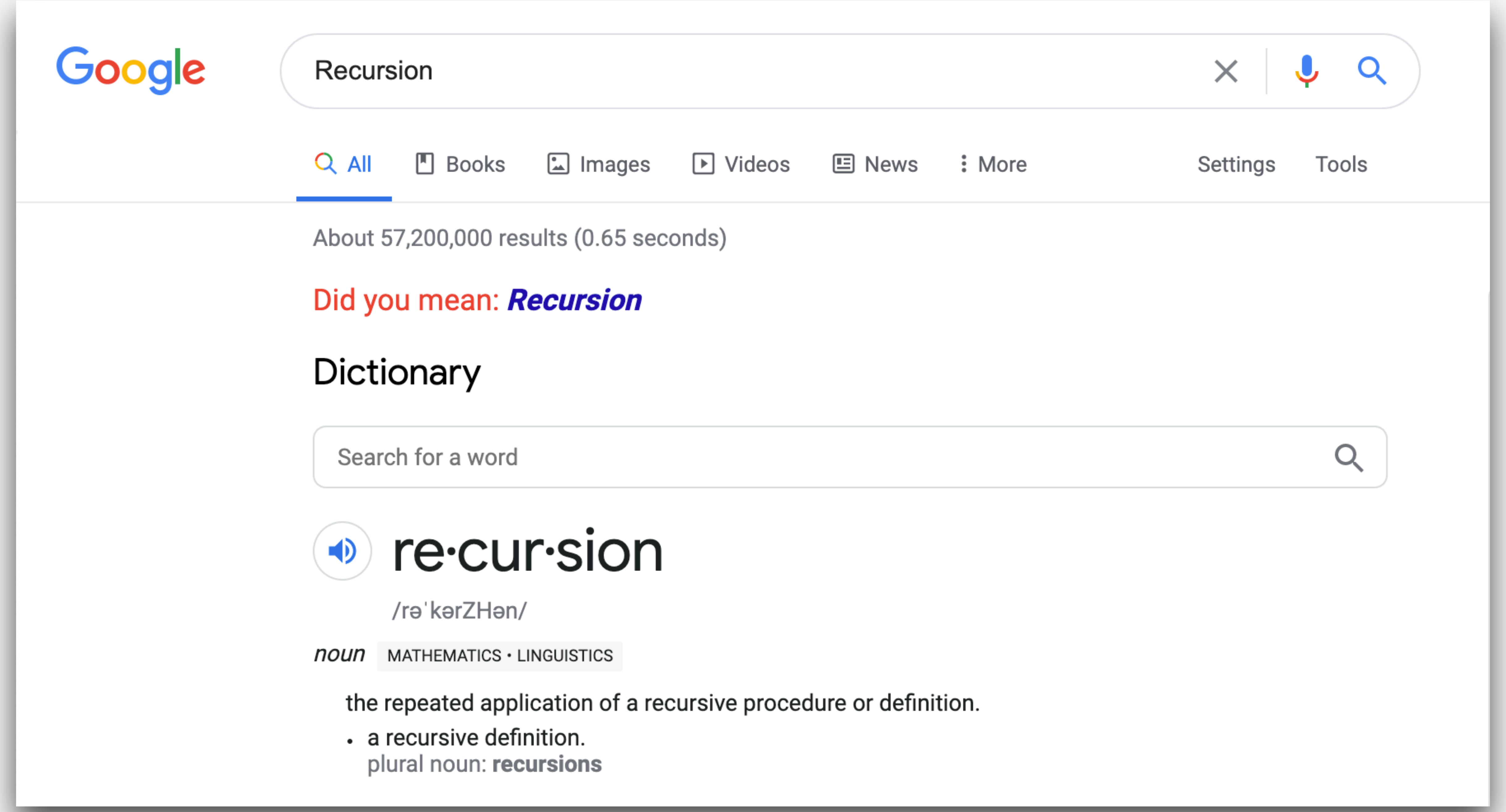
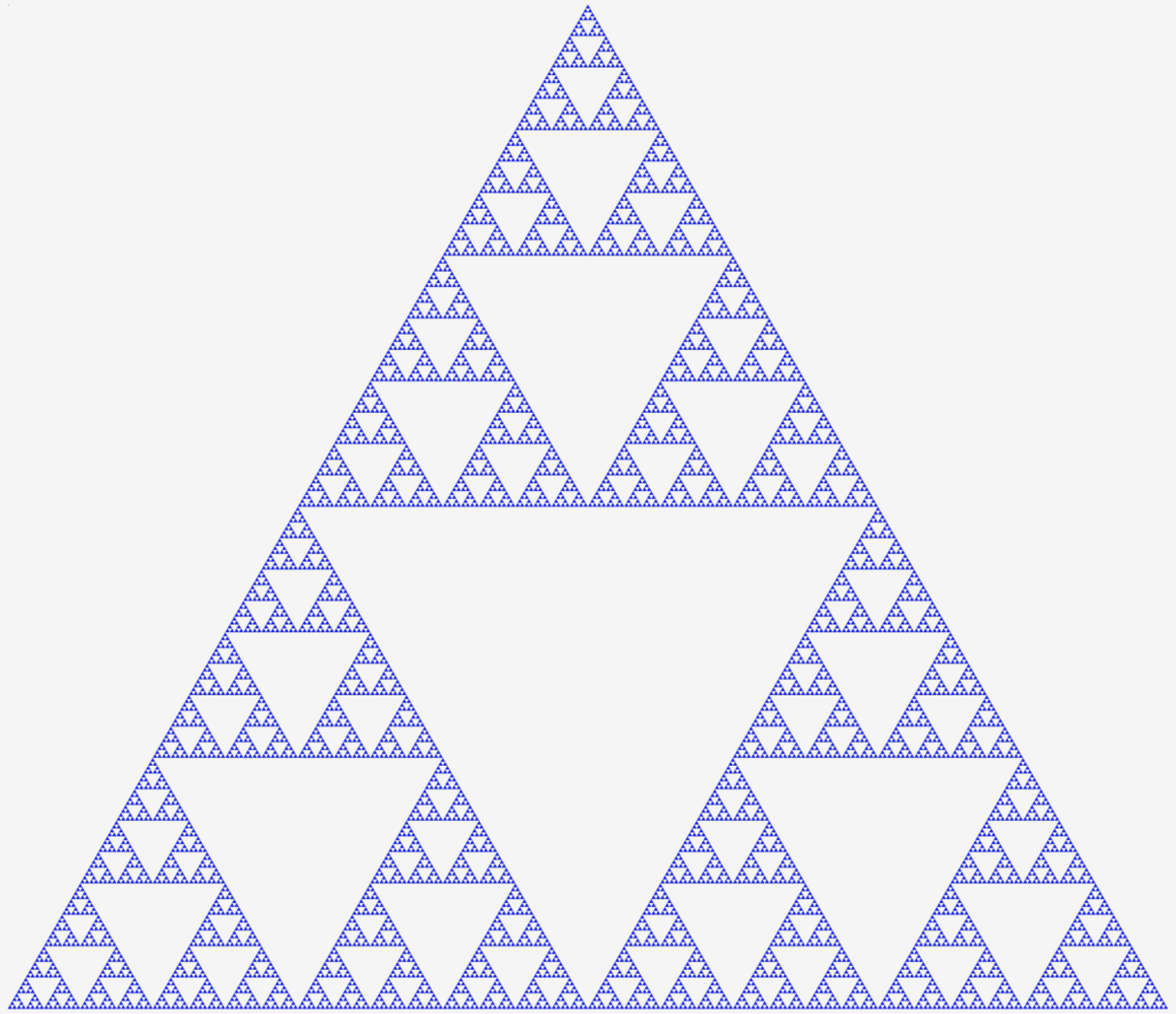
Chapter 8

