2305. Fair Distribution of Cookies

Medium | Bit Manipulation | Array | Dynamic Programming | Backtracking | Bitmask | Leetcode Link

Problem Description

In this problem, we are given an array called `cookies`, where each element `cookies[i]` represents the number of cookies in the `i-th` bag. We are also given an integer `k` which represents the number of children to whom we have to distribute all these bags of cookies. The important condition here is that all the cookies in one bag must go to the same child, and we cannot split one bag's contents between children.

Our aim is to find the distribution of cookies among the `k` children such that the unfairness is minimized. The unfairness is defined as the maximum total number of cookies that any single child receives. So, if one child ends up with a lot more cookies than the others, the unfairness is high, and we want to avoid that.

Intuition

To arrive at a solution for this problem, we need to find a way to distribute the cookies such that no child receives an amount of cookies that exceeds our current best (minimal unfairness) distribution.

The intuition behind the solution is to consider the problem as deep-first search (DFS) across all possible distributions and to track the amount of cookies each child has received at any point. We will follow these steps:

1. Sort the `cookies` array in descending order. This helps to consider larger bags first, potentially leading to an optimization that may prune the search space.
2. Start a recursive DFS function that tries to add the current bag of cookies to each child's total.
3. Track the number of cookies each child has in an array `cnt`, and keep updating the minimum unfairness `ans` as we try different distributions.
4. While performing DFS, we check two conditions before choosing to add a bag to a child's total: a. If adding the current bag of cookies makes this child's total exceed the current best unfairness score `ans`, we should not continue this path, as it won't lead to an improvement. b. If two successive children have the same amount of cookies and we are considering assigning more to the latter, we skip this to avoid duplicate distributions that have the same effect.
5. Once we have considered adding the current bag to each child (and recursively for all following bags), we would have explored all distributions.
6. We then return the minimum unfairness that we found.

The recursive nature of the function allows us to explore each possible distribution and update the minimum unfairness accordingly. By using the heuristic of sorting bags in descending order, and by skipping certain branches where the unfairness would only increase, we ensure that the solution is efficient enough to explore all relevant possibilities without unnecessary computations.

Solution Approach

The implementation of this problem uses a Depth-First Search (DFS) approach to explore all possible distributions of cookies to the `k` children, aiming to find the distribution that yields the minimum unfairness. The key elements of the implementation are:

- **Sorting the `cookies` array**: We first sort the array in reverse order (descending). This is a heuristic that can potentially reduce the search space by considering larger quantities first, as they have a more significant impact on the unfairness.

- **Recursive DFS function**: The core of the implementation is the recursive function `dfs(i)`, which explores distributions starting from the `i-th` bag.

- **State tracking with `cnt` array**: We use an array `cnt` of size `k` to keep track of how many cookies each child currently has in the ongoing distribution scenario.

- **Pruning with an unfairness limit `ans`**: We utilize the variable `ans` to store the minimum unfairness found so far, initialized to an infinite value. As the search proceeds, `ans` gets updated with the best (lowest) unfairness score. We use this value to make decisions on pruning the search: if adding a bag of cookies to a child's total surpasses `ans`, we know this path will not yield a better result, so we can backtrack.

- **Avoidance of duplicate states**: When iterating to add cookies to a child's total, if the previous child (`j-1`) has the same number of cookies as the current child (`j`), we skip this step to prevent exploring duplicate distributions that would not affect the outcome.

- **Recursive DFS exploration**: In each step of the recursion, we attempt to add the current bag to each child's total and recursively call `dfs(i + 1)` to consider the subsequent bag. If we add the bag's cookies, we then backtrack by subtracting those cookies before moving on to the next child.

Here is a step-by-step walkthrough of the recursive function:

1. If we have considered all bags (`i >= len(cookies)`), then we have reached a complete distribution. Update the unfairness limit `ans` with the maximum number of cookies any child has received in this distribution (`max(cnt)`), and return as there's no more exploration needed for this path.

