1891. Cutting Ribbons

Medium Array Binary Search

Leetcode Link

Problem Description

In this problem, you are given an array of integers named `ribbons`, where each element represents the length of a particular ribbon. Additionally, you are given an integer `k`, which represents the target number of ribbons that you want to obtain. The challenge is to cut these ribbons into segments of equal length. Importantly, the length of each segment must be a positive integer, and you're looking for the maximum possible length of these segments that will allow you to create exactly `k` segments.

You have the flexibility to cut the ribbons in various ways or even choose not to cut a particular ribbon at all. Any leftover ribbon after achieving your target of `k` segments can be discarded. If it is not possible to get `k` segments with the same positive integer length, then the answer will be `0`.

The task is to determine the longest segment length that you can achieve to get `k` equal-length segments from the initial ribbons.

Intuition

To arrive at the solution for this problem, we utilize a binary search approach due to the monotonic nature of the problem's conditions. The key insight is that if you can make `x` ribbons of a certain length `y`, then you can certainly make `k` ribbons of any length less than `x`. This forms the basis of a binary search where we can efficiently narrow down the maximum length that meets the condition.

The binary search starts with the left boundary at `0` and the right boundary at the maximum length of the ribbons available, since the segment length cannot exceed the longest ribbon that we have. We continuously adjust our search range based on whether the mid-value we are testing allows us to obtain `k` or more ribbons of that length. If it does, we have a potential solution and we try to find an even longer length by moving our search range rightward (toward longer lengths). If it does not, we must look for shorter lengths by moving the search range leftward.

By repeating this process, our search range converges to the maximum length that allows us to obtain exactly `k` ribbons of that length.

Solution Approach

The solution employs a binary search algorithm to efficiently find the maximum length of ribbon that we can cut to achieve exactly `k` segments of equal length. Binary search is chosen as the algorithm because of the monotonic relationship between ribbon length and the number of pieces — more ribbons can be obtained with shorter lengths and fewer with longer lengths. The purpose of binary search here is to maximize the length of the ribbons while still meeting the target quantity `k`.

To employ binary search, we establish two pointers `left` at `0` and `right` at the maximum length in the `ribbons` array, which represents the range within which we'll search for the maximum possible length. The algorithm proceeds iteratively, dividing this range by choosing a midpoint `mid`. In binary search, mid is typically calculated as `(left + right) / 2`, but in this solution, we use a slightly different formula, `mid = (left + right + 1) >> 1` to avoid integer underflows when using larger int values, and shift the result rightward by one bit, which is equivalent to integer division by 2.

For each `mid` value chosen during the binary search:

1. We calculate `cnt` which is the total number of ribbons that can be cut at the given `mid` length. This is done by iterating through the `ribbons` array and dividing each ribbon's length by `mid` (the proposed length). We use integer division here because we can only use complete segments.
2. If `cnt` is greater than or equal to (>=) `k`, it means that cutting the ribbons into `mid` length segments satisfies our requirement for at least `k` segments. Therefore, we update `left` to `mid`, as there might exist a longer segment length that also meets the requirement.
3. If `cnt` is less than `< k`, then we can't obtain enough ribbons at the proposed length and have to try shorter lengths. Hence, we set the new `right` to `mid - 1`.

This search range narrowing down continues until `left` and `right` converge, meaning `left` will point to the maximum length that we can use to cut the ribbons into exactly `k` segments.

Finally, we return the `left` pointer's value, as this will have converged to the correct maximum segment length through the iterations.

Overall, this solution approach takes advantage of the sorted and monotonic nature of the resultant ribbon lengths to apply binary search, which reduces the time complexity from potentially linear in nature to logarithmic — a significant efficiency boost, especially with larger input sizes.

Example Walkthrough

Let's consider a small example with `ribbons = [9, 7, 5]` and `k = 5`. We want to cut these ribbons into 5 segments of equal length. We'll use the binary search approach described to find the maximum length of these segments.

