1231. Divide Chocolate

Hard Array Binary Search Leetcode Link

Problem Description

In this problem, you are given an array `sweetness`, where each element represents the sweetness level of a chunk of chocolate. You also have `k` friends with whom you want to share the chocolate. You plan to make `k` cuts to the chocolate bar, resulting in `k + 1` pieces, each composed of consecutive chunks.

Your goal is to be generous with your friends and, therefore, you will choose the least sweet piece for yourself. However, you want to optimize this so that this least sweet piece is as sweet as possible. The task is to find the maximum sweetness you can get for your piece after making the `k` cuts in the best possible way.

Intuition

The solution leverages a binary search algorithm to find the optimal maximum sweetness you can achieve for your chocolate piece. The key insight is to realize that there is a range of possible sweetness levels for your piece – at the minimum, it could be the sweetness of the least sweet chunk, and at the maximum, it could be the total sweetness of the entire chocolate bar divided by `k + 1`.

The binary search approach gradually narrows down this range by guessing a potential maximum sweetness (`mid`), then checking if it's possible to cut the bar into `k + 1` pieces where each piece has at least this level of sweetness. The `check` function is used to determine whether the current guess can be achieved by summing up the chunks' sweetness until it meets or exceeds the guessed level before making a cut. If the function can make more than `k` such valid cuts, the guess is feasible, and it means you might be able to increase the sweetness level for your piece, so you search in the higher half. Otherwise, the guess is too high, and you search in the lower half.

By repeatedly applying the binary search, you narrow down to the optimal `l` value, which represents the maximum sweetness you can achieve for your chocolate piece.

Solution Approach

The solution uses binary search to efficiently find the optimal cut points within the chocolate bar to maximize the sweetness of the piece you will eat. Below is the step-by-step explanation of the implementation:

- First, we establish the search boundaries for binary search. The `l` (left boundary) is initialized to `0`, assuming we cannot have a negative sweetness value, and `r` (right boundary) is initialized to the sum of all chunks' sweetness since this is the maximum sweetness we could potentially achieve if we didn't have to share with anyone.
- We create a helper function `check(x)` with the parameter `x` representing the current guess for the maximum sweetness of the piece. It will return `True` if it is possible to cut the bar into more than `k` pieces where each piece has at least `x` sweetness. The function does this by iterating over the sweetness values, accumulating them until they reach or exceed `x`, then making a virtual cut and proceeding with the rest of the chunks.
- Inside the while loop, which continues until `l < r`, we calculate the midpoint `mid` with `(l + r + 1) >> 1`. The operation `>> 1` is equivalent to integer division by 2, so this finds the middle value between `l` and `r`. The `+1` is important as it helps to avoid an infinite loop when `l` and `r` are close together.
- The `if` condition inside the while loop uses the `check(mid)` function to decide which half of the current range to discard:
  - If `check(mid)` returns `True`, it means it's possible to cut the chocolate into pieces where each has at least `mid` sweetness. Therefore, we should try to find a potentially higher minimum sweetness, and we update `l` to `mid`.
  - If `check(mid)` returns `False`, it means `mid` is too high of a value for the minimum piece sweetness that we can assure for ourselves after doing the cuts. Thus, we update `r` to `mid - 1`.
- The loop will exit when `l` equals `r`, which means we found the maximum sweetness that can still satisfy the conditions. The solution returns `l`, which is the optimal maximum sweetness we can achieve for our piece of chocolate.

This algorithm ensures we arrive at the ideal maximum minimum sweetness while making sure that the rest of the chocolate pieces are fair enough to be distributed to the `k` friends.

Example Walkthrough

Let's go through an example to illustrate the solution approach. Suppose the `sweetness` array is `[1, 2, 3, 4, 5, 6, 7, 8, 9]`, and you have `k=2` friends.

Our goal is to find the maximum sweetness for the least sweet piece we can have after `k` cuts.

