CS 115 Lecture 20

Recursion

Neil Moore

Department of Computer Science
University of Kentucky
Lexington, Kentucky 40506
neil@cs.uky.edu

1 December 2015

Recursion

Problems—computational, mathematical, and otherwise—can be defined and solved **recursively.**

- That is, in terms of themselves.

Recursion

Problems—computational, mathematical, and otherwise—can be defined and solved **recursively.**

- That is, in terms of themselves.
- A compound sentence is two sentences with “and” between them.

Recursion

Problems—computational, mathematical, and otherwise—can be defined and solved **recursively.**

- That is, in terms of themselves.
- A compound sentence is two sentences with “and” between them.
- A Python expression may contain two expressions with an operator between them: `(3 + 2) * (4 - 9)`.

Recursion

Problems—computational, mathematical, and otherwise—can be defined and solved **recursively.**

- That is, in terms of themselves.
- A compound sentence is two sentences with "and" between them.
- A Python expression may contain two expressions with an operator between them: `(3 + 2) * (4 - 9)`.
- Point a video camera at its own display—hall of mirrors.

Recursion

Problems—computational, mathematical, and otherwise—can be defined and solved **recursively.**

- That is, in terms of themselves.
- A compound sentence is two sentences with "and" between them.
- A Python expression may contain two expressions with an operator between them: `(3 + 2) * (4 - 9)`.
- Point a video camera at its own display—hall of mirrors.
- Many mathematical structures are defined recursively.
  - Fibonacci numbers, factorials, fractals, ...

Recursion

Problems—computational, mathematical, and otherwise—can be defined and solved **recursively.**

- That is, in terms of themselves.
- A compound sentence is two sentences with "and" between them.
- A Python expression may contain two expressions with an operator between them: `(3 + 2) * (4 - 9)`.
- Point a video camera at its own display—hall of mirrors.
- Many mathematical structures are defined recursively.
  - Fibonacci numbers, factorials, fractals, ...
  - Mathematicians call this **induction** (same thing as recursion).
  - It's also a common method of mathematical proof.

Recursion

Problems—computational, mathematical, and otherwise—can be defined and solved **recursively.**

- That is, in terms of themselves.
- A compound sentence is two sentences with "and" between them.
- A Python expression may contain two expressions with an operator between them: `(3 + 2) * (4 - 9)`.
- Point a video camera at its own display—hall of mirrors.
- Many mathematical structures are defined recursively.
  - Fibonacci numbers, factorials, fractals, ...
  - Mathematicians call this **induction** (same thing as recursion).
  - It's also a common method of mathematical proof.
- Search for recursion on Google.
  - Note the search suggestion.

Recursion

Problems—computational, mathematical, and otherwise—can be defined and solved **recursively.**

- That is, in terms of themselves.
- A compound sentence is two sentences with "and" between them.
- A Python expression may contain two expressions with an operator between them: `(3 + 2) * (4 - 9)`.
- Point a video camera at its own display—hall of mirrors.
- Many mathematical structures are defined recursively.
  - Fibonacci numbers, factorials, fractals, . . .
  - Mathematicians call this **induction** (same thing as recursion).
  - It's also a common method of mathematical proof.
- Search for recursion on Google.
  - Note the search suggestion.

Recursion in programming

- The idea behind recursion in programming:
  - Break down a complex problem into a simpler version of *the same problem*.

Recursion in programming

- The idea behind recursion in programming:
  - Break down a complex problem into a simpler version of *the same problem*.
  - Implemented by functions *that call themselves*.

Recursion in programming

- The idea behind recursion in programming:
  - Break down a complex problem into a simpler version of *the same problem*.
  - Implemented by functions *that call themselves*.
    - **Recursive functions**.

Recursion in programming

- The idea behind recursion in programming:
  - Break down a complex problem into a simpler version of *the same problem*.
  - Implemented by functions *that call themselves*.
    - **Recursive functions**.
  - The same computation recurs (occurs repeatedly).
    - This is not the same as iteration (looping)!

Recursion in programming

- The idea behind recursion in programming:
  - Break down a complex problem into a simpler version of *the same problem*.
  - Implemented by functions *that call themselves*.
    - **Recursive functions**.
  - The same computation recurs (occurs repeatedly).
    - This is not the same as iteration (looping)!
    - But it is possible to convert iteration to recursion, and vice versa.

Recursion in programming

- The idea behind recursion in programming:
  - Break down a complex problem into a simpler version of *the same problem*.
  - Implemented by functions *that call themselves*.
    - **Recursive functions**.
  - The same computation recurs (occurs repeatedly).
    - This is not the same as iteration (looping)!
    - But it is possible to convert iteration to recursion, and vice versa.
- Recursion is often the most natural way of thinking about a problem.

Recursion in programming

- The idea behind recursion in programming:
  - Break down a complex problem into a simpler version of *the same problem*.
  - Implemented by functions *that call themselves*.
    - **Recursive functions**.
  - The same computation recurs (occurs repeatedly).
    - This is not the same as iteration (looping)!
    - But it is