2217. Find Palindrome With Fixed Length Medium Array Math

Leetcode Link

The task here is to find certain special numbers called "positive palindromes." A positive palindrome is a number that reads the same

Problem Description

positive integer intlength, our goal is to determine the intlength-digit palindrome corresponding to each query. For each integer in queries, we interpret it as the queries[i]-th smallest positive palindrome of length intLength. If this palindrome exists, we add it to our answer array; otherwise, we append -1 to indicate there's no such palindrome for that query.

both forward and backward, and it does not have any leading zeros. Given two inputs: an array of integers named queries and a

If we think about the nature of palindromes, we can realize that for a palindrome of a given length, the first half of the number dictates the second half due to the mirror-like property of palindromes. Therefore, finding a palindrome can be done by constructing its first half and then mirroring it to create the second half.

The problem requires us to handle the fact that palindromes of even and odd lengths behave slightly differently: an odd-length palindrome will have a single digit in the middle that isn't mirrored.

The solution utilizes the pattern that palindromes of a certain length can be constructed by taking a base number and mirroring its digits. The base number is essentially the first half of the palindrome.

To construct the smallest intLength-digit palindrome, we need to start with the smallest base possible that, once mirrored, forms a

palindrome of that length. This smallest base number is 10^(l-1) where l is the length of the base. The base itself is half of intLength: l = (intLength + 1) // 2.

The scope of possible bases ranges from this smallest base to 10^l - 1. This is because 10^l would result in a palindrome

exceeding intLength digits, violating the problem constraints.

For intLength that is odd, the middle digit is part of the base.

With this understanding, the solution consists of the following steps:

• For intLength that is even, the base directly mirrors itself to form the whole palindrome.

half of intLength, rounded up. 2. Determine the starting point (start) and the endpoint (end) for possible bases—the starting point being 10^(1-1) and the endpoint being 10^l - 1.

1. Calculate the length 1 of the base number for the palindrome. If intLength is even, 1 is half of intLength. If intLength is odd, 1 is

3. Iterate over each query in queries:

 \circ Calculate a tentative palindrome base v by adding the 0-based index of the query (q - 1) to the starting point.

exists. o Otherwise, create a string (s) for the first half of the palindrome (which is v), mirror it, and append it to itself. For odd

detailed breakdown of the implementation:

2. Define Start and End Range for Bases:

1. Determine Base Length (1):

1 l = (intLength + 1) >> 1

intLength, ignore the last digit of the mirrored portion to prevent duplication of the middle digit. Convert the resulting string back to an integer and append it to ans. 4. Return the completed ans list, which now contains the queries[i]-th smallest palindrome or -1 for each query tested.

∘ If this base is greater than the range's endpoint (end), append −1 to the answers array (ans), signifying no such palindrome

- **Solution Approach** The solution approach leverages simple integer and string operations to generate palindromes of a specific length. Here's a more
- constructed. This length is found by dividing intlength by 2 and rounding up if necessary. It uses the right shift operator >> 1 which is equivalent to dividing by 2.

• The first step in the algorithm involves calculating the length of the base number 1, from which the entire palindrome can be

10**(1 - 1). The endpoint is the largest integer of that half-length, 10**1 - 1. These define the range of numbers that can be the first half of a palindrome. 1 start, end = 10 ** (l - 1), 10 ** l - 1

3. Iterate Over Queries and Construct Palindromes:

The main logic happens in a loop iterating over each query. For each query, a base value v for the palindrome is created by

adding the query's 0-based index to the start of the range. Then the solution checks whether v exceeds the end, in which

• To form palindromes of intLength, the starting point is the smallest possible positive integer of half that length, which is

case it adds -1 to the answer array, indicating that no such palindrome exists within the given length constraint. 1 v = start + q - 12 **if** v > end:

ans.append(-1)

1 s += s[::-1][intLength % 2 :]

1 ans.append(int(s))

Example Walkthrough

4. Constructing the Full Palindrome String:

5. Finalize and Return the Answer: After constructing the full palindrome string for each query, it is converted back to an integer and appended to the answer array ans. Once all queries are processed, ans is returned as the result containing the requested palindromes or -1 when no such palindrome exists.

The algorithm effectively uses string manipulation to leverage the inherent symmetry of palindromes, constructing them in an

optimal manner by only dealing with half the number and mirroring it to get the full length. This means the time complexity is

Let's walk through a small example to illustrate the solution approach with queries = [1, 2, 4] and intLength = 3.

primarily determined by the number of queries and the computational complexity of string manipulation, both of which are managed

• When a valid base v is found, it is converted to a string s. To construct the whole palindrome, s is concatenated with its

% 2:] trims the first character from the reversed string before concatenation if intLength is odd.

reverse (s[::-1]). For palindromes of an odd length, the middle character should not be duplicated, so the slice [intLength

 \circ Since intLength is 3, which is odd, we calculate 1 as (3 + 1) >> 1, yielding 1 = 2.