1. We initialize `l=0` and `r=45` because 45 is the total sweetness (sum of all chunks) and the `k+1` pieces implies that the maximum single piece's sweetness can't be more than `45/3=15`.
2. We create a helper function, `check(x)`, which will go through the sweetness array, summing the values until it reaches or exceeds the current guess x, and then count a cut.
3. Enter the binary search while-loop:
   - First iteration: `l=0`, `r=45`, so `mid=(0+45+1) >> 1 = 23`.
     - Check if it's possible to cut the bar so that each piece is at least `23` sweetness: No, because the highest sum we can get before the total sum exceeds 23 is `1+2+3+4+5+6=21`, then `7+8+9=24`, which only makes 1 cut. So, update `r` to `mid - 1`, which is `22`.
   - Second iteration: `l=0`, `r=22`, so `mid=(0+22+1) >> 1 = 11`.
     - Check if it's possible with `11`: Yes, one piece can be `1+2+3+4=10` (which is just under 11), the next can be `5+6=11`, and the last piece will be `7+8+9=24` (far exceeding 11, but it's the last piece, so it's okay). We managed more than `k` cuts here, so update `l` to `mid`, which is `11`.
   - Third iteration: `l=11`, `r=22`, so `mid=(11+22+1) >> 1 = 17`.
     - With `17`, we can't reach the minimum sweetness before we've hit the end of the array (again, we get chunks like `1+2+3+4+5=15` and then `6+7+8+9=30`), resulting in only 1 cut, so update `r` to `mid - 1`, which is `16`.
   - Fourth iteration: `l=11`, `r=16`, so `mid=(11+16+1) >> 1 = 14`.
     - Attempt with `14`: We can have the pieces `1+2+3+4+5=15` and `6+7=13` (below 14, so we keep adding) then `8+9=17`, we're just making 2 cuts. Since we succeeded in making `k` cuts before the sum fell short of `14`, we can continue searching between `14` to `16`.
   - The process will continue, narrowing down the range until `l` equals `r`.
4. By the end of the last iteration, when `l` equals `r`, we have found the right cut-off (`mid`) where we can still have more than `k` cuts with each piece being at least as sweet as our final value of `l`.
5. The final value of `l` will be our answer. In this example, the maximum sweetness for the least sweet piece you can have would be `11` after iterating this process to the point where `l` and `r` converge.

Python Solution

```python
from typing import List

class Solution:
    def maximizeSweetness(self, sweetness: List[int], k: int) -> int:
        # Helper function to check if the minimum sweetness value 'min_sweetness'
        # allows us to divide the chocolate into more than k+1 pieces
        def can_divide(min_sweetness: int) -> bool:
            total = 0
            pieces = 0
            # Go through each piece of the chocolate
            for piece_sweetness in sweetness:
                total += piece_sweetness
                # If the total sweetness is at least 'min_sweetness',
                # we can form a new piece
                if total >= min_sweetness:
                    total = 0
                    pieces += 1
            # The +1 is because we need to divide the chocolate into k+1 pieces
            return pieces > k

        # Initialize binary search bounds
        # 'left' is the minimum possible sweetness, 'right' is the maximum possible sweetness
        left, right = 1, sum(sweetness)
        # Perform binary search to find the maximum minimum sweetness
        while left < right:
            # Calculate the middle point of the current search range
            mid = (left + right + 1) // 2
            # If we can divide the chocolate with 'mid' sweetness,
            # we try to find a higher minimum sweetness value
            if can_divide(mid):
                left = mid
            else:
                # Otherwise, if we cannot, we look for a smaller minimum sweetness value
                right = mid - 1
        # Return the maximum minimum sweetness value we found
        return left