2. For each child `j` in range `k`, check if adding the current bag to this child's total (`cnt[j] + cookies[i]`) would not exceed `ans`. If it does exceed, or if we have a duplicate state as described above, skip this child.

3. If it doesn't exceed, add the current bag's cookies to this child's total (`cnt[j] += cookies[i]`) and recursively explore the next bag (`dfs(i + 1)`).

4. After the recursive call, backtrack by subtracting the cookies from the child's total (`cnt[j] -= cookies[i]`) to restore the state before exploring other possibilities.

The recursion ensures that all combinations are considered and by pruning the search space, the algorithm remains efficient enough to find the minimum unfairness across all distributions.

Example Walkthrough

Let's consider a small example using the solution approach provided above. Suppose we have an array `cookies = [8, 7, 5]` and `k = 2`, meaning we have 3 bags of cookies with 8, 7, and 5 cookies respectively, which need to be distributed to 2 children.

Here's a walkthrough of the process:

1. We sort the cookies array in descending order, resulting in `cookies = [8, 7, 5]`.

2. Initialize the `cnt` array, ensuring it's of size `k` (the number of children). In this example, `cnt = [0, 0]` as we have 2 children.

3. Set `ans` to a large number to represent infinity since we haven't found any distribution yet. We'd update this value every time we find a better (smaller) unfairness.

Now, we start the recursive DFS function `dfs(i)` with `i = 0`.

4. At the first level, we have two options: give the first bag (with 8 cookies) to either of the children.
   - If we give it to the first child, `cnt = [8, 0]`.
   - Next, we call the recursive function `dfs(1)` to distribute the next bag.
5. Again, at the second level, we have the option to add the next bag with 7 cookies to either of the children's totals.
   - Adding to the first child isn't an option as it exceeds the current `ans` (which remains effectively infinite for now), so we explore giving to the second child, and the `cnt` array updates to `[8, 7]`.
   - We now call `dfs(2)` to consider the last bag.
6. Finally, for the last bag with 5 cookies, we try both options again but we have to ensure not to exceed the current `ans`.
   - If we add it to the first child, `cnt` becomes `[13, 7]`.
   - If we add it to the second child, `cnt` becomes `[8, 12]`.

In the end, the minimum unfairness `ans` is updated to the best distribution found. Here, the smallest maximum we could get from any distribution is 12, by giving out the cookies in such a manner that the first child gets 8 cookies and the second child gets 12 cookies (`[8, 12]`).

Thus, the unfairness of the distribution is 12, which is the maximum number of cookies any child receives in the best possible distribution of cookies to the 2 children.

Python Solution

```python
from typing import List

class Solution:
    def distributeCookies(self, cookies: List[int], k: int) -> int:
        # Recursive depth-first search function to distribute cookies
        def dfs(index):
            # Base case: if all cookies have been considered
            if index >= len(cookies):
                # Record the maximum cookies any child has to minimize it
                self.best_distribution = min(self.best_distribution, max(children_cookies))
                return

            # Iterate through each child
            for j in range(k):
                # Skip this distribution if:
                # 1. Current distribution already exceeds the best distribution found
                # 2. To avoid duplicate distributions, skip if the current child has
                #    the same number of cookies as the previous child
                if children_cookies[j] + cookies[index] >= self.best_distribution or (j > 0 and children_cookies[j] == children_cooki
                    continue

                # Distribute current cookie to child j and recurse
                children_cookies[j] += cookies[index]
                dfs(index + 1)
                # Backtrack: remove the current cookie from child j
                children_cookies[j] -= cookies[index]

        # Initialize best distribution as infinity
        self.best_distribution = float('inf')
        # Initialize list to keep track of cookies each child has
        children_cookies = [0] * k
        # Sort cookies in descending order to distribute larger cookies first
        cookies.sort(reverse=True)
        # Start recursive distribution
        dfs(0)
        # Return the minimum of the maximum number of cookies any child has
        return self.best_distribution

