483. Smallest Good Base

Binary Search

Problem Description

Math

Hard

base k for n means that if we were to write n in base k, all its digits would be 1's. For example, the number 7 in base 2 is written as 111, so 2 is a good base for 7. The challenge is to find the smallest such base, which is always greater than or equal to 2.

The LeetCode problem is asking us to find the *smallest good base* for a given integer n that is represented as a string. A *good*

which all digits are 1's.

Intuition To approach this problem, the key observation is that n can be expressed as a sum of geometric series when using a base k in

 $n = 1 + k^1 + k^2 + ... + k^{m-1}$

- This is a geometric series, and we need to find the smallest k (good base) for which there is some m that satisfies the equation.
- Since n consists entirely of 1's for a good base k, the larger the base k, the fewer digits m we'll need, and vice versa. This means

Mathematically, for a base k and m digits in base k representation, n can be expressed as:

that for larger m, k will be smaller.

The maximum m is bounded due to the size of n - n's binary representation is the longest m could be since binary (base 2) has the most 1's for any given number. Therefore, the loop counter m starts from 63 (since a 64-bit integer has a maximum of 63 1's plus

1 sign bit). The solution works by iterating over possible values of m from this maximum m down to 2. For each m, it performs a binary search to find the smallest k such that the sum of the geometric series equals n. Once we find such k, we return it as the result. If we fail

To speed up the process, the cal(k, m) function is defined to calculate the sum of the geometric series efficiently. This avoids recalculating powers of k multiple times. The binary search is conducted within the range of [2, num - 1] for each m. Whenever the cal(mid, m) function, which

to find such k for all m, then the smallest good base is n-1 (because the only representation of n with all 1's using base n-1 is 11).

represents the sum of the series for base mid and m digits, yields a value less than num (our original number), we know that the base is too small and needs to be larger; thus, we adjust our search range accordingly. The solution, therefore, combines the understanding of geometric series with binary search to find the smallest good base within an optimized time complexity.

the algorithm utilized: Convert the input string n to an integer num to perform numerical operations.

1's. The number 63 is used because for a 64-bit integer, the maximum length of pure 1's (excluding the sign bit) is 63.

Loop through m starting from 63 down to 2. This represents the possible lengths of the number n when written in base k, all in

For each value of m, perform a binary search to find the smallest base k that satisfies the condition that all digits of n base k

The solution involves the implementation of an iterative approach combined with binary search. Let's go through the steps and

are 1's. Set the initial search range with 1 (left) as 2 and r (right) as $\frac{1}{1}$ and $\frac{1}{1}$, indicating the minimum and maximum potential

within the binary search.

(num - 1) to find the smallest k.

6. New mid is $(2 + 3) \gg 1$ which equals 2.

4. Find the new mid, which is now (2 + 6) >> 1 which equals 4.

1. Conduct a binary search for 'k' between '2' and '12'.

Example Walkthrough

Solution Approach

bases, respectively. Conduct the <u>binary search</u>:

 \circ Compute the middle point mid between 1 and r as (1 + r) >> 1, where >> 1 is a bitwise right shift equivalent to division by 2.

 Calculate the sum of the geometric series using cal(mid, m). ■ The cal function takes a base k and a length m, iteratively multiplies the base (geometric progression), and accumulates the result in s,

initializing with 1 (for the first digit, which is always 1).

the right range from mid to r does not contain the smallest base.

 Otherwise, update the left range boundary 1 to mid + 1, as the current base mid is too small to represent num with all 1's. • This process narrows down the search space until the left and right boundaries converge. After the binary search loop concludes, the function checks if the value discovered at 1 produces a sum equal to num using

cal(l, m). If it does, this base l is the smallest good base for the given m, and it is returned as a string.

• Check if the computed series sum is greater than or equal to num. If so, update r to mid, indicating that the current base may be too large, or

If no suitable base k is found across all m values in the loop, which means n cannot be written as all 1's in any other base than itself minus 1, the function returns num - 1 as a string. This corresponds to the base n-1 since any number n in base n-1 is 11.

The solution makes efficient use of binary search within an iterative loop to significantly narrow down the possible candidates for

a good base and arrive at the smallest possible one. It leverages the mathematical properties of geometric series for verification

Starting with m equal to 63 and decreasing, we're looking for the smallest base k such that 13 can be written as a series of 1's this would take too long computationally, though, so for the sake of the example, let's consider smaller m values. We will start with

When m = 3, our equation $n = 1 + k^1 + k^2$ should equal 13. So in this step, we'll perform a binary search between 2 and 12

m = 3, as larger values of m would result in smaller values of k, and we are searching for the smallest k.

5. Calculating cal(4, 3) gives us 1 + 4 + 16 = 21, which is again greater than 13, so we adjust r again to mid - 1, now 3.

3. Since 57 is greater than 13, mid is too large. We adjust our range and set r to mid -1, which is now 6.

7. Calculate cal(2, 3) giving us 1 + 2 + 4 = 7, which is less than 13, so we adjust 1 to mid + 1, now 3.

Let's assume n is given as the string "13". First, we convert this string to an integer num = 13 to perform arithmetic operations.

During each iteration of the binary search, we: 1. Find mid. For the first iteration, 1 is 2, r is 12, so mid will be (2 + 12) >> 1 which equals 7. 2. Calculate cal(mid, m). Using mid = 7, we find that cal(7, 3) = 1 + 7 + 49 = 57.

We find that using k = 3, cal(3, 3) equals 1 + 3 + 9 = 13, which matches our num. Therefore, base k = 3 can represent the

Solution Implementation

num = int(n)

return sum_product

Convert input string to integer

left, right = 2, num - 1

while left < right:</pre>

else:

return str(num - 1)

for term_count in range(63, 1, -1):

right = mid

return str(left)

long power = 1; // Start with k^0

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

return Long.MAX VALUE;

if (Long.MAX_VALUE - sum < power) {</pre>

if (Long.MAX_VALUE / power < base) {</pre>

power = Long.MAX_VALUE;

sum += power; // Add current power of base to sum

// Check if next multiplication would cause overflow

// Function to find the smallest good base of a number as a string

// Calculate the maximum possible value of m, assuming base 2 (binary)

// Initialize multiplier (mul) and sum (s) for geometric progression

// If sum equals to the value, we've found the smallest good base

// since a K-base number system of K+1 (here v) would always be written as 10...0 (which equals K+1).

// Calculate the base k for the current m using nth root

// Calculate the sum of the sequence with m terms

mul *= base; // multiply by base each time

// If no other base found, the smallest good base is value $-\ 1$

// Function to find the smallest good base for a number given as a string

sum += mul; // add the term to the sum

// Convert the input number n to a long integer

// Start iterating from the largest possible m to 1

int maxM = floor(log(value) / log(2));

int base = pow(value, 1.0 / m);

for (int i = 0; i < m; ++i) {

return to_string(base);

// Import the required function from JavaScript Math object

for (int m = maxM; m > 1; --m) {

long mul = 1, sum = 1;

if (sum == value) {

return to_string(value - 1);

import { log10, pow, floor } from 'math';

function smallestGoodBase(n: string): string {

// Convert the input string to a number

power *= base; // Otherwise, multiply power by base

long sum = 0;

} else {

string smallestGoodBase(string n) {

long value = stol(n);

return sum;

left = mid + 1

If no good base is found, return num - 1,

which is always a good base $(n = k^1 + 1)$

Binary search for the good base

mid = (left + right) // 2

if calculate_sum(mid, term_count) >= num:

if calculate_sum(left, term_count) == num:

Check if we found the exact sum that matches the given number

Python

8. Since 1 now equals r, the search concludes.

number 13 as 111 in base 3, and given this is the smallest k we've found, we return k = 3 as the smallest good base for the number 13.

For the given integer `n` of value "13" (which we convert to the integer `num = 13` for processing), we aim to find t We start with m = 3, a possible length of the all 1's representation (i.e., 111):

- First iteration: mid = 7. Calculate cal(7, 3) = 57. This is greater than 13, so set r to 6.

- Second iteration: mid = 4. Calculate cal(4, 3) = 21. Still greater than 13, set r to 3. - Third iteration: mid = 2. Calculate cal(2, 3) = 7. Less than 13, set l to 3. 2. When `l` and `r` converge, we find that `cal(3, 3)` equals `13`. 3. Return base k = 3 as the smallest good base, which represents 13 as 111 in this base.

Try to find the smallest base by iterating from the largest term count down to 2

class Solution: def smallestGoodBase(self, n: str) -> str: # Helper function to calculate the sum of a geometric series def calculate_sum(base, term_count): power_product = sum_product = 1 for i in range(term_count): power_product *= base sum_product += power_product

Java class Solution { // Finds the smallest base for a number with the properties of a good base

```
public String smallestGoodBase(String n) {
    long num = Long.parseLong(n); // Convert string to long integer
   // loop to check all possible lengths starting from the highest possible
    for (int length = 63; length >= 2; --length) {
        long base = getBaseForGivenLength(length, num);
        if (base != -1) {
            return String.valueOf(base); // if a valid base is found, return it
   // If no good base is found, return n-1 as base as per the mathematical property
    return String.valueOf(num - 1);
// Helper method to get a base for a given range and target number
private long getBaseForGivenLength(int length, long targetNumber) {
    long left = 2, right = targetNumber - 1;
    while (left < right) { // Binary search to find the good base</pre>
        long mid = (left + right) >>> 1; // Use unsigned right shift for division by 2
        long result = calculatePowerSum(mid, length);
        if (result >= targetNumber) {
            right = mid; // Adjust right boundary
        } else {
            left = mid + 1; // Adjust left boundary
    return calculatePowerSum(right, length) == targetNumber ? right : −1;
// Helper method to calculate the sum of powers for a given base and length
private long calculatePowerSum(long base, int length) {
```

};

TypeScript

C++

public:

class Solution {

```
let value: number = parseInt(n);
      // Calculate the maximum possible value of m assuming the base is 2 (binary) system
      let maxM: number = floor(log10(value) / log10(2));
      // Iterate from the largest possible value of m to 1
      for (let m = maxM; m > 1; m--) {
          // Calculate the base (k) for the current value of m using the nth root
          let base: number = pow(value, 1.0 / m);
         // Initialize variables for the geometric progression
          let mul: number = 1; // Multiplier
          let sum: number = 1; // Sum of the geometric sequence
         // Calculate the sum of the sequence with m terms
          for (let i = 0; i < m; i++) {
             mul *= base; // Multiply by the base for each term
              sum += mul; // Add the computed term to the sum
         // If the sum is equal to the original number, we've found the smallest good base
          if (sum === value) {
              return base.toString();
      // If no base was found, return value - 1, which is always a good base for any number
      return (value - 1).toString();
  // Please note the usage of 'log10' instead of 'log' in TypeScript.
  // TypeScript uses the built-in JavaScript Math object's log10 method for base 10 logarithms.
class Solution:
   def smallestGoodBase(self, n: str) -> str:
       # Helper function to calculate the sum of a geometric series
       def calculate_sum(base, term_count):
           power_product = sum_product = 1
           for i in range(term_count):
               power_product *= base
               sum_product += power_product
           return sum_product
       # Convert input string to integer
       num = int(n)
       # Try to find the smallest base by iterating from the largest term count down to 2
       for term_count in range(63, 1, -1):
           # Binary search for the good base
```

The time complexity of the algorithm is determined by the nested loop: The outer loop runs for each possible value of m, which ranges from 63 to 2, resulting in a maximum of 62 iterations. This is

Time Complexity

Space Complexity

left, right = 2, num - 1

right = mid

return str(left)

left = mid + 1

If no good base is found, return num - 1,

which is always a good base $(n = k^1 + 1)$

mid = (left + right) // 2

if calculate_sum(mid, term_count) >= num:

if calculate_sum(left, term_count) == num:

Check if we found the exact sum that matches the given number

while left < right:</pre>

else:

return str(num - 1)

Time and Space Complexity

The inner loop is a binary search, which runs in O(log(n)) time, where n is the given number. In each iteration of this binary search, the function cal is called which performs, at maximum, m multiplications.

this context is $2^64 - 1$, the maximum length of m is 63.

Given that m is at most 63, and for each m we perform a binary search which takes O(log(n)) time, the overall time complexity of the inner loop is 0(m * log(num)).

3. The cal function itself runs in O(m) time, since it contains a loop that iterates m times.

Combining these aspects together, the total time complexity of the code is $0(m * m * \log(num))$ or, more concretely, 0(63 *log(num)) because m is a constant at most 63.

because m represents the maximum length of digits in base k representation for the number n, and since the largest number in

- The space complexity of the algorithm is 0(1): The space used by the algorithm is constant, as there are only a few integer variables being used and no additional space (like data structures) that grow with the size of the input.
- The cal function uses a constant amount of space as well, as the variables p and s are just integers and do not require space that scales with the input size. Hence, there is no significant space usage that scales with the size of the input.