Recursion

Recursion

To understand ***recursion*** you must first understand ***recursion.***





Sierpinksi Triangle

Recursion

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

A ***recursive algorithm*** is a category of algorithms in which a problem is solved by relying on repetitions of the same algorithm.

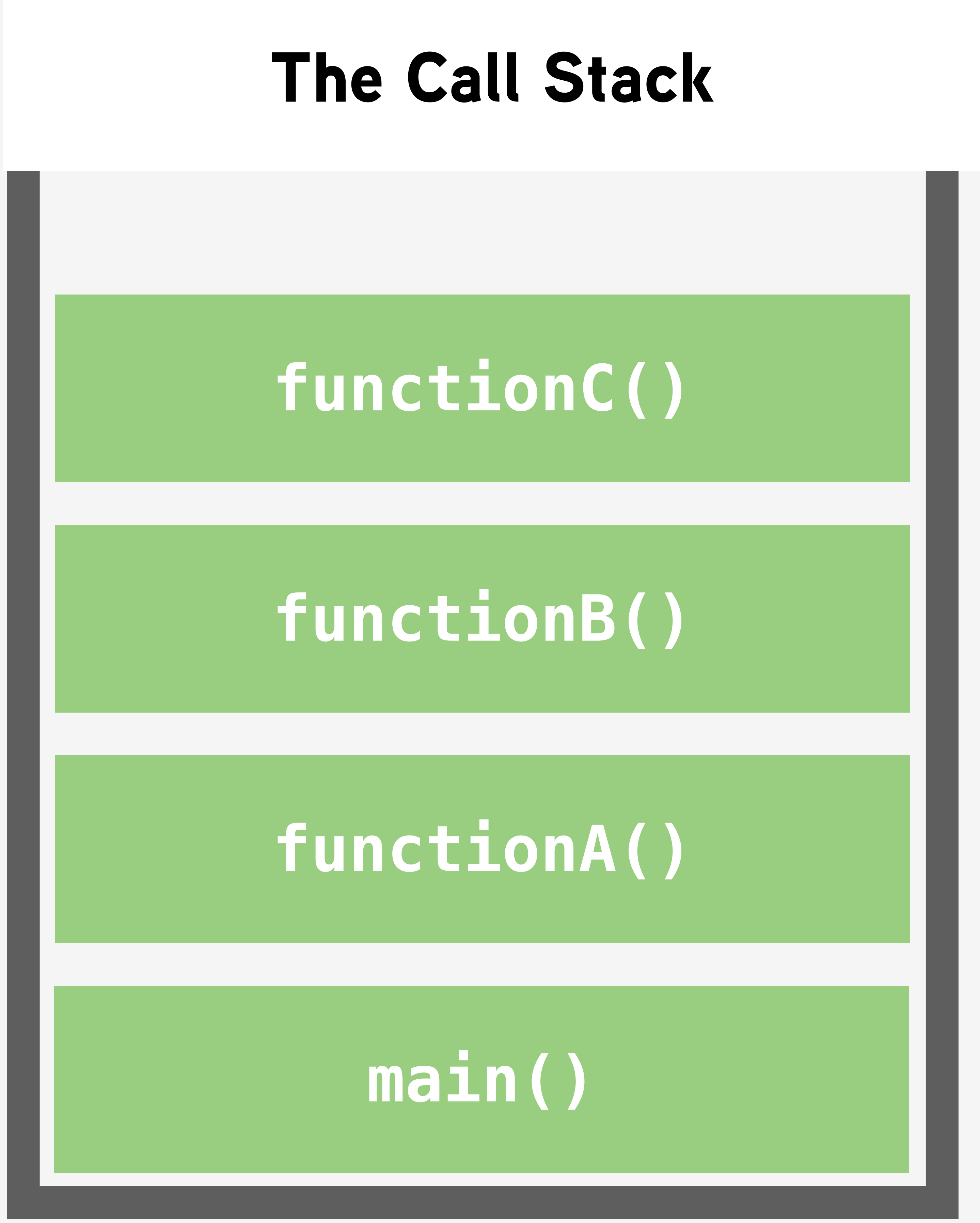
More simply: A recursive function, is a function that calls itself.

```
void myFunction() {
  myFunction();
}
```

examples: basic_recursion.cpp

Remember functions, and the call stack!

```cpp
void functionA();
void functionB();
void functionC();

int main() {
  functionA();

  return 0;
}

void functionA() {
  cout << "Starting functionA()" << endl;
  functionB();
  cout << "Ending functionA()" << endl;
}

void functionB() {
  cout << "Starting functionB()" << endl;
  functionC();
  cout << "Ending functionB()" << endl;
}

void functionC() {
  cout << "Starting functionC()" << endl;
  cout << "Ending functionC()" << endl;
}
```



examples: function_overview.cpp

Recursive Example:

Consider a function / algorithm for running a race…

Define function ***race():***

1. If you cross the finish line, STOP
2. Take one step forward
3. ***race()***

Recursive Example:

Consider a function / algorithm for running a race…

Define function ***race():***

1. If you cross the finish line, STOP
2. Take one step forward
3. ***race()***

The Base Case

The base case defines the end point of
your program, or the stopping point of your recursive algorithm

Recursive Example:

Consider a function / algorithm for running a race…

Define function ***race():***

1. If you cross the finish line, STOP
2. Take one step forward
3. ***race()***

The Work

The second part of our recursive algorithm contains the work that is being done to get to our stopping point

Recursive Example:

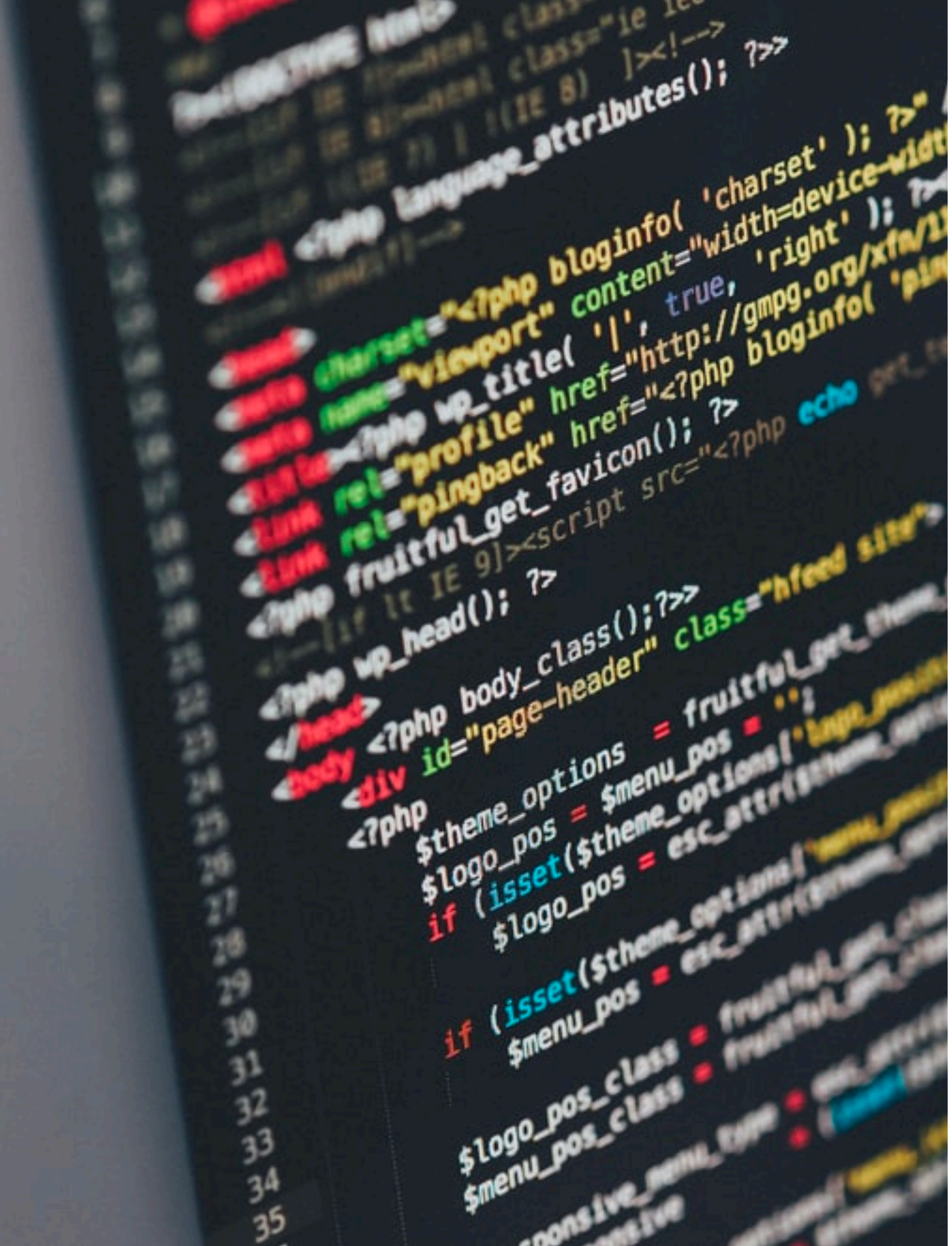
Consider a function / algorithm for running a race…

Define function ***race():***

1. If you cross the finish line, STOP
2. Take one step forward
3. ***race()***

The Recursive Call

Lastly, we have the function call, to itself.



Recursion!

Implement a function that counts down from a number N and then prints “GO” once it reaches 0.

examples: count_down.cpp

Creating a recursive function

- All recursive functions ***must*** have a base case so that they can finish execution
  - Remember our first example:

```
void myFunction() {
  myFunction();
}
```

Is this a good recursive function?

  - Define the work being done and then the recursive function call.

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

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

}
```

Example: Factorial

We can represent factorial (***!***) as the product of all integers below it.

- Examples:

2! = 2 * 1

5! = 5 * 4 * 3 * 2 * 1

7! = 7 * 6 * 5 * 4 * 3 * 2 * 1

example: factorial.cpp

Example: Factorial

We can represent factorial (***!***) as the product of all integers below it.

- Examples:

2! = 2 * 1 = 2

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

7! = 7 * 6 * 5 * 4 * 3 * 2 * 1 = 5040

Mathematically we can represent that factorial as:

N! = N * (N - 1)!

example: factorial.cpp

Greatest Common Divisor

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

The Greatest common divisor is typically solved by using the ***Euclidean algorithm.***

- Works by repeatedly subtracting the smaller of two numbers from the larger, until they are equal, yielding 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 -> 4
```

example: gcd.cpp

Greatest Common Divisor

Alternatively, we can also solve the GCD problem using the ***modulo euclidean algorithm.***

This works by using the modulo operator to find the greatest common divisor

General formula for GCD modulo:

```
GCD(a, b) = GCD(b % a, a)
```

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

```
GCD(10,8) = GCD(8, 10 % 8)

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

GCD(2, 0) = 2
```

example: gcd_modulo.cpp

Fibonacci (Recursion fan out)

One problem with recursion is the use of a finite space or stack frames in order to find a solution.

