2554. Maximum Number of Integers to Choose From a Range I

Medium | Greedy | Array | Hash Table | Binary Search | Sorting | Leetcode Link

Problem Description

The problem presents a task involving selection from a range of integers while adhering to certain constraints:

- You have to choose integers from a range between 1 and `n` (inclusive).
- Each integer in the range can be chosen at most once.
- There is a given list of banned integers (`banned`), which you cannot choose.
- The sum of the chosen integers cannot exceed a specified sum (`maxSum`).

Your goal is to select the maximum number of integers possible without violating any of the above rules. The result that needs to be returned is the count of such integers.

Intuition

To solve this problem, a greedy approach can be utilized. The intuition here is that to maximize the number of integers to select, one should start from the smallest possible integer and move up, because smaller integers add less to the total sum, thus allowing for more integers to be included without exceeding `maxSum`.

Here are the steps for the given greedy approach:

1. Initialize a counter (`ans`) to keep track of the count of integers chosen and a sum (`s`) to keep track of the sum of chosen integers.
2. Create a set (`ban`) from the `banned` list for efficient lookup of banned integers.
3. Start iterating over the integers in the range from 1 to `n` inclusive.
4. For each integer:
   - Check if adding this integer to the current sum (`s`) would exceed `maxSum`. If it would, break out of the loop as no more integers can be chosen without violating the sum constraint.
   - Check if the integer is not in the `ban` set. If the integer is not banned, increment the count (`ans`) by 1 and add the integer to the sum (`s`).
5. Return the count (`ans`) as the final answer.

By following this approach, each decision contributes to the overall objective of maximizing the count by choosing the smallest non-banned integers first, while also ensuring the total sum does not go over `maxSum`.

Solution Approach

The implementation of the solution in Python follows the steps outlined in the intuition.

Here's a step-by-step explanation of the code provided in the Reference Solution Approach, referring to the algorithm and data structures used:

1. Initialize two variables:
   - `ans` to store the running count of chosen numbers, initially set to 0.
   - `s` to store the current sum of chosen numbers, also initially set to 0.
2. Convert the `banned` list of integers into a set called `ban`. A set is used because it provides O(1) time complexity for checking if an item exists in it, which is far more efficient than using a list, especially when the `banned` list is large.
3. Iterate over the integers from 1 to `n` inclusive using a for loop. This is the range from which you can choose the integers.
4. In each iteration, perform two checks before choosing an integer:
   - If the current sum `s` plus the current integer `i` is greater than `maxSum`, terminate the loop. This check ensures that the solution doesn't exceed the allowed sum constraint.
   - If the current integer `i` is not in the `ban` set, increment `ans` by 1 and add `i` to the sum `s`. This action effectively chooses the integer and updates the count and sum accordingly.
5. Once the loop ends (either by exceeding `maxSum` or after checking all possible integers), return `ans`. This final value represents the maximum number of integers you can choose that satisfy the given constraints.

This algorithm is an efficient way to solve the problem because it works incrementally to build the solution, ensuring at each step that the conditions are met. By starting from the smallest possible number and moving up, it adheres to the greedy approach strategy, which in this case is optimal.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach. Assume we have `n = 10`, `banned = [3, 5, 7]`, and `maxSum = 15`.

1. Initialize `ans = 0` and sum `s = 0`.
2. Convert the banned list into a set: `ban = {3, 5, 7}`.
3. Start iterating from `i = 1` to `n = 10`.
   - For `i = 1`: `s + i = 1` <= `maxSum = 15`, and `1` is not in `ban`. So, `ans = 1` and `s = 1`.
   - For `i = 2`: `s + i = 3` <= `maxSum`, and `2` is not in `ban`. Now `ans = 2` and `s = 3`.
   - For `i = 3`: Skip since `3` is in `ban`.
   - For `i = 4`: `s + i = 7` <= `maxSum`, and `4` is not in `ban`. Update `ans = 3` and `s = 7`.
   - For `i = 5`: Skip since `5` is in `ban`.
   - For `i = 6`: `s + i = 13` <= `maxSum`, and `6` is not in `ban`. Update `ans = 4` and `s = 13`.
   - For `i = 7`: Skip since `7` is in `ban`.
   - For `i = 8`: `s + i = 21` > `maxSum`, so we break the loop as adding `8` would exceed `maxSum`.
4. Having completed the iteration or stopped when we could no longer add integers without exceeding `maxSum`, we have a final `ans = 4`.

This walk-through illustrates how the algorithm makes choices and updates both the number of integers chosen and the current sum while respecting the constraints. The value of `ans` after the loop ends (in this case, `4`) is the number we would return as the maximum count of integers we can choose within the rules.

