Chapter 6

Recursion

Recursion

- An **algorithm** is a sequence of steps or procedures for solving a specific problem

- A **recursive algorithm** is an algorithm that solves a problem by relying on repetitions of the same algorithm.

- Example: Function *Race()*

  1. If you are at the finish line, Stop

  2. Take one step forward

  3. *Race()*

Parts of Recursion

- The first part of a recursive algorithm is the **base case**

- A base case defines some procedure or stopping point that the recursive algorithm is working towards

Function *Race()*

1. If you are at the finish line, Stop

2. Take one step forward

3. *Race()*

Parts of Recursion

- The first part of a recursive algorithm is the **base case**

- A base case defines some procedure or stopping point that the recursive algorithm is working towards

Function *Race()*

1. If you are at the finish line, Stop **Base Case**

2. Take one step forward

3. *Race()*

Parts of Recursion

- Second, we work toward the base case
- Here we reduce the problem into a smaller size

Function *Race()*

1. If you are at the finish line, Stop
2. Take one step forward
3. *Race()*

Work Toward Base Case

Parts of Recursion

- Lastly we call the recursive method again

- Usually we perform the recursive function on a smaller set of the problem determined by the function.

Function *Race()*

1. If you are at the finish line, Stop

2. Take one step forward

3. *Race()*

Work Toward Base Case

Recursive Functions

- A function that calls itself in C++ is known as a **recursive function**

- *Example*: Create a function that counts down from a number N and then prints GO once it reaches 0.

example: simple_recursion.cpp
char_recursion.cpp

Creating a Recursive Function

- **Write the base case:** Every Recursive Function *must* have a base case in order to finish execution

- **Write the recursive case:** Contains the logic that recursively calls itself.

```
int recursiveFunction(/* params */) {

  if (/* base case */) {
    return value;
  }
  /* recursive case */
  else {
    // call function again
  }

}
```

Exercise

Write a recursive function that reads in a number and outputs the number factorial.

Ex: *5! = 5 * 4 * 3 * 2 * 1 = 120*

```
int factorial(int n) {
   // define base case

   // define recursive case
}
```

Exercise

Write a recursive function that reads in a number and outputs the number factorial.

Ex: *5! = 5 * 4 * 3 * 2 * 1 = 120*

We can represent N! in a more concise definition:

$$N! = N * (N - 1)!$$

$$5! = 5 * 4!$$

answer: factorial.cpp

Greatest Common Divisor

Greatest common divisor is the largest number that divides evenly into two numbers.

Euclidean algorithm: Subtract smaller number from larger number until they are equal. That yields the *GCD*

Example:

$$GCD(8, 12) = GCD(8, 12 - 8) = GCD(8,4)$$

$$GCD(8, 4) = GCD(8 - 4, 4) = GCD(4, 4).$$

$$GCD(4,4) = 4 == 4 \rightarrow \mathbf{4}$$

Greatest Common Divisor

GCD(num1, num2)

Base Case:

num1 == num2

Recursive Case:

If num1 > num2: GCD(num1 - num2, num2)

If num2 > num1: GCD(num2 - num1, num1)

example: gcd.cpp

Greatest Common Divisor

We can also use the **Modulo Euclidean algorithm** for finding the Greatest Common Divisor.

This uses the modulo operator to find the greatest common divisor.

Full algorithm here: https://www.geeksforgeeks.org/euclidean-algorithms-basic-and-extended/

Greatest Common Divisor (Modulo)

General formula for GCD *modulo:*

$$GCD(a, b) = GCD(b \% a, a)$$

Once *a* equals 0 we know that the GCD is b.