- Consider the Fibonacci sequence:

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

- The Fibonacci sequence is a number set where each number is the sum of two preceding numbers. The formula is as follows:

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

- This causes recursive ***fan out.***

example: fibonacci.cpp

Fibonacci (C++)

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

```
int recursion(int val1, int val2) {
  ...

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

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

Types of Recursion

Direct Recursion - When a function contains a call to itself, within it's own function body.

Indirect Recursion - When a function contains a call to a secondary function which in turn calls the first function again, repeatedly placing multiple ***different*** stack frames on the call stack to find a solution.

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

Iteration vs Recursion

Both ***iteration*** and ***recursion*** are based on some sort of control statement.

Both involve repetitions of some block of code:

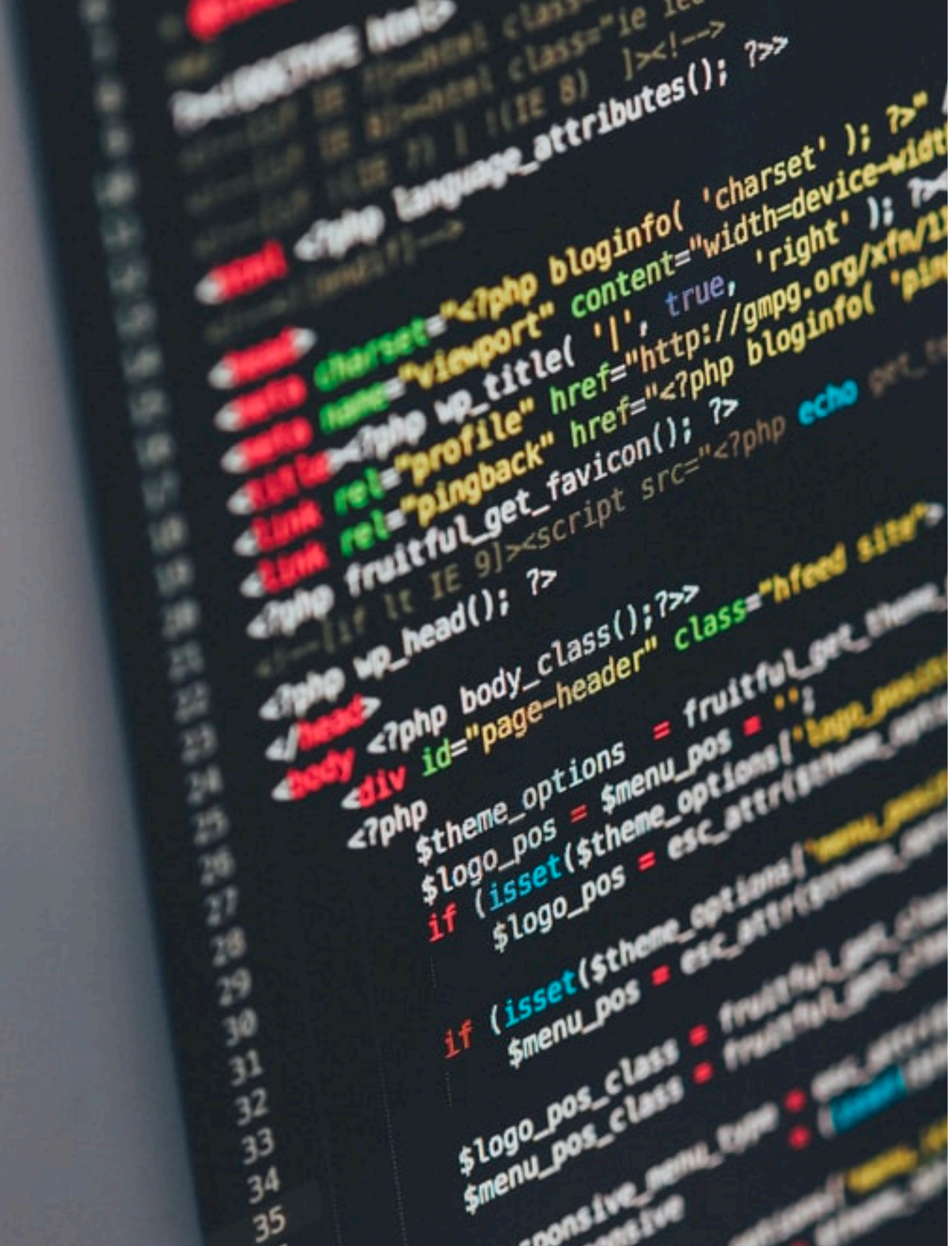
- Iteration uses ***iteration blocks***
- Recursion uses repeated ***function calls*** or ***stack frames*** within the call stack

Iteration and recursion both require some sort of termination test

- Iteration - uses conditional
- Recursion - uses base case

https://learn.zybooks.com/zybook/SMUCS1342GabrielsenSpring2021/chapter/6/section/7?content_resource_id=46786481

example: palindrome.cpp



Palindrome

Implement a function that determines whether or not a cstring is a palindrome. Use an ***iterative*** approach first, then ***recursive***

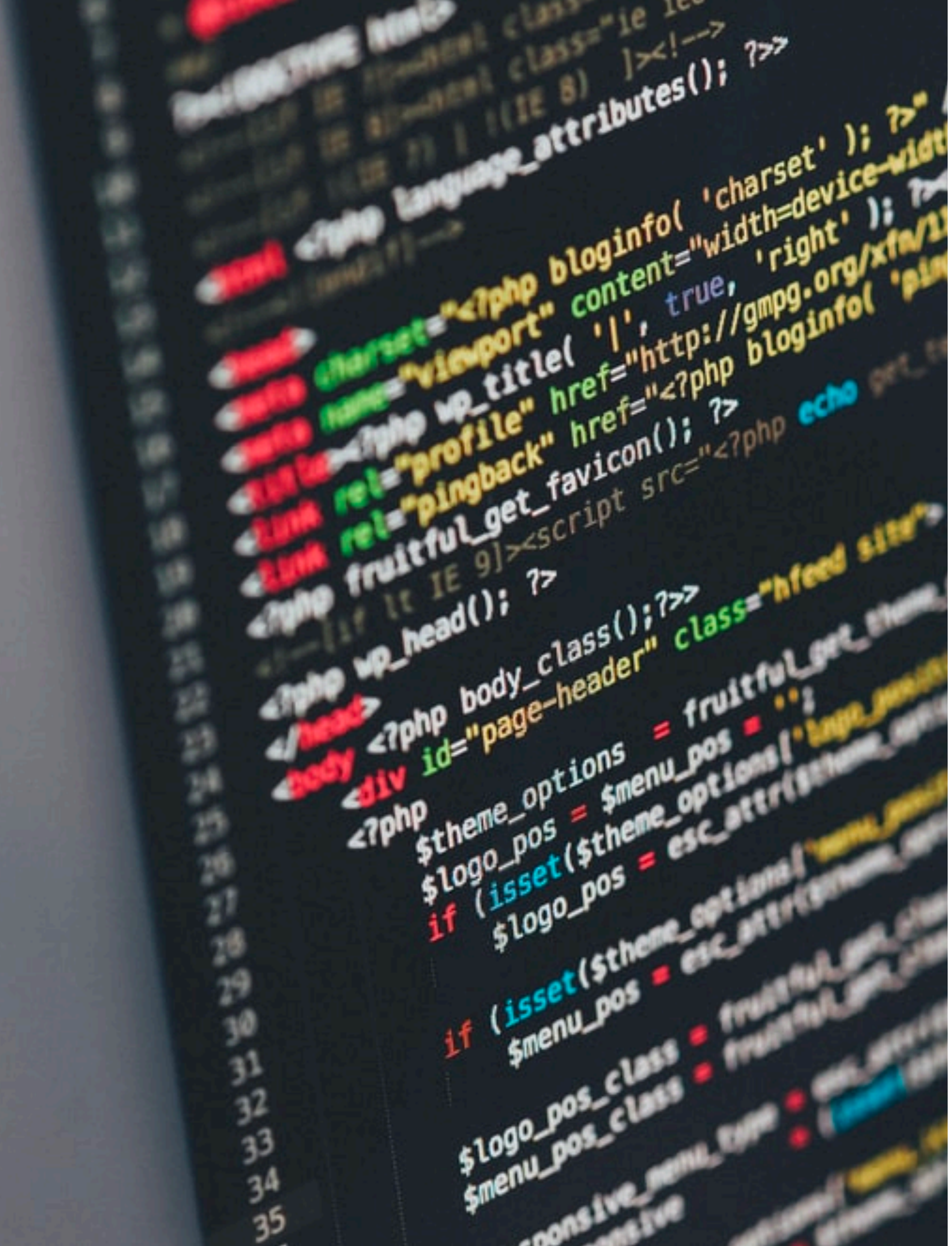
examples: palindrome.cpp

Palindrome: ***iterative***