Python Solution

```python
class Solution:
    def maxCount(self, banned_numbers: List[int], n: int, max_sum: int) -> int:
        count = 0  # Initialize the count of numbers that aren't banned and whose sum is within max_sum
        current_sum = 0  # Initialize the current sum of numbers
        banned_set = set(banned_numbers)  # Convert the banned list to a set for faster lookups

        # Iterate through the numbers from 1 to n
        for i in range(1, n + 1):
            # Check if adding the current number would exceed the maximum allowed sum
            if current_sum + i > max_sum:
                break  # If it would, break out of the loop early

            # Check if the current number is not banned
            if i not in banned_set:
                count += 1  # Increment the count of valid numbers
                current_sum += i  # Add the current number to the sum

        # Return the final count of valid numbers
        return count

```

Java Solution

```java
class Solution {
    // Method that returns the maximum count of distinct integers that can be chosen
    // from 1 to n such that their sum does not exceed maxSum and they are not in the banned list
    public int maxCount(int[] bannedNumbers, int n, int maxAllowedSum) {
        // Create a hash set to store the banned numbers for easy access
        Set<Integer> bannedSet = new HashSet<>(bannedNumbers.length);

        // Populate the bannedSet with numbers from the bannedNumbers array
        for (int number : bannedNumbers) {
            bannedSet.add(number);
        }

        // Initialize answer to store the count of possible numbers and sum to track current sum
        int count = 0;
        int currentSum = 0;

        // Iterate through integers from 1 to n
        for (int i = 1; i <= n && currentSum + i <= maxAllowedSum; ++i) {
            // If the current number is not in the banned set
            if (!bannedSet.contains(i)) {
                // Increment the answer count and add the number to the current sum
                ++count;
                currentSum += i;
            }
        }

        // After the loop, return the total count of selectable numbers that meet the criteria
        return count;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <unordered_set>
using namespace std;

class Solution {
public:
    // Function to calculate the maximum number of unique integers we can sum up
    // to maxSum without including any integers from the banned list.
    // Parameters:
    // - banned - a list of integers to be excluded
    // - n - the range of positive integers to consider (1 to n)
    // - maxSum - the maximum allowable sum
    // Returns the maximum count of unique integers within the limit of maxSum, excluding banned ones.
    int maxCount(vector<int>& banned, int n, int maxSum) {
        // Convert the vector of banned numbers into an unordered_set for efficient lookup
        unordered_set<int> bannedNumbers(banned.begin(), banned.end());

        // Initialize a counter for the answer and a variable to keep the current sum
        int answer = 0;
        int currentSum = 0;

        // Iterate through the numbers from 1 to n
        for (int i = 1; i <= n && currentSum + i <= maxSum; ++i) {
            // Check if the current number is not in the banned set
            if (!bannedNumbers.count(i)) {
                // Increment the count of usable numbers
                ++answer;
                // Add the current number to the running sum
                currentSum += i;
            }
        }
        // Return the maximum count of unique integers found that we can sum up to maxSum
        return answer;
    }
};

```

Typescript Solution

```typescript
function maxCount(banned: number[], n: number, maxSum: number): number {
    // Create a Set from the banned array to facilitate O(1) look-up times.
    const bannedSet = new Set(banned);

    // Initialize sum to keep track of the total sum of non-banned numbers.
    let sum = 0;

    // Initialize ans to keep track of the count of numbers that can be added without exceeding maxSum.
    let ans = 0;

    // Iterate through numbers from 1 to n.
    for (let i = 1; i <= n; i++) {
        // If adding the current number exceeds maxSum, exit the loop.
        if (i + sum > maxSum) {
            break;
        }

        // If the current number is in the banned set, skip to the next iteration.
        if (bannedSet.has(i)) {
            continue;
        }

        // Add the current non-banned number to the sum.
        sum += i;

        // Increment the count since we have successfully added a non-banned number without exceeding maxSum.
        ans++;
    }

    // Return the count of non-banned numbers added without exceeding maxSum.
    return ans;
}

```

Time and Space Complexity

The time complexity of the provided code is `O(n)`, where `n` is the input parameter that defines the range of numbers we are considering (`1` to `n`). This is because the code iterates over each number from `1` to `n` at most once. During each iteration, the algorithm performs a constant-time operation: checking if the number is not in the banned set and if adding the current number would exceed `maxSum`. If neither condition is true, it performs two additional constant-time operations: incrementing `ans` and adding the number to the running total `s`.

The space complexity of the code is `O(b)`, where `b` is the length of the `banned` list. This accounts for the space needed to store the banned numbers in a set for quick lookup.