Example:

$$GCD(10,8) = GCD(8, 10 \% 8)$$

$$GCD(8, 2) = GCD(2, 8 \% 2)$$

$$GCD(2, 0) = 2$$

example: gcd_modulo.cpp

Recursion - Fan Out

Recursion can also be used to *fan out* operations:

```
int recursion(int val1, int val2) {
   ...
   int total = recursion(val1, val2) + recursion(val1, val2);
}
```

```
recursion(val1, val2)   recursion(val1, val2);
```

Fibonacci Sequence

The Fibonacci sequence is a mathematical number set where each number is the sum of the two preceding ones.

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …

$$F_n = F_{n-1} + F_{n-2}$$

Fibonacci Sequence

```
int fib(int num) {
   if (num <= 1)
      return num;
   return fib(num-1) + fib(num-2);
}
```

example: fibonacci.cpp

Types Of Recursion

Direct Recursion: When a function contains a call to itself within its own function body.

Example: *factorial.cpp*

Indirect Recursion: When a function calls a second function which in turn calls the first function again.

Indirect Recursion

```
void g() {
   f(); // indirect recursive call
}
void f() {
   g();
}
int main() {
   f();
}
```

example: indirect_recursion.cpp

Iteration vs Recursion

- Both Iteration and recursion are **based on a control statement:**
- Both involve iteration / repetition
  - Iteration uses an ***iteration statement***
  - Recursion uses repeated ***function calls*** and utilizes the stack
- Iteration and recursion both require a termination test.
  - Iteration - uses conditions
  - Recursion - uses the **base case**

Which is better?

Short Answer: *It depends*

1. Recursion is generally less performant than iteration (Using the stack involves a lot of overhead)

2. However Recursion can often times take less time to code and represent the algorithm better

Iteration vs Recursion

Iterative approach: *Palindrome*

```
bool palindrome(char word[], int lowerBound, int upperBound) {
  bool pflag = true;

  while(lowerBound < upperBound && pflag) {
    if (word[lowerBound] != word[upperBound]) {
      pflag = false;
    } else {
      lowerBound++;
      upperBound--;
    }
  }

  return pflag;
}
```

example: palindrome_iter.cpp

Iteration vs Recursion

Recursive approach: *Palindrome*

```
bool pdrome(char word[], int lowerBound, int upperBound) {
  if (lowerBound >= upperBound) {
    return true;
  } else if (word[lowerBound] != word[upperBound]) {
    return false;
  } else {
    return pdrome(word, ++lowerBound, --upperBound);
  }
}
```

example: palindrome_recursive.cpp

Guessing Game

- Think of a number between 1 and 100 and create a program that repeatedly guesses the number until the correct number has been guessed.

- This can also be thought of as a search algorithm.

Number = 45

1

Guess 1
Guess 2
Guess 3
.
.
.

