2829. Determine the Minimum Sum of a k-avoiding Array

Medium Greedy Math Leetcode Link

Problem Description

In this problem, we are working with a very specific type of array called a "k-avoiding" array. An array is considered to be k-avoiding if there are no two distinct elements within it that add up to the number `k`. Our objective is to construct a k-avoiding array that has a length of `n`, where `n` is a given integer, and we want to make sure that the sum of the elements of this array is as small as possible. We're asked to find and return this minimum possible sum.

Intuition

The key to solving this problem is to avoid any pairs of numbers that sum up to `k`. A straightforward approach is to start from the smallest positive integer and keep adding distinct numbers to the array, making sure that none of these integer pairs sum to `k`.

We can do this by maintaining a set of integers that should not be included in the array (to avoid summing to `k`). As we iterate and add a new element `i` to the array, we also add `k - i` to the set of integers to avoid because if `k - i` was ever included later, it would sum with `i` to make `k`, violating the k-avoiding property.

If we encounter a number that is already in the set of numbers to avoid, we simply increment `i` to find the next suitable candidate.

This approach will ensure that we are always adding the smallest possible integers to our array, hence achieving the minimum possible sum for a k-avoiding array of length `n`.

Solution Approach

The solution involves a greedy approach and a set to keep track of visited numbers and their complementary numbers that sum to `k`. Here is a step-by-step breakdown of the algorithm using the provided solution code.

1. Initialize a sum `s` to `0`, which will hold the sum of the k-avoiding array values, and an index `i` to `1`, which is the smallest positive integer we can add to our array.

2. Initialize an empty set `vis` to keep track of numbers that are already either used in the array or that should be avoided because their corresponding pairs would sum up to `k`.

3. Iterate `n` times, since we need to add `n` distinct integers to our k-avoiding array. For each iteration:

    - Increment `i` until you find a value that is not present in the `vis` set. This is the next smallest number that can be added to the array without risking a sum of `k` with any other number already in the array.
    - Add `i` to the `vis` set because it is now part of our k-avoiding array.
    - Add `k - i` to the `vis` set to avoid any future numbers from creating a pair with `i` that sums to `k`.
    - Add `i` to our running total `s`, as `i` is now a confirmed element of our k-avoiding array.
4. After we have added `n` numbers to the array, we return `s`, which now holds the minimum possible sum of the k-avoiding array.

The two key data structures and patterns used in this solution are:

- **Greedy Algorithm**: By always choosing the next smallest integer not yet visited or restricted, we ensure that we are minimizing the sum at each step, leading to a global minimum sum.
- **Set**: By using a set for `vis`, we can quickly check if a number has already been used or is off-limits and thus efficiently manage our pool of potential array elements.

This algorithm is efficient because it operates on a simple principle of avoidance and uses sets for fast lookup, adding up to an overall time complexity of O(n) since we perform a constant amount of work for each of the n iterations to construct the k-avoiding array.

Example Walkthrough

Let's consider constructing a k-avoiding array with `k = 7` and length `n = 5`. We want to end up with an array which has no two distinct elements that add up to 7, and with the smallest possible sum.

1. We initialize the sum `s = 0` and index `i = 1` which is our starting point for choosing array elements. Our set `vis` is also initialized to an empty set.

2. Now, we will add numbers to our k-avoiding array, while also updating the set `vis` with each new number and the number that when combined with it adds up to our target k (7). Our loop will run `n` times.

3. On the first iteration, `i = 1` is not in `vis`, so we can add it to our array. We then update `vis` to include `1` and `7 - 1 = 6` (to avoid later). The sum `s` now becomes `s = 1`.

4. On the second iteration, we try `i = 2`. Since `2` is not in `vis`, we can use it. We update `vis` by adding `2` and `7 - 2 = 5`. The sum `s` becomes `s = 1 + 2 = 3`.

5. On the third iteration, `i = 3` is not in `vis`, and its pair that would sum to `7` is `4` which is not in `vis` yet either, so we can add `3` safely. We update `vis` with `3` and `4`. The sum `s` is now `s = 3 + 3 = 6`.

6. The fourth iteration, however, is interesting—`i` is incremented to `4`, but `4` is in `vis`, indicating a sum with `3` would equal `k = 7`, so we cannot use it. We increment `i` to `5`, which is also in `vis`. We keep incrementing until we reach `i = 8`, which is safe to use. We add `8` and its pair `7 - 8 = -1` (although negative numbers are not relevant for our case, we're working with positive integers) to `vis`, and update our sum to `s = 6 + 8 = 14`.

7. On our final iteration, `i = 9` is not in `vis`, so we can use it safely in our k-avoiding array. We add `9` and its pair `7 - 9 = -2` to `vis`, even though those negative numbers won't impact our positive integer set. The final sum `s` is `14 + 9 = 23`.

Our final k-avoiding array might look like `[1, 2, 3, 8, 9]` which has no elements that add up to `7`, and the sum of its elements is minimum possible, which is `23`.