1. Determine the Base Length (1):

For the first query (1):

reversing.

5. Finalize and Return the Answer:

2. Define Start and End Range for Bases:

○ The starting point start is 10**(2 - 1), which equals 10.

■ We calculate v as start + 1 - 1, which is 10.

■ We calculate v as start + 2 - 1, which is 11.

■ We calculate v as start + 4 - 1, which is 13.

Our final answer ans is [101, 111, 131].

half_length = (int_length + 1) >> 1

Initialize an empty list to store the answers

Convert the first half to a string

answers.append(int(palindrome))

Iterate over each query to find the k-th palindrome

start = 10 ** (half_length - 1)

value = start + query - 1

half_str = str(value)

Return the list of answers

return answers

end = $(10 ** half_length) - 1$

v is within the range, so we proceed to construct the palindrome.

■ The palindrome becomes '101', and we append the integer 101 to our answer ans.

■ This is also within range, so the palindrome would be '111', and we add 111 to ans.

○ The endpoint end is 10**2 - 1, which equals 99.

efficiently within the provided problem constraints.

3. Iterate Over Queries and Construct Palindromes: • The algorithm will loop over the queries [1, 2, 4] to find the corresponding palindromes. 4. Constructing Palindromes:

■ The string representation of v is '10', and because intLength is 3 (odd), we don't repeat the middle digit when

For the second query (2):

Python Solution

class Solution:

12

13

14

15

16

25

26

27

28

29

30

31

32

33

34

35

36

37

6

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

38

37 }

C++ Solution

#include <vector>

3 #include <algorithm>

#include <string>

class Solution {

public:

48

49

50

51

52

53

54

55

56

58

4

8

9

10

11

12

13

14

15

16

17

19

20

21

23

32

34

33 }

57 };

2 #include <cmath>

from typing import List

answers = []

for query in queries:

For the fourth query (4):

■ This base is not greater than end, so we construct the palindrome '131' and add 131 to ans.

∘ We also note that no queries in this example exceed our range, so there is no need to add -1 at any point.

The algorithm has successfully determined the queries[i]-th smallest positive palindrome of length intLength for each provided

Since there is no third query, we do not perform any action for it as it was not given in queries.

query. The palindromes are [101, 111, 131] for the 1st, 2nd, and 4th smallest palindromes of length 3, respectively.

def kth_palindrome(self, queries: List[int], int_length: int) -> List[int]:

Calculate the number of digits in the first half of the palindrome

Define the start and end of the range for the first half of the palindrome

19 20 # If the value exceeds the end boundary, the palindrome doesn't exist if value > end: 21 answers.append(-1)continue 24

Calculate the value in the first half by offsetting the start with the query index

Construct the full palindrome by concatenating the first half and its reverse

Append the palindrome to the answers list, converting it back to an integer

If the integer length is odd, skip the last digit of the reversed half

// Initialize an array for the answers with the same length as the queries array

// If the value exceeds the upper bound of halfLength palindrome, set the result as -1

new StringBuilder(halfPalindrome).reverse().substring(intLength % 2);

palindrome = half_str + half_str[::-1][int_length % 2:]

// Function to find kth palindromic number with specified length

// Calculate the length of the first half of the palindromes

public long[] kthPalindrome(int[] queries, int intLength) {

// Calculate the value for current palindrome

String halfPalindrome = Long.toString(value);

palindromes[i] = Long.parseLong(fullPalindrome);

// Return the array containing all the palindrome numbers

// Add to the results after parsing the string back to a long

std::string fullPalindromeStr = firstHalf + secondHalf;

palindromes.push_back(std::stoll(fullPalindromeStr));

// Return the vector containing all the found palindromes.

function kthPalindrome(queries: number[], intLength: number): number[] {

// Calculate the palindrome's non-reversed part.

const nonReversedPartOfPalindrome = baseNumber + query - 1;

const reversedPartOfPalindrome = nonReversedPartOfPalindrome

// Determine if the integer's length is odd.

const isOdd = intLength % 2 === 1;

const maxQueryValue = baseNumber * 9;

if (query > maxQueryValue) {

return queries.map(query => {

.toString()

Time and Space Complexity

.split('')

return palindromes;

Typescript Solution

// Convert the string to a long long and add to the results.

