# distinct digit numbers hackerrank solution

The "Distinct Digit Numbers" problem on HackerRank presents a fascinating challenge for programmers looking to hone their understanding of algorithms and data structures. This article will delve deep into various approaches and solutions for efficiently identifying numbers with distinct digits. We'll explore the fundamental concepts behind this problem, analyze different programming techniques to tackle it, and provide clear, actionable insights for achieving an optimal HackerRank distinct digit numbers solution. Whether you're a beginner seeking a foundational understanding or an experienced coder aiming for peak performance, this comprehensive guide will equip you with the knowledge to conquer this common coding interview question.

## Table of Contents


- Understanding the Distinct Digit Numbers Problem
- Core Concepts for Solving Distinct Digit Numbers
- Algorithm Design for Distinct Digit Numbers
- Common Approaches to HackerRank Distinct Digit Numbers
- Detailed Solution Breakdown: Using Sets
- Alternative Solution: String Manipulation and Iteration
- Optimizing Your Distinct Digit Numbers Solution
- Common Pitfalls and How to Avoid Them
- Example Test Cases and Explanations
- Conclusion: Mastering the Distinct Digit Numbers Challenge


## Understanding the Distinct Digit Numbers Problem

The "Distinct Digit Numbers" problem on HackerRank typically involves determining how many numbers within a given range possess a unique characteristic: all of their digits are distinct. This means that no digit

appears more than once within the number. For instance, 123 has distinct digits, while 112 or 232 do not. The challenge lies in efficiently iterating through a potentially large range of numbers and performing this check for each one. Understanding this core requirement is the first step towards devising an effective strategy for your HackerRank distinct digit numbers solution.

The problem statement usually specifies a lower bound and an upper bound for the range. Your task is to count all integers 'x' such that lower_bound <= x <= upper_bound, where 'x' is a distinct digit number. This seemingly simple condition can lead to surprisingly complex algorithmic considerations when the range becomes extensive. Therefore, a robust and well-thought-out approach is crucial for success.

# Core Concepts for Solving Distinct Digit Numbers

To effectively tackle the distinct digit numbers problem, several fundamental programming concepts are paramount. These concepts form the building blocks for any efficient solution you might devise. Understanding these will not only help you solve this specific HackerRank challenge but also equip you with transferable skills for a wide array of algorithmic problems.

## Digit Extraction

The most crucial step in determining if a number has distinct digits is to be able to access and examine each of its individual digits. There are several common methods for digit extraction. One involves using the modulo operator (%) to get the last digit and integer division (/) to remove it, repeating this process until the number becomes zero. Another approach is to convert the number to a string and then iterate through the characters of the string.

## Uniqueness Checking

Once you have extracted the digits, you need a mechanism to check if they are all unique. This is where data structures designed for efficient membership testing and duplicate detection come into play. The core idea is to keep track of the digits encountered so far. If a digit is encountered that has already been seen, the number is not a distinct digit number.

## Iteration and Range Management

The problem inherently involves iterating through a range of numbers. This means you'll need a loop that starts from the lower bound and goes up to the

upper bound. For each number within this range, you will apply your distinct digit checking logic. Efficient range management, especially for very large ranges, might involve considering more advanced techniques if a brute-force iteration becomes too slow.

# Algorithm Design for Distinct Digit Numbers

Designing an efficient algorithm is key to a successful HackerRank distinct digit numbers solution. A naive approach might work for small ranges but will quickly become intractable as the bounds increase. We need an algorithm that balances clarity with computational efficiency.

## Brute-Force Iteration

The most straightforward approach is to iterate through every number from the lower bound to the upper bound. For each number, we check if it has distinct digits. If it does, we increment a counter. While simple to understand, this approach can be very slow for large ranges, often leading to Time Limit Exceeded (TLE) errors on platforms like HackerRank.

## Optimized Iteration with Distinct Digit Check

To improve upon the brute-force method, we need to ensure our distinct digit check is as efficient as possible. The choice of data structure for tracking seen digits significantly impacts performance. Using a boolean array or a hash set can offer near constant-time lookups, making the check for each digit very fast.