The steps effectively demonstrate a greedy approach, continuously choosing the next smallest integer not in the avoidance set, and efficiently constructing a k-avoiding array with the smallest possible sum.

Python Solution

```python
class Solution:
    def minimumSum(self, num_elements: int, target: int) -> int:
        # Initialize the sum and the starting integer
        current_sum, current_int = 0, 1

        # This set will keep track of the visited or used integers
        visited = set()

        # Iterate for the number of elements required
        for _ in range(num_elements):
            # Avoid using integers already in the visited set
            while current_int in visited:
                current_int += 1

            # Add the current integer to the visited set
            visited.add(current_int)

            # Add the complement of current_int with respect to the target
            visited.add(target - current_int)

            # Add the current integer to the sum
            current_sum += current_int

        # Return the calculated sum
        return current_sum

# Example usage:
# solution = Solution()
# print(solution.minimumSum(4, 10)) # Example call to the method

```

Java Solution

```java
class Solution {
    public int minimumSum(int numElements, int cancellationFactor) {
        int sum = 0; // Variable to keep track of the sum of the chosen elements
        int smallestEligible = 1; // Variable to store the smallest eligible number that can be used
        boolean[] visited = new boolean[numElements * numElements + cancellationFactor + 1]; // Array to mark visited numbers

        // Repeat until all 'numElements' elements have been added
        while (numElements-- > 0) {
            // Loop to find the next unvisited number starting from smallestEligible
            while (visited[smallestEligible]) {
                ++smallestEligible;
            }

            // Mark this number as visited
            visited[smallestEligible] = true;

            // If the number is large enough, mark its complementary number (relative to 'cancellationFactor') as visited
            if (cancellationFactor >= smallestEligible) {
                visited[cancellationFactor - smallestEligible] = true;
            }

            // Add the number to the sum
            sum += smallestEligible;
        }

        // Return the computed sum
        return sum;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <cstring>

class Solution {
public:
    int minimumSum(int numElements, int delta) {
        int sum = 0; // Used to store the sum of chosen elements
        int currentElement = 1; // Starting element for selection
        // Vector to keep track of visited elements, initialized to false
        std::vector<bool> visited(numElements * numElements + delta + 1, false);

        // Process each element, decrementing numElements as we go
        while (numElements--) {
            // Find the first unvisited element starting from currentElement
            while (visited[currentElement]) {
                ++currentElement;
            }
            // Mark this element as visited
            visited[currentElement] = true;
            // If delta is sufficient, mark the corresponding element as visited
            if (delta >= currentElement) {
                visited[delta - currentElement] = true;
            }
            // Add the current element to the sum
            sum += currentElement;
        }
        return sum; // Return the computed sum
    }
};

```

Typescript Solution

```typescript
function minimumSum(numElements: number, offset: number): number {
    let sum = 0; // Will hold the sum of the minimum elements
    let currentElement = 1; // Start with the first natural number
    // An array to keep track of visited numbers; initialized with 'false'
    const visited = Array<boolean>(numElements * numElements + offset + 1).fill(false);

    // Iterate until we have selected 'numElements' elements
    while (numElements--) {
        // Find the next current element that has not been visited
        while (visited[currentElement]) {
            ++currentElement;
        }

        // Mark the current element as visited
        visited[currentElement] = true;

        // If the offset minus the current element is non-negative, mark it as visited
        if (offset >= currentElement) {
            visited[offset - currentElement] = true;
        }

        // Accumulate the sum of selected elements
        sum += currentElement;
    }

    // Return the sum of the minimum elements
    return sum;
}

```

Time and Space Complexity

The given Python code aims to find the minimum sum of a sequence of `n` unique numbers, whereby for each selected number `i`, the number `k - i` is not selected. It initializes a sum `s` to 0, an index `i` to 1, and a set `vis` to track visited numbers.

Time Complexity

The time complexity of the code depends on the loop that runs `n` times – once for each number we need to find. Inside this loop, there is a `while i in vis` loop that keeps incrementing `i` until it finds a non-visited number. In the worst-case scenario, this inner while loop can run for each number from 1 to `n` if all are previously marked as visited. However, in practice, the loop will run considerably fewer times than `n` for each outer loop iteration, because it only needs to find the next unvisited number.

Assuming `U(n)` to be the average number of iterations for finding the next unvisited number over all `n` iterations, the time complexity can be approximated as `O(n * U(n))`. Since it's difficult to precisely define `U(n)`, we can consider it as a factor that is less than `n`. However, for the sake of upper-bound estimation, let's consider `U(n)` in the worst case to be `n`, which would give us a time complexity of `O(n^2)` in the worst case.

Space Complexity

The space complexity is easier to analyze. The code only uses a set to store the visited numbers. At most, this set will contain `2n` elements (each element and its complement with respect to `k`), so the space complexity is `O(n)`.