2557. Maximum Number of Integers to Choose From a Range II

Problem Description

We are given an integer array `banned`, a positive integer `n` indicating the upper limit of an integer range starting from 1, and a positive integer `maxSum` which serves as the cap for the sum of integers we can select. The task is to choose a collection of distinct integers within this range from 1 to `n`, ensuring two conditions: first, that none of the integers are present in the `banned` array, and second, that the cumulative sum of all chosen integers does not exceed `maxSum`. The objective is to maximize the number of integers chosen while adhering to the stated constraints.

Intuition

To maximize the number of chosen integers, while considering the `maxSum` limitation and banned list, we must take a strategic approach in picking the integers. A naive approach might be to start from 1 and keep adding the next non-banned integer until we reach or exceed `maxSum`. However, this might not optimize for the maximum number of integers since larger values will use up the allowed sum more quickly.

Instead, the intuition is to proceed in a way that we try to include as many small non-banned numbers as possible because smaller numbers have smaller sums, enabling us to pick more numbers while staying under `maxSum`.

We can achieve this by:

- Extending the `banned` list with two sentinel values, 0 and `n+1`, which represent the lower and upper boundaries beyond which we can't select integers.
- Sorting the extended `banned` list to be able to iterate through banned ranges efficiently.
- Using binary search to find the maximum count of integers that can be selected between each pair of consecutive banned numbers, without exceeding `maxSum`.

This can be seen in the provided solution code where the `banned` list is augmented and then sorted. The `pairwise` function (assumed to be similar to itertools.pairwise) iterates through adjacent elements providing pairs of banned integers that we can calculate the potential numbers between them.

The binary search within the loop efficiently finds the maximum number of selectable integers in each range. If adding one more integer exceeds `maxSum`, the binary search moves left, else it moves right to expand the count. Once the range is found, we add it to the `ans` tally and deduct the sum of those integers from `maxSum` to ensure we don't exceed the overall allowed sum.

The answer is returned as soon as `maxSum` is not enough to add more integers, guaranteeing the focus on maximizing quantity with the smallest possible numbers while adhering to `maxSum`.

Solution Approach

The code below implements a strategic method to solve the problem by maximizing the number of integers chosen within the given constraints. Here's a deeper dive into the solution's methodology:

First, the solution uses both the concept of sorting and binary search, combining these algorithms for efficient computation. The `banned` list is modified to include two extra elements, `0` and `n + 1`, to handle edge cases seamlessly.

The data structures used include:

- A list (`banned`) that contains the original banned numbers plus the two sentinels (0 and `n + 1`).
- A set, to ensure all elements in `banned` are unique.
- A sorted list version of that set, to allow for binary search operations.

The code then traverses the sorted `banned` list, searching for the maximum number of selectable integers between each pair of consecutive banned numbers (now including the sentinels). This is done using the `pairwise` iteration which is assumed to yield a tuple `(i, j)` where `i` and `j` are adjacent banned numbers.

For each pair `(i, j)`, the algorithm finds the potential gap (`j - i - 1`) between them to determine the number of selectable integers. It then employs binary search within this gap to find the maximum count of integers (`cnt`) which can be chosen without exceeding `maxSum`. The binary search checks if the sum of an arithmetic series from `i + 1` to `i + mid` is less than or equal to `maxSum`.

The arithmetic series sum is calculated with the formula:

```
1 (i + 1 + i + mid) * mid / 2
```

This sum formula computes the sum of the first `mid` terms of the natural numbers starting at `i + 1`.

After finding the `left` value for each range, the algorithm updates the answer (`ans`) with the count of new integers added, and subtracts the sum of those integers from `maxSum`.

The process stops when `maxSum` becomes zero or negative, indicating that no additional integers can be chosen without exceeding the maximum sum constraint at which point the current value of `ans` is returned.

Through this approach, the solution efficiently maximizes the number of integers chosen without exceeding `maxSum` and while respecting the `banned` list.

Example Walkthrough

Let's consider an example to illustrate the solution approach.

Suppose we are given the following inputs:

- `banned = [3, 6, 10]`
- `n = 15`
- `maxSum = 25`

Now, let's use the solution approach step by step:

1. **Extend and Sort the `banned` List:** The `banned` array is extended to include two sentinel values, which here would be `0` and `n+1 = 16`. After including these, the `banned` array is sorted. The updated `banned` list looks like this: `[0, 3, 6, 10, 16]`.
2. **Iterate Using Pairwise:** Using the `pairwise` function, we obtain the adjacent pairs of banned numbers: `(0, 3)`, `(3, 6)`, `(6, 10)`, and `(10, 16)`.
3. **Binary Search on Each Range:** For each pair `(i, j)`, we perform a binary search to find the maximum count of integers that can be selected between `i+1` and `j-1` without exceeding `maxSum`. Here's how it works for each pair:
   - For `(0, 3)`, the selectable range is `1` and `2`. Since the sum `1 + 2` is `3`, which is less than `maxSum`, we include both numbers. Now, `maxSum` is reduced to `22`.
   - For `(3, 6)`, the selectable range is `4` and `5`. The sum `4 + 5` is `9`, which is within the remaining `maxSum` of `22`. We include these, and `maxSum` reduces to `13`.
   - Next, for `(6, 10)`, the selectable range is `7`, `8`, and `9`. The binary search finds that we can only take `7` and `8` without exceeding the new `maxSum` of `13`, because their sum is `15`, and adding `9` would bring the total to `24`, over the cap. So we add `7` and `8` and reduce `maxSum` to `13 - (7 + 8) = -2`. Now `maxSum` is negative.
   - At this point, we cannot choose any more numbers because the remaining `maxSum` is not enough to include more integers. We stop and do not proceed to the pair `(10, 16)`.
4. **End Result:** The maximum number of integers chosen without exceeding `maxSum` and avoiding the `banned` list is `[1, 2, 4, 5, 7, 8]`. The `ans` would be `6`, which is the count of these chosen integers.

This example walk-through demonstrates how the problem is solved using the described approach, ensuring that we maximize the number of integers chosen within the constraints.

Python Solution

```python
class Solution:
    def maxCount(self, banned: List[int], n: int, maxSum: int) -> int:
        # Extend the banned list to include 0 and n+1 to simplify range calculations
        banned.extend([0, n + 1])
        # Ensure the banned list is unique and sorted
        banned = sorted(set(banned))

        # Initialize the answer to 0
        ans = 0

        # Loop through each pair of consecutive banned numbers (Using pairwise requires itertools in Python 3.8+)
        for lower_bound, upper_bound in zip(banned, banned[1:]):
            # We use binary search to find the maximum count of numbers
            # between a pair of banned numbers that we can sum without exceeding maxSum
            left, right = 0, upper_bound - lower_bound - 1

            while left < right:
                # Calculate the midpoint for binary search
                mid = (left + right + 1) // 2

                # Calculate the sum of an arithmetic series from lower_bound+1 to lower_bound+mid
                if ((lower_bound + 1 + lower_bound + mid) * mid) // 2 <= maxSum:
                    # If the sum is less than or equal to maxSum, move the left bound up
                    left = mid
                else:
                    # Otherwise, move the right bound down
                    right = mid - 1

            # Add the count of numbers found to the answer
            answer += left
            # Decrease maxSum by the sum of numbers we've added to the answer
            maxSum -= (lower_bound + 1 + lower_bound + left) * left // 2

            # If maxSum becomes zero or negative, we cannot add any more numbers
            if maxSum <= 0:
                break

        # Return the total count of numbers we can sum
        return answer
```

Java Solution

```java
class Solution {
    // Method to calculate the maximum count of consecutive numbers not exceeding maxSum
    // with the given constraints.
    public int maxCount(int[] banned, int n, long maxSum) {
        // Create a hash set to store banned numbers along with the boundaries.
        Set<Integer> bannedIndicesSet = new HashSet<>();
        bannedIndicesSet.add(0); // Add lower boundary
        bannedIndicesSet.add(n + 1);  // Add upper boundary

        // Add all banned numbers to the set
        for (int bannedNumber : banned) {
            bannedIndicesSet.add(bannedNumber);
        }

        // Convert the set to a list and sort it to process intervals between banned numbers.
        List<Integer> bannedIndicesList = new ArrayList<>(bannedIndicesSet);
        Collections.sort(bannedIndicesList);

        // Initialize the answer which will store the maximum count of consecutive numbers
        int maxCount = 0;

        for (int k = 1; k < bannedIndicesList.size(); ++k) {
            // Get the interval boundaries from the sorted list
            int intervalStart = bannedIndicesList.get(k - 1);
            int intervalEnd = bannedIndicesList.get(k);

            // Initialize binary search bounds
            int left = 0;
            int right = intervalEnd - intervalStart - 1;

            while (left < right) {
                // Use unsigned right shift for division by 2 to avoid negative overflow
                int mid = (left + right + 1) >>> 1;

                // Check if the sum of consecutive numbers from intervalStart2 to intervalStart+mid
                // fits in the maximum using the arithmetic progression sum formula
                if ((intervalStart + 1 + intervalStart + mid) * 1L * mid / 2 <= maxSum) {
                    left = mid;
                } else {
                    right = mid - 1;
                }
            }

            // Update maxCount and decrease maxSum by the sum of consecutive numbers found
            maxCount += left;
            maxSum -= (intervalStart + 1 + intervalStart + left) * 1L * left / 2;

            // If maxSum is exhausted, break out of the loop
            if (maxSum <= 0) {
                break;
            }
        }

        // Return the calculated maximum count
        return maxCount;
    }
}
```

C++ Solution

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

class Solution {
public:
    // This function calculates the maximum count of numbers that can be added
    // to the intervals between banned numbers without exceeding maxSum.
    int maxCount(vector<int>& banned, int n, long long maxSum) {
        // Add boundary markers for the start and end of possible number range.
        banned.push_back(0);     // Add a banned number at the start.
        banned.push_back(n + 1);  // Add a banned number at the end.

