746. Min Cost Climbing Stairs

Easy Array Dynamic Programming

Problem Description

The problem presents a scenario where you have a staircase with each step associated with a certain cost given in an integer array called `cost`. The goal is to figure out the minimum total cost required to reach the top of the staircase. The unique aspect of this problem is that you can choose between taking one step or two steps at a time after paying the corresponding cost. Furthermore, you have the flexibility to start from either the first step (`cost[0]`) or the second step (`cost[1]`). The top of the staircase is considered to be one step past the last step, hence you need to decide step by step, which is the cheaper option to achieve the minimum cost to reach the top.

Intuition

The intuition behind the solution comes from dynamic programming concepts, specifically the idea of solving smaller subproblems and building up to the final solution. Since at each step you can make a decision based on the previous two steps, we can work our way up the staircase from bottom to top, calculating the minimum cost needed to reach each step.

The critical observation is that the minimum cost to reach a given step only depends on the minimum costs to reach the two preceding steps. We don't need to remember the path we took, only the costs. This leads us to understand that at each step `i`, the minimum cost (`minCost[i]`) is the minimum of the cost of getting to the previous step plus the cost associated with the previous step (`minCost[i-1] + cost[i-1]`), or the cost of getting to the step before it plus the cost associated with two steps behind (`minCost[i-2] + cost[i-2]`).

However, to optimize space complexity, we realize that we don't need to keep an array to store all previous `minCosts`, we can simply use two variables to keep track of these costs as we iterate through the `cost` array. By repeatedly updating these two variables, we ensure that they always represent the minimum costs to reach the last two steps. At the end of the iteration, the second variable (`b` in the code) will contain the minimum cost to reach the top of the staircase.

Solution Approach

The solution uses a bottom-up dynamic programming approach where we solve for the smaller subproblems first and use their results to build up to the complete solution. In this scenario, each step has a subproblem that consists of finding the minimum cost to get to that step.

The algorithm uses two variables, `a` and `b`, to keep track of the minimum costs up to the two previous steps, respectively. These two variables eliminate the need for an additional auxiliary array that would keep track of the minimum costs for all steps, thus saving space. In algorithmic terms, we reduce the space complexity from O(n) to O(1) by using this approach.

Let's walk through the implementation steps referencing the Python code provided:

1. We initialize two variables, `a` and `b`, both set to `0`. These will track the minimum cost to reach the one step behind the current (`a`) and the current step (`b`), respectively.

```
a = b = 0
```

2. We loop through each step in the array from the second step until the end. Since we can either start at step `0` or `1`, we don't need to calculate the cost to reach step `0`.

```
for i in range(1, len(cost)):
```

3. Within the loop, we compute the minimum cost to reach the next step, which is determined by the minimum of the cost to get to the current step plus the cost of the current step, and the cost to get to the previous step plus the cost of the previous step.

```
a, b = b, min(a + cost[i - 1], b + cost[i])
```

Here's what happens in the above line:

- `min(a + cost[i - 1], b + cost[i])`: This expression finds the minimum cost between taking one step from `i-1` or two steps from `i-2` to reach the ith step.
- `a, b = b, ...`: We then update `a` to be the previous value of `b` (since we're moving to the next step, the "current" becomes "previous"), and `b` is updated to the just-calculated minimum cost.

4. Lastly, after exiting the loop, variable `b` holds the minimal cost to reach the top of the staircase. We return `b` as the solution.

```
return b
```

Note that the top of the floor is one step past the end of the array, and the way the loop is constructed ensures that after iterating through the array, `b` will represent the cost of getting to the top, not just to the last step.

Example Walkthrough

Let's walk through an example to illustrate how the solution approach works. Suppose we have the following `cost` array:

```
cost = [10, 15, 20]
```

The cost array indicates that it costs 10 to move to the first step, 15 to move to the second, and 20 to move to the third. We want to find the minimum cost to reach the top of the staircase, which is one step past the last step (the third step in this case). The top of the staircase is not associated with any cost.

