2195. Append K Integers With Minimal Sum

Medium Greedy Array Math Sorting

Leetcode Link

Problem Description

In this problem, you are given an array of integers named `nums` and an integer `k`. The goal is to append `k` unique positive integers to the `nums` array, which are not already present in `nums`, so that the total sum after appending these new integers is the smallest possible. The question requires you to find the sum of the `k` integers that you append to the array.

For example, if `nums = [1,4,25]` and `k = 2`, the two unique integers to append could be 2 and 3, as they are the smallest positive integers not present in `nums`. The sum we are looking for would be 2 + 3 = 5.

It's essential to note that the integers appended to `nums` must be positive and non-repeating.

Intuition

The key intuition behind the solution is to find the smallest gaps in the sorted `nums` array where the missing positive integers can be placed. To do this efficiently, we first add two boundaries to the `nums` array: 0 at the beginning and 2*10^9 at the end (which is sufficiently large to ensure that we can find `k` unique positive integers).

The array is then sorted to facilitate the process of finding missing numbers between the contiguous elements of `nums`.

After sorting, we look into the gaps between consecutive elements. Specifically, we check the difference between each pair of neighbouring elements to calculate how many unique positive integers could fit into each gap. We prioritize smaller numbers and only insert as many integers as we need (up to `k`).

This process is repeated iteratively, reducing `k` by the number of integers inserted each time until `k` becomes 0, which means no more numbers need to be appended. The sum of the numbers added within each gap is calculated using the arithmetic progression formula: `sum = n/2 * (first_term + last_term)`, where `n` is the number of terms to add, `first_term` is the start of the sequence to append, and `last_term` is the end of the sequence to append.

The loop breaks immediately once we have appended `k` integers, and we return the cumulative sum of all the appended integers.

Solution Approach

The implementation of the solution follows these steps:

1. **Append Boundaries to `nums`:** We start by appending the integer 0 at the beginning and a very large integer (here, `2 * 10^9`) at the end of the `nums` array. These boundaries help us handle edge cases where the smallest or largest possible integers must be appended.
2. **Sort the `nums` Array:** We sort the `nums` array to make it easier to go through it sequentially and find gaps between the existing numbers where new integers can be added.
3. **Iterate Over Pairs of Elements:** Utilizing the `pairwise` function (new in Python 3.10), we iterate over the `nums` array in pairs to check for the difference (gap) between each pair of neighbouring elements.
4. **Calculate the Number to Append:** For each pair `(a, b)`, we calculate the number of unique positive integers that can be appended in the gap `(b - a - 1)`. We limit this to the minimum of the gap size and `k` as we are only interested in appending up to `k` numbers.
5. **Sum Calculation:** Using the formula for the sum of an arithmetic sequence, we calculate the sum of the consecutive integers from `(a + 1)` to `(a + n)`, where `n` is the number of integers to include in the particular gap. The sum is calculated by the formula `(a + 1 + a + n) * n // 2`, taking advantage of the fact that the sum of a sequence can be found by multiplying the average of the first and last term by the number of terms.
6. **Update `k` and the Total Sum:** We subtract `n` from `k`, reducing the number of integers left to append, and add the sum of the appended integers to `ans`, which holds the cumulative sum of integers appended so far.
7. **Break Condition:** As soon as `k` reaches 0, we break out of the loop since we have appended enough integers to meet the requirement.

The solution approach effectively uses sorting to organize the data and then a sweep algorithm with an arithmetic sequence sum calculation to quickly derive the sum without an explicit enumeration of each number that needs to be appended.

```python
class Solution:
    def minimalKSum(self, nums: List[int], k: int) -> int:
        nums.append(0)
        nums.append(2 * 10**9)
        nums.sort()
        ans = 0
        for a, b in pairwise(nums):
            n = min(k, b - a - 1)
            if n <= 0:
                continue
            k -= n
            ans += (a + 1 + a + n) * n // 2
            if k == 0:
                break
        return ans
```

The beauty of this solution lies in its efficiency, as it makes a single pass over the sorted array and calculates the sums in a constant number of operations per gap, thus achieving a time complexity that is linearithmic (O(n*log(n))) due to sorting, where `n` is the number of elements in the initial `nums` array.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach. Suppose we have `nums = [3, 5, 7]` and `k = 3`. We want to append `k = 3` unique positive integers to `nums` such that the total sum is minimized.

