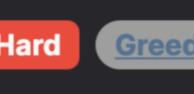
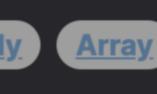
## 2350. Shortest Impossible Sequence of Rolls







Hash Table Greedy Array **Leetcode Link** 

## **Problem Description**

In this problem, we are given two inputs: an integer array rolls of size n, which represents the outcomes of rolling a k-sided dice n times, with k being the second input — a positive integer. The dice have sides numbered from 1 to k.

We need to find the length of the shortest sequence of rolls that is not present in the given rolls array. A sequence of rolls is

defined by the numbers that are obtained when rolling the k-sided dice some number of times. The key point is that we're not looking for a sequence that must be in consecutive order in rolls, but the numbers must appear in the same order as they would in the dice rolls. To put it in simple words: imagine you are writing down each result of rolling a k-sided dice on a piece of paper, one after another.

You now have to find the smallest length of a list of numbers that you could never write down only by using the numbers in the exact same order they appear in the rolls array.

### The solution to this problem relies on the observation that the length of the shortest absent rolls sequence depends directly on the

Intuition

look at the input array as a series of segments where, once we encounter all k unique dice numbers, we can start looking for a new sequence. The approach for the solution is to track the unique numbers we roll with a set s. As we iterate over the rolls:

variety of numbers in each segment of the array. Since we want to find the shortest sequence that can't be found in rolls, we can

2. We check if the size of the set s has reached k. If it has, it means we've seen all possible outcomes from the dice in this

segment.

1. We add each number to the set s.

- 3. Once we see that all numbers have appeared, it implies that any sequence of length equal to the current answer ans can be constructed from the segment, so we begin searching for the next longer sequence that cannot be constructed, by incrementing
- ans by 1. 4. We then clear the set s to start checking for the next segment. With this strategy, each time we complete a full set of k unique numbers, the length of the shortest missing sequence increases,
- since we're able to construct any shorter sequence up to that point. The increment in ans represents the sequential nature of sequences that can't be found in rolls. When we have gone through all the elements in rolls, the value stored in ans will be the

length of the shortest sequence that didn't appear in the rolls.

be generated. Thus, we increment ans by 1.

**Solution Approach** The solution utilizes a greedy approach which aims to construct the shortest missing sequence incrementally.

## Algorithm:

1. We initialize an empty set s. The purpose of this set is to store unique dice roll outcomes from the rolls array as we iterate through it.

## 2. We also define an integer ans, which is initialized at 1. This variable represents the length of the current shortest sequence that

- cannot be found in rolls. 3. We iterate through every integer v in the rolls array and add the roll outcome v to the set s every time. This is our way of
- once. If a particular number repeats in rolls, it does not affect our counting in the set s. 4. After each insertion, we check if the size of the set s has reached the size k. This step is key to the implementation, since reaching a set size of k implies that we have seen all possible outcomes that could be the result of a dice roll.

keeping track of the outcomes we have seen so far. By using a set, we automatically ensure that each outcome is only counted

- If and when the set size is k, it means we have found a sequence of rolls of length ans that can be created using rolls. Consequently, we can't be looking for sequences of this length anymore; we need to look for a longer sequence that cannot
- unique dice outcomes. 5. The loop repeats until all dice rolls in rolls have been examined. By now, ans will be one more than the length of the longest sequence of rolls that can be found in rolls. Therefore, ans will be the shortest sequence length that cannot be constructed

To start looking for the next shortest missing sequence, we need to clear the set s. By doing so, we reset our tracking of

**Data Structures:** • The use of a set is essential in this approach. The set allows us to maintain a collection of unique items, which is perfect for keeping track of which dice numbers we have encountered. As a set doesn't store duplicates, it's also very efficient for this use

## • The use of an integer, ans, to represent the length of the sequence we're currently looking for.

case.

from rolls.

Patterns: The pattern in the solution approach follows a greedy algorithm. Greedy algorithms make the locally optimum choice at each step with the intent of finding the global optimum.

In summary, while iterating over the array rolls, we are greedily increasing the length of the sequence ans whenever we confirm that

a sequence of that length can indeed be created using the rolls seen so far. The final value of ans when we have completed our

iteration is the length of the shortest sequence that cannot be constructed from rolls.

No increment to ans is needed since the set does not yet contain all k unique rolls.

• We still don't need to increment ans as s does not contain all k unique rolls.

• The set still doesn't contain all k unique numbers, so we don't increment ans.

Let's illustrate the solution approach with a small example: Assume we are given an array rolls = [1, 2, 3, 3, 2], and the dice has k = 3 sides. Our goal is to find the length of the shortest

1. First, we initialize an empty set s and an integer ans with a value of 1. The set s will track unique dice roll outcomes, and ans

### represents the length of the current shortest sequence not found in rolls. 2. We begin iterating over each element in rolls:

Example Walkthrough

We add 1 to the set s, which becomes {1}.

sequence of dice rolls that is not represented in rolls.

