135. Candy

Hard Greedy Array

Problem Description

In this LeetCode problem, we are tasked with distributing candies to children in a line based on their ratings. Each child in the line has a rating value, and our goal is to allocate candies following two rules:

1. Every child must receive at least one candy.
2. A child with a higher rating than their immediate neighbors must receive more candies.

We are required to determine the minimum number of candies that we need to distribute to satisfy the above rules.

Intuition

To solve this problem, we can consider the two rules that govern how we should distribute candies. The first rule is straightforward: we can start by giving each child one candy. The second rule requires a bit more thought because we need to ensure that each child with a higher rating than their neighbor receives more candies.

One approach is to evaluate how each child compares to their neighbors from both the left and right sides separately, using two arrays, `left` and `right`. Here's our process for arriving at the solution:

1. We create two arrays, `left` and `right`, with the same length as the `ratings` array, and initialize all elements to 1, which accounts for the first rule that every child receives at least one candy.
2. We scan through the `ratings` array from left to right, comparing each child's rating to their previous neighbor. If a child's rating is higher than that of the previous child, we increment the value in the corresponding `left` array to one plus the value for the previous child.
3. Next, we'll need to consider the `right` neighbors, too. We scan through the `ratings` array from right to left this time. If we find that a child's rating is higher than that of the next child, we increment the `right` array's corresponding value to one plus the value for the next child.
4. We then take the maximum value between the corresponding elements in `left` and `right` for each child. This step ensures that the requirements for both neighbors are satisfied.
5. Lastly, we sum up the maximum values that we've obtained for each child, giving us the minimum number of candies needed to distribute to all the children while meeting the rules.

This algorithm effectively caters to all the possible scenarios and ensures that each child gets the correct number of candies based on their ratings and allows us to calculate the minimum total number of candies required.

Solution Approach

The solution can be broken down into a few steps using simple algorithms and data structures pattern:

1. **Initial Setup**: Two arrays named `left` and `right` of the same length as `ratings` are created. Both are filled with `1`s, which ensures that each child gets at least one candy, adhering to the first rule.

2. **Left-to-Right Pass**: Starting from the second child (at index 1), we iterate over the `ratings` array. If we find that the current child has a higher rating than the one to the left (at index `i-1`), we give the current child more candies than the left one by incrementing the `left[i]` value to `left[i-1] + 1`.

```
for i in range(1, len(ratings)):
    if ratings[i] > ratings[i - 1]:
        left[i] = left[i - 1] + 1
```

3. **Right-to-Left Pass**: In the next step, we start from the second-to-last child and go back to the first child. Similar to the left-to-right pass, if the current child has a higher rating than the one to the right (at index `i+1`), we perform a similar increment on the `right[i]`.

```
for i in range(len(ratings) - 2, -1, -1):
    if ratings[i] > ratings[i + 1]:
        right[i] = right[i + 1] + 1
```

4. **Combining Results**: Now that we have gone through the ratings and considered each child in comparison to their left and right neighbors, we need to combine results. The final candy count for each child should be the maximum value out of `left[i]` and `right[i]`. This is because a child's final candy count must satisfy both its left and right neighbors.

```
total_candies = sum(max(left[i], right[i]) for i in range(len(ratings)))
```

5. **Final Result**: The sum of the maximum of the `left` and `right` values for each child gives us the minimum number of candies we need to distribute.

By using two arrays, we are applying a form of Dynamic Programming where we store intermediate results ("the number of candies needed considering the left or right neighbor") to solve overlapping subproblems ("the number of candies a child should get"). The solution is efficient both in terms of time and space complexity and ensures that we adhere to the problem constraints while obtaining the correct minimum total of candies.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach described above.

Consider the ratings array `[1, 0, 2]`. We need to distribute candies to the children in such a way that all conditions are met.