```
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;
}
```

Palindrome: ***recursive***

```
bool palindrome(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);
  }
}
```

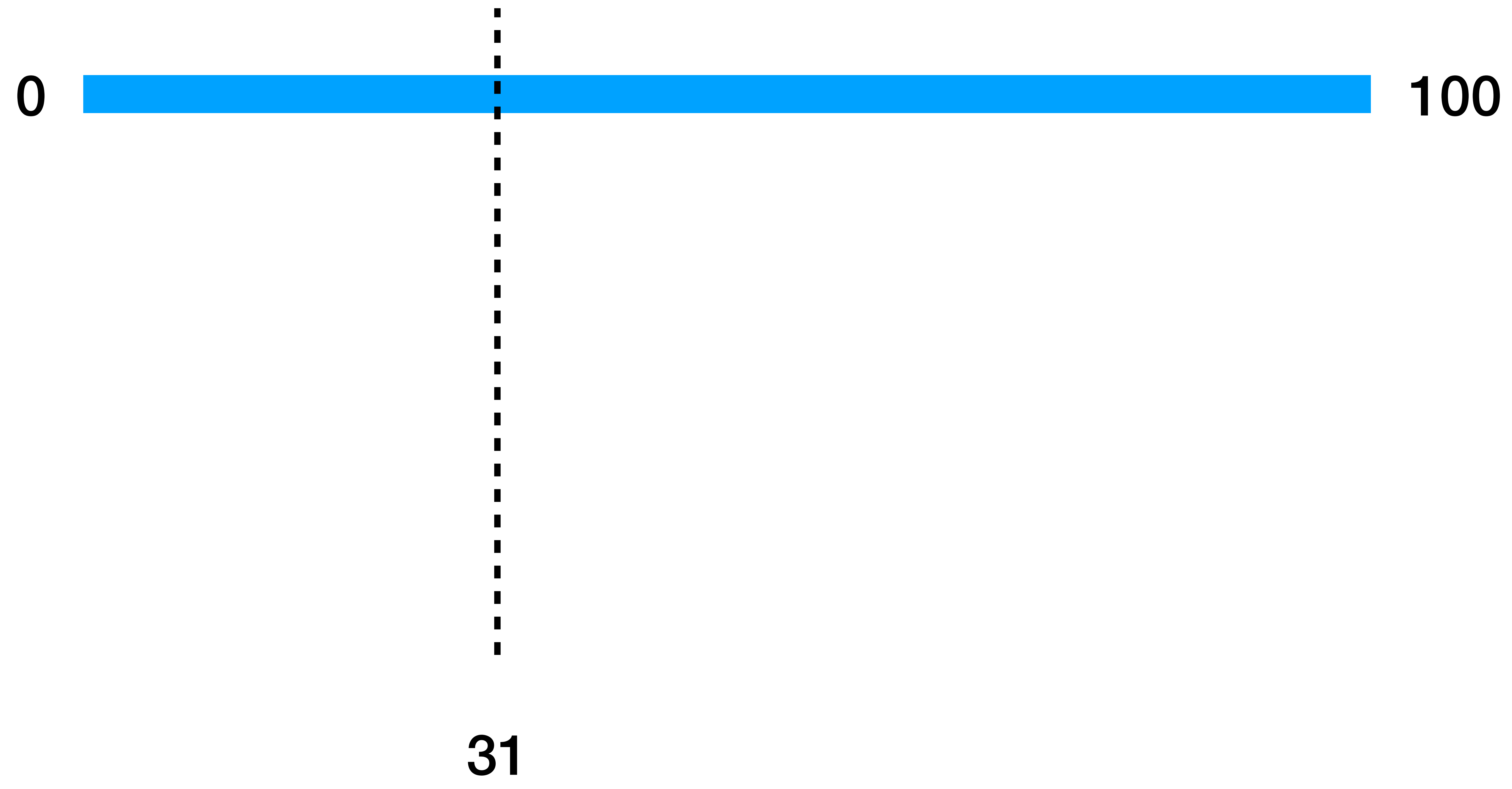


Guessing Game

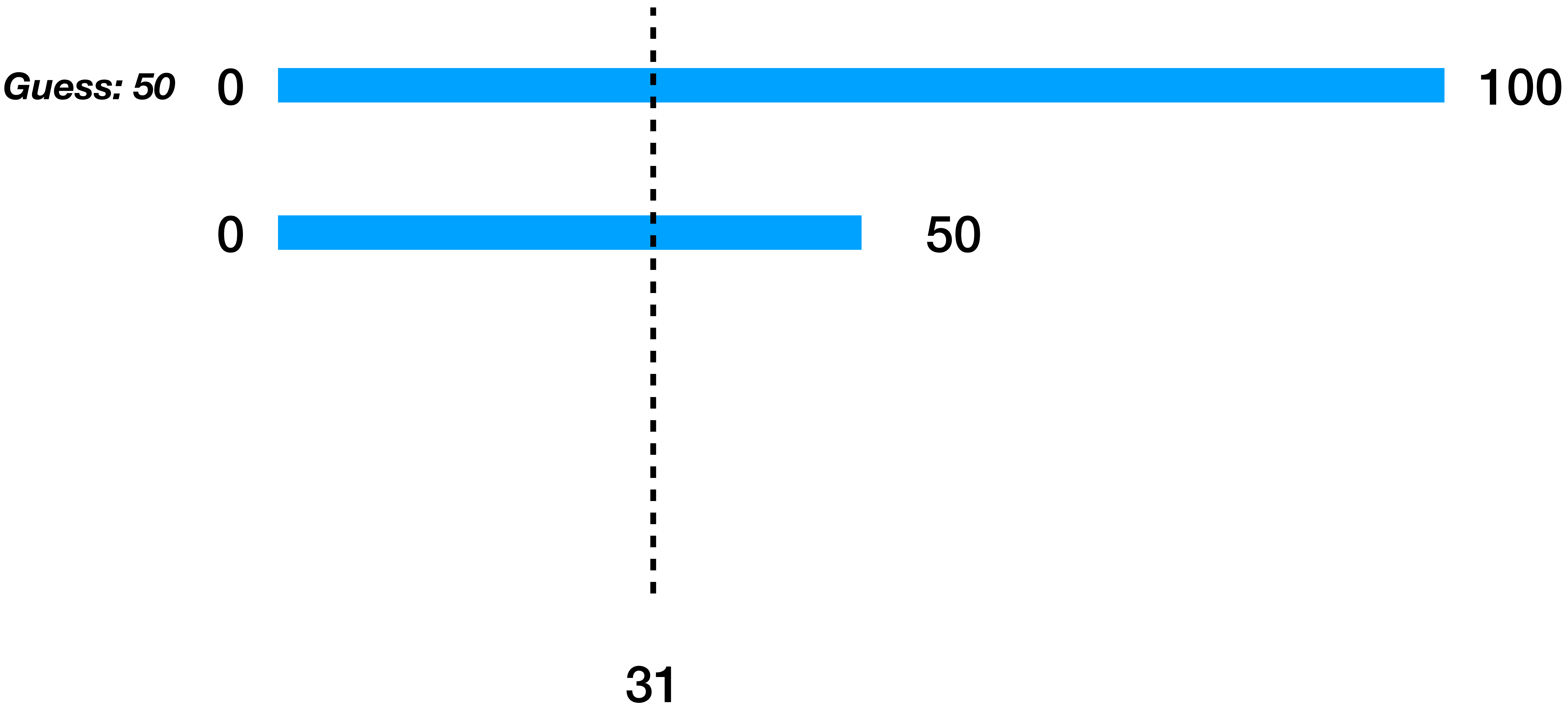