1. **Append Boundaries to `nums`:** We first add the boundaries, so `nums` becomes `[0, 3, 5, 7, 2000000000]`.
2. **Sort the `nums` Array:** `nums` is already in sorted order after appending the boundaries.
3. **Iterate Over Pairs of Elements:** We now consider each pair of neighbouring elements and the gaps between them:
   - Between `0` and `3`, the gap is `2` (`[1, 2]`). We can append two numbers here.
   - Between `3` and `5`, no gap exists.
   - Between `5` and `7`, no gap exists.
   - Between `7` and `2000000000`, there's a large gap, but we will only fill as many numbers as we need to reach `k`.
4. **Calculate the Number to Append:**
   - For the pair `(0, 3)`, we calculate the gap is `2` and `k = 3`. We can use this entire gap, so we append `1` and `2`.
5. **Sum Calculation:**
   - The sum for the gap between `(0, 3)` will be `(1 + 2)`, which equals `3`.
6. **Update `k` and the Total Sum:**
   - After filling the gap between `(0, 3)` with `1` and `2`, `k` reduces from `3` to `1` (`k = k - 2`), and our running total (the answer `ans`) becomes `3`.
7. **Continue Iterating Until k is 0:**
   - Since `k` is not yet `0`, we continue to the next gap. However, there are no gaps between `3` and `5`, nor between `5` and `7`.
   - Finally, we look at the gap between `7` and `2000000000`.
   - We have `1` left to append (`k = 1`).
   - The next number we can append is `8` (the first number after `7`).
8. **Sum Calculation:**
   - We append `8`, and our sum (`ans`) becomes `3 + 8 = 11`.
9. **Update `k` and the Total Sum:**
   - `k` is now `0`, and we've achieved our goal of appending `3` unique positive integers that were not already in `nums`.

Therefore, the sum of the appended numbers is `11`. This corresponds to appending the numbers `1`, `2`, and `8` to the array.

This walkthrough demonstrates how the algorithm efficiently calculates the sum of the minimal `k` unique positive integers that can be appended to `nums` to achieve the smallest possible total sum.

Python Solution

```python
from typing import List

class Solution:
    def minimalKSum(self, nums: List[int], k: int) -> int:
        # Append 0 and a large number at the end of nums which serves as the boundaries.
        nums.append(0)
        nums.append(2 * 10**9)
        # Sort the given list to find intervals between numbers easily.
        nums.sort()

        # Initialize the answer
        min_k_sum = 0

        # Use zip to create pairwise tuples to iterate through nums and find missing integers.
        for current, next_num in zip(nums, nums[1:]):
            # The number of missing integers between the current and next_num
            num_missing = min(k, next_num - current - 1)

            # If there are no missing integers or the count is exhausted, skip this pair.
            if num_missing <= 0:
                continue

            # Decrease k by the number of missing integers we add to the sum.
            k -= num_missing

            # Calculate the sum of the arithmetic series of missing integers
            # from current + 1 to current + num_missing and add it to min_k_sum.
            min_k_sum += (current + 1 + current + num_missing) * num_missing // 2

            # If k is depleted, break out of the loop since we've found enough integers.
            if k == 0:
                break

        # Return the total minimum sum of 'k' distinct positive integers.
        return min_k_sum
```

Java Solution

```java
import java.util.Arrays;

class Solution {

    // Function to find minimal sum of 'k' distinct integers
    // that are not present in the input array 'nums'.
    public long minimalKSum(int[] nums, int k) {
        // Create a new array with additional room for
        // the smallest and largest possible integer values
        int[] sortedNums = new int[nums.length + 2];
        sortedNums[sortedNums.length - 1] = (int) 2e9; // Set the last element to a very high number

        // Copy the original array into this new array, starting from index 1
        for (int i = 0; i < nums.length; ++i) {
            sortedNums[i + 1] = nums[i];
        }

        // Sort the new array to ensure proper ordering
        Arrays.sort(sortedNums);

        // Initialize the answer as a long due to potential for large summation values
        long sum = 0;

        // Iterate over sortedNums to find missing numbers
        for (int i = 1; i < sortedNums.length; ++i) {
            int current = sortedNums[i - 1];
            int next = sortedNums[i];

            // Calculate how many numbers can be taken between current and next
            // without duplicating any existing numbers in the array
            int count = Math.min(k, next - current - 1);

            // If no numbers can be taken here, continue to the next
            if (count <= 0) {
                continue;
            }

            // Decrease 'k' by the number of new numbers being added
            k -= count;

            // Calculate the sum of the consecutive numbers using the arithmetic sum formula:
            // sum = (first number + last number) * count / 2
            sum += (long) (current + 1 + current + count) * count / 2;

            // If k becomes 0, we've found enough numbers, break the loop
            if (k == 0) {
                break;
            }
        }
        // Return the minimal sum of the 'k' distinct missing integers
        return sum;
    }
}
```