1. **Initial Setup**: We initialize the `left` and `right` arrays to `[1, 1, 1]` because each child must get at least one candy according to the first rule.
2. **Left-to-Right Pass**: We iterate through the `ratings` array starting from the second element. Comparing each element with its left neighbor:
   - For the second child (rating `0`), since `0 < 1`, we do not change `left[1]`.
   - For the third child (rating `2`), since `2 > 0`, we increment `left[2]` to `left[1] + 1`, so `left` becomes `[1, 1, 2]`.
3. **Right-to-Left Pass**: We iterate from right to left, starting from the second-to-last element:
   - For the second child (rating `0`), since `0 < 2`, we increment `right[1]` to `right[2] + 1`, so `right` becomes `[1, 2, 1]`.
   - For the first child (rating `1`), since `1 > 0`, no change is needed as `right[0]` is already greater than `right[1]`.
4. **Combining Results**: We take the maximum value for each index between `left` and `right`.
   - The combined array will be `max([1,1],[1,2]), max([1,2],[1,1]), max([2,1],[2,1])`, which results in `[1, 2, 2]`.
5. **Final Result**: We sum up the values of the combined array: `1 + 2 + 2 = 5`. Therefore, we need a minimum of 5 candies to distribute to the children following the rules.

By following this step-by-step process for each child's rating, we can satisfy both conditions and compute the minimum number of candies required, which in this example is 5.

Solution Implementation

Python

```
from typing import List

class Solution:
    def candy(self, ratings: List[int]) -> int:
        # Length of the ratings list
        num_children = len(ratings)

        # Initialize lists to represent the minimum candies for each child from left and right perspectives
        candies_from_left = [1] * num_children
        candies_from_right = [1] * num_children

        # Calculate the minimum candies required from the left perspective
        for i in range(1, num_children):
            # If the current child has a higher rating than the previous child,
            # give one more candy than the previous child
            if ratings[i] > ratings[i - 1]:
                candies_from_left[i] = candies_from_left[i - 1] + 1

        # Calculate the minimum candies required from the right perspective
        for i in range(num_children - 2, -1, -1):
            # If the current child has a higher rating than the next child,
            # give one more candy than the next child
            if ratings[i] > ratings[i + 1]:
                candies_from_right[i] = candies_from_right[i + 1] + 1

        # Sum the max number of candies required from both perspectives for each child
        # to ensure all conditions are met
        total_candies = sum(max(candies_left, candies_right) for candies_left, candies_right in zip(candies_from_left, candies_from_right))

        return total_candies
```

Java

```
class Solution {
    public int candy(int[] ratings) {
        int n = ratings.length; // Total number of children
        int increasingCount = 0; // Counter for increasing ratings
        int decreasingCount = 0; // Counter for decreasing ratings
        int peakCandy = 0; // Number of candies at the peak (highest point in the current iteration)
        int totalCandies = 1; // Start with one candy for the first child

        // Iterate through each child starting from the second one
        for (int i = 1; i < n; i++) {
            if (ratings[i - 1] < ratings[i]) {
                // Rating is higher than the previous child's rating
                increasingCount++;
                peakCandy = increasingCount + 1; // The current child gets one more candy than the increasing count
                decreasingCount = 0; // Reset decreasing count
                totalCandies += peakCandy; // Update total candies with the current child's candies
            } else if (ratings[i] == ratings[i - 1]) {
                // Rating is equal to the previous child's rating
                // Reset the peak, increasing count, and decreasing count
                peakCandy = 0;
                increasingCount = 0;
                decreasingCount = 0;
                totalCandies++; // Each child gets at least one candy
            } else {
                // Rating is lower than the previous child's rating
                decreasingCount++;
                increasingCount = 0; // Reset increasing count
                // Add the number of candies for decreasing sequence.
                // If peak candy count is not enough to cover the decreasing sequence, an extra candy is given to the peak
                totalCandies += decreasingCount + (peakCandy > decreasingCount ? 0 : 1);
            }
        }

        // Return the computed total number of candies needed
        return totalCandies;
    }
}
```