Think of a number between 1 and 100.

Create a program that repeatedly guesses a number until the correct number has been guessed.

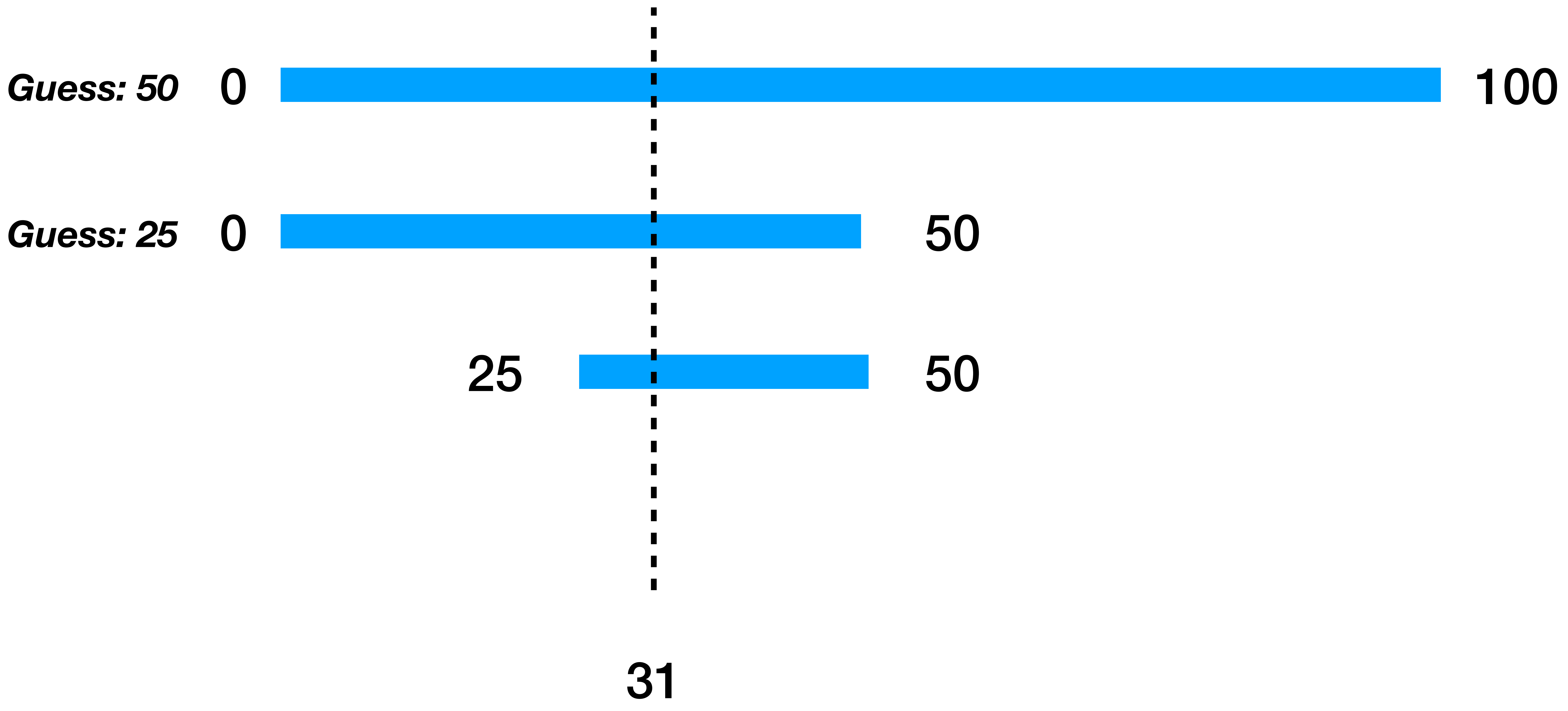
Guessing Game



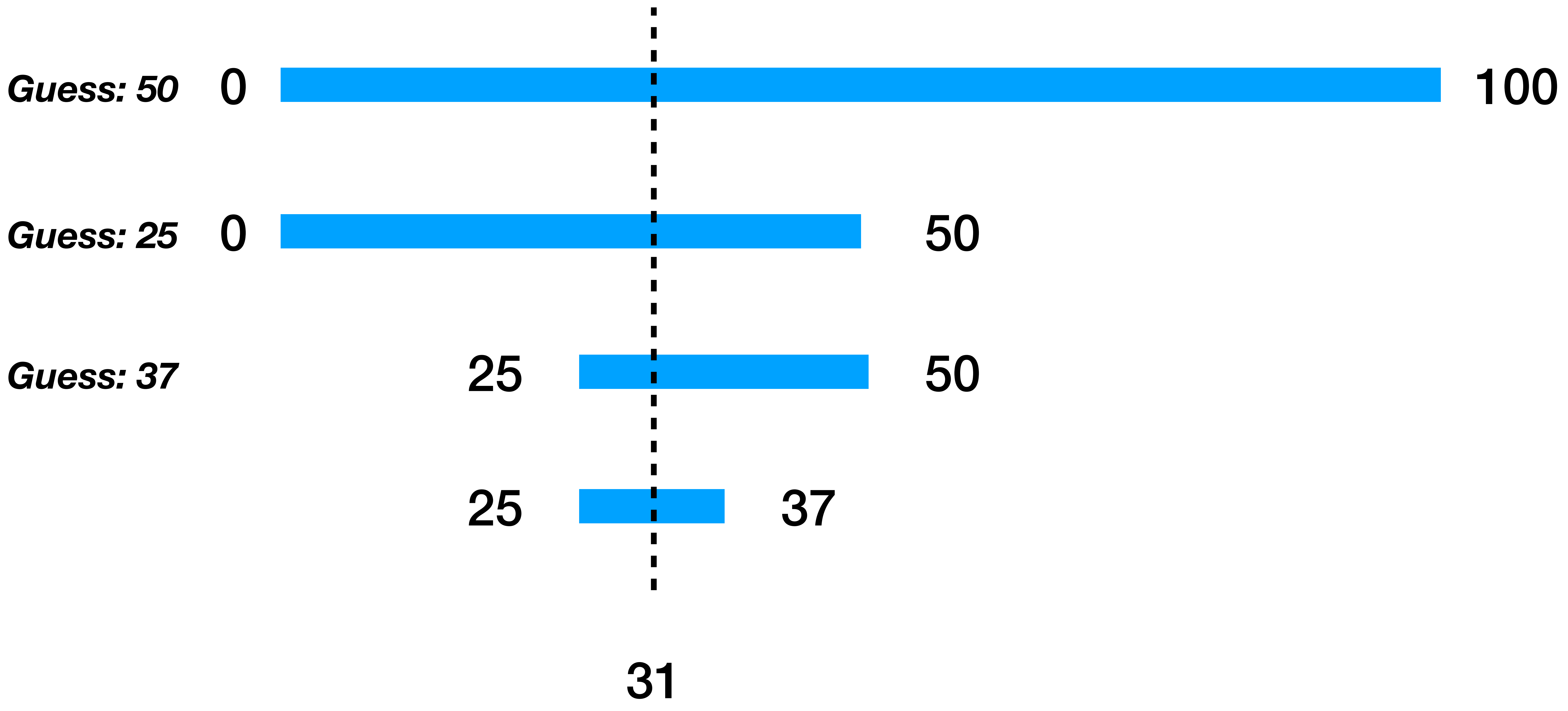
Guessing Game



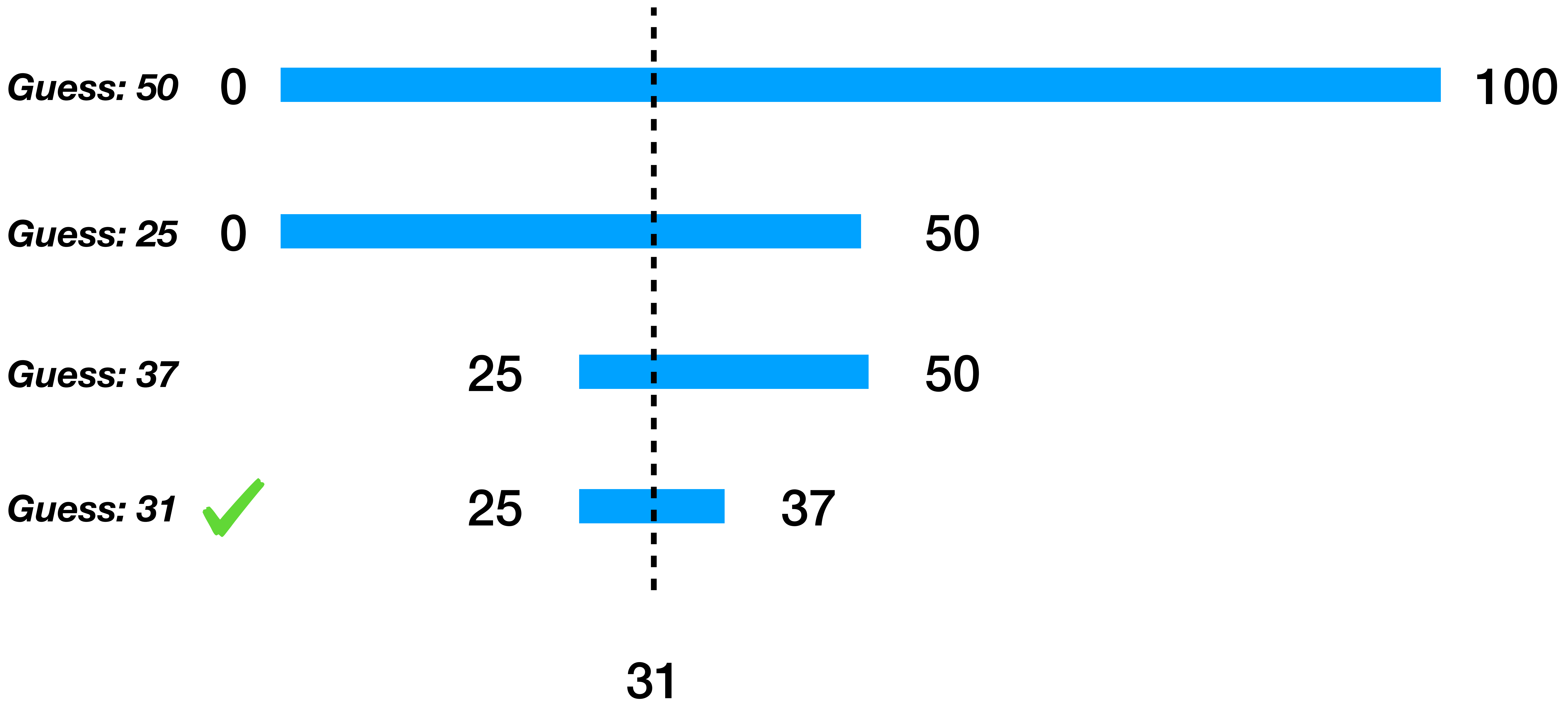
