1296. Divide Array in Sets of K Consecutive Numbers Sorting Medium **Greedy** Array Hash Table

Problem Description

Given an integer array nums and a positive integer k, the task is to determine whether the array can be divided into several sets of k consecutive numbers. To divide the array into sets of k consecutive numbers means to group the elements of the array such that each group contains k numbers and each number in a group follows the one before it with a difference of 1. The function should return true if such division is possible, otherwise, it should return false.

Intuition

times a number appears in nums without sorting the entire array.

To solve this problem, the intuition is to use a greedy approach. The steps followed are:

Start from the smallest number in the array and try to build a consecutive sequence of length k. If the sequence cannot be formed due to a missing number, return false.

Count the occurrences of each number in the array using a hash map or counter. This step helps to quickly find how many

- If a consecutive sequence of length k starting from a number is successfully formed, reduce the count of the numbers used in the sequence. If the count drops to zero, remove the number from the counter to avoid unnecessary checks in further
- iterations. Repeat steps 2 and 3 until all numbers are used to form valid sequences or until you find a sequence that cannot be

Using this approach, the solution checks in a sorted order if there are enough consecutive numbers following the current number

to form a group of k elements. If at any point there aren't enough consecutive numbers to form a group of k, the function returns

- false. On the other hand, if all numbers can be grouped successfully, the function returns true.
- **Solution Approach** The provided solution implements a greedy algorithm to solve the problem by using the following steps: Counting Elements: The solution uses Python's Counter from the collections module to count the frequency of each integer

in the input nums. This Counter acts like a hash map, and it stores each unique number as a key and its frequency as the

if cnt[v]:

if cnt[x] == 0:

cnt[x] -= 1

Example Walkthrough

return False

for x in range(v, v + k):

be formed, and the function returns False.

Let's illustrate the solution approach with an example:

Counter({3: 2, 4: 2, 1: 1, 2: 1, 5: 1, 6: 1})

from collections import Counter

nums = [1, 2, 3, 3, 4, 4, 5, 6]

cnt = Counter(nums)

3, 4, 5, 6].

corresponding value.

cnt = Counter(nums)

completed.

Sorting and Iterating: After counting, the code sorts the unique numbers and iterates over them. The sorting ensures that we

check for consecutive sequences starting with the smallest number.

- for v in sorted(nums): Forming Consecutive Groups: Inside the loop, we check if the current number's count is non-zero, indicating that it hasn't been used up in forming a previous group. If the count is non-zero, the nested loop tries to form a group starting from this
- number v up to v + k.
 - **Updating the Counter:** When a number x is found, its count is decremented since it's being used to form the current sequence. If the count reaches zero after decrementing, the number x is removed from the counter to prevent future unnecessary checks.

Completing the Iteration: This process continues until either a missing number is found (in which case False is returned), or

Through these steps, the algorithm ensures that all possible consecutive sequences of length k are checked and formed,

all numbers are successfully grouped (in which case True is returned when the loop finishes).

validating the possibility of dividing the array into sets of k consecutive numbers accurately.

Validating Consecutive Numbers: For each number in the expected consecutive range [v, v + k), check if the current

number x is present in the counter (i.e., its count is not zero). If it is zero, this indicates that a consecutive sequence cannot

- **if** cnt[x] == 0: cnt.pop(x)
- into sets of k (4) consecutive numbers. Following the solution approach: **Counting Elements:** We count the frequency of each number using the Counter.

Suppose we have an array nums = [1, 2, 3, 3, 4, 4, 5, 6] and k = 4. We want to find out if it's possible to divide this array

Sorting and Iterating: We sort the nums and iterate over the distinct values. Here, the sorted unique values would be [1, 2,

Forming Consecutive Groups: We start from the smallest number and try to form a group of k consecutive numbers. Starting

Validating Consecutive Numbers: We check if each of these consecutive numbers is present in the counter with a non-zero

count.

with 1, then 2, 3, and 4.

