374. Guess Number Higher or Lower

Binary Search Interactive **Easy**

Problem Description

goal is to find the number selected by the game with the help of a clue provided after each guess. When you make a guess by choosing a number within the range, the game responds by indicating if your guess is higher, lower, or equal to the secret number it has chosen. The responses are given through a pre-defined guess function with the following possible return values:

In the Guess Game, you have to guess a number chosen by the game from a range of numbers starting from 1 to n. The game's

If your guess is too high, the function returns −1.

If your guess is correct, meaning it matches the secret number, the function returns 0.

If your guess is too low, the function returns 1.

- Your task is to use this guess function to determine and return the secret number the game has picked.

Intuition

The straightforward approach to solving this problem would involve using a binary search method. Since the potential numbers

narrowing it down until we find the exact number. However, this solution uses a specialized Python library called bisect that is capable of performing binary search operations efficiently. Here's the intuition behind the approach in the solution provided: • bisect.bisect: This is a function from the bisect module which is used to find a position in a list where an element should be inserted to keep

range from 1 to n, we can start by guessing in the middle of this range. Depending on the feedback from the guess function, we

would know if the target number is higher or lower than our current guess, and we would adjust our search range accordingly,

the list sorted. • Since we know the guess function returns -1 if our guess is high and 1 if our guess is low, we can use this information as a key function for the bisect method. By negating the guess function in the key, we essentially transform the guess function's return value to use it for bisect.

- The range function generates a sequence of numbers, which represents our possible guesses. • By using bisect with the negated guess as a key, it will effectively perform a binary search to find the correct position, i.e., where the return value of the guess function would be 0, which means the guess matches the picked number.
- In doing so, the solution code bypasses the more common iterative or recursive implementations of binary search and instead

uses the bisect library to find the solution, showing yet another powerful aspect of Python's standard library.

(or just bisect as an alias for bisect_right), which are used for binary search.

Solution Approach The implementation of the provided solution uses the bisect module from Python, which is designed to make binary searches

simple and efficient. Here's the breakdown of the solution approach: Leveraging Python's Standard Library: The bisect module contains two main functions — bisect_left and bisect_right

represents the entire list of potential guesses.

narrowing in on the correct number with each iteration.

leveraging the **bisect** module to find this secret number.

middle point is 6.5, so our next guess will be 6.

guess function returns 1 since our guess is lower than the secret number.

manually implemented binary search algorithm.

Example Walkthrough

chosen by the game.

@param num, your quess

import bisect

Java

/**

*/

/**

class Solution:

def guess(num: int) -> int:

The quess API is already defined for you.

def guessNumber(self, n: int) -> int:

def guess compare(x):

public class Solution extends GuessGame {

otherwise return 0

@return -1 if num is higher than the picked number

1 if num is lower than the picked number

Use binary search to find the guessed number.

Define the bisect function to compare guess results.

Negative result if our guess is higher (-1),

where the guess result is 0, hence bisect_left is more appropriate.

// Initialize the upper bound of the search range.

int mid = left + (right - left) / 2;

int apiResponse = guess(mid);

if (apiResponse <= 0) {</pre>

return bisect.bisect_left(range(1, n + 1), 0, key=guess_compare)

Note: The original implementation used bisect.bisect, which by default assumes

* The guessNumber method aims to find the number which the API is thinking of.

an insertion point that would maintain order. In this case, we want the exact position

* This class is an extension of the GuessGame class which contains a guess API method.

* Guesses the number by using binary search to minimize the number of calls to the guess API.

// Continue searching as long as the range has not been narrowed down to a single element.

// If the quess is too high or correct, narrow the range to the lower half (inclusive of mid).

// Calculate the midpoint of the current range to use as our guess.

// Make a guess using the midpoint and get the response from the guess API.

// At this point, lowerBound == upperBound, and we have found the secret number

Return the result of the guess API call:

Understanding the bisect function: The bisect bisect function is used to find the insertion point for a given element in a sorted list to maintain the sorted order. This means that it can be used to find where in the list a new element should go such that the list will still be in order.

- Applying bisect with a key function: In this problem, the bisect function is given a key argument, which is a function that will be called on each element in the array before making comparisons during the binary search. Here, the key function is a lambda function that returns the negated result of the guess function: lambda x: -guess(x). The negation is useful because
- guess returns 0, when we negate 0, it remains 0, which bisect would interpret as the correct insertion point or, in our case, the correctly guessed number. Using the range: The range function, range(1, n + 1), creates a sequence of numbers starting from 1 to n. This range

bisect is typically used for finding insertion points and expects the condition to be sorted in ascending order. Since a correct

Finding the correct guess: The bisect bisect function is called with the range and key function defined above. It will

effectively perform a binary search over this range, using the key function to compare the guess to the secret number until the insertion point (where the guess is correct) is located. **Returning the correct number**: The function will return the position where the guess(x) is 0, which is the selected number. The algorithm used in this solution has an average-case and worst-case time complexity of O(log n) because it is a binary search

algorithm. It works by essentially cutting down the search space by half with each incorrect guess that it makes, thereby

Overall, this is an elegant approach that takes full advantage of Python's capabilities, providing an alternative to the more

Let's assume the secret number the game has picked is 6 and our range n is 10. We will use the provided solution approach

Let's illustrate the binary search mechanism step by step. First Iteration: The middle of our range 1 to 10 is 5.5 (we would round this to 6 in implementation, but let's follow the strict binary process). We guess 5 as our first attempt (since we typically use integer division which truncates the decimal). The

Setting Up the Problem: The game has chosen a secret number from 1 to 10, and our guess should be within this range.