Guessing Game

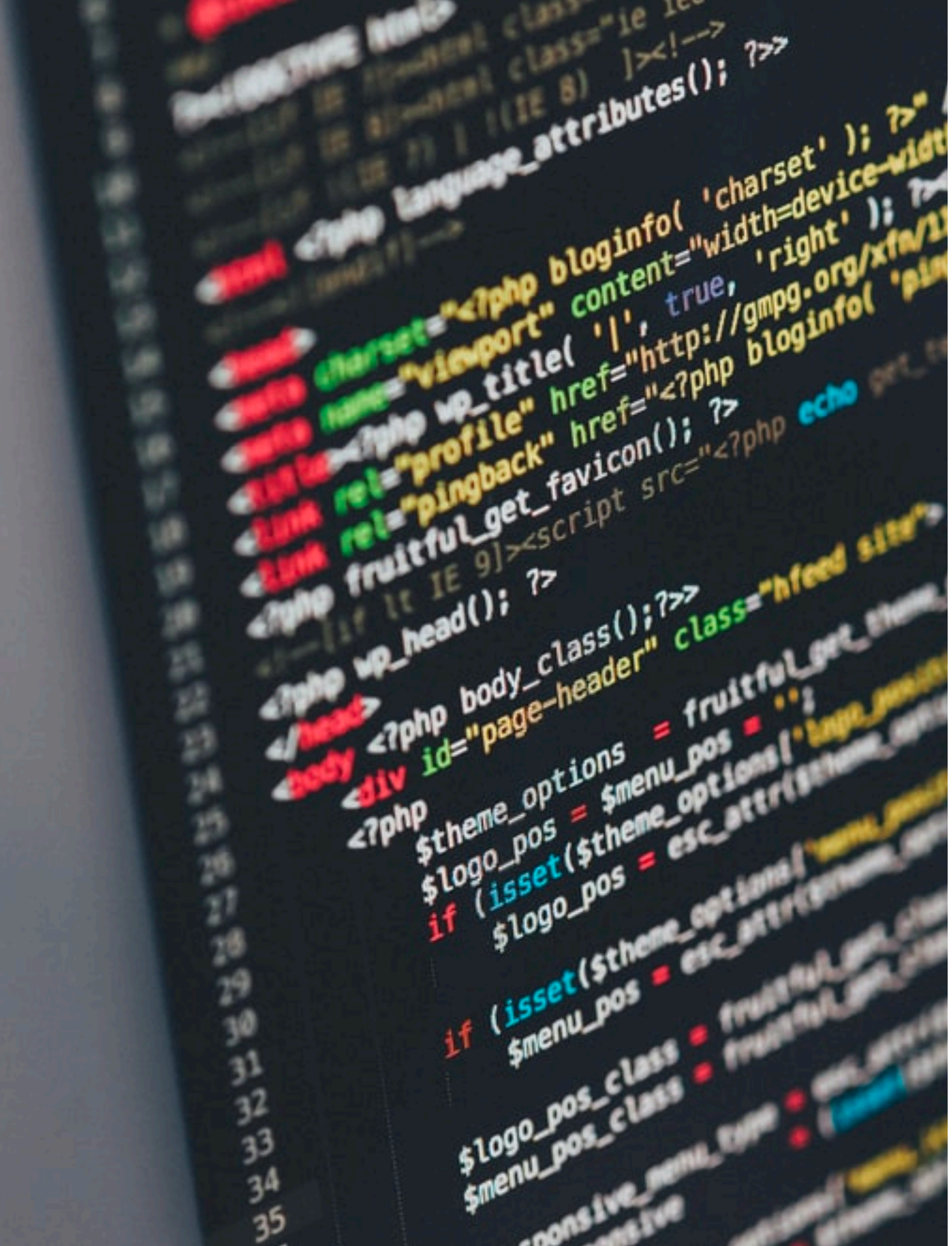


Guessing Game



Guessing Game





Guessing Game

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

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);
  }
}
```

Binary Search

The ***Binary Search*** algorithm is a category of search algorithms that is used to find a value amongst an ordered list.

- binary search works exactly like the Guessing Game!
- 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.

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 (Think fibonacci)