1. Initialize `left = 0` and `right = 9` (the longest ribbon length).
2. The `mid` value is calculated as `(left + right + 1) >> 1`. Initially, that would be `(0 + 9 + 1) >> 1 = 5`.
3. We check if we can get at least `k` segments of length `5` from the ribbons. We have:
   - `9 // 5` gives us `1` segment from the first ribbon
   - `7 // 5` gives us `1` segment from the second ribbon
   - `5 // 5` gives us `1` segment from the third ribbon In total, we have `3` segments, which is less than `k`, so we cannot use length `5`.
4. Since the count of segments is less than `k`, we adjust the `right` to be `mid - 1`, which is now `5 - 1 = 4`.
5. Recalculate `mid` with the new boundaries: `(left + right + 1) >> 1` which now is `(0 + 4 + 1) >> 1 = 2`.
6. Check if we can get at least `k` segments of length `2` from the ribbons. This time, we have:
   - `9 // 2` gives us `4` segments from the first ribbon
   - `7 // 2` gives us `3` segments from the second ribbon
   - `5 // 2` gives us `2` segments from the third ribbon In total, `4 + 3 + 2` gives us `9` segments, which is more than `k`. We can use length `2`, and there might be a possibility for a longer length, so we continue.
7. Now, we move the left boundary up to `mid`, making `left = 2`.
8. With the boundaries `left = 2` and `right = 4`, we recalculate `mid = (2 + 4 + 1) >> 1 = 3`.
9. We check for length `3`:
   - `9 // 3` gives us `3` segments
   - `7 // 3` gives us `2` segments
   - `5 // 3` gives us `1` segment In total, we get `6` segments, which is again more than `k`.
10. Since we can still get more than `k` segments, we move `left` up to `mid`, setting `left` to `3`.
11. Now `left` and `right` are equal at `4`, and we calculate `mid` one more time: `(3 + 4 + 1) >> 1 = 4`.
12. Checking length `4`, we find:
    - `9 // 4` gives us `2` segments
    - `7 // 4` gives us `1` segment
    - `5 // 4` gives us `1` segment The total is `4`, which is less than `k`.
13. Having found that length `4` is not enough, we adjust `right` to `mid - 1`, resulting in `right = 3`, where `left` was already `3`.

At this point, the search range has narrowed such that `left` and `right` converge to `3`. We have determined that the maximum segment length that can be achieved to get exactly `k` segments from the initial ribbons is `3`. Thus, we return `3` as the answer.

Python Solution

```python
from typing import List

class Solution:
    def maxLength(self, ribbons: List[int], k: int) -> int:
        # Initialize the search boundaries.
        left, right = 0, max(ribbons)

        # Perform the binary search to find the maximum length.
        while left < right:
            # Calculate the middle value between left and right, rounding up.
            mid = (left + right + 1) // 2

            # Calculate the number of ribbons after cutting with the current mid length.
            count = sum(ribbon // mid for ribbon in ribbons)

            # If the current mid allows us to make at least k ribbons, search for a longer length.
            # Otherwise, search for a shorter length.
            if count >= k:
                left = mid
            else:
                right = mid - 1

        # Return the maximum length found where at least k ribbons can be made.
        return left
```