```

Java Solution

```java
class Solution {
    public int maximizeSweetness(int[] sweetness, int k) {
        // Initialize the range of sweetness we will search within.
        int left = 1; // The minimum sweetness can't be less than 1 (assuming sweetness array is positive).
        int right = 0; // The maximum sweetness possible.

        // Calculate the total sweetness in the array, which is the upper limit for binary search.
        for (int sweet : sweetness) {
            right += sweet;
        }

        // Perform a binary search to find the maximum sweetness that can be achieved.
        while (left < right) {
            // Use the mid point of the current range to test the sweetness level.
            int mid = (left + right + 1) >>> 1;

            // Check if it's possible to have more than k pieces with at least 'mid' sweetness.
            if (canSplit(sweetness, mid, k)) {
                left = mid; // If possible, search towards the higher end of the range.
            } else {
                right = mid - 1; // Otherwise, search towards the lower end of the range.
            }
        }

        // Return the maximum sweetness level found.
        return left;
    }

    private boolean canSplit(int[] sweetnessArray, int minimumSweetness, int k) {
        int currentSum = 0; // Current piece sweetness sum.
        int pieces = 0;     // Number of pieces formed.

        // Sum up sweetness pieces and count how many are >= minimumSweetness.
        for (int sweet : sweetnessArray) {
            currentSum += sweet;
            // When we reach the minimum sweetness for the current piece, increment piece count.
            if (currentSum >= minimumSweetness) {
                pieces++;
                currentSum = 0; // Reset the sum for the next piece.
            }
        }

        // Check if we can have one more piece for the divider (k pieces means k+1 friends).
        // Return true if pieces count is greater than k.
        return pieces >= k + 1;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <numeric>

class Solution {
public:
    // Function to maximize the minimum sweetness of the pieces divided
    // Parameters:
    // sweetness: vector of integers where each integer represents the sweetness of a piece
    // k: number of friends to share the sweetness with (pieces to divide into k+1)
    int maximizeSweetness(vector<int>& sweetness, int k) {
        int left = 1; // minimum possible sweetness
        int right = accumulate(sweetness.begin(), sweetness.end(), 0); // maximum possible sweetness

        // Helper lambda to check if a given minimum sweetness value is achievable
        auto canAchieveSweetness = [&](int minSweetness) {
            int currentSum = 0, cuts = 0;
            for (int pieceSweetness : sweetness) {
                currentSum += pieceSweetness;
                if (currentSum >= minSweetness) {
                    currentSum = 0;
                    ++cuts; // Increase count of number of cuts/pieces
                }
            }
            return cuts >= k + 1; // Check if we can make k+1 pieces
        };

        // Binary search to find the maximum of the minimum sweetness that we can achieve
        while (left < right) {
            int mid = left + (right - left + 1) / 2; // Midpoint and to avoid integer overflow

            // If the current mid can achieve the minimum sweetness, search towards the higher end
            if (canAchieveSweetness(mid)) {
                left = mid;
            } else {
                right = mid - 1; // Otherwise, search towards the lower end
            }
        }

        // left will be the maximum of the minimum sweetness we can achieve
        return left;
    }
};

```

Typescript Solution

```typescript
function maximizeSweetness(sweetness: number[], k: number): number {
    let left = 0; // Initialize the lower bound of binary search
    let right = sweetness.reduce((a, b) => a + b); // Initialize the upper bound as the sum of sweetness

    // Define a helper function to check if it's possible to split the sweetness array
    // into more than k parts where each part has a minimum total sweetness of x
    const canSplit = (minimumSweetness: number): boolean => {
        let currentSweetness = 0;
        let partsCount = 0;
        for (const value of sweetness) {
            currentSweetness += value; // Add the sweetness value to the current sum
            // If the current sum reaches the minimum sweetness threshold, we start a new part
            if (currentSweetness >= minimumSweetness) {
                currentSweetness = 0; // Reset the current sweetness for the next part
                partsCount++; // Increment the count of parts
            }
        }
        // Check if we can have more than k parts with the specified minimum sweetness
        return partsCount > k;
    };

    // Use binary search to find the maximum minimum sweetness
    while (left < right) {
        const mid = (left + right + 1) >> 1; // Equivalent to Math.floor((left + right + 1) / 2)
        // If it's possible to split into more than k parts with mid as minimum sweetness,
        // it means we can try a higher minimum. So we update left to mid.
        if (canSplit(mid)) {
            left = mid;
        } else {
            // Otherwise, we need to look for a smaller minimum sweetness
            right = mid - 1;
        }
    }
    // After binary search, left will be the maximum minimum sweetness we can achieve
    return left;
}

```

Time and Space Complexity

The given Python code aims to maximize the sweetness of the piece a person can get after dividing the chocolate `sweetness` array into `k + 1` pieces. It uses a binary search approach to find the solution.

Time Complexity

The binary search runs while `l < r`, which typically gives us a `O(log n)` complexity, where `n` is the range of values to be searched—in this case, the total sum of the `sweetness` array (`sum(sweetness)`).

Within each iteration of the binary search, the `check` function is called, which itself runs through the entire array once. Therefore, this part has a time complexity of `O(m)`, where `m` is the length of the `sweetness` array.

Combining these two parts together, we get a total time complexity of `O(m * log n)` for the entire algorithm, taking into account the binary search and the linear scan in the `check` function for each mid value.

Space Complexity

The space complexity is `O(1)`. No additional space is allocated that grows with the size of the input, as the variables `l`, `r`, `mid`, `s`, `cnt`, and `x` only use a constant amount of space.