2226. Maximum Candies Allocated to K Children

Medium Array Binary Search Leetcode Link

Problem Description

In this problem, you have an array `candies`, where each element represents a pile of candies (with index starting from 0). Each pile contains a certain number of candies denoted by `candies[i]`. The array gives you the flexibility to split any pile into smaller sub-piles but does not allow you to combine two different piles into one.

Additionally, you are given an integer `k`, which represents the number of children to whom the candies must be distributed. The goal is to distribute the candies to `k` children in such a way that each child receives exactly the same number of candies. However, each child can receive at most one pile (which can potentially be a sub-pile of one of the original piles), and it's possible that not all piles will be distributed.

The objective is to find the maximum number of candies that you can give to each child under these conditions.

Intuition

To solve this problem, we need to determine the *maximum number of candies each child can get*. The key here is understanding that if we can give `x` candies to each child, then surely we can give any amount less than `x` candies as well (by dividing the piles into smaller ones). However, we cannot always give more than `x` candies. So, our solution will likely involve finding the optimal `x`.

One way to approach this is through binary search. The minimum number of candies a child could get is `0` (if `k` is larger than the total number of candies available) and the maximum is the size of the largest pile in `candies`.

Using binary search, we can find the largest possible value for `x` in the range `[0, max(candies)]` such that we can still distribute the candies to `k` children with each child receiving `x` candies. We consider a middle value `mid` within this range and check if it is possible to divide the piles in such a way that at least `k` children can receive `mid` candies each. If it is possible, then we know we can try to increase `x` and search in the upper half of the range; otherwise, we search in the lower half of the range.

The provided code implements this binary search approach to efficiently determine the maximum amount of candies that can be distributed to each child such that every child receives the same amount.

Solution Approach

The solution provided is a binary search approach, which solves the problem by iterating over a possible range of candies that can be distributed to each child and narrowing down that range to find an optimal solution.

Here's a step-by-step explanation of the implementation:

1. We initialize two pointers, `left` and `right`. `left` represents the lower bound (minimum number of candies we can give per child), set to 0, since it's possible that we cannot give any candies to the children if there are fewer candies than `k`. `right` represents the upper bound (maximum number of candies we can give to each child), which would be the largest pile in `candies`, since no child can receive more than the size of the largest pile.

```
left, right = 0, max(candies)
```

2. We perform the binary search by checking the middle value between the `left` and `right` pointers. This middle value, `mid`, posits a hypothetical maximum number of candies to be distributed per child.

```
mid = (left + right + 1) >> 1
```

3. For the current middle value `mid`, we calculate whether it is possible to achieve this distribution by summing up the number of children we can satisfy if each were to be given `mid` candies. This is done by dividing each pile size by `mid` and summing those values across all piles.

```
cnt = sum(v // mid for v in candies)
```

4. If `cnt` (the number of potential children receiving `mid` candies) is greater than or equal to `k`, then distributing `mid` candies to each child is possible, and we can potentially give more, so we adjust the `left` pointer to `mid`.

```
if cnt >= k:
    left = mid
```

5. If `cnt` is less than `k`, then we cannot give out `mid` candies to each child, and we might need to reduce the number of candies we give to each child. Therefore, we adjust the `right` pointer to `mid - 1`.

```
else:
    right = mid - 1
```

6. This search process continues until `left` and `right` converge, which happens when `left == right`. When the binary search terminates, `left` will hold the maximum number of candies that we can distribute to each child.

7. Finally, the function returns the value of `left` as the answer.

```
return left
```

The algorithm effectively uses binary search on a sorted range of possible candy distributions, utilizing the idea that if a certain amount `x` is doable, so is any amount less than `x`. We use division to check the feasibility of each `mid` point, counting the number of children that can be satisfied with `mid` candies and adjusting our search space accordingly until we find the maximum feasible distribution.

