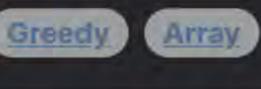
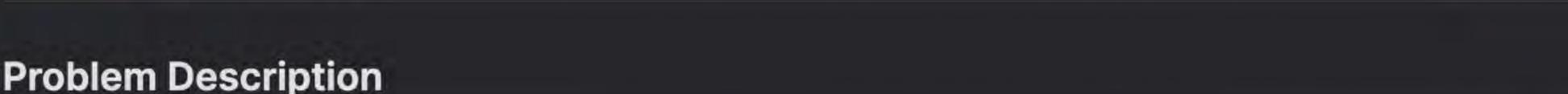
1798. Maximum Number of Consecutive Values You Can Make









Leetcode Link

The problem presents us with an array coins, each element representing the value of a coin that we have in our possession. The goal is to determine the maximum number of consecutive integer values that we can create by combining these coins, starting from and including the value 0. It is also mentioned that it's possible to have multiple coins of the same value.

The task can be visualized as a game where we are trying to create a continuous sequence of values starting from zero by selecting coins from the array. The challenge is to find out how long this sequence can be before we encounter a gap that cannot be filled with the available coins.

The solution to the problem lies in sorting the array and then iteratively adding the value of coins to a running sum, maintaining the

Intuition

maximum consecutive sequence that can be produced. We start with an answer ans initialized to 1, which represents the smallest sum we aim to create. Intuitively, if we include value 0

The sorted array allows us to approach the problem in an incremental manner. We iterate through each coin, and for each coin, we check if its value is greater than the current sum ans.

which doesn't require any coins, we can surely create at least 1 as the next consecutive integer value using our coins.

 If the coin's value is less than or equal to ans, we can add this coin to the previous sum to extend our consecutive sequence up to the new sum.

- If the coin's value is more than ans, this indicates that there's a gap we cannot fill using our coins as all smaller coins have already been processed. So, we break the loop.
- After processing each coin or upon encountering a gap, the current value of ans represents the maximum number of consecutive integer values attainable.

The reason this approach works is that once coins are sorted, combining them from the smallest to the largest ensures that we are filling in the smallest possible gaps first, thus extending the consecutive sequence without missing any possible value.

Solution Approach

First, we sort the array. This is essential for the greedy algorithm to work because we want to consider coins in ascending order to

build up the consecutive sequence without any gaps.

accumulator and the tracker for the consecutive numbers that can be made with the coins seen so far.

The implementation follows a greedy approach using simple control structures and a sorting algorithm.

1 for v in sorted(coins): Once the coins are sorted, we use a for-loop to iterate through each coin. The variable ans is initialized to 1, which acts as both the

```
1 ans = 1
```

we could already create.

We iterate through each coin's value in the sorted list. For each value v, we check if v is larger than ans. If it is not, we add the value

of v to ans. This addition is the act of creating a new consecutive number by adding the value of the coin to the sum of values that

```
The above conditional is used to check for the presence of a gap. If the value of the current coin is greater than the current ans
value, there is a break in our consecutive numbers, and we cannot extend our sequence further with the current coin. In this case,
```

made consecutively with what we have). The break statement ends the loop when a gap is found. If all coins are processed without encountering a gap, the loop concludes naturally. Finally, we return ans, which now indicates the maximum sum we could reach consecutively with the given coins.

we break out of the loop and return the maximum consecutive number that we can make until now (since further numbers cannot be

This algorithm is efficient because the costly operation is the initial sorting which typically has a time complexity of O(n log n), where

n is the number of coins. The subsequent iteration is an O(n) operation, making the total complexity O(n log n). The space complexity

is O(1) since we are not using any additional data structures proportional to the input size.

1 return ans

Let's apply the solution approach to a small set of coins to see how it works. Suppose we have the following coins: coins = [1,2,3,4]. Our task is to determine the maximum number of consecutive integer values we can create starting from 0. Let's walk through the steps specified in the solution approach:

2. We initialize ans to 1. This represents the smallest sum we aim to create, understanding that a sum of 0 can always be created

without using any coins.

to and including 2.

sums up to and including 7.

sums up to and including 11.

to 11 with the given coins.

Python Solution

class Solution:

13

14

17

10

12

13

14

15

16

19

20

21

22

24

25

Example Walkthrough

3. We iterate through each coin in the array. The sorted array is coins = [1,2,3,4]. ∘ For the first coin (v = 1), since v <= ans (1 <= 1), we add v to ans. Now, ans = ans + v = 2. We can now create all sums up

• The next coin is v = 2. Since v <= ans (2 <= 2), we can add v to ans, making ans = 4. Now we can create all sums up to and including 4.

1. First, we sort the coins, but since they are already in ascending order (1, 2, 3, 4), we don't need to do anything.

- We move to the next coin (v = 3). Again, v <= ans (3 <= 4), so we add v to ans, which becomes ans = 7. We can create all
 - Since we did not encounter any coin v such that v > ans throughout the entire iteration, we do not hit the break statement in our loop.

