455. Assign Cookies

Easy Greedy Array Two Pointers Sorting

Problem Description

The problem describes a scenario where a parent wants to distribute cookies to their children in a way that maximizes happiness, given certain constraints. Each child has a greed factor, `g[i]`, which indicates the minimum size of a cookie that will make them content. Similarly, each cookie has a size, `s[j]`. A cookie can only make a child content if the size of the cookie is greater than or equal to the child's greed factor. The objective is to assign cookies to children in such a manner that the maximum number of children are content. A child can receive at most one cookie, and a cookie can only be assigned to one child. The task is to calculate the maximum number of content children based on the distribution of cookies according to the children's greed factors.

Intuition

The intuition behind the solution is to prioritize giving cookies to children with the smallest greed factor first. We can sort both the children's greed factors (`g`) and the cookie sizes (`s`) in non-decreasing order. We then iterate through the sorted greed factors, trying to find the smallest cookie that meets or exceeds that greed factor. If such a cookie is found, we assign it to the child, making them content, and move on to the next child. Once a cookie has been assigned, it can no longer be given to any other child, so we move to the next available cookie.

The reason for sorting is to use the smallest cookies for children with the lowest greed factor, which ensures we don't waste a large cookie on a child who would be content with a smaller one. By efficiently matching cookies to children starting with the least greedy ones, we aim to maximize the total number of content children.

As we iterate through the greed factors, if we reach a point where there are no more cookies left to satisfy a child, we return the number of children that have been made content so far. If all children have been assigned a cookie, we return the total number of children.

Solution Approach

The solution approach uses a greedy algorithm which is an algorithmic paradigm that builds up a solution piece by piece, always choosing the next piece that offers the most immediate benefit. In this case, the immediate benefit is making a child content with the smallest cookie possible.

The algorithm works as follows:

1. Sort the greed factors array `g` and the cookie sizes array `s` in ascending order. This sorting is crucial as it allows us to efficiently assign the smallest available cookie that will make each child content.
2. Initialize a pointer `i` to `0`, which will track the current index in the cookie sizes array `s`.
3. Loop through each child's greed factor in `g`, with the index `i` iterating through `g`.
4. For the current child's greed factor (`g[i]`), iterate through the cookies starting from the current index `j` in the array `s` until you find a cookie large enough to satisfy this child. This inner while loop continues as long as `j < len(s)` and the current cookie is smaller than the current child's greed factor (`s[j] < g[i]`). If the current cookie is too small, increment `j` to check the next cookie.
5. If `j` is equal to or greater than the length of `s`, meaning there are no more cookies left to distribute, return the index `i` as the total number of content children.
6. If a suitable cookie is found (one that is equal to or larger than `g[i]`), increment `j` to move to the next cookie and continue with the next child by proceeding to the next iteration of the for loop.
7. If all children have been considered, return the length of `g` as all children have been made content.

The performance of this algorithm hinges on the sorting process (which typically runs in O(n log n) time). The following loop operations are linear, making the total time complexity O(n log n) primarily due to sorting, where `n` is the number of elements in the larger of the two arrays `g` or `s`. Space complexity is O(1) since the solution sorts and iterates over the input arrays in place without using extra space for auxiliary data structures.

Example Walkthrough

Let's consider the following example to illustrate the solution approach. Imagine a scenario where we have three children with greed factors `g = [1, 2, 3]` and an assortment of four cookies with sizes `s = [1, 1, 3, 4]`.

Following the steps outlined in the solution approach:

1. First, sort the arrays `g` and `s`. Since `g` is already sorted, we only need to sort `s`, which becomes `s = [1, 1, 3, 4]`. Both arrays are now sorted in ascending order.
2. Initialize a pointer `j` to `0` to track the current index in the sorted cookie sizes array `s`.
3. Loop through each child's greed factor in `g`. We start with the first child `g[0]`, which is `1`.
4. For the current child's greed factor (`g[i]`), loop through the cookies starting from the current `j` index. We compare the child's greed factor to the current cookie size:
   - For the first child (`g[0] = 1`), the first cookie (`s[0] = 1`) is of size `1`, which satisfies the child's greed factor. We increment `j` to `1` and move to the next child.
5. We now check for the second child (`g[1] = 2`). Since we have incremented `j`, the cookie we are looking at is now `s[1]` which is of size `1`. However, `1` is smaller than the greed factor `2`. So, we increment `j` to `2`.
   - Now looking at cookie `s[2]` which is of size `3`, it satisfies the second child's need (`g[1] = 2`). We increment `j` to `3` and move to the next child.
6. For the third child (`g[2] = 3`), the next cookie is `s[3] = 4`, which is also sufficient. We increment `j` to `4` (which now is outside the boundary of array `s`).
7. Since all the children's greed factors have been considered and satisfied, we return the length of `g`, which is `3`. This indicates that all children have been made content.

By following the algorithm, we have maximized the number of content children with the available cookies. The sorted arrays allowed us to efficiently match the smallest possible cookie to each child according to their greed factor.

Solution Implementation

Python

```python
class Solution:
    def findContentChildren(self, greed_factors: List[int], cookie_sizes: List[int]) -> int:
        # Sort the greed factors of the children and the sizes of the cookies
        greed_factors.sort()
        cookie_sizes.sort()

        # Initialize the cookie index
        cookie_index = 0

        # Iterate through each greed factor
        for child_index, greed in enumerate(greed_factors):
            # Move through the cookie sizes until we find a cookie that satisfies the current greed factor
            while cookie_index < len(cookie_sizes) and cookie_sizes[cookie_index] < greed:
                cookie_index += 1

            # If there are no more cookies left, return the number of content children so far
            if cookie_index >= len(cookie_sizes):
                return child_index

            # Move to the next cookie index assuming the current cookie has been used
            cookie_index += 1

        # All children have been content, return the total number of children
        return len(greed_factors)
```

Java

```java
class Solution {
    // Method to find the maximum number of content children given the greed factors of children and the sizes of cookies
    public int findContentChildren(int[] greedFactors, int[] cookieSizes) {
        // Sort the arrays to make the greedy assignment possible
        Arrays.sort(greedFactors);
        Arrays.sort(cookieSizes);

        int numberOfChildren = greedFactors.length;   // Total number of children
        int numberOfCookies = cookieSizes.length;     // Total number of cookies available

        // Initialize the count for content children
        int contentChildrenCount = 0;

        // Initialize pointers for greedFactors and cookieSizes arrays
        int greedFactorIndex = 0;
        int cookieSizeIndex = 0;

        // Loop through each child's greed factor
        while (greedFactorIndex < numberOfChildren) {

            // Find the first cookie that satisfies the current child's greed factor
            while (cookieSizeIndex < numberOfCookies && cookieSizes[cookieSizeIndex] < greedFactors[greedFactorIndex]) {
                cookieSizeIndex++;  // Increment cookie index until a big enough cookie is found
            }

            // If a cookie that satisfies the current child's greed factor is found, consider the child content
            if (cookieSizeIndex < numberOfCookies) {
                contentChildrenCount++;      // Increment the count of content children
                cookieSizeIndex++;           // Move to the next cookie
            } else {
                // If no more cookies are available to satisfy any more children, break out of the loop
                break;
            }

            // Move to the next child
            greedFactorIndex++;
        }

        // Return the final count of content children
        return contentChildrenCount;
    }
}
```

C++

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

class Solution {
public:
    int findContentChildren(std::vector<int>& children, std::vector<int>& cookies) {
        // Sort the greed factors of the children
        std::sort(children.begin(), children.end());
        // Sort the sizes of the cookies
        std::sort(cookies.begin(), cookies.end());

        int numChildren = children.size(); // Number of children
        int numCookies = cookies.size();   // Number of cookies
        int contentChildren = 0;           // Counter for content children

        // Iterate through each child
        for (int childIndex = 0, cookieIndex = 0; childIndex < numChildren; ++childIndex) {
            // Find the first cookie that can satisfy the current child's greed factor
            while (cookieIndex < numCookies && cookies[cookieIndex] < children[childIndex]) {
                ++cookieIndex;
            }