# Common Approaches to HackerRank Distinct Digit Numbers

Several common algorithmic patterns and data structures are frequently employed when solving the distinct digit numbers problem on HackerRank. Understanding these will give you a solid foundation for developing your own solutions.

## Using a Set for Uniqueness

Sets are ideal for this problem because they inherently store only unique elements. When checking a number's digits, you can add each digit to a set. If the size of the set after processing all digits is equal to the total

number of digits in the number, then all digits are distinct. This is a very clean and Pythonic way to solve the problem.

## Using a Boolean Array/Frequency Map

For problems involving digits (0-9), a boolean array of size 10 can be very efficient. You can initialize all elements to false. As you extract each digit, you check the corresponding index in the array. If it's already true, the number has duplicate digits. Otherwise, you mark that index as true. This method is often faster than using a hash set due to direct array access.

## String Conversion and Iteration

Converting the number to a string allows for easy iteration over its digits as characters. You can then use either a set or a frequency map to check for duplicate characters (digits) within the string. This approach can be more readable for some programmers.

# Detailed Solution Breakdown: Using Sets

One of the most elegant and commonly used methods for solving the distinct digit numbers problem involves leveraging the properties of sets. Sets, by definition, store only unique elements, making them a natural fit for identifying distinct digits within a number. Let's break down how this approach works in detail.

## The Set-Based Algorithm

The core idea is to iterate through each number within the specified range (from the lower bound to the upper bound). For every number, we perform the following steps:

1. Initialize an empty set, say `seen_digits`.

2. Extract digits from the current number. A common way to do this is by repeatedly taking the number modulo 10 to get the last digit and then dividing the number by 10 to remove the last digit, until the number becomes 0.

3. For each extracted digit, try to add it to the `seen_digits` set.

4. After processing all digits of the number, compare the size of the `seen_digits` set with the total number of digits in the original number. If they are equal, it means all digits were unique, and we

increment our count of distinct digit numbers.

## Example Implementation Logic (Conceptual)

Consider the number 123.

- Initialize `seen_digits = {}`.
- Digit 3 is extracted. Add 3 to `seen_digits`. `seen_digits = {3}`.
- Number becomes 12. Digit 2 is extracted. Add 2 to `seen_digits`. `seen_digits = {3, 2}`.
- Number becomes 1. Digit 1 is extracted. Add 1 to `seen_digits`. `seen_digits = {3, 2, 1}`.
- Number becomes 0. Stop.
- The size of `seen_digits` is 3. The original number 123 has 3 digits. Since 3 == 3, the number 123 has distinct digits.

Now consider the number 121.

- Initialize `seen_digits = {}`.
- Digit 1 is extracted. Add 1 to `seen_digits`. `seen_digits = {1}`.
- Number becomes 12. Digit 2 is extracted. Add 2 to `seen_digits`. `seen_digits = {1, 2}`.
- Number becomes 1. Digit 1 is extracted. Try to add 1 to `seen_digits`. Since 1 is already in the set, the set remains `{1, 2}`.
- Number becomes 0. Stop.
- The size of `seen_digits` is 2. The original number 121 has 3 digits. Since 2 != 3, the number 121 does not have distinct digits.

This set-based approach is straightforward to implement and generally performs well for most typical HackerRank constraints.

# Alternative Solution: String Manipulation and Iteration

While using sets is a common and efficient method, an alternative approach for solving the distinct digit numbers problem involves direct string manipulation combined with iterative checking. This method can sometimes be more intuitive for those comfortable with string operations.

## String Conversion and Character Checking

The fundamental idea here is to convert the integer into its string representation. Once the number is a string, each character (which represents a digit) can be accessed individually. The challenge then becomes how to efficiently check for duplicate characters within this string.

## Using a Frequency Array or Dictionary

A common technique with string manipulation is to use a frequency map (or an array acting as a frequency map) to count the occurrences of each character (digit). For example, you could use a simple array of size 10, indexed by the digit value. Iterate through the string representation of the number. For each digit character, convert it back to an integer and increment its count in the frequency array. If at any point you find that a digit's count is already 1 before incrementing it, you know there's a duplicate, and the number is not a distinct digit number. You can then break early from checking this number.

