2834. Find the Minimum Possible Sum of a Beautiful Array

Medium Greedy Math

Problem Description

being defined as beautiful:

The problem presents us with two positive integers, n and target. We are asked to find an array nums meeting specific criteria,

• The array nums should have a length equal to n.

- All elements in nums must be distinct positive integers.
- There must not be any two distinct indices i and j within the range [0, n 1] for which nums[i] + nums[j] equals target.
- The goal is to determine the minimum possible sum of a beautiful array, with the added detail that the result should be given modulo 10^9 + 7. This modulus operation ensures that the numbers stay within a reasonable range, given the constraints of large

number handling in most programming languages. Intuition

The solution relies on constructing the beautiful array iteratively while maintaining two key insights: 1. Since we want to minimize the sum of nums, we should start adding the smallest positive integers available, which are naturally consecutive

starting from 1. 2. To prevent any two numbers from being able to sum up to target, when we add a new number i to nums, we must make sure that target - i is

2. ans will hold the running sum of the nums array as we find the correct integers to include.

- not possible to be added in future steps. This is because if we have i in the array, and later add target i, we would violate the condition that no two numbers can sum to target.
- We use a set, vis, to keep track of numbers that cannot be a part of nums to satisfy the condition above. This is so we can quickly check if a number is disallowed before adding it to our array. Each iteration, we look for the smallest unvisited number starting from 1, add it to the sum, mark its complement with respect to target in vis, and move to the next smallest number.

the sum remains as small as possible. **Solution Approach**

This iterative process is repeated n times to fill the nums array while ensuring all conditions for a beautiful array are met and that

The implementation of the solution is straightforward and methodical. Let's break down the steps in the solution code: 1. Initialize an empty set vis which will store the integers that cannot be included in the nums array, to prevent summing up to the target.

3. We start iterating through numbers to include in the nums array starting with i = 1, which ensures we start with the smallest positive integer.

increment i to the next number and check again.

potentially storing up to n integers in the vis set.

Following the steps outlined in the solution approach:

-1 = 9}. 6. Iterate to the next number: i = 2.

-3 = 7}. 6. Iterate to the next number: i = 4.

2. Iterate through numbers starting with i = 1: Our goal is to iterate 5 times, as n = 5.

32-bit or 64-bit signed integer range commonly used in programming languages.

def minimum_possible_sum(self, n: int, target: int) -> int:

Initialize the answer (sum) to zero.

while current_int in visited:

visited.add(target - current_int)

Return the computed sum after 'n' additions.

// Return the final sum which is the minimum possible sum

// such that no pair of integers adds up to 'target'.

long long minimumPossibleSum(int n, int target) {

memset(visited, false, sizeof(visited));

// Add the current number to our sum.

def minimum_possible_sum(self, n: int, target: int) -> int:

Initialize the answer (sum) to zero.

while current_int in visited:

Add the current integer to the sum.

visited.add(target - current_int)

Move to the next integer.

current_int += 1

answer += current_int

current_int += 1

Initialize a set to keep track of visited numbers.

Initialize the current integer we are going to add to the sum.

Loop 'n' times to find 'n' unique numbers to add to the sum.

Find the next unvisited integer to add to the sum.

Mark the counterpart (target - current_int) as visited.

bool visited[n + target];

for (int i = 1; n > 0; ++i) {

while (visited[i]) {

long long sum = 0;

++i;

sum += i;

// Function to calculate the minimum possible sum of 'n' unique positive integers

// Create an array to keep track of the numbers that should not be used

// because they would add up to the target with a number already in use.

// Initialize the sum to 0, the sum will be accrued over the iteration.

// Iterating over the potential numbers starting from 1.

// Initialize the 'visited' array to false, indicating no numbers have been used yet.

// Skip the numbers that we can't use because they have a pair already in use.

// Check if the counterpart of the current number (target - i) can potentially

Move to the next integer.

answer += current_int

current_int += 1

Initialize a set to keep track of visited numbers.

Initialize the current integer we are going to add to the sum.

Loop 'n' times to find 'n' unique numbers to add to the sum.

Find the next unvisited integer to add to the sum.

Mark the counterpart (target - current_int) as visited.

number that could combine with our current i to sum to target.

6. **Iterate**: We increment i to consider the next integer in the following iteration.

Now we go into a loop that runs n times – once for each number that we need to add to our nums array:

- 4. Check vis for the next available number: Since no two numbers should add up to target, before considering the integer i to add to ans, we check if it's already in the set vis. If it is, it means its complementary number (that would sum to target) is already part of the nums array, so we
- 5. Update the sum and set: Once we find a number that is not in vis, it means we can safely add it to ans without "violating" the target sum condition. We add i to ans, then we add target - i to vis. Adding target - i ensures that in the subsequent iterations, we don't pick a
- Finally, we return the total sum ans modulo 10^9 + 7. This modulus ensures that our final answer fits within the limits for integer values as prescribed by the problem and is a common practice in competitive programming to prevent integer overflow.
- constant time complexity 0(1) checks and insertions, providing us with an efficient way to track and prevent selecting numbers that could pair up to form target. The overall time complexity of the solution is O(n) since we iterate over n elements, and the space complexity is also O(n) due to

In this implementation, we use a greedy algorithm starting with the smallest possible integer and moving up. The set vis ensures