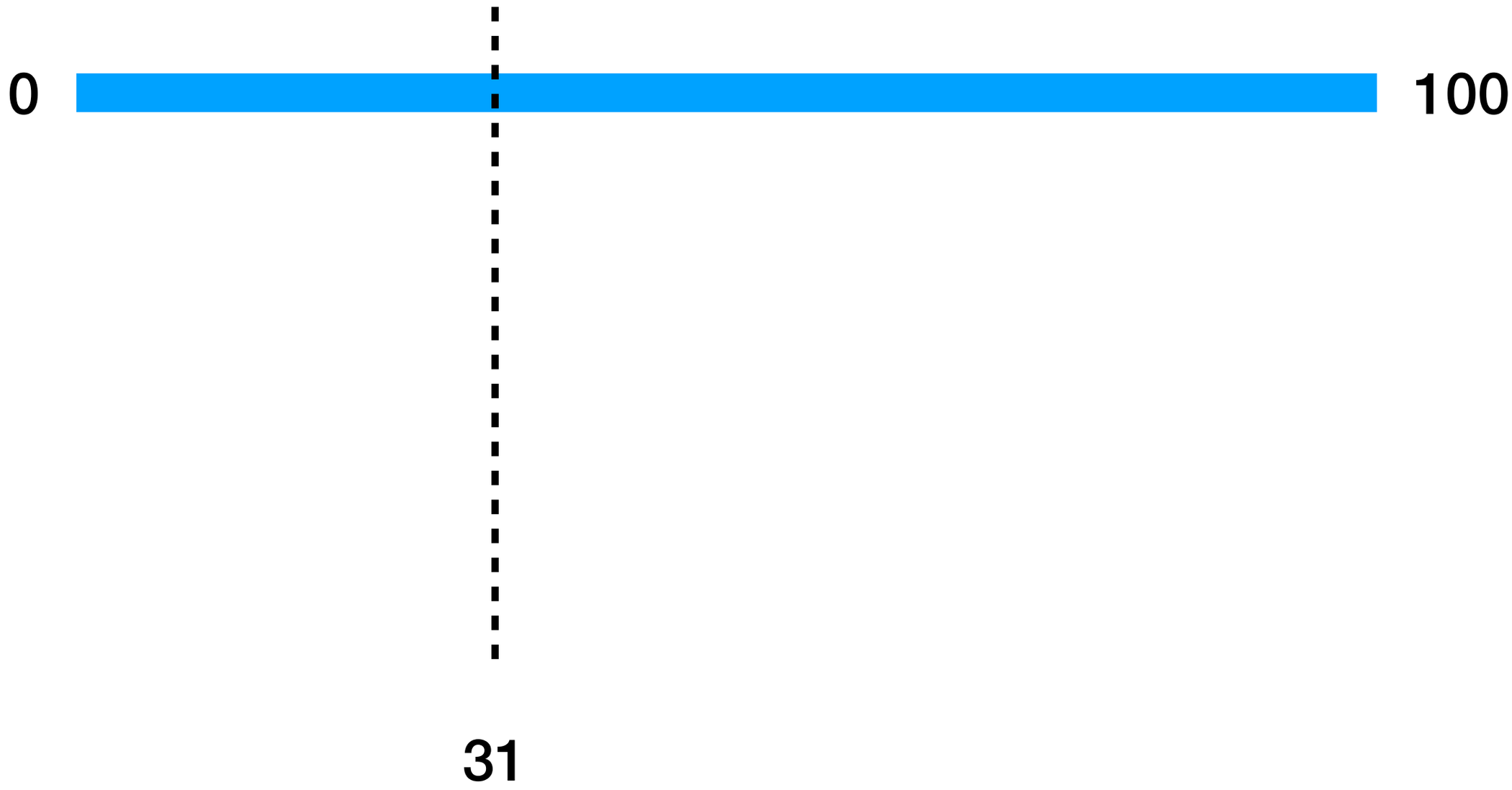
2

Guess 10
Guess 20
Guess 30
.
.
.