## Early Exit Strategy

An important optimization when using string manipulation is the "early exit" strategy. As soon as you detect a duplicate digit within a number (by finding that a digit's count is already 1 before incrementing it), you can immediately stop processing that particular number and move on to the next one in the range. This avoids unnecessary computations and improves overall efficiency, especially for numbers with duplicates appearing early in their digit sequence.

Consider the number 4894.

- Convert to string: "4894".
- Initialize a frequency array `counts` of size 10 to all zeros.
- Process '4': Convert to integer 4. `counts[4]` becomes 1.

- Process '8': Convert to integer 8. `counts[8]` becomes 1.
- Process '9': Convert to integer 9. `counts[9]` becomes 1.
- Process '4': Convert to integer 4. Check `counts[4]`. It is already 1. This indicates a duplicate. The number 4894 is not a distinct digit number. Stop processing this number and move to the next in the range.

This string-based method, particularly with an early exit, can be a competitive alternative to the set-based approach.

# Optimizing Your Distinct Digit Numbers Solution

Achieving an optimal solution for the distinct digit numbers problem on HackerRank often requires more than just a correct algorithm; it demands performance optimization. Even with a well-chosen approach like using sets, there are nuances that can significantly impact execution time, especially when dealing with large input ranges.

## Pre-computation (if applicable)

In some scenarios, if the range of possible numbers is bounded and known beforehand, you might consider pre-computing whether each number within that maximum possible range has distinct digits. You could store these results in a boolean array or a similar data structure. Then, for any given query range, you can simply sum up the pre-computed results within that range. This shifts the computational cost to an initial setup phase, making subsequent queries very fast. However, for HackerRank problems where the range is directly provided as input and can be very large, direct pre-computation of the entire possible range might be infeasible due to memory or time constraints.

## Algorithmic Complexity Analysis

Understanding the time complexity of your chosen algorithm is crucial.

- A brute-force approach with a simple digit check (e.g., nested loops to compare digits) might have a complexity of roughly O(N D^2), where N is the size of the range and D is the maximum number of digits.
- Using a set or a frequency array for digit checking within the loop typically brings the complexity down to O(N D), as checking for uniqueness of D digits takes O(D) time with efficient data structures.
- The maximum number of digits (D) for numbers within typical integer

limits is small (e.g., for a 32-bit integer, it's around 10 digits). Therefore, O(N D) is usually acceptable if N is not excessively large.

## Efficient Digit Extraction

Ensure your digit extraction method is efficient. While both modulo/division and string conversion work, performance can vary slightly across programming languages. For languages like Python, string conversion might be slightly less performant than repeated modulo and division operations for very large numbers due to the overhead of string object creation and manipulation. However, the difference is often marginal for typical problem constraints.

# Common Pitfalls and How to Avoid Them

When developing a HackerRank distinct digit numbers solution, several common mistakes can lead to incorrect results or performance issues. Being aware of these pitfalls can save you significant debugging time.

## Off-by-One Errors in Ranges

Ensure your loops correctly handle the inclusive nature of the given range bounds. A common mistake is to iterate up to `upper_bound - 1` when it should be `upper_bound` or to start from `lower_bound + 1` when it should be `lower_bound`. Always double-check your loop conditions.

## Handling Zero and Single-Digit Numbers

Numbers like 0 and single-digit numbers (1 through 9) inherently have distinct digits. Your logic should correctly identify these. For example, 0 has one digit (0) which is distinct. A number like 5 has one digit (5) which is distinct. Ensure your digit extraction and uniqueness checking mechanisms don't incorrectly exclude these valid cases.

## Integer Overflow Issues

While less common for the distinct digit check itself, if you're performing intermediate calculations or manipulations on very large numbers, be mindful of potential integer overflow issues if your programming language has fixed-size integer types. HackerRank environments usually provide sufficiently large integer types, but it's good practice to be aware.