4. After processing all the coins, the current value of ans is 11, which means we can create every consecutive integer value from 0

In conclusion, with the given array of coins = [1,2,3,4], we can create all consecutive integer values from 0 up to 11. The greedy

algorithm efficiently allows us to determine this by adding coins in ascending order and checking for any possible gaps.

• Lastly, we have the coin v = 4. It is also less than or equal to ans (4 <= 7), so we add v to ans to get ans = 11. We can create

- 5. We return ans which is 11 in this case, indicating the maximum number we can reach consecutively.
 - # amount that can be obtained with the current set of coins. $max_consecutive = 1$ # Loop through the coins sorted in ascending order

The variable `max_consecutive` is used to track the highest consecutive

If the current coin value is greater than the highest consecutive

amount that can be formed, we cannot create the next consecutive

Otherwise, we add the value of the current coin to the highest

// Sort the coins array to consider coins in increasing order

// we cannot extend the consecutive sequence any further

// Return the maximum consecutive integer that cannot be formed

// Initialize the answer to 1, since we start checking from the first positive integer

// and adding the current coin allows us to extend this sequence further

// If the current coin's value is greater than the current maximum consecutive integer,

// Otherwise, increase the maximum consecutive integer by the value of the current coin

// This is because we can create all values from 1 to current maxConsecutive with the coins seen so far

consecutive amount to increase the range of consecutive amounts

def getMaximumConsecutive(self, coins: List[int]) -> int:

for coin in sorted(coins):

break

Arrays.sort(coins);

int maxConsecutive = 1;

for (int coin : coins) {

break;

// Iterate through the sorted coins

if (coin > maxConsecutive) {

maxConsecutive += coin;

number, so we break the loop.

if coin > max_consecutive:

that can be formed.

max_consecutive += coin

```
19
20
           # After processing all the coins we can, return the highest consecutive
21
           # amount that we were able to reach.
22
           return max_consecutive
23
Java Solution
   import java.util.Arrays; // Import Arrays utility for sorting
   class Solution {
       // Method to find the maximum consecutive integer that cannot be created using a given set of coins
       public int getMaximumConsecutive(int[] coins)
```

26 return maxConsecutive; 28 } 29

```
C++ Solution
1 #include <vector> // Include necessary header for vector
   #include <algorithm> // Include necessary header for sort function
   class Solution {
   public:
       // Function to find the maximum consecutive value that cannot be obtained with a given set of coins
       int getMaximumConsecutive(std::vector<int>& coins) {
           // Sort the coins in non-decreasing order
           std::sort(coins.begin(), coins.end());
           // Initialize the answer to 1 (the smallest positive integer)
           int maxConsecutive = 1;
12
13
           // Iterate through the sorted vector of coins
14
15
           for (int coin : coins) {
               // If the current coin value is greater than the current possible consecutive value, break the loop
16
               if (coin > maxConsecutive) break;
18
               // Otherwise, add the value of the coin to the maxConsecutive to extend the range of possible consecutive values
19
20
               maxConsecutive += coin;
21
           // Return the first maximum consecutive value that cannot be obtained
24
           return maxConsecutive;
25
```

```
// Import necessary functionalities
```

26 };

27

```
import { sort } from 'algorithm'; // TypeScript does not have these, assuming they exist.
   // Declare the function to find the maximum consecutive value that cannot be obtained with a given set of coins
   function getMaximumConsecutive(coins: number[]): number {
       // Sort the coins in non-decreasing order
       coins.sort((a, b) => a - b);
 8
9
       // Initialize the answer to 1(the smallest positive integer)
       let maxConsecutive = 1;
10
       // Iterate through the sorted array of coins
       for (let coin of coins) {
           // If the current coin value is greater than the current possible consecutive value, break the loop
           if (coin > maxConsecutive) break;
15
16
           // Otherwise, add the value of the coin to maxConsecutive to extend the range of possible consecutive values
17
           maxConsecutive += coin;
19
20
21
       // Return the first maximum consecutive value that cannot be obtained
22
       return maxConsecutive;
23 }
24
Time and Space Complexity
```

11 12 13 14

Typescript Solution

Time Complexity The provided code snippet has a time complexity of O(n log n) due to the sorting operation, where n is the number of elements in the coins list. The sorted function in Python uses the Timsort algorithm, which has this time complexity on average and in the worst case. Following the sorting, the code iterates through the sorted coins once, which has a time complexity of O(n). However, since

sorting is the dominant operation, the overall time complexity of the code remains $O(n \log n)$. Space Complexity

The space complexity of the code is O(n). This is because the sorted function returns a new list containing the sorted elements, therefore it requires additional space proportional to the size of the input list. The other variables used in the function (ans and v) use

constant space and do not depend on the size of the input, so they do not affect the overall space complexity.