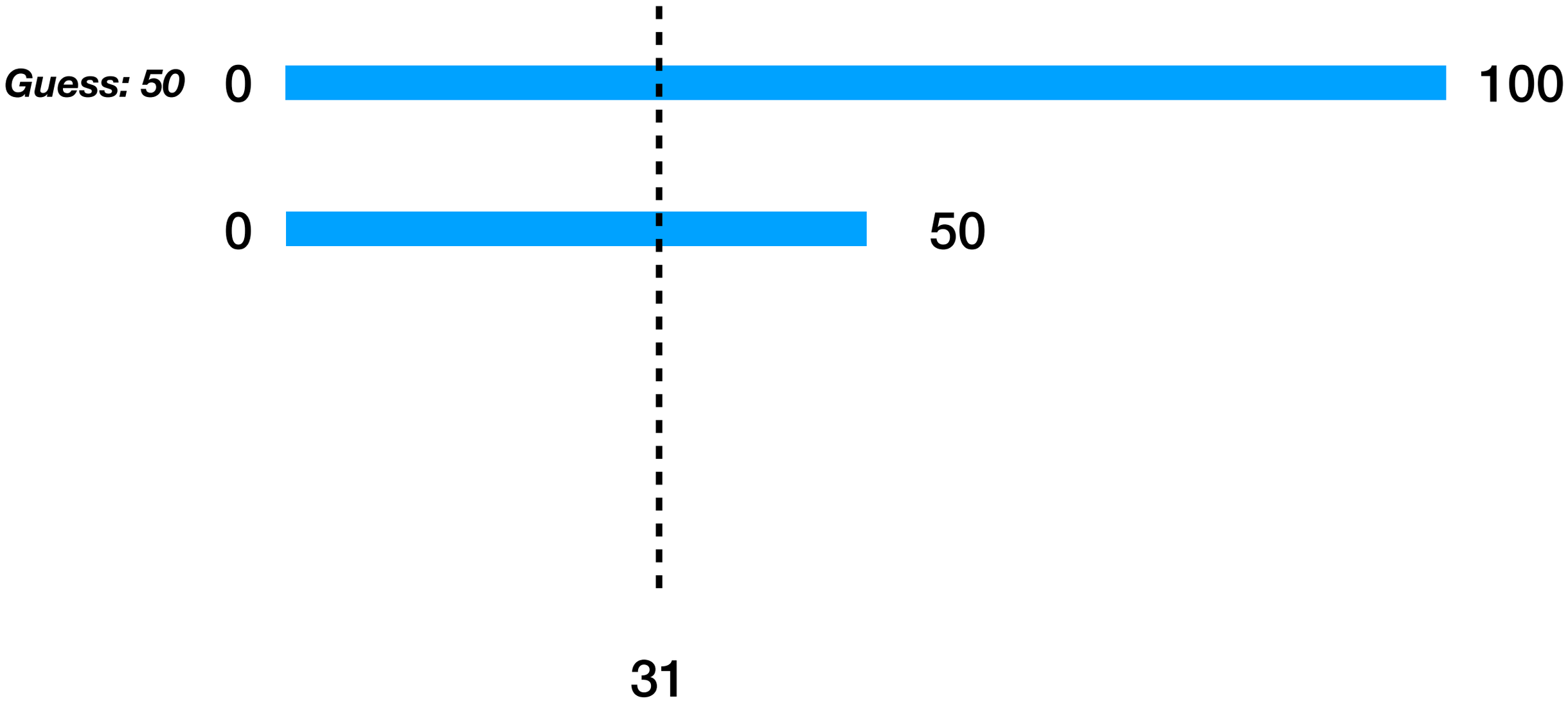
3

Guess 50
Guess 25
Guess 37
.
.
.