Example Walkthrough

Let's take an example to illustrate the solution approach. Suppose we have an array of piles of candies given as `candies = [4, 7, 12, 19]`, and the number of children `k = 3`.

We want to distribute these candies such that each of the 3 children gets the same number of candies, and we are tasked with finding the maximum number of candies each child can get.

1. We begin by initializing `left = 0` and `right = 19` (19 being the largest pile of candies).
2. We perform binary search. Initially, `mid = (0 + 19 + 1) >> 1` is `10`. We check if we can distribute 10 candies to each child.
3. To check the feasibility, we sum up how many children can receive 10 candies:
   - Pile 1 can be divided into `4 // 10` children, which is `0`.
   - Pile 2 can be divided into `7 // 10` children, which is `0`.
   - Pile 3 can be divided into `12 // 10` children, which is `1`.
   - Pile 4 can be divided into `19 // 10` children, which is `1`.

   The total number of children who can receive candies is `0 + 0 + 1 + 1 = 2`.
4. Since we can only satisfy 2 children, but we need to satisfy 3, `10` candies per child is too high. So, we set `right = mid - 1`, which is `9`.
5. We repeat the binary search. Now, `mid = (0 + 9 + 1) >> 1` is `5`. We calculate:
   - Pile 1 can be divided into `4 // 5` children, which is `0`.
   - Pile 2 can be divided into `7 // 5` children, which is `1`.
   - Pile 3 can be divided into `12 // 5` children, which is `2`.
   - Pile 4 can be divided into `19 // 5` children, which is `3`.

   Now, the total is `0 + 1 + 2 + 3 = 6`.
6. Since the total number of children who can receive candies is now 6, and we only need to satisfy 3, we have found that giving 5 candies per child is possible. We could potentially give more, so we adjust `left` to `mid`, which is `5`.
7. We continue this process, but now as `left` and `right` converge, both will be equal to `5`, and the loop ends.
8. The binary search concludes with `left = 5`, which is the maximum number of candies we can distribute to each child.

Each child can receive 5 candies, which can be distributed as follows:

- Child 1 receives 5 candies from pile 2.
- Child 2 receives 5 candies from pile 3.
- Child 3 receives 5 candies from pile 4.

2 candies will remain in pile 3, and 14 candies will remain in pile 4, but since we can't combine them and we can only give one pile to each child, this is the most we can distribute evenly.

Python Solution

```python
class Solution:
    def maximumCandies(self, candies: List[int], k: int) -> int:
        # Initialize left to 0 (minimum possible result) and
        # right to the maximum number of candies in any pile
        left, right = 0, max(candies)

        # Binary search for the highest number of candies per child
        while left < right:
            # Mid is the number of candies to distribute per child; add 1 and
            # shift right by 1 to get the upper middle for even numbers
            mid = (left + right + 1) // 2

            # Count how many children can receive 'mid' number of candies
            count = sum(candy // mid for candy in candies)

            # If the number of children that can receive 'mid' candies is at least k,
            # mid can be a potential answer, so we move left pointer up to mid
            if count >= k:
                left = mid
            # Otherwise, we decrease the right pointer, as we need less candies per child
            else:
                right = mid - 1

        # left is now the maximum number of candies per child that we can distribute
        return left