For 1, cnt[1] = 1, thus we can use one 1.

For 2, cnt [2] = 1, we can use one 2.

For 3, cnt [3] = 2, we can use one 3. For 4, cnt [4] = 2, we can use one 4.

After using the numbers for the first group [1, 2, 3, 4], our counter updates to:

Counter({3: 1, 4: 1, 5: 1, 6: 1}) Completing the Iteration: We repeat the process for the next smallest number with a non-zero count, which in this updated counter is 3. We try to form the next group starting from 3, and we would need a sequence [3, 4, 5, 6].

Updating the Counter: After decrementing, if any count becomes zero, we remove the number from the counter.

• For 3, cnt[3] = 1, use one 3. For 4, cnt [4] = 1, use one 4. • For 5, cnt[5] = 1, use one 5.

After this sequence, our counter is empty, which means we have successfully used all numbers to form groups of k

Since no step failed and we could form two groups [1, 2, 3, 4] and [3, 4, 5, 6] each with 4 consecutive numbers, the

function would return True. Hence, it is possible to divide the given array nums into sets of k (4) consecutive numbers.

Solution Implementation **Python**

from collections import Counter

num_count = Counter(nums)

for num in sorted(nums):

if num_count[num]:

def isPossibleDivide(self, nums: List[int], k: int) -> bool:

Loop over each number after sorting nums

for x in range(num, num + k):

if num count[x] == 0:

return False

if num_count[x] == 0:

public boolean isPossibleDivide(int[] nums, int k) {

if (frequencyMap.containsKey(num)) {

return false;

for (int i = num; i < num + k; ++i) {</pre>

if (!frequencyMap.containsKey(i)) {

for (int num : nums) {

Arrays.sort(nums);

for (int num : nums) {

Map<Integer, Integer> frequencyMap = new HashMap<>();

// Create a map to store the frequency of each number in the input array.

frequencyMap.put(num, frequencyMap.getOrDefault(num, 0) + 1);

// Sort the input array to ensure the numbers are in ascending order.

// Iterate over the sorted array to check if division into subsequences is possible.

// Only start a sequence if the current number is still in the frequencyMap.

// Attempt to create a subsequence of length k starting with the current number.

// If the current number is not in the frequencyMap, division is not possible.

num_count.pop(x)

num count[x] -= 1

Create a frequency count for all the numbers in nums

If this number is still in the count dictionary

Attempt to create a consecutive sequence starting at this number

If the count drops to zero, remove it from the dictionary

If any number required for the sequence does not exist, return False

Decrease the count for this number since it's used in the sequence

class Solution:

For 6, cnt[6] = 1, use one 6.

consecutive numbers.

If the entire loop completes without returning False, it means all sequences can be formed return True Java

```
/**
* Checks if it is possible to divide the array into consecutive subsequences of length k.
 * @param nums Input array of integers.
* @param k Length of the consecutive subsequences.
* @return true if division is possible, false otherwise.
```

*/

import java.util.Arrays;

import java.util.Map;

class Solution {

import java.util.HashMap;

```
// Decrease the frequency of the current number.
                    frequencyMap.put(i, frequencyMap.get(i) - 1);
                    // If the frequency of a number becomes 0, remove it from the map.
                    if (frequencyMap.get(i) == 0) {
                        frequencyMap.remove(i);
       // If the loop completes, then division into subsequences of length k is possible.
       return true;
C++
#include <vector>
#include <unordered_map>
#include <algorithm>
class Solution {
public:
   // Function to determine if it is possible to divide the vector of integers into groups of size 'k'
    // with consecutive numbers.
    bool isPossibleDivide(std::vector<int>& nums, int k) {
       // Creating a frequency map to count occurrences of each number
        std::unordered_map<int, int> frequencyMap;
        for (int num : nums) {
            ++frequencyMap[num];
       // Sort the input vector to process numbers in ascending order
       std::sort(nums.begin(), nums.end());
       // Process each number in the sorted vector
        for (int num : nums) {
           // If there is still a count for this number, we need to form a group starting with this number
            if (frequencyMap.find(num) != frequencyMap.end()) {
                // Attempt to create a group of 'k' consecutive numbers
                for (int i = num; i < num + k; ++i) {</pre>
                    // If any number required to form the group is missing, return false
                    if (!frequencyMap.count(i)) {
                        return false;
                    // Decrement the count for the current number in the group
                    if (--frequencyMap[i] == 0) {
                        // If the count reaches zero, remove the number from the frequency map
                        frequencyMap.erase(i);
```

