1894. Find the Student that Will Replace the Chalk

Medium | Array | Binary Search | Prefix Sum | Simulation

Problem Description

In this problem, you're given an integer array `chalk` and an integer `k`, which represent the number of chalk pieces available for a group of students to use. Each element in the `chalk` array corresponds to the amount of chalk that each student will use to solve a problem. For example, `chalk[0]` is the amount of chalk the first student will use, `chalk[1]` the amount for the second student, and so on, in a repeating cycle.

The sequence starts with the first student (index 0) and goes to the last student (index `n - 1`). After the teacher has distributed problems to all students, the cycle starts again with the first student. This continues until there aren't enough chalk pieces for a student to use. At that point, the student will be asked to replace the chalk.

The task is to find out which student will be asked to replace the chalk. In other words, you need to return the index of the first student who cannot proceed because the chalk pieces left are fewer than what they need.

Intuition

To solve this problem, it's efficient to use a prefix sum and binary search.

Prefix Sum: First, we calculate the cumulative chalk usage for each student, which gives us a sequence where each element tells us the total amount of chalk used by all students up to that point. This can be done using the `accumulate` function from the `itertools` module in Python, which creates a list of accumulated sums.

Binary Search: Then, because the cycle starts again after reaching the last student, we can take the total amount of chalk `k` and find the remainder when divided by the total accumulated chalk. This tells us how much chalk would be left after an integer number of full cycles, and the next student to take a problem would need more chalk than what we have. Using this leftover amount of chalk, we can perform a binary search to find the first student who would need more chalk than the remnants. The built-in function `bisect_right` from the `bisect` module in Python can help us here. It searches for the place to insert the leftover chalk amount in order to maintain the sorted order. This position, effectively the index of the student, is the output of our function.

Solution Approach

The solution approach for this problem is to first calculate the prefix sum of the chalk requirements for each student and then use a binary search algorithm to find the index of the student who will replace the chalk.

Here's a step-by-step breakdown of the implementation described by the provided solution code:

1. **Calculate Prefix Sum**: Using the `accumulate` function from the `itertools` module, we calculate the cumulative sum of the `chalk` array. The result is a new list `s` where `s[i]` represents the total amount of chalk used by the first `i + 1` students. This step is crucial because it allows us to understand the total chalk consumption in one complete cycle through all students.
2. **Modulo Operation**: The value of `k` may be very large, possibly larger than the total amount of chalk used in a full cycle by all students. To handle this, we use the modulo operation `k %= s[-1]` which gives us the remainder of `k` after dividing by the total chalk used in one cycle (`s[-1]` represents the total chalk used after all students have taken a problem once). This effectively reduces the problem to a single incomplete cycle, making it easier to find the student who will replace the chalk.
3. **Binary Search**: Now, we need to identify the first student who cannot proceed due to insufficient chalk. This is done by using the `bisect_right` function from the `bisect` module, which performs a binary search. The function finds the position where the leftover chalk `k` would be inserted to maintain the sorted order of the accumulated chalk list `s`. The returned index `bisect_right(s, k)` is the index of the first student whose chalk requirement exceeds the amount of leftover chalk `k`.

Combining these steps, the code successfully determines the student who will replace the chalk:

```python
class Solution:
    def chalkReplacer(self, chalk: List[int], k: int) -> int:
        s = list(accumulate(chalk))  # step 1
        k %= s[-1]                   # step 2
        return bisect_right(s, k)    # step 3
```

Even though this approach appears straightforward, it's highly efficient because both the prefix sum and binary search operate in `O(n)` and `O(log n)` time complexities, respectively, making the whole solution quite performant for even large input sizes.

Example Walkthrough

Let's walk through the solution approach with a small example to illustrate how it works in practice. Consider the `chalk` array `[5, 1, 3]` and `k = 11`:

1. We first calculate the prefix sum, which tells us the cumulative amount of chalk used by the students in one round:

```
Original chalk array: [5, 1, 3]
Prefix sum: [5, 6, 9]
```

With this, we can see that one full cycle of chalk usage by all students amounts to `9`.

2. Now we apply the modulo operation to `k` with the total chalk used in one cycle:

```
k = 11
Total chalk in one cycle = 9 (from the last element of the prefix sum)
k % 9 = 2
```

The modulo gives `2`, so after completing one full round, we have `2` chalks left.

3. Using binary search, we find where `2` (the remaining chalk pieces) would be inserted in our prefix sum array to maintain the sorted order:

```
Prefix sum array: [5, 6, 9]
Leftover: 2
```

A binary search for `2` in the array `[5, 6, 9]` would place it before the first element since `2` is less than `5`. Therefore, it would be inserted at index `0`.

Thus, the student at index `0` (the first student) would be the one who can't proceed because they require `5` chalks, and we only have `2` remaining. Hence, the answer is `0`, the index of the first student in our `chalk` array.