- $\circ$  We move to the next roll, adding 2 to the set. Now  $s = \{1, 2\}$ .
  - to start tracking the next sequence.  $\circ$  The next roll is a 3. We add it to the now-empty set s, resulting in  $s = \{3\}$ .

 $\circ$  Finally, we add the last roll, 2, to the set. Now  $s = \{2, 3\}$ .

 $\circ$  We add the next roll, 3, to the set. Now  $s = \{1, 2, 3\}$ .

• Since we have reached the end of the rolls array and the set s does not contain k unique numbers, we stop here. 3. After completing the iteration, the value of ans stands at 2.

Therefore, the length of the shortest sequence of rolls that cannot be found in rolls is 2. This means there is no sequence of two

• Since s now contains k unique numbers (all possible outcomes of the dice), we have seen at least one of each possible roll.

At this point, we can construct any sequence of length 1 with the numbers in s, so we increment ans to 2 and clear the set s

from typing import List class Solution: def shortestSequence(self, rolls: List[int], k: int) -> int: # `shortestSeqLength` will hold the length of the shortest sequence that is not a subsequence of `rolls`

# Clear the set to start tracking a new subsequence

# Return the length of the shortest sequence that is not a subsequence of `rolls`

rolls in the order they were rolled that we did not see at least once in the given rolls array.

15 # Check if we have seen all `k` different numbers 16 17 if len(uniqueNumbers) == k: # If true, we can form a new subsequence which will not be a subsequence of the current `rolls` 18 shortestSeqLength += 1 19

# `uniqueNumbers` will keep track of the unique numbers we have seen in the current subsequence

# Java Solution

class Solution {

**Python Solution** 

9

10

11

12

14

20

21

22

23

24

26

shortestSeqLength = 1

uniqueNumbers = set()

for number in rolls:

uniqueNumbers.add(number)

uniqueNumbers.clear()

public int shortestSequence(int[] rolls, int k) {

int shortestSequence(vector<int>& rolls, int k) {

// Iterate over the roll values

for (int roll : rolls) {

return shortestSeqLength

# Iterate over each number in the `rolls` list

# Add the number to the set of unique numbers

#### // Initialize a set to keep track of unique elements Set<Integer> set = new HashSet<>(); // Initialize the answer variable which represents the shortest sequence int answer = 1; // Iterate through each number in the rolls array for (int number : rolls) { 9 // Add the current number to the set set.add(number); // Check if the set size equals k, meaning all numbers are present 13 **if** (set.size() == k) { // Reset the set for the next sequence 14 set.clear(); // Increment the answer value, as we've completed a sequence 16 17 answer++; 18 19 20 // Return the count of the shortest sequence return answer; 22 23 24 C++ Solution

// Function to find the shortest sequence that contains every number from 1 to k

numbers.insert(roll); // Insert the current roll value into the set

// If the size of the set equals k, we have found a full sequence

unordered\_set<int> numbers; // Set to store unique numbers

int sequenceLength = 1; // Initialize the sequence length as 1

#### 16 17 18 19

1 #include <vector>

class Solution {

public:

6

9

10

11

12

14

#include <unordered\_set>

```
if (numbers.size() == k) {
15
                   numbers.clear(); // Clear the set for the next sequence
                   ++sequenceLength; // Increment the sequence length counter
20
21
           return sequenceLength; // Return the shortest sequence length
22
23 };
24
Typescript Solution
   // Importing Set from the ES6 standard library
  import { Set } from "typescript-collections";
   // Function to find the shortest sequence that contains every number from 1 to k
   function shortestSequence(rolls: number[], k: number): number {
       const numbers: Set<number> = new Set(); // Set to store unique numbers
       let sequenceLength: number = 1; // Initialize the sequence length as 1
 8
       // Iterate over the roll values
9
       rolls.forEach(roll => {
           numbers.add(roll); // Insert the current roll value into the set
```

### 13 14 15

10 11 // If the size of the set equals k, we have found a full sequence if (numbers.size === k) { numbers.clear(); // Clear the set for the next sequence sequenceLength++; // Increment the sequence length counter 16

# **Time Complexity**

Time and Space Complexity

}); return sequenceLength; // Return the shortest sequence length 19 20 } 21

## size of the set when cleared. Space Complexity

The space complexity is O(k) because the set s is used to store at most k unique values from the rolls list at any given time. The other variables, and v, use a constant amount of space, hence they contribute 0(1), which is negligible in comparison to 0(k).

The time complexity of the given code is O(n) where n is the length of the input list rolls. This is because the code iterates through

reaches the size k, it is cleared, which is also an 0(1) operation because it happens at most n/k times and does not depend on the

each roll exactly once. Inside the loop, adding an element to the set s and checking its length are both 0(1) operations. When s