// Calculate the base number, which is the smallest number of the given integer length.

// Map each query to its corresponding palindrome or -1 if the query is out of bounds.

// Calculate the maximum valid query based on the integer length, used for bounds checking.

return -1; // Query value exceeds the upper limit of possible palindromes.

const baseNumber = 10 ** (Math.floor(intLength / 2) + (isOdd ? 1 : 0) - 1);

// It's used to generate palindromes by prefixing it to its reverse (or almost reverse if the length is odd).

// Convert the number to a string, reverse it, and slice off the first digit if the length is odd.

long value = startNum + queries[i] - 1;

long[] palindromes = new long[queries.length];

// Iterate through all the queries

palindromes[i] = -1;

// Convert the number to a string

// Generate the full palindrome string

String fullPalindrome = halfPalindrome +

if (value > endNum) {

continue;

return palindromes;

for (int i = 0; i < queries.length; ++i) {</pre>

int halfLength = (intLength + 1) >> 1; 8 // Determine the start position of palindromes 9 10 long startNum = (long) Math.pow(10, halfLength - 1); 11 // Determine the end position of palindromes 12 long endNum = (long) Math.pow(10, halfLength) - 1; 13

Java Solution

1 class Solution {

```
// This function finds the kth smallest palindrome of a given length.
        std::vector<long long> kthPalindrome(std::vector<int>& queries, int intLength) {
 9
           // Calculate the length of half of the palindrome
10
           // because a palindrome reads the same backward as forward.
11
            int halfLength = (intLength + 1) >> 1; // equivalent to (intLength + 1) / 2 but faster
12
13
14
           // Starting value for half of the palindrome (e.g., 100...0 with halfLength number of digits)
15
            long long start = std::pow(10, halfLength - 1);
16
17
            // Ending value for half of the palindrome (e.g., 999...9 with halfLength number of digits)
            long long end = std::pow(10, halfLength) - 1;
18
19
20
           // Vector to store the resulting palindromes.
21
            std::vector<long long> palindromes;
22
23
           // Loop through each query to find the respective palindrome.
            for (int q : queries) {
24
25
                // Calculate the value for this query by offsetting from the starting value.
26
                long long value = start + q - 1;
27
                // If the calculated number exceeds the largest number with halfLength digits, it is not possible
28
29
               // to generate the palindrome, so we add -1.
                if (value > end) {
30
                    palindromes.push_back(-1);
31
32
                    continue;
33
34
35
                // Convert the number to the first half of the palindrome.
                std::string firstHalf = std::to_string(value);
36
                // Prepare the second half by reversing the first half.
38
39
                std::string secondHalf = firstHalf;
40
                std::reverse(secondHalf.begin(), secondHalf.end());
41
42
                // If the palindrome is of odd length, we omit the last digit in the second half.
                if (intLength % 2 == 1) {
43
                    secondHalf = secondHalf.substr(1);
44
45
46
47
                // Combine both halves to form the full palindrome.
```

.reverse() .slice(is0dd ? 1 : 0) 26 27 .join(''); 28 29 // Construct the full palindrome and return it as a number. const palindrome = Number(nonReversedPartOfPalindrome.toString() + reversedPartOfPalindrome); 30 return palindrome;

});

Time Complexity

complexity.

Space Complexity

for each element in queries.

digits in the integer.

the space complexity beyond O(n).

2. **The loop**: The main loop runs once for each query in queries, hence, if there are n queries, the loop runs n times. 3. String operations within the loop: o Creating the s string: This involves creating a string representation of an integer which is O(L) where L is the number of

The time complexity of the given code can be broken down into a couple of key operations that occur within the loop running once

1. Calculation of the half-length: This is done only once before the loop, taking a constant time, so it does not affect the total time

• String slicing and concatenation: The slicing s[::-1] takes O(L) time and the concatenation s[::-1] [intLength % 2 :]

- Creating the ans list append operation: Appending to the list is 0(1). Since L (length of the palindrome) is at most half of intLength and intLength itself is a constant with respect to the length of
- queries, the operations inside the loop that depend on intLength can be considered to occur in constant time as well. Thus, the total time complexity for the loop can be approximated as O(n) where n is the number of queries.
- For space complexity, we are mostly concerned with additional space that the program uses aside from the input and output. 1. The ans list: This list increases proportionally with the number of queries n, so the space complexity contribution here is O(n).

2. Temporary variables v and s: These are used for each query and do not depend on the number of queries. They do not increase

Therefore, the overall space complexity is O(n) where n is the number of queries.

also takes O(L) time, as the length of the slice is proportional to L.