            // If we found a cookie that can satisfy the child
            if (cookieIndex < numCookies) {
                // Give the cookie to the child
                ++contentChildren;
                // Move to the next cookie
                ++cookieIndex;
            } else {
                // If no more cookies are available that can satisfy any child, break out of the loop
                break;
            }
        }

        // The number of children who can be content with the cookies we have
        return contentChildren;
    }
};
```

TypeScript

```typescript
/**
 * Finds the maximum number of content children given the children's greed factors and the sizes of cookies.
 * A child will be content if they receive a cookie that is equal to or larger than their greed factor.
 * @param {number[]} greedFactors - An array representing the greed factors of each child.
 * @param {number[]} cookieSizes - An array representing the sizes of cookies available.
 * @returns {number} The maximum number of content children.
 */
function findContentChildren(greedFactors: number[], cookieSizes: number[]): number {
    // Sort greed factors and cookie sizes in non-decreasing order.
    greedFactors.sort((a, b) => a - b);
    cookieSizes.sort((a, b) => a - b);

    // Initialize counters for children and cookies.
    const totalChildren = greedFactors.length;
    const totalCookies = cookieSizes.length;

    // Initialize the index for greed factors and cookie sizes.
    let childIndex = 0;
    let cookieIndex = 0;

    // Loop through the greed factors of each child.
    while (childIndex < totalChildren) {
        // Find a cookie that is equal to or larger than the greed factor of the current child.
        while (cookieIndex < totalCookies && cookieSizes[cookieIndex] < greedFactors[childIndex]) {
            cookieIndex++;
        }

        // If a suitable cookie is found, move on to the next child and cookie.
        if (cookieIndex < totalCookies) {
            childIndex++;
            cookieIndex++;
        } else {
            // No more cookies available, return the number of content children.
            return childIndex;
        }
    }

    // All children have been matched with cookies, return the total number of children.
    return totalChildren;
}
```

```python
class Solution:
    def findContentChildren(self, greed_factors: List[int], cookie_sizes: List[int]) -> int:
        # Sort the greed factors of the children and the sizes of the cookies
        greed_factors.sort()
        cookie_sizes.sort()

        # Initialize the cookie index
        cookie_index = 0

        # Iterate through each greed factor
        for child_index, greed in enumerate(greed_factors):
            # Move through the cookie sizes until we find a cookie that satisfies the current greed factor
            while cookie_index < len(cookie_sizes) and cookie_sizes[cookie_index] < greed:
                cookie_index += 1

            # If there are no more cookies left, return the number of content children so far
            if cookie_index >= len(cookie_sizes):
                return child_index

            # Move to the next cookie index assuming the current cookie has been used
            cookie_index += 1

        # All children have been content, return the total number of children
        return len(greed_factors)
```

Time and Space Complexity

Time Complexity

The time complexity of the provided code can be analyzed based on two main operations: sorting the lists `g` and `s` and then iterating through them to find content children.

1. **Sorting**: Both the lists `g` and `s` are sorted at the beginning of the function. The sorting operation typically has a time complexity of `O(n log n)` for an average sorting algorithm like Timsort (used in Python's `sort()` method), where `n` is the number of elements in the list. If `g` has `N` elements and `s` has `M` elements, the total time for sorting would be `O(N log N + M log M)`.
2. **Iterating through Lists**: After sorting, the code uses a while loop within a for loop to find matches for `g[i]`. In the worst-case scenario, every element of `g` would be compared with every element of `s`, which gives us a time complexity of `O(N*M)`, as each list is traversed at most once.

Combining both steps, the overall time complexity is `O(N log N + M log M)` for the sorting part, which is dominant, plus `O(N*M)` for the iteration part. Therefore, the total time complexity is:

`O(N log N + M log M)`.

Space Complexity

The space complexity is the amount of extra space or temporary space used by an algorithm. The provided code only uses a fixed number of variables (`i`, `x`, `j`) and does not create any auxiliary data structures proportional to the size of the input. The space taken up by the input lists themselves is not counted towards the space complexity, as they are part of the input.

Therefore, the space complexity is:

`O(1)`.