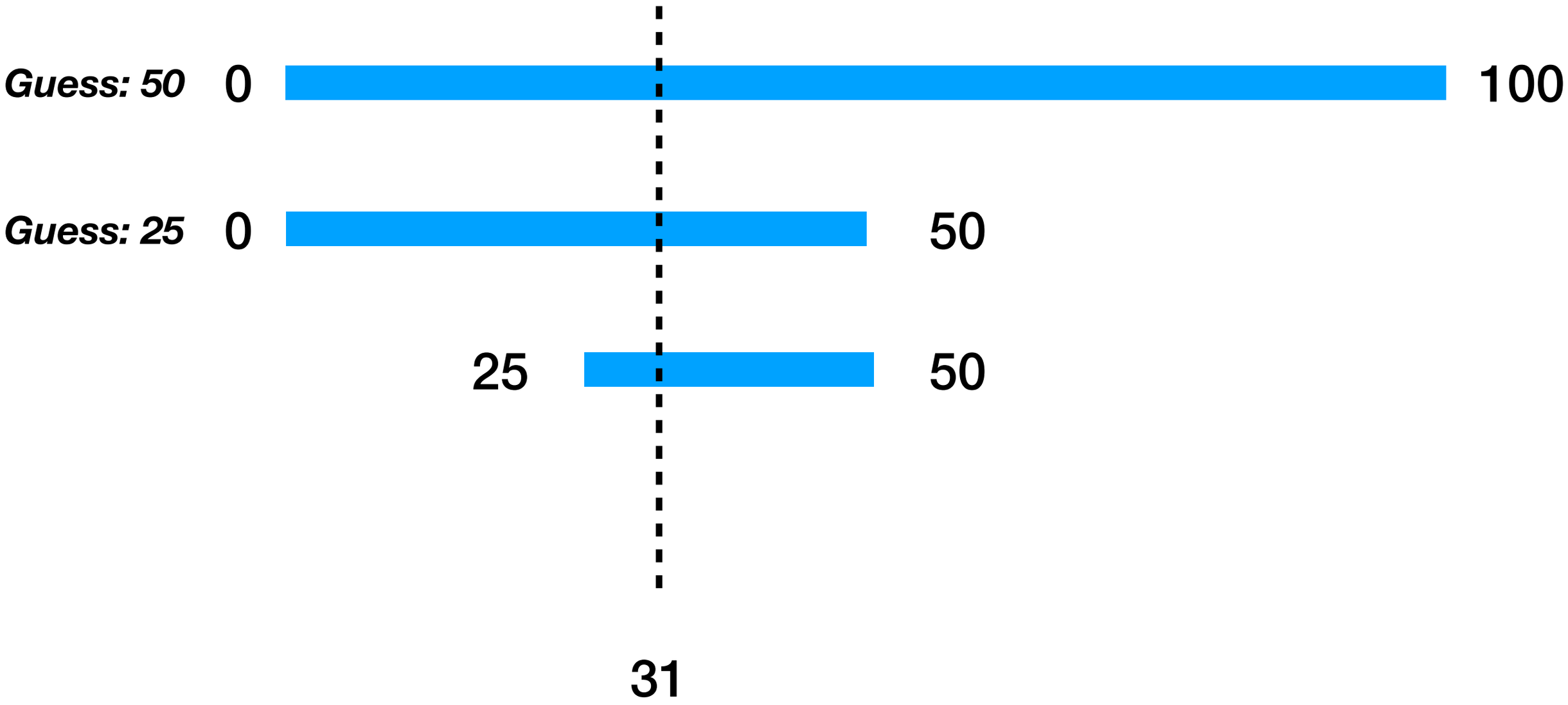
Guessing Game



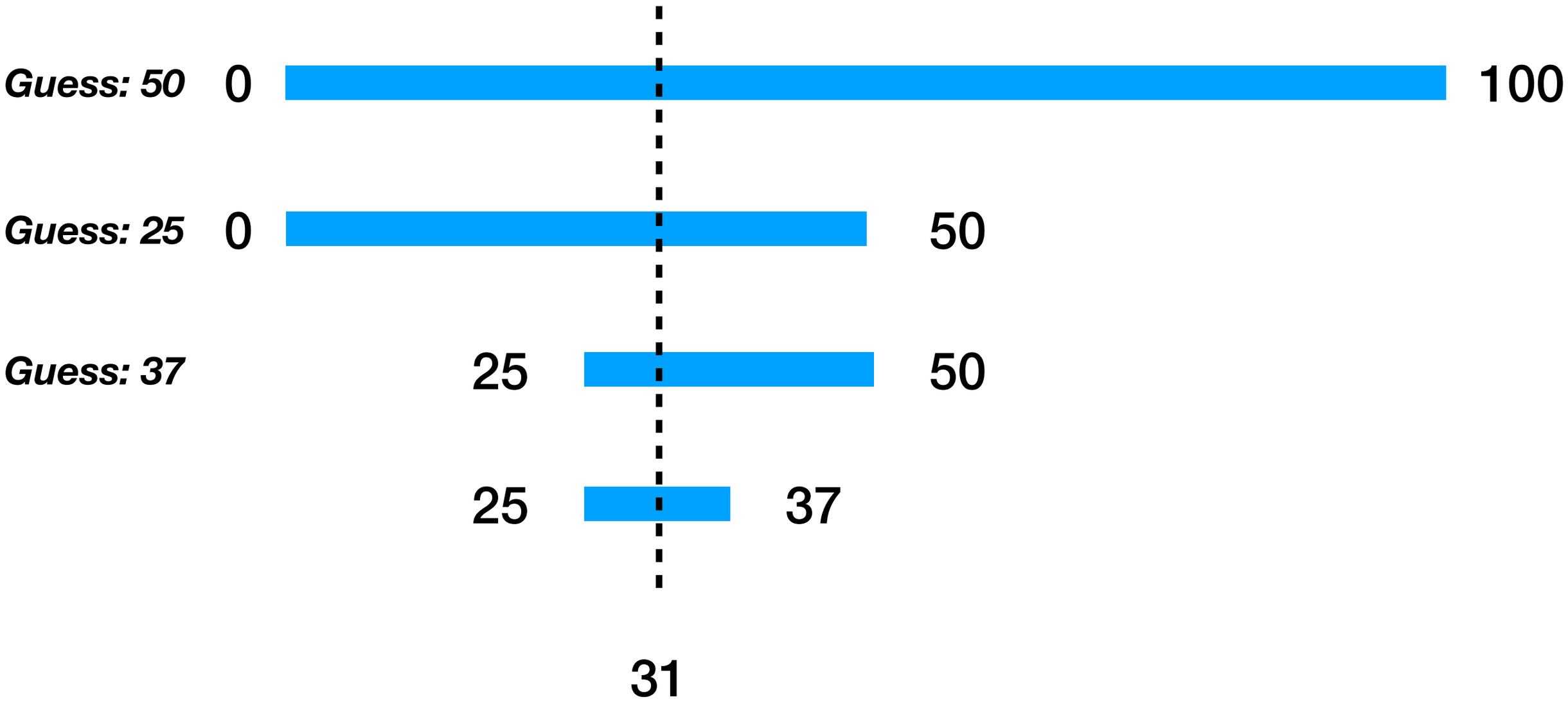
Guessing Game



Guessing Game

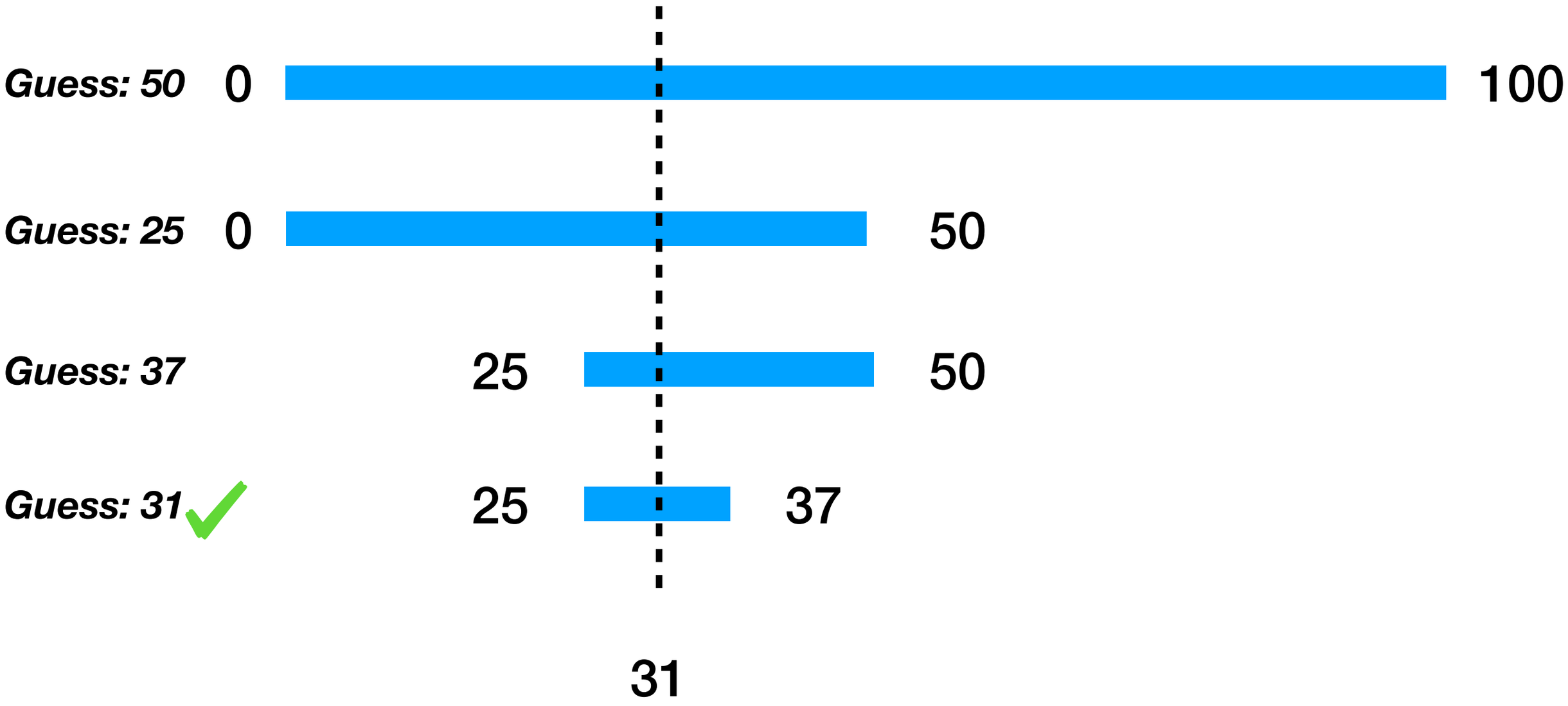


Guessing Game



example: binary_search.cpp

Guessing Game



Guessing Game

```
void Guess(int lowVal, int highVal) {
  int midVal = (highVal + lowVal) / 2;
  char response;

  cout << "Is your number " << midVal << "? (h/l/y)" << endl;
  cin >> response;

  if (response == 'y') {
    cout << "Yay!" << endl;
  } else if (response == 'h') {
    Guess(midVal, highVal);
  } else {
    Guess(lowVal, midVal);
  }
}
```

example: guessing_game.cpp

Binary Search

- This is more commonly known as the **binary search** algorithm