// If the function hasn't returned false, it's possible to divide the numbers as required

// If there is still a count for this number, we need to form a group starting with it

// If any number required to form the group is missing, return false

// If the count reaches zero, remove the number from the frequency map

Attempt to create a consecutive sequence starting at this number

If the count drops to zero, remove it from the dictionary

If any number required for the sequence does not exist, return False

Decrease the count for this number since it's used in the sequence

If the entire loop completes without returning False, it means all sequences can be formed

// If the function hasn't returned false, it's possible to divide the numbers as required return true;

class Solution:

return true;

for (const num of nums) {

nums.sort((a, b) \Rightarrow a - b);

for (const num of nums) {

if (num in frequencyMap) {

return false;

from collections import Counter

return True

it's removed.

Time and Space Complexity

num_count = Counter(nums)

for num in sorted(nums):

if num_count[num]:

function isPossibleDivide(nums: number[], k: number): boolean {

const frequencyMap: Record<number, number> = {};

// Process each number in the sorted array

for (let i = num; i < num + k; i++) {</pre>

if (!(i in frequencyMap)) {

if (--frequencyMap[i] === 0) {

delete frequencyMap[i];

frequencyMap[num] = (frequencyMap[num] || 0) + 1;

// Create a frequency map to count occurrences of each number

// Sort the input array to process numbers in ascending order

// Attempt to create a group of 'k' consecutive numbers

def isPossibleDivide(self, nums: List[int], k: int) -> bool:

Loop over each number after sorting nums

for x in range(num, num + k):

if num count[x] == 0:

return False

if num_count[x] == 0:

num_count.pop(x)

num_count[x] -= 1

Create a frequency count for all the numbers in nums

If this number is still in the count dictionary

// Decrement the count for the current number in the group

TypeScript

```
Time Complexity
  The time complexity of the given code is determined by a few factors: the sorting of the input list nums, the construction of the
  counter cnt, and the nested loop where we check and decrement the count for each element over the range from v to v + k.
     Sorting nums: The sort operation on the list nums has a time complexity of O(N log N), where N is the number of elements in
     nums.
      Counter Construction: Constructing the counter cnt is O(N) because we go through the list nums once.
     Nested Loop: The nested loop involves iterating over each number in the sorted nums and then an inner loop that iterates up
      to k times for each unique number that has a non-zero count.
```

The total number of decrements across all elements cannot exceed N (since each decrement corresponds to one element of nums), and since we have that outer loop that potentially could visit all N elements, we would multiply this by k giving us 0(Nk).

So, combining these together, the total time complexity is $0(N \log N + N + N k) = 0(N \log N + N k)$. Since for large N, N log N

• The counter can have at most M entries where M is the number of unique numbers in nums. In the worst case, if all numbers are unique, M is equal

dominates N, and Nk may either dominate or be dominated by N log N depending on the values of N and k, the final time

It may seem like this gives us 0(Nk), but this is not entirely correct because each element is decremented once, and once it

hits zero, it is popped from the counter and never considered again. Therefore, each element contributes at most 0(k) before

Space Complexity The space complexity is mostly determined by the counter cnt which stores a count of each unique number in nums.

complexity is often written with both terms.

- to N, giving us a space complexity of O(N). Therefore, the overall space complexity of the code is O(N).