1. We start by initializing two variables, `a` and `b`, to 0. These represent the minimum costs to reach the prior step (`a`) and the current step (`b`).

```
a = b = 0
```

2. Now, we will loop through each step in the array starting from the second step. Since the first step can be chosen as a starting point, we do not need to consider the cost at index `0` at the beginning.

```
for i in range(1, len(cost)):
```

At this point, `i = 1`.

3. We compute the minimum cost to the next step.

```
a, b = b, min(a + cost[i - 1], b + cost[i])
```

For `i = 1`:

- `cost[i-1]` is `cost[0]`, which is `10`.
- `cost[i]` is `cost[1]`, which is `15`.
- `a + cost[i - 1]` is `0 + 10 = 10`.
- `b + cost[i]` is `0 + 15 = 15`.
- `min(a + cost[i - 1], b + cost[i])` is `min(10, 15)` which is `10`.

Therefore, `a` becomes `0` and `b` becomes `10`. Now, the minimum cost to reach the second step is `10`.

Continue the loop for `i = 2`.

For `i = 2`:

- `cost[i-1]` is `cost[1]`, which is `15`.
- `cost[i]` is `cost[2]`, which is `20`.
- `a + cost[i - 1]` is `10 + 15 = 15`.
- `b + cost[i]` is `10 + 20 = 30`.
- `min(a + cost[i - 1], b + cost[i])` is `min(15, 30)` which is `15`.

Now, `a` takes the previous value of `b`, and `b` takes the new minimum. So, `a` becomes `10`, and `b` becomes `15`.

4. Since we've reached the end of the array, the `b` value now represents the minimum cost to reach the top of the staircase. Thus, the minimum cost to get to the top is `15`.

```
return b
```

In this example, the cheapest way to reach the top is by starting on the first step with a cost of 10 and then jumping two steps to the top with no additional cost, resulting in a total minimum cost of 10 + 0 = 10. However, due to how the algorithm is structured, `b` ends up as 15 at the end, which is the correct answer because we are looking for the minimal cost to get to the top, which includes the scenario where we start at the second step (`cost[1] = 15`) and take one step to the top. The algorithm effectively accounts for both starting points as per the problem requirements.

Solution Implementation

Python

```
from typing import List

class Solution:
    def minCostClimbingStairs(self, cost: List[int]) -> int:
        # Initialize variables to store the minimum cost of reaching the previous two steps.
        previous_step_cost = current_step_cost = 0

        # Iterate over the cost array starting from the second element,
        # since we can start either from step 0 or step 1.
        for i in range(2, len(cost) + 1):
            # Compute the minimum cost to reach the current step by taking
            # a single step from the previous step or a double step from the one before previous.
            # This compares the cost of reaching the previous step and one step before that,
            # added to the cost of the step that would be taken.
            temp = current_step_cost
            current_step_cost = min(previous_step_cost + cost[i - 2], current_step_cost + (cost[i - 1] if i != len(cost) else 0))

            # Update the previous step cost for the next iteration
            previous_step_cost = temp

        # Return the minimum cost to reach the top of the floor, which is the same as reaching the
        # end of the cost array, since we can either end at the last step or bypass it.
        return current_step_cost
```

Java

```
class Solution {
    // Method to calculate the minimum cost to climb stairs
    public int minCostClimbingStairs(int[] cost) {
        // Two variables to store the minimum cost to reach the step before the current step (prevStep)
        // and the step before the previous step (prevPrevStep).
        int prevPrevStepCost = 0, prevStepCost = 0;

        // Looping through the array from the second element since the cost of climbing from
        // the ground (before the first step) to the first or second step is already provided in 'cost' array.
        for (int i = 2; i <= cost.length; ++i) {
            // Calculate the minimum cost to reach the current step by comparing the cost of
            // taking one step from the previous step and the cost of taking two steps from the step before that.
            int currentStepCost = Math.min(prevStepCost + cost[i - 1], prevPrevStepCost + cost[i - 2]);

            // Update the costs for the next iteration. The previous step becomes the step before the previous step,
            // and the current step becomes the previous step.
            prevPrevStepCost = prevStepCost;
            prevStepCost = currentStepCost;
        }

        // Return the minimum cost to reach the top of the stairs which is either from the last or second-last step.
        return prevStepCost;
    }
}
```