- This will find a number in an ordered list in at most *log(n)* iterations, where *n* is the number of items in the list

- This is more preferable to a linear search where we just check each item in the array one by one, performing at most *n* iterations.

log(n) < n

Binary Search (Iterative)

```
int binarySearch(int numbers[], int lb, int ub, int value) {
  int half = 0;
  bool found = false;
  while (lb <= ub && !found) {
    half = (lb + ub) / 2;
    if (numbers[half] == value) {
      found = true;
    } else if (numbers[half] > value) {
      ub = half - 1;
    } else {
      lb = half + 1;
    }
  }

  return (found) ? half : -1;
}
```

example: binary_search_iterative.cpp

Binary Search (Recursive)

```
int binarySearch(int numbers[], int lb, int ub, int value) {
  int half;
  if (lb > ub) {
    return -1;
  }
  half = (lb + ub) / 2;
  if (numbers[half] == value) {
    return half;
  } else if (numbers[half] > value) {
    return binarySearch(numbers, lb, half - 1, value);
  } else {
    return binarySearch(numbers, half + 1, ub, value);
  }
}
```

example: binary_search_iterative.cpp

Stack Overflow

- **Stack Overflow:** Deep recursion could fill the stack region and cause a stack overflow, meaning a stack frame extends beyond the memory region allocated for stack

```
int overflow(int value) {
  return overflow(value + 1);
}
```

- Keep in mind that even recursive algorithms that do terminate eventually may cause stack overflow depending on the amount of calls needed