```

Java Solution

```java
import java.util.Arrays;

class Solution {
    // Array to hold the value of cookies.
    private int[] cookies;
    // Array to hold the current distribution count for each child.
    private int[] childCookieCount;
    // Number of children to distribute cookies to.
    private int numChildren;
    // Total number of cookies available.
    private int numCookies;
    // The minimized maximum number of cookies any child gets.
    private int minMaxCookies = Integer.MAX_VALUE;

    public int distributeCookies(int[] cookies, int k) {
        numCookies = cookies.length;          // Get the total number of cookies.
        childCookieCount = new int[k];        // Initialize the distribution count array.
        Arrays.sort(cookies);                 // Sort the cookies array.
        this.cookies = cookies;               // Assign cookies array to class variable for easy access.
        this.numChildren = k;                 // Set the number of children.
        distributeCookiesToChildren(numCookies - 1); // Start the distribution from the last index.
        return minMaxCookies;                 // Return the result.
    }

    private void distributeCookiesToChildren(int cookieIndex) {
        // If cookies have been considered, update the minMaxCookies with the maximum cookies any child got.
        if (cookieIndex < 0) {
            for (int count : childCookieCount) {
                minMaxCookies = Math.max(minMaxCookies, count);
            }
            return;  // Exit since all cookies are distributed.
        }
        // Try to distribute the current cookie to each child.
        for (int i = 0; i < numChildren; ++i) {
            // Pruning: if addition exceeds current answer or children have the same count as previous, skip.
            if (childCookieCount[i] + cookies[cookieIndex] >= minMaxCookies ||
                (i > 0 && childCookieCount[i] == childCookieCount[i - 1])) {
                continue;
            }
            // Add the current cookie to the child's count and recurse for the remaining cookies.
            childCookieCount[i] += cookies[cookieIndex];
            distributeCookiesToChildren(cookieIndex - 1);
            // Backtrack: remove the cookie to try another distribution.
            childCookieCount[i] -= cookies[cookieIndex];
        }
    }
}

```

C++ Solution

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

// "Solution" class implements a method to distribute cookies among 'k' children
// in such a way that the child with the maximum number of cookies gets the least
// possible number, provided each child must receive at least one cookie.

class Solution {
public:
    // The "distributeCookies" method takes a vector of integers where each element
    // represents the size of a cookie, and an integer 'k', the number of children.
    // It returns an integer which is the minimized maximum number of cookies
    // a child gets after distribution.
    int distributeCookies(vector<int>& cookies, int k) {
        // First, sort cookies in non-increasing order to start assigning
        // larger cookies first.
        sort(cookies.rbegin(), cookies.rend());

        // Initialize an array to keep track of the cookies count for each child.
        int count_per_child[k];
        memset(count_per_child, 0, sizeof count_per_child);

        // Get the total number of cookies.
        int num_cookies = cookies.size();

        // Initialize 'answer' with a high value.
        int answer = INT_MAX;

        // Define a lambda function for depth-first search to distribute the cookies.
        function<void(int)> distribute = [&](int index) {
            // If all cookies have been considered, update the 'answer' with the current maximum.
            if (index >= num_cookies) {
                answer = *max_element(count_per_child, count_per_child + k);
                return;
            }
            // Loop through each child.
            for (int child = 0; child < k; ++child) {
                // Prune the search if current distribution exceeds the current answer
                // or to avoid identical distributions when previous child has the same count.
                if (count_per_child[child] + cookies[index] >= answer ||
                    (child > 0 && count_per_child[child] == count_per_child[child - 1])) {
                    continue;
                }
                // Add the current cookie to the child's count and recurse to the next cookie.
                count_per_child[child] += cookies[index];
                distribute(index + 1);
                // Remove the cookie from the child's count before backtrack.
                count_per_child[child] -= cookies[index];
            }
        };