C++ Solution

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

class Solution {
public:
    long long minimalKSum(vector<int>& nums, int k) {
        // Adding boundary values to handle edge cases automatically.
        nums.push_back(0);     // Lower bound value for the sequence.
        nums.push_back(2e9);   // Upper bound value to ensure we can find k unique numbers.

        // Sort the input vector to easily find missing integers.
        sort(nums.begin(), nums.end());

        // Initialize the sum.
        long long sum = 0;

        // Iterate over the sorted numbers to find the sum of the first k missing positive integers.
        for (int i = 1; i < nums.size(); ++i) {
            int current = nums[i - 1];
            int next = nums[i];

            // Calculate the count of missing numbers between current and next number.
            int missingCount = min(k, next - current - 1);

            // No missing numbers, move to the next pair.
            if (missingCount <= 0) continue;

            k -= missingCount; // Reduce k by the number of missing numbers found.

            // Calculate the sum of the missing numbers in the range and add it to the sum.
            sum += 1ll * (current + 1 + current + missingCount) * missingCount / 2;

            // If we have found k numbers, break the loop.
            if (k == 0) break;
        }
        return sum; // Return the calculated sum.
    }
};
```

Typescript Solution

```typescript
function minimalKSum(nums: number[], k: number): number {
    // Adding boundary values to handle edge cases automatically.
    nums.push(0);    // Lower bound value for the sequence.
    nums.push(2 * Math.pow(10, 9));   // Upper bound value to ensure we can find k unique numbers.

    // Sort the input array to easily find missing integers.
    nums.sort((a, b) => a - b);

    // Initialize the sum.
    let sum: number = 0;

    // Iterate over the sorted numbers to find the sum of the first k missing positive integers.
    for (let i = 1; i < nums.length; ++i) {
        const current: number = nums[i - 1];
        const next: number = nums[i];

        // Calculate the count of missing numbers between current and next number.
        const missingCount: number = Math.min(k, next - current - 1);

        // No missing numbers, move to the next pair.
        if (missingCount <= 0) continue;

        k -= missingCount; // Reduce k by the number of missing numbers found.

        // Calculate the sum of the missing numbers in the range and add it to the sum.
        sum += ((current + 1 + current + missingCount) * missingCount) / 2;

        // If we have found k numbers, break the loop.
        if (k === 0) break;
    }
    return sum; // Return the calculated sum.
}
```

Time and Space Complexity

Time Complexity

The time complexity of the `minimalKSum` method can be split into a few distinct parts:

1. `nums.append(0)` and `nums.append(2 * 10**9)` - These operations are constant-time, i.e., `O(1)`.
2. `nums.sort()` - Sorting the array `nums` takes `O(n log n)` time, where `n` is the length of the array.
3. The `for` loop and `pairwise(nums)` - Iterating over the array takes `O(n)` time. Inside the loop, operations are constant time, such as arithmetic operations and comparison. The `pairwise` function iterates over the list, generating tuples of adjacent elements. The complexity of generating each pair is `O(1)`, thus pairwise itself does not add to the asymptotic complexity of the loop.
4. Note that the inner operations that include arithmetic and the `if` condition are all `O(1)` operations.

Combining these, the time complexity is dominated by the sorting step, which results in an overall time complexity of `O(n log n)`.

Space Complexity

For space complexity:

1. An additional `0` and `2 * 10**9` are appended to the existing list `nums`, this is `O(1)` additional space.
2. `pairwise(nums)` function - This function generates an iterator, which does not create a full copy of `nums`, and thus takes `O(1)` space.
3. No additional data structures are used that depend on the size of the input.

Thus, the space complexity of the method is `O(1)` — constant space, aside from the input list itself.