C++

```
// The Solution class provides a method to find the minimum cost to reach the top
// of a staircase, where each step has a certain cost associated with stepping on it.
class Solution {
public:
    /// The minCostClimbingStairs function calculates the minimum cost to reach the top
    /// of the staircase given a vector where each element represents the cost of a step.
    int minCostClimbingStairs(vector<int>& cost) {
        // Initialize two variables to store the minimum cost to reach the current step
        // and the previous step.
        int minCostToCurrent = 0, minCostToPrevious = 0;

        // Iterate through the steps, starting from the second step (first step is index 0).
        for (int i = 2; i <= cost.size(); ++i) {
            // Calculate the minimum cost to reach the current step by comparing:
            // 1. The cost to reach the previous step (one step behind) and step on current step.
            // 2. The cost to reach the second previous step (two steps behind) and skip the previous step.
            int minCostToNext = min(minCostToPrevious + cost[i - 1], minCostToCurrent + cost[i - 2]);

            // Move the 'current' and 'previous' costs forward for the next iteration.
            minCostToPrevious = minCostToCurrent;
            minCostToCurrent = minCostToNext;
        }

        // After the loop, minCostToCurrent holds the minimum cost to reach the top of the stairs
        // since one can step on either the last step or one step before last, so we do not need
        // to specifically handle the top step.
        return minCostToCurrent;
    }
};
```

TypeScript

```
/**
 * Computes the minimum cost to climb stairs. You can either climb one or two steps at a time.
 * After you pay the cost, you can either start from the step with index 0, or the step with index 1.
 *
 * @param {number[]} cost - The cost of each step.
 * @return {number} - The minimum cost to reach the top of the floor.
 */
function minCostClimbingStairs(cost: number[]): number {
    // Initialize two variables to store the cumulative cost from two steps back (prevCost)
    // and one step back (currentCost), starting at the base, which is zero cost.
    let prevCost = 0;
    let currentCost = 0;

    // Iterate through the array of costs, starting from the second,
    // since the first step's cost is already accounted for by the initial costs of 0.
    for (let i = 2; i <= cost.length; ++i) {
        // Calculate the new cost by taking the minimum of the previous two costs
        // plus the cost of either taking one step from the previous step or two steps back.
        let newCost = Math.min(prevCost + cost[i - 2], currentCost + cost[i - 1]);

        // Update previous and current costs for the next iteration.
        prevCost = currentCost;
        currentCost = newCost;
    }

    // After iterating through all the steps, the currentCost holds the minimum cost
    // to reach the top of the floor.
    return currentCost;
}
```

```
from typing import List

class Solution:
    def minCostClimbingStairs(self, cost: List[int]) -> int:
        # Initialize variables to store the minimum cost of reaching the previous two steps.
        previous_step_cost = current_step_cost = 0

        # Iterate over the cost array starting from the second element,
        # since we can start either from step 0 or step 1.
        for i in range(2, len(cost) + 1):
            # Compute the minimum cost to reach the current step by taking
            # a single step from the previous step or a double step from the one before previous.
            # This compares the cost of reaching the previous step and one step before that,
            # added to the cost of the step that would be taken.
            temp = current_step_cost
            current_step_cost = min(previous_step_cost + cost[i - 2], current_step_cost + (cost[i - 1] if i != len(cost) else 0))

            # Update the previous step cost for the next iteration
            previous_step_cost = temp

        # Return the minimum cost to reach the top of the floor, which is the same as reaching the
        # end of the cost array, since we can either end at the last step or bypass it.
        return current_step_cost
```

Time and Space Complexity

The given code is a dynamic programming approach to solve the minimum cost climbing stairs problem. Let's analyze the time and space complexity.

Time Complexity:

The time complexity of the code is determined by the loop that iterates over the range `1` to `len(cost)`. Since it goes through each step once, the time complexity is `O(n)`, where `n` is the number of elements in the cost list.

Space Complexity:

As for the space complexity, the code uses two variables `a` and `b` to keep track of the previous two costs only, not needing additional space that is dependent on the input size. Thus, the space complexity is `O(1)`, constant space, as it does not grow with the size of the input.