Let's use a small example to illustrate the solution approach. Suppose we have the following input: • target = 10

First Iteration (i = 1): 4. Check vis: 1 is not in vis, so we can consider it. 5. Update ans and vis: ans = 0 + 1 = 1, vis = {10

Second Iteration (i = 2): 4. Check vis: 2 is not in vis, so it's safe to add. 5. Update ans and vis: ans = 1 + 2 = 3, vis = $\{9, 6\}$

Third Iteration (i = 3): 4. Check vis: 3 is not in vis, so we take it. 5. Update ans and vis: ans = 3 + 3 = 6, vis = $\{9, 8, 10\}$

(since 10 - 5 = 5), and we're trying to add 5 now, so it's okay to choose it as 5 will not be added again. 5. Update ans and

vis: Since 5 can be added to ans, it's updated to ans = 10 + 5 = 15, and vis would include its target complement, but since

Now let's walk through each iteration:

Example Walkthrough

10 - 2 = 8}. 6. Iterate to the next number: i = 3.

1. Initialize a set and sum variable: We start with an empty set $vis = \{\}$ and an integer for our running sum ans = 0.

8, 7, 10 - 4 = 6}. 6. Iterate to the next number: i = 5. Fifth Iteration (i = 5): 4. Check vis: 5 is not in vis, but adding it we need to consider that its target complement would be 5

it's the same, no new entry is added to vis. 6. There are no more iterations, as we've reached n additions.

Finally, we return ans $% (10^9 + 7)$, which in this case is $15 % (10^9 + 7) = 15$, since 15 is already less than $10^9 + 7$. The final beautiful array that satisfies all conditions could be [1, 2, 3, 4, 5] with the minimum possible sum being 15. The set

Remember, in scenarios where n and target are much larger, the modulo operation would assure that the output fits within the

vis helped us to avoid including the numbers which would sum up to the target with any other number already in the array.

Solution Implementation **Python**

current_int += 1 # Add the current integer to the sum.

return answer

return sum;

#include <cstring>

class Solution {

Java

C++

public:

visited = set()

current int = 1

for _ in range(n):

answer = 0

class Solution:

```
class Solution {
   public long minimumPossibleSum(int n, int target) {
       // Create an array to keep track of visited numbers
       boolean[] visited = new boolean[n + target];
       // Initialize the answer (sum) to 0
        long sum = 0;
       // Loop over the numbers starting from 1 up to n
        for (int i = 1; n > 0; --n, ++i) {
           // If the current number has been visited, skip to the next one
            while (visited[i]) {
                ++i;
            // Add the smallest unvisited number to the sum
            sum += i;
           // If the target is greater than or equal to the current number, mark the corresponding number as visited
            if (target >= i && (target - i) < visited.length) {</pre>
                visited[target - i] = true;
```

class Solution:

visited = set()

current_int = 1

for _ in range(n):

answer = 0

```
// be used in the future, and mark it as visited to avoid using it.
            if (target >= i) {
                visited[target - i] = true;
            // Decrease the count of remaining numbers to add to our sum.
            --n;
        return sum; // Return the final sum.
};
TypeScript
/**
* Calculates the minimum possible sum of 'n' distinct integers where
 * each integer 'i' in the array does not equal 'target - i'.
 * @param {number} n - The count of distinct integers to sum
* @param {number} target - The target value that must not be met by the expression 'i' + 'array[i]'
 * @returns {number} - The minimum possible sum of 'n' distinct integers.
function minimumPossibleSum(n: number, target: number): number {
   // Initialize a boolean array with false values to track visited numbers
    const visited: boolean[] = Array(n + target).fill(false);
    let sum = 0; // Initialize the sum of integers.
    // Iterate over the range of possible values until 'n' distinct integers are found
    for (let i = 1; n > 0; ++i, --n) {
       // Skip over the numbers that have already been visited
       while (visited[i]) {
            ++i;
       // Add the current integer to the sum
       sum += i;
       // Mark the corresponding pair value as visited if it falls within the array bounds
       if (target >= i && (target - i) < visited.length) {</pre>
            visited[target - i] = true;
   // Return the minimum sum of 'n' distinct integers
    return sum;
```

```
# Return the computed sum after 'n' additions.
       return answer
Time and Space Complexity
 The provided Python code snippet finds the minimum possible sum of a sequence of n integers such that each value and its
 complement with respect to target are unique in the sequence.
```

• A set named vis is used to keep track of visited numbers, which would at most store n elements because for every element we add, we loop n

There is a for loop that goes n times, which is O(n). • Inside the loop, there is a while loop that continues until i is not in vis. Since the while loop increments i each time a collision with vis is

Time Complexity

Here is the breakdown:

Here is the breakdown:

detected and the number of possible collisions is limited by n, the amortized time complexity due to the while loop is 0(1). • Inserting and checking the presence of an item in a set in Python is 0(1) on average, as set is implemented as a hash table.

The time complexity of the code is O(n).

- Therefore, the time complexity for the complete for loop is essentially O(n).
- **Space Complexity**
- The space complexity of the code is O(n).
- times. • Other than vis, only a few variables are used (ans, i) with constant space requirement.

Thus, the set vis dictates the space complexity, which is O(n).