        // Initiate the search process from the first cookie.
        distribute(0);

        // Return the answer - the minimized maximum number of cookies among children.
        return answer;
    }
};

```

Typescript Solution

```typescript
// Function to find the minimum possible maximum number of cookies one child can get,
// when the cookies are distributed among 'k' children.
function distributeCookies(cookies: number[], k: number): number {
    // Initialize an array to track the total cookies each child has.
    const cookiesPerChild = new Array(k).fill(0);
    // Set an initial high value for the answer to compare against later.
    let minValueOfMaxCookies = Number.MAX_SAFE_INTEGER;

    // Depth-first search helper function to try out different distributions.
    const dfs = (index: number) => {
        // Check if all cookies have been considered.
        if (index >= cookies.length) {
            // Update the minimum value with the maximum cookies any child currently has.
            minValueOfMaxCookies = Math.min(minValueOfMaxCookies, Math.max(...cookiesPerChild));
            return;
        }
        for (let j = 0; j < k; ++j) {
            // Skip the distribution if it would surpass the current minimum value of max cookies,
            // or if the current and previous child would have the same amount (to avoid duplicate distributions).
            if (cookiesPerChild[j] + cookies[index] >= minValueOfMaxCookies ||
                (j > 0 && cookiesPerChild[j] === cookiesPerChild[j - 1])) {
                continue;
            }
            // Add the current cookie to the child 'j' and move to the next cookie.
            cookiesPerChild[j] += cookies[index];
            // Continue exploring with the next cookie.
            dfs(index + 1);
            // Backtrack: remove the current cookie from the child 'j'.
            cookiesPerChild[j] -= cookies[index];
        }
    };

    // Start the depth-first search with the first cookie.
    dfs(0);

    // After exploring all distributions, return the minimum possible maximum number of cookies.
    return minValueOfMaxCookies;
}

```

Time and Space Complexity

The given code is a depth-first search algorithm aiming to distribute cookies among `k` children such that the maximum number of cookies given to any single child is minimized. The provided Python function lacks a return type for the DFS helper function, and it should be corrected before analyzing the code's computational complexity.

Time Complexity:

The time complexity of this algorithm is quite high due to its backtracking nature which explores every possible distribution of the cookies. Let's break it down:

- We have `len(cookies)` cookies to distribute, and for each cookie, we have `k` choices of children to whom the cookie can be given. The branching factor is thus `k`.
- The `dfs` function is called recursively for each cookie and at every level of the recursion tree determines for each child whether to continue or not based on certain condition checks (`cnt[j] + cookies[i] >= ans` and `j and cnt[j] == cnt[j - 1]`).
- Each child can have at most `len(cookies)` cookies, meaning there are `len(cookies)^k` possible ways to distribute the cookies without any checks or pruning.

However, due to the pruning conditions:

- If the current count `cnt[j]` plus the cookie `cookies[i]` is greater or equal to the current answer `ans`, the branch will prune and not further search down that path.
- If the current child has the same count as the previous child, to avoid redundant distributions, the branch is pruned.

With these pruning conditions, the worst-case run-time complexity is less than `O(k^n)`, where `n` is the number of cookies. However, it is challenging to quantify the exact impact of the pruning on the average or upper bound, as it heavily depends on the distribution of the `cookies` list and the choice of `k`.

Space Complexity:

- The `cnt` array holds a count for each child, which has a size `k`, resulting in `O(k)` space.
- Since the `dfs` function is called recursively for each cookie, there will be `len(cookies)` (which is `n`) activation records on the call stack at most, if we consider `k` to be constant, then the space complexity contributed by the recursive stack is `O(n)`.

In total, the space complexity of the algorithm is `O(n + k)`, simplifying to `O(n)` if `k` is constant or if `n` is significantly larger than `k`.