**Solution Article**

**Approach 1: Brute Force**

We check every number from 1 to n-1 and pass it to the guessguess*guess* function. The number  
for which a 0 is returned is the required answer.

/\* The guess API is defined in the parent class GuessGame.

   @param num, your guess

   @return -1 if my number is lower, 1 if my number is higher, otherwise return 0

      int guess(int num); \*/

public class Solution extends GuessGame {

    public int guessNumber(int n) {

        for (int i = 1; i < n; i++)

            if (guess(i) == 0)

                return i;

        return n;

    }

}

**Complexity Analysis**

* Time complexity : O(n)O(n)*O*(*n*). We scan all the numbers from 1 to n.
* Space complexity : O(1)O(1)*O*(1). No extra space is used.

**Approach 2: Using Binary Search**

**Algorithm**

We can apply Binary Search to find the given number. We start with the mid  
number. We pass that number to the guessguess*guess* function. If it returns a -1, it implies that the guessed number is larger than the required one. Thus, we use Binary Search for numbers lower than itself. Similarly, if it returns a 1, we use Binary Search  
for numbers higher than itself.

/\* The guess API is defined in the parent class GuessGame.

   @param num, your guess

   @return -1 if my number is lower, 1 if my number is higher, otherwise return 0

      int guess(int num); \*/

public class Solution extends GuessGame {

    public int guessNumber(int n) {

        int low = 1;

        int high = n;

        while (low <= high) {

            int mid = low + (high - low) / 2;

            int res = guess(mid);

            if (res == 0)

                return mid;

            else if (res < 0)

                high = mid - 1;

            else

                low = mid + 1;

        }

        return -1;

    }

}

**Complexity Analysis**

* Time complexity : O(log⁡2n)O\big(\log\_2 n\big)*O*(log2​*n*). Binary Search is used.
* Space complexity : O(1)O(1)*O*(1). No extra space is used.

**Approach 3: Ternary Search**

**Algorithm**

In Binary Search, we choose the middle element as the pivot in splitting. In Ternary Search, we choose two pivots (say m1m1*m*1 and m2m2*m*2) such that the given range is divided into three equal parts. If the required number numnum*num* is less than m1m1*m*1 then we apply ternary search on the left segment of m1m1*m*1. If numnum*num* lies between m1m1*m*1 and m2m2*m*2, we apply ternary search between m1m1*m*1 and m2m2*m*2. Otherwise we will search in the segment right to m2m2*m*2.

/\* The guess API is defined in the parent class GuessGame.

   @param num, your guess

   @return -1 if my number is lower, 1 if my number is higher, otherwise return 0

      int guess(int num); \*/

public class Solution extends GuessGame {

    public int guessNumber(int n) {

        int low = 1;

        int high = n;

        while (low <= high) {

            int mid1 = low + (high - low) / 3;

            int mid2 = high - (high - low) / 3;

            int res1 = guess(mid1);

            int res2 = guess(mid2);

            if (res1 == 0)

                return mid1;

            if (res2 == 0)

                return mid2;

            else if (res1 < 0)

                high = mid1 - 1;

            else if (res2 > 0)

                low = mid2 + 1;

            else {

                low = mid1 + 1;

                high = mid2 - 1;

            }

        }

        return -1;

    }

}

**Complexity Analysis**

* Time complexity : O(log⁡3n)O\big(\log\_3 n \big)*O*(log3​*n*). Ternary Search is used.
* Space complexity : O(1)O(1)*O*(1). No extra space is used.

**Follow up**

It seems that ternary search is able to terminate earlier compared to binary search. But why is binary search more widely used?

**Comparisons between Binary Search and Ternary Search**

Ternary Search is worse than Binary Search. The following outlines the recursive formula to count comparisons of Binary Search in the worst case.

T(n)=T(n2 )+2,T(1)=1T(n2 )=T(n22 )+2∴T(n)=T(n22 )+2×2=T(n23 )+3×2=…=T(n2log⁡2n )+2log⁡2n=T(1)+2log⁡2n=1+2log⁡2n\begin{aligned} T(n) &= T\bigg(\frac{n}{2} \ \bigg) + 2, \quad T(1) = 1 \\ T\bigg(\frac{n}{2} \ \bigg) &= T\bigg(\frac{n}{2^2} \ \bigg) + 2 \\ \\ \therefore{} \quad T(n) &= T\bigg(\frac{n}{2^2} \ \bigg) + 2 \times 2 \\ &= T\bigg(\frac{n}{2^3} \ \bigg) + 3 \times 2 \\ &= \ldots \\ &= T\bigg(\frac{n}{2^{\log\_2 n}} \ \bigg) + 2 \log\_2 n \\ &= T(1) + 2 \log\_2 n \\ &= 1 + 2 \log\_2 n \end{aligned}*T*(*n*)*T*(2*n*​ )∴*T*(*n*)​=*T*(2*n*​ )+2,*T*(1)=1=*T*(22*n*​ )+2=*T*(22*n*​ )+2×2=*T*(23*n*​ )+3×2=…=*T*(2log2​*nn*​ )+2log2​*n*=*T*(1)+2log2​*n*=1+2log2​*n*​

The following outlines the recursive formula to count comparisons of Ternary Search in the worst case.

T(n)=T(n3 )+4,T(1)=1T(n3 )=T(n32 )+4∴T(n)=T(n32 )+2×4=T(n33 )+3×4=…=T(n3log⁡3n )+4log⁡3n=T(1)+4log⁡3n=1+4log⁡3n\begin{aligned} T(n) &= T\bigg(\frac{n}{3} \ \bigg) + 4, \quad T(1) = 1 \\ T\bigg(\frac{n}{3} \ \bigg) &= T\bigg(\frac{n}{3^2} \ \bigg) + 4 \\ \\ \therefore{} \quad T(n) &= T\bigg(\frac{n}{3^2} \ \bigg) + 2 \times 4 \\ &= T\bigg(\frac{n}{3^3} \ \bigg) + 3 \times 4 \\ &= \ldots \\ &= T\bigg(\frac{n}{3^{\log\_3 n}} \ \bigg) + 4 \log\_3 n \\ &= T(1) + 4 \log\_3 n \\ &= 1 + 4 \log\_3 n \end{aligned}*T*(*n*)*T*(3*n*​ )∴*T*(*n*)​=*T*(3*n*​ )+4,*T*(1)=1=*T*(32*n*​ )+4=*T*(32*n*​ )+2×4=*T*(33*n*​ )+3×4=…=*T*(3log3​*nn*​ )+4log3​*n*=*T*(1)+4log3​*n*=1+4log3​*n*​

As shown above, the total comparisons in the worst case for ternary and binary search are 1+4log⁡3n1 + 4 \log\_3 n1+4log3​*n* and 1+2log⁡2n1 + 2 \log\_2 n1+2log2​*n* comparisons respectively. To determine which is larger, we can just look at the expression 2log⁡3n2 \log\_3 n2log3​*n* and log⁡2n\log\_2 nlog2​*n* . The expression 2log⁡3n2 \log\_3 n2log3​*n* can be written as 2log⁡23×log⁡2n\frac{2}{\log\_2 3} \times \log\_2 nlog2​32​×log2​*n* . Since the value of 2log⁡23\frac{2}{\log\_2 3}log2​32​ is greater than one, Ternary Search does more comparisons than Binary Search in the worst case.