C++

```
#include <vector>
#include <algorithm> // Included to use std::max

class Solution {
public:
    /**
     * Calculate the minimum number of candies to distribute to children,
     * given different ratings, where each child must have at least one candy,
     * and children with a higher rating than their neighbors must have more candies.
     *
     * @param ratings A vector of integers where each element represents the rating of a child.
     * @return The minimum number of candies required.
     */
    int candy(vector<int>& ratings) {
        // Get the number of children
        int numChildren = ratings.size();

        // Initialize left and right vectors with 1 candy for each child
        vector<int> candiesFromLeft(numChildren, 1);
        vector<int> candiesFromRight(numChildren, 1);

        // Distribute candies from left to right, ensuring each child with a higher rating
        // than the left neighbor gets more candies
        for (int i = 1; i < numChildren; ++i) {
            if (ratings[i] > ratings[i - 1]) {
                candiesFromLeft[i] = candiesFromLeft[i - 1] + 1;
            }
        }

        // Distribute candies from right to left with the same logic
        for (int i = numChildren - 2; i >= 0; --i) {
            if (ratings[i] > ratings[i + 1]) {
                candiesFromRight[i] = candiesFromRight[i + 1] + 1;
            }
        }

        // Calculate the total amount of candies by taking the maximum of
        // candies from left and right at each position
        int totalCandies = 0;
        for (int i = 0; i < numChildren; ++i) {
            totalCandies += std::max(candiesFromLeft[i], candiesFromRight[i]);
        }

        // Return the total number of candies needed
        return totalCandies;
    }
};
```

TypeScript

```
function candy(ratings: number[]): number {
    const numChildren = ratings.length;
    const candiesFromLeft = new Array(numChildren).fill(1);
    const candiesFromRight = new Array(numChildren).fill(1);

    // Traverse from left to right, and assign candies ensuring that each child
    // with a higher rating than the left neighbor receives more candies
    for (let i = 1; i < numChildren; ++i) {
        if (ratings[i] > ratings[i - 1]) {
            candiesFromLeft[i] = candiesFromLeft[i - 1] + 1;
        }
    }

    // Traverse from right to left, and assign candies ensuring that each child
    // with a higher rating than the right neighbor receives more candies
    for (let i = numChildren - 2; i >= 0; --i) {
        if (ratings[i] > ratings[i + 1]) {
            candiesFromRight[i] = candiesFromRight[i + 1] + 1;
        }
    }

    // Calculate the minimum required candies by selecting the maximum number
    // of candies assigned from either direction for each child
    let totalCandies = 0;
    for (let i = 0; i < numChildren; ++i) {
        totalCandies += Math.max(candiesFromLeft[i], candiesFromRight[i]);
    }

    // Return the total number of candies needed
    return totalCandies;
}
```

```
from typing import List

class Solution:
    def candy(self, ratings: List[int]) -> int:
        # Length of the ratings list
        num_children = len(ratings)

        # Initialize lists to represent the minimum candies for each child from left and right perspectives
        candies_from_left = [1] * num_children
        candies_from_right = [1] * num_children

        # Calculate the minimum candies required from the left perspective
        for i in range(1, num_children):
            # If the current child has a higher rating than the previous child,
            # give one more candy than the previous child
            if ratings[i] > ratings[i - 1]:
                candies_from_left[i] = candies_from_left[i - 1] + 1

        # Calculate the minimum candies required from the right perspective
        for i in range(num_children - 2, -1, -1):
            # If the current child has a higher rating than the next child,
            # give one more candy than the next child
            if ratings[i] > ratings[i + 1]:
                candies_from_right[i] = candies_from_right[i + 1] + 1

        # Sum the max number of candies required from both perspectives for each child
        # to ensure all conditions are met
        total_candies = sum(max(candies_left, candies_right) for candies_left, candies_right in zip(candies_from_left, candies_from_right))

        return total_candies
```

Time and Space Complexity

The time complexity of the code is `O(n)`. Here's the analysis:

- There are two single-pass for-loops that go through the list of `ratings` which contains `n` elements. Each loop runs in `O(n)` and they are run sequentially, not nested. So the time complexity remains `O(n)`.
- The final return statement uses a generator expression with `zip` to combine the two lists (`left` and `right`) and sum the maximum values of corresponding elements. `zip` function pairs up the elements of both lists, which takes `O(n)` time, and the summation also takes `O(n)` time since each element is considered once.

Combining these linear operations still results in a total time complexity of `O(n)`.

The space complexity of the code is `O(n)`. This is due to:

- Two additional lists `left` and `right`, each of size `n`, are created to store the incrementing sequences based on the `ratings`.
- No other additional space that scales with the input size is used, so the total space complexity is the sum of the space used by these two lists, which is `2 * O(n)` simplified to `O(n)` as constants are dropped in big O notation.