## Inefficient Uniqueness Check

Using an inefficient method for checking digit uniqueness, such as nested loops to compare every digit with every other digit for each number, will drastically slow down your solution. Always opt for data structures like sets or frequency arrays/maps for O(1) or O(log D) average-case lookups, where D is the number of digits.

# Example Test Cases and Explanations

To solidify your understanding and verify the correctness of your distinct digit numbers solution, let's walk through a few example test cases. These examples illustrate how different numbers are processed and why they are or are not counted.

## Test Case 1: Simple Range

**Input Range:** 1 to 20

**Expected Output:** 19

**Explanation:**

- Numbers from 1 to 9 all have distinct digits (single digit).
- 10: Distinct (1, 0).
- 11: Not distinct (1 repeated).
- 12: Distinct (1, 2).
- 13: Distinct (1, 3).
- 14: Distinct (1, 4).
- 15: Distinct (1, 5).
- 16: Distinct (1, 6).
- 17: Distinct (1, 7).
- 18: Distinct (1, 8).
- 19: Distinct (1, 9).
- 20: Distinct (2, 0).
- The only number excluded is 11. So, 20 numbers in total, minus 1

excluded number, gives 19 distinct digit numbers.

## Test Case 2: Range with Duplicates

**Input Range:** 95 to 105

**Expected Output:** 8

**Explanation:**

- 95: Distinct (9, 5).
- 96: Distinct (9, 6).
- 97: Distinct (9, 7).
- 98: Distinct (9, 8).
- 99: Not distinct (9 repeated).
- 100: Not distinct (0 repeated).
- 101: Not distinct (1 repeated).
- 102: Distinct (1, 0, 2).
- 103: Distinct (1, 0, 3).
- 104: Distinct (1, 0, 4).
- 105: Distinct (1, 0, 5).
- Numbers counted: 95, 96, 97, 98, 102, 103, 104, 105. Total = 8.

## Test Case 3: Edge Cases

**Input Range:** 1 to 1

**Expected Output:** 1

**Explanation:** The number 1 has a single digit, which is distinct.

**Input Range:** 11 to 11

**Expected Output:** 0

**Explanation:** The number 11 has duplicate digits.

# Conclusion: Mastering the Distinct Digit Numbers Challenge

The "Distinct Digit Numbers" problem on HackerRank serves as an excellent benchmark for assessing a programmer's grasp of fundamental algorithmic principles and data structure application. By understanding the core requirement of identifying numbers with unique digits and employing efficient techniques for digit extraction and uniqueness checking, you can construct robust and performant solutions. Whether you choose the elegance of sets or the directness of string manipulation with frequency tracking, the key lies in optimizing for speed and accuracy. Avoiding common pitfalls like off-by-one errors and inefficient uniqueness checks is crucial for success. With a solid understanding of these concepts and the ability to analyze algorithmic complexity, you are well-equipped to master the distinct digit numbers challenge and enhance your problem-solving toolkit.

# Frequently Asked Questions

## What's the core problem being solved in the Distinct Digit Numbers problem on HackerRank?

The problem asks to count the number of integers within a given range [L, R] that have only distinct digits. This means no digit can repeat within a number.

## What is the typical approach to solving the Distinct Digit Numbers problem efficiently?

A common and efficient approach involves digit DP (Dynamic Programming). This technique allows us to build numbers digit by digit and keep track of whether digits have been repeated.

## Can you explain the state definition for a typical digit DP solution for this problem?

A typical state might include: current position (index of the digit being placed), tight constraint (whether we are restricted by the upper bound of the range), isLeadingZero (to handle leading zeros), and a bitmask representing which digits have already been used.

## How does the 'tight' constraint work in digit DP for this problem?

The 'tight' constraint is a boolean flag. If true, it means we are currently

building a number that matches the prefix of the upper bound. This restricts the current digit we can place to be at most the corresponding digit in the upper bound. If false, we can place any digit from 0 to 9.

## Why is handling leading zeros important in Distinct Digit Numbers?

Leading zeros don't affect the 'distinctness' of digits in the way we usually think (e.g., 007 is just 7, and 7 has distinct digits). The `isLeadingZero` flag in the DP helps to correctly skip placing digits if we are at the beginning of the number and the current digit being considered is 0, ensuring we don't count numbers like '012' as distinct from '12' prematurely.

