of express service at current station

            // Calculate the new minimum cost of reaching the current station via regular service
            long long newCostRegular = std::min(costRegular + regularCost, costExpress + regularCost);
            // Calculate the new minimum cost of reaching the current station via express service
            long long newCostExpress = std::min(costRegular + expressCost + expressCostAtStation,
                                                costExpress + expressCostAtStation);

            // Update the minimum costs for regular and express service at current station
            costRegular = newCostRegular;
            costExpress = newCostExpress;

            // The minimum cost for the current station is the smaller of the two minimum costs
            minCosts[i] = std::min(costRegular, costExpress);
        }

        return minCosts; // Return the vector of minimum costs for each station
    }
};

```

Typescript Solution

```typescript
function minimumCosts(regularCosts: number[], expressCosts: number[], expressLaneCost: number): number[] {
    // The number of days
    const numDays = regularCosts.length;

    // Minimum accumulated cost using the regular lane
    let minRegularCost = 0;

    // Minimum accumulated cost using the express lane (initialized to a large number)
    let minExpressCost = Number.MAX_SAFE_INTEGER;

    // Array to store the minimum cost for each day
    const totalCosts: number[] = new Array(numDays).fill(0);

    // Iterate over each day
    for (let day = 0; day < numDays; ++day) {
        // Cost of using the regular lane on the current day
        const currentRegularCost = regularCosts[day];

        // Cost of using the express lane on the current day
        const currentExpressCost = expressCosts[day];

        // Compute the minimum cost for the current day using the regular lane
        const newMinRegularCost = Math.min(minRegularCost + currentRegularCost, minExpressCost + currentRegularCost);

        // Compute the minimum cost for the current day using the express lane
        const newMinExpressCost = Math.min(minRegularCost + expressLaneCost + currentExpressCost, minExpressCost + currentExpressCost

        // Update minimum costs for both lanes
        minRegularCost = newMinRegularCost;
        minExpressCost = newMinExpressCost;

        // Record the minimum total cost for the current day
        totalCosts[day] = Math.min(minRegularCost, minExpressCost);
    }

    // Return the array of minimum total costs for each day
    return totalCosts;
}

```

Time and Space Complexity

Time Complexity

The provided code snippet goes through the lists `regular` and `express` exactly once, performing a constant number of operations for each element. The `enumerate` function is used to iterate over both lists simultaneously, and for each element, a comparison and a few arithmetic operations are conducted. These operations are constant time, and since the iteration is done once per element in the list, the time complexity is **`O(n)`**, where `n` is the length of the `regular` list (and `express` list, as they are of the same length).

Space Complexity

The space complexity of the code is primarily dependent on the `cost` list that is being created to store the result at each step. Since this list is the same length as the input lists (`regular` and `express`), the space required by the `cost` list is **`O(n)`**.

The rest of the variables (`f`, `g`, `a`, `b`, `ff`, `gg`, and `i`) use a constant amount of space, so they do not add to the complexity in terms of `n`. The constants `inf` (representing infinity) and `expressCost` are also not dependent on `n`, so the overall space complexity remains linear with respect to the length of the inputs.

In summary, the space complexity is also **`O(n)`**.