Solution Implementation

Python

```python
# First, we need to import the required functions from the itertools and bisect modules.
from itertools import accumulate
from bisect import bisect_right

class Solution:
    def chalkReplacer(self, chalk: List[int], k: int) -> int:
        # Accumulate the chalk requirements in a list
        # to find out the total chalk needed for one round.
        total_chalk_per_round = list(accumulate(chalk))

        # The remaining chalk after multiple complete rounds
        # is the remainder of k divided by the total chalk per round.
        k %= total_chalk_per_round[-1]

        # Find the index of the smallest number in the accumulated
        # chalk list that is greater than k using binary search.
        # This is the index of the student who will be the next to receive chalk.
        return bisect_right(total_chalk_per_round, k)

# We need to import the typing module to use List[int] as a type hint in the function signature
from typing import List
```

Java

```java
class Solution {
    public int chalkReplacer(int[] chalk, int k) {
        int numberOfStudents = chalk.length;

        // Define a prefix sum array with an extra space for easier calculations
        long[] prefixSum = new long[numberOfStudents + 1];

        // Calculate the prefix sum of chalk requirements
        // Here, starting index of prefixSum should be 1 to match the chalk indices.
        prefixSum[1] = chalk[0];
        for (int i = 1; i < numberOfStudents; ++i) {
            prefixSum[i + 1] = prefixSum[i] + chalk[i];
        }

        // Find the remaining chalk after it has been distributed once
        k %= prefixSum[numberOfStudents];

        // Initialize search boundaries for binary search
        int left = 0, right = numberOfStudents;

        // Performing binary search to find the minimum index 'left'
        // such that prefixSum[left] is greater than 'k'
        while (left < right) {
            int mid = (left + right) >> 1; // Equivalent to (left + right) / 2 but more efficient

            // If the sum up to mid is greater than k, search the left half
            if (prefixSum[mid] > k) {
                right = mid;
            } else { // Otherwise, search the right half
                left = mid + 1;
            }
        }

        // The index 'left' points to the first student that cannot finish
        // their chalk round, that's the student to start the next round
        return left - 1; // Adjusting the index because prefixSum started from 1 instead of 0
    }
}
```

C++

```cpp
#include <vector>
#include <algorithm> // Needed for std::upper_bound

class Solution {
public:
    // Function to find the index of the student who will receive the last chalk piece before the chalk runs out
    int chalkReplacer(vector<int>& chalk, int k) {
        int numStudents = chalk.size();
        vector<long long> prefixSum(numStudents, chalk[0]);
        // Calculate prefix sums of chalk requirements
        for (int i = 1; i < numStudents; ++i) {
            prefixSum[i] = prefixSum[i - 1] + chalk[i];
        }
        // The amount of chalk K is now how much chalk will be needed in the last round
        // because we are interested in a full cycle where every student has received chalk exactly once
        k %= prefixSum[numStudents - 1];

        // Use upper bound to return the first element in the prefix sum which is greater than k
        // this will be the first student who will not be able to finish writing as the chalk will run out before this student
        return std::upper_bound(prefixSum.begin(), prefixSum.end(), k) - prefixSum.begin();
    }
};
```

TypeScript

```typescript
// Import the necessary functions from `lodash` for calculating the prefix sums and finding the upper bound.
import _ from 'lodash';

// Function to calculate prefix sums of an array.
const calculatePrefixSums = (chalk: number[]): number[] => {
  const prefixSums: number[] = [];
  chalk.reduce((acc, current) => {
    const sum = acc + current;
    prefixSums.push(sum);
    return sum;
  }, 0);
  return prefixSums;
};

// Main function to find the index of the student who will receive the last chalk piece before the chalk runs out.
const chalkReplacer = (chalk: number[], k: number): number => {
  const numStudents = chalk.length;
  const prefixSums: number[] = calculatePrefixSums(chalk);
  // Adjust the number of chalk (k) by taking modulus with the total sum to find remainder chalk after full cycles.
  k %= prefixSums[prefixSums.length - 1];

  // Use lodash's sortedIndex to find the smallest index at which k should be inserted
  // into `prefixSums` in order to maintain the array's sorted order.
  const studentIndex: number = _.sortedIndex(prefixSums, k);

  // This index represents the student who cannot finish writing as the chalk runs out before their turn.
  return studentIndex;
};

export { chalkReplacer };
```


```python
# First, we need to import the required functions from the itertools and bisect modules.
from itertools import accumulate
from bisect import bisect_right

class Solution:
    def chalkReplacer(self, chalk: List[int], k: int) -> int:
        # Accumulate the chalk requirements in a list
        # to find out the total chalk needed for one round.
        total_chalk_per_round = list(accumulate(chalk))

        # The remaining chalk after multiple complete rounds
        # is the remainder of k divided by the total chalk per round.
        k %= total_chalk_per_round[-1]

        # Find the index of the smallest number in the accumulated
        # chalk list that is greater than k using binary search.
        # This is the index of the student who will be the next to receive chalk.
        return bisect_right(total_chalk_per_round, k)

# We need to import the typing module to use List[int] as a type hint in the function signature
from typing import List
```


Time and Space Complexity

Time Complexity

The time complexity of the given code is determined by two main operations: accumulating the `chalk` array and performing the binary search.

- The `accumulate` function computes the prefix sums of the `chalk` list, which takes `O(n)` time, where `n` is the length of `chalk`.
- The modulus operation `k %= s[-1]` is constant time, `O(1)`.
- `bisect_right` is a binary search function, which takes `O(log n)` time to find the insert position for `k` in the sorted list `s`.

Combining these complexities, we get `O(n + log n)` which simplifies to `O(n)` since `O(n)` dominates `O(log n)`.

Space Complexity

The space complexity of the code comes from storing the prefix sums in the list `s`. Since this list has the same length as the input list `chalk`, the space complexity is `O(n)`.