```

Java Solution

```java
class Solution {
    public int maximumCandies(int[] candies, long k) {
        // Initialize the search range for the maximum number of candies per child.
        int low = 0;
        int high = (int) 1e7;

        // Use a binary search to find the maximum number of candies
        // that can be distributed to `k` children evenly.
        while (low < high) {
            // Calculate the middle point of the current search range.
            // Use `+ 1` to ensure the `mid` is biased towards the higher part of the range.
            // The bitwise right shift operator `>> 1` effectively divides the sum of `low` and `high` by 2.
            int mid = (low + high + 1) >> 1;

            // Count the total number of children that can receive `mid` candies.
            long count = 0;
            for (int candy : candies) {
                count += candy / mid;
            }

            // If the total number of children that can receive `mid` candies is at least `k`,
            // then we know that it's possible to give out at least `mid` candies to each child.
            // So we update `low` to `mid` to search in the higher half of the range next.
            if (count >= k) {
                low = mid;
            }
            // If not, we need to search in the lower half of the range, so we adjust `high`.
            else {
                high = mid - 1;
            }
        }

        // `low` will represent the highest number of candies that can be distributed evenly to `k` children.
        return low;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int maximumCandies(vector<int>& candies, long long totalChildren) {
        // Initialize the search range
        int minCandies = 0, maxCandies = 1e7;

        // Perform a binary search to find the maximum number of candies
        // that can be distributed to each child
        while (minCandies < maxCandies) {
            // Calculate the mid-point for the current search range
            int mid = (minCandies + maxCandies + 1) >> 1;

            // Counter to record the total number of children we can distribute
            // candies to with the current 'mid' number of candies per child
            long long childCount = 0;

            // Iterate through each pile of candies
            for (int candiesInPile : candies) {
                // Increment the count by the number of children we can satisfy
                // with the current pile using 'mid' candies per child
                childCount += candiesInPile / mid;
            }

            // If we can give at least 'mid' candies to 'totalChildren' or more,
            // adjust the lower bound of the search range
            if (childCount >= totalChildren) {
                minCandies = mid;
            } else {
                // Otherwise, reduce the upper bound of the search range
                maxCandies = mid - 1;
            }
        }

        // After narrowing down the search range, 'minCandies' will hold the
        // maximum number of candies that can be distributed to each child
        return minCandies;
    }
};

```

Typescript Solution

```typescript
// Function to calculate the maximum number of candies
// that can be distributed to each child
function maximumCandies(candies: number[], totalChildren: number): number {
    // Initialize the search range
    let minCandies: number = 0;
    let maxCandies: number = 1e7;

    // Perform a binary search to find the maximum number of candies
    // that can be distributed to each child
    while (minCandies < maxCandies) {
        // Calculate the mid-point for the current search range
        let mid: number = Math.floor((minCandies + maxCandies + 1) / 2);

        // Counter to record the total number of children we can distribute
        // candies to with the current 'mid' number of candies per child
        let childCount: number = 0;

        // Iterate through each pile of candies
        for (let candiesInPile of candies) {
            // Increment the count by the number of children we can satisfy
            // with the current pile using 'mid' candies per child
            childCount += Math.floor(candiesInPile / mid);
        }

        // If we can give at least 'mid' candies to 'totalChildren' or more,
        // adjust the lower bound of the search range
        if (childCount >= totalChildren) {
            minCandies = mid;
        } else {
            // Otherwise, reduce the upper bound of the search range
            maxCandies = mid - 1;
        }
    }

    // After narrowing down the search range, 'minCandies' will hold the
    // maximum number of candies that can be distributed to each child
    return minCandies;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the given code is mainly determined by the while loop and the sum operation within that loop. The `while` loop uses a binary search pattern which runs until `left` is less than `right`. In terms of the binary search, this takes `O(log(max(candies)))` time, because we are repeatedly halving the search space that starts with the maximum value in `candies`.

Inside this loop, there is a sum operation which includes a list comprehension that iterates over all elements in the `candies` list. This operation has a time complexity of `O(n)` where `n` is the number of elements in `candies`, because it must visit each element once.

Since the list comprehension is done every time the while loop runs, the combined time complexity is the product of the two, resulting in `O(n * log(max(candies)))`.

Space Complexity

The space complexity of the given code is `O(1)`. Despite the list comprehension inside the sum function, it does not create a new list due to the generator expression; instead, it calculates the sum on the fly. Thus, the only additional space used is for a few variables (`left`, `right`, `mid`, `cnt`), which is constant and does not depend on the input size. Therefore, the space used is constant and does not grow with the size of the input.