        // Sort the banned numbers to process them in order.
        sort(banned.begin(), banned.end());

        // Remove duplicate entries if any, to handle only unique banned numbers.
        banned.erase(unique(banned.begin(), banned.end()), banned.end());

        int answer = 0;  // Initialize the answer to zero.

        // Iterate through the intervals between banned numbers.
        for (int index = 1; index < banned.size(); ++index) {
            int start = banned[index - 1], end = banned[index];
            int low = 0, high = end - start - 1; // Low and high limits for current interval.

            // Perform binary search within the interval to find the max count.
            while (low < high) {
                int mid = (low + high + 1) / 2;
                // Calculate the sum of arithmetic progression.
                if ((start + 1 + start + mid) * 1LL * mid / 2 <= maxSum) {
                    low = mid;  // If the sum is within limit, search right side.
                } else {
                    high = mid - 1;  // If sum exceeds limit, search left side.
                }
            }

            answer += low;  // Add the found count to the answer.
            // Subtract the sum of the found numbers from maxSum.
            maxSum -= (start + 1 + start + low) * 1LL * low / 2;

            if (maxSum <= 0) {  // If there's no sum left to add numbers, break out.
                break;
            }
        }

        return answer;  // Return the total count of numbers that can be added.
    }
};

// Example usage:
// int[] bannedNumbers = {3, 6, 10};
// int n = 15;
// int maxSum = 25;
// console.log(maxCount(bannedNumbers, n, maxSum)); // Should print the result of the function.
```

Typescript Solution

```typescript
function maxCount(banned: number[], n: number, maxSum: number): number {
    // Add boundary markers for the start and end of the possible number range.
    banned.push(0);     // Add a banned number at the start.
    banned.push(n + 1);  // Add a banned number at the end.

    // Sort the banned numbers to process them in order.
    banned.sort((a, b) => a - b);

    // Remove duplicate entries if any, to handle only unique banned numbers.
    let bannedSet = Array.from(new Set(banned));

    let answer = 0;  // Initialize the answer to zero.

    // Iterate through the intervals between banned numbers.
    for (let index = 1; index < bannedSet.length; ++index) {
        let start = bannedSet[index - 1], end = bannedSet[index];
        let low = 0, high = end - start - 1; // Low and high limits for the current interval.

        // Perform binary search within the interval to find the max count.
        while (low < high) {
            let mid = low + Math.floor((high - low + 1) / 2);
            // Calculate the sum of the arithmetic progression.
            if ((start + 1 + start + mid) * mid / 2 <= maxSum) {
                low = mid;  // If the sum is within limit, search right side.
            } else {
                high = mid - 1;  // If sum exceeds limit, search left side.
            }
        }

        answer += low;  // Add the found count to the answer.
        // Subtract the sum of the found numbers from maxSum.
        maxSum -= (start + 1 + start + low) * low / 2;

        if (maxSum <= 0) {  // If there's no sum left to add numbers, break out.
            break;
        }
    }

    return answer;  // Return the total count of numbers that can be added.
}

// Example usage:
// let bannedNumbers = [3, 6, 10];
// let n = 15;
// let maxSum = 25;
// console.log(maxCount(bannedNumbers, n, maxSum)); // Should print the result of the function.
```

Time and Space Complexity

Time Complexity

The time complexity of the code is determined by mainly two operations: sorting the `banned` list and performing a binary search for each pair in the list after incorporating the additional elements and removing duplicates.

1. Sorting the `banned` list has a time complexity of $O(K * \log(K))$, where K is the number of elements in the `banned` list after extending it with `[0, n + 1]`. Since `ban` is constructed by converting `banned` to a set and sorting, the number of elements k is at most the original length of `banned` plus 2.
2. The `pairwise` function results in $O(K - 1)$ iterations since it will create pairs of adjacent elements in the sorted `ban` list.
3. For each pair `(i, j)` produced by `pairwise`, there is a binary search running to determine how many integers can be counted between `i` and `j` without exceeding `maxSum`. The binary search runs in $O(\log N)$ time in the worst case, where N is the difference `(j - i - 1)`.

Therefore, if N is the maximum value in the list before sorting (and after adding `0` and `n + 1`), then the worst-case scenario for the binary search is $O(\log N)$ time for each of $K - 1$ iterations.

Combining these, the total time complexity is $O(K * \log(K) + (K - 1) * \log(N)) \approx O(K * \log(K) + K * \log(N))$. Since K can be at most $n + 2$, the time complexity in terms of n is $O((n + 2) * \log(n + 2) + (n + 1) * \log(n))$.

Space Complexity

The space complexity of the code involves the additional space used by the `ban` list and the space for the variables used in the binary search.

1. The `ban` list stores at most K elements, which is at most the length of `banned` plus 2, thus giving us a space complexity of $O(K)$.
2. The variables used in the binary search (`left`, `right`, `mid`, `ans`, and `maxSum`) require constant space $O(1)$.

Therefore, the space complexity of the algorithm is $O(K)$. This is $O(n)$ in terms of the maximum possible length of the `ban` list when n is large compared to the number of banned elements.