Java Solution

```java
class Solution {
    public int maxLength(int[] ribbons, int k) {
        int minLength = 0; // Set the minimum possible length of a ribbon to 0.
        int maxLength = 0; // This will hold the maximum length of a ribbon found in the array.

        // Find the longest ribbon in the array.
        for (int ribbonLength : ribbons) {
            maxLength = Math.max(maxLength, ribbonLength);
        }

        // Establish a binary search to find the maximum length of the ribbon
        // that can be used to cut into k pieces or more.
        while (minLength < maxLength) {
            // Calculate the middle length between the current minLength and maxLength,
            // biased towards the upper bound to avoid infinite loop.
            int midLength = (minLength + maxLength + 1) >>> 1;

            // Initialize a counter to keep track of how many ribbons we can cut.
            int count = 0;

            // Calculate the number of ribbons that can be cut with midLength.
            for (int ribbonLength : ribbons) {
                count += ribbonLength / midLength; // Integer division automatically floors the result.
            }

            // If we can cut at least k ribbons of midLength, increase minLength.
            // This implies we can try longer lengths.
            if (count >= k) {
                minLength = midLength;
            } else {
                // Otherwise, decrease maxLength to try shorter lengths.
                maxLength = midLength - 1;
            }
        }

        // Return the maximum length we found that can cut at least k ribbons.
        return minLength;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    // Function to find the maximum length of ribbon pieces that can be cut
    // so that we have at least 'k' ribbon pieces.
    int maxLength(vector<int>& ribbons, int k) {
        // Initialize the 'left' boundary of our binary search to 0
        // and 'right' boundary to the length of the longest ribbon.
        int left = 0, right = *max_element(ribbons.begin(), ribbons.end());

        // Use binary search to find the maximum length of ribbon pieces.
        while (left < right) {
            // 'mid' is the midpoint length that we will test to see if we can
            // get at least 'k' ribbon pieces of that length.
            // We add 1 before dividing by 2 to round up for parity, alternative to bitwise operation
            // Initialize count of total ribbon pieces we can get with the current 'mid'.
            int mid = (left + right + 1) / 2, count = 0;

            // Calculate how many pieces of length 'mid' we can get for each ribbon.
            for (int ribbon : ribbons) {
                count += ribbon / mid;
            }

            // If we can get at least 'k' pieces, try finding a longer length;
            // otherwise, we try shorter lengths.
            if (count >= k) {
                left = mid; // Notice we use 'mid' rather than 'mid+1' because we're already considering mid
            } else {
                right = mid - 1;
            }
        }

        // Return the maximum length of ribbon we found; 'left' is our best valid guess.
        return left;
    }
};
```

Typescript Solution

```typescript
// Define the function with a clear signature, describing the parameter types and return type.
// This function finds the maximum length of the ribbon pieces that we can cut such that we have at least 'k' pieces.
function maxLength(ribbons: number[], k: number): number {
    // Initialize the search bounds for binary search.
    let lowerBound = 0;
    let upperBound = Math.max(...ribbons);

    // Utilize a binary search to find the maximum length.
    while (lowerBound < upperBound) {
        // Calculate the middle value of the current search range.
        const middle = Math.floor((lowerBound + upperBound + 1) / 2);

        // Initialize the counter for how many ribbon pieces we can cut.
        let count = 0;

        // Calculate the number of pieces we can cut from each ribbon.
        for (const ribbonLength of ribbons) {
            count += Math.floor(ribbonLength / middle);
        }

        // If we can cut at least 'k' pieces, the piece length 'middle' is valid, so move the lower bound up.
        if (count >= k) {
            lowerBound = middle;
        } else {
            // Otherwise, the piece length is too high, so we reduce the upper bound.
            upperBound = middle - 1;
        }
    }

    // Return the maximum length of ribbon pieces we can cut.
    return lowerBound;
}
```

Time and Space Complexity

Time Complexity

The time complexity of the code is `O(n * log M)`. Here `n` refers to the number of ribbons, and `M` is the maximum length of the ribbons. The code uses a binary search process to converge on the maximum length of the ribbon that can be cut to produce at least `k` ribbons of equal length. The search range is between 0 and the maximum ribbon length, which gives us the `log M` term because the search space is being halved in each iteration of the while loop. In each iteration, we go through all `n` ribbons to calculate the total number of ribbons of the current length (`mid`), leading to the multiplier of `n`.

Space Complexity

The space complexity of the code is `O(1)`. This is because the space used does not grow with the input size. Only a constant amount of extra space is used for variables `left`, `right`, `mid`, and `cnt`, irrespective of the size of the input array `ribbons`.