## How do we convert the count for a range [L, R] using a helper function?

Most digit DP solutions calculate the count of numbers with distinct digits from 0 up to a given number `N`. To find the count for [L, R], we typically calculate `count(R) - count(L-1)`.

# Additional Resources

Here are 9 book titles and descriptions related to distinct digit numbers and problem-solving, with a focus on algorithmic thinking that might apply to a Hackerrank solution:

1. _The Art of Computer Programming, Volume 1: Fundamental Algorithms_
This foundational text by Donald Knuth delves deep into the core concepts of algorithms and data structures. It covers everything from sorting and searching to number theory and combinatorial algorithms, providing a rigorous understanding of how to design and analyze efficient computational processes. For a problem involving distinct digits, understanding basic arithmetic and combinatorial principles from this book would be invaluable for devising a robust solution.

2. _Cracking the Coding Interview: 189 Programming Questions and Solutions_
This practical guide by Gayle Laakmann McDowell is essential for anyone preparing for technical interviews. It presents a wide range of common interview problems, including those involving number manipulation and properties. The systematic approach to problem decomposition and solution development outlined here is directly applicable to tackling challenges on platforms like HackerRank.

3. _Introduction to Algorithms_
Often referred to as "CLRS," this comprehensive textbook by Cormen, Leiserson, Rivest, and Stein provides a thorough treatment of algorithms. It covers a vast array of topics, from basic data structures to advanced graph algorithms and computational geometry. Understanding concepts like time

complexity and optimization techniques from this book would be crucial for ensuring an efficient solution to any digit-related problem.

4. _Algorithmic Thinking: How to Solve Problems with Algorithms_
Written by Daniel Zingaro, this book focuses on developing the crucial skill of algorithmic thinking itself. It breaks down the process of translating problem statements into working code, emphasizing logic, decomposition, and efficiency. For a HackerRank problem focused on distinct digits, this book would guide the solver in systematically breaking down the constraints and exploring different algorithmic approaches.

5. _Number Theory for Computer Science_
This specialized book by Donald Knuth (part of his larger series, but a good standalone concept) would be highly relevant. It explores the mathematical underpinnings of numbers and their properties, which is directly applicable to problems involving digit analysis. Concepts like divisibility, prime numbers, and number representations would be key tools for solving distinct digit challenges.

6. _Data Structures and Algorithms Made Easy_
This book by Narasimha Karumanchi offers a more accessible approach to learning data structures and algorithms. It covers essential topics with clear explanations and numerous examples, making complex concepts understandable. For someone tackling a HackerRank problem, this book provides the building blocks and practical examples needed to implement efficient solutions.

7. _A Mathematical Approach to Computer Science_
This title suggests a book that bridges the gap between mathematical theory and computational practice. It likely covers topics like discrete mathematics, logic, and foundational mathematical concepts that are critical for understanding number properties. Such a book would equip a solver with the theoretical knowledge to analyze the behavior of numbers and their digits.

8. _Competitive Programming 3_
By Steven Halim and Felix Halim, this book is a well-regarded resource for those participating in competitive programming. It focuses on common algorithmic patterns and techniques used to solve problems efficiently under time constraints. A problem involving distinct digits might require knowledge of string manipulation, number theory, and optimized search algorithms, all of which are likely covered here.

9. _Problem Solving with Algorithms and Data Structures Using Python_
This book by Bradley N. Miller and David L. Ranum provides a practical introduction to algorithms and data structures with a focus on the Python programming language. It explains fundamental concepts with code examples, making it easy to apply them to real-world problems. For a HackerRank solution involving distinct digits, this book would offer the programming tools and conceptual understanding to build an effective solution.

# Distinct Digit Numbers Hackerrank Solution

Distinct Digit Numbers Hackerrank Solution

## Related Articles

- dirty passage from 50 shades of grey
- diary of a wimpy kid script
- dimensional analysis practice problems with answers

Back to Home