Adjusting the Range: With the feedback that our guess was too low, we adjust our range from 6 to 10. The new middle of this range is 8. **Second Iteration**: We now guess 8, and the guess function returns -1 indicating our guess is higher than the secret number.

Narrowing Down Further: The secret number is, therefore, between 6 and 7 (since we've ruled out 8 and above). The new

Third Iteration and Correct Guess: Guessing 6, the guess function returns 0, signalling that we've found the correct number.

Throughout this process, if we were using bisect.bisect with the negated guess function as the key, it would have negated the

return values of the guess function and performed these steps internally. We wouldn't actually see the returned values of 1 or

-1; bisect would use them to perform the binary search: • At the first iteration, with the middle element as 5 and the guess function returning 1 (guess too low), negating it gives us -1, which guides the search to go higher.

• At the second iteration, with the middle element as 8 and the guess function returning -1 (guess too high), negating it gives us 1, which

guides the search to go lower. • Finally, at the correct guess 6, the guess function returns 0, and negated or not, it remains 0, indicating the correct position has been found. So with the bisect bisect function, the correct number 6 would be returned as the position where the guess(x) equates to 0.

This example walkthrough illustrates how the binary search and, by extension, the bisect function would find the secret number

Solution Implementation Python

Positive if our guess is lower (1), # Zero if our quess is correct (0) return -guess(x) # Use the bisect function to find the correct number. # We make use of the `key` parameter to provide our custom comparison # function which uses the guess API as the comparison logic.

```
* @param n the upper bound of the range to search.
* @return
             the number that matches the guess API's hidden number.
public int quessNumber(int n) {
   // Initialize the lower bound of the search range.
```

int left = 1;

int right = n;

while (left < right) {</pre>

return lowerBound;

// Forward declaration of the guess API.

// @return 0 if num is the guess number

declare function guess(num: number): number;

function guessNumber(n: number): number {

// Initialize the search range

let left = 1;

let right = n;

// @return -1 if num is lower than the guess number

// @return 1 if num is higher than the guess number

* @return {number} - The correct guessed number.

* Finds the number guessed by the guess API within the range [1, n].

* @param $\{number\}$ n - The maximum range within which the guessed number falls.

// @param {number} num - Your guess

```
right = mid;
            } else {
                // If the guess is too low, narrow the range to the upper half (exclusive of mid).
                left = mid + 1;
        // When the loop exits, left and right converge to the target number; return it.
        return left;
C++
/**
 st The guessNumber function tries to find a secret number within a range between 1 and n.
 * It relies on a pre-defined API "guess" which takes an integer as an input (your guess) and returns
 * -1 if the guess is lower than the secret number, 1 if it is higher, and 0 when your guess matches the secret number.
 * @param n An integer representing the upper bound of the range within which to guess the number.
 * @return The secret number.
class Solution {
public:
    int quessNumber(int n) {
        int lowerBound = 1; // Initialize lower bound of the search interval
        int upperBound = n; // Initialize upper bound of the search interval
        // Perform a binary search in the interval from lowerBound to upperBound
        while (lowerBound < upperBound) {</pre>
            // Calculate the middle value in the current interval
            int mid = lowerBound + ((upperBound - lowerBound) >> 1); // Equivalent to (lowerBound + upperBound) / 2 but avoids overfl
            // Use the guess API method to compare the middle value with the secret number
            if (quess(mid) <= 0) {
                // If the guess is greater than or equal to the secret number, narrow the search to the lower half
                upperBound = mid;
            } else {
                // If the guess is less than the secret number, narrow the search to the upper half
                lowerBound = mid + 1;
```

```
// Use binary search to find the guessed number
```

};

/**

*/

TypeScript

```
while (left < right) {</pre>
        // Calculate the midpoint to minimize the search range
        // Using right shift by 1 to ensure no overflow happens
        const mid = left + ((right - left) >>> 1);
        // Guess the middle of the range and reduce the range based on the response
        const result = quess(mid);
        if (result <= 0) {
            // If the guessed number is less than or equal to mid, narrow the range to the left half
            right = mid;
        } else {
            // If the quessed number is greater than mid, narrow the range to the right half
            left = mid + 1;
    // At the end of the loop, left should be equal to the guessed number
    return left;
# The guess API is already defined for you.
# @param num, your quess
# @return -1 if num is higher than the picked number
          1 if num is lower than the picked number
           otherwise return 0
# def guess(num: int) -> int:
import bisect
class Solution:
    def guessNumber(self. n: int) -> int:
        # Use binary search to find the guessed number.
        # Define the bisect function to compare guess results.
        def guess compare(x):
            # Return the result of the guess API call:
            # Negative result if our guess is higher (-1),
            # Positive if our guess is lower (1),
            # Zero if our guess is correct (0)
            return -guess(x)
```

Time and Space Complexity

Use the bisect function to find the correct number.

where the guess result is 0, hence bisect_left is more appropriate.

function which uses the guess API as the comparison logic.

return bisect.bisect_left(range(1, n + 1), 0, key=guess_compare)

Note: The original implementation used bisect.bisect, which by default assumes

We make use of the `kev` parameter to provide our custom comparison

an insertion point that would maintain order. In this case, we want the exact position

The time complexity of the provided code is $O(\log n)$. This is because the bisect function employed here uses a binary search algorithm to find the insertion point for 0 in the sorted sequence. It repeatedly divides the sequence in half to locate the position, which results in a logarithmic number of steps with respect to the sequence's length.

The space complexity of the code is 0(1) since it uses a constant amount of extra space. The key function in the bisect function does not store any additional data structures related to the size of the input n, and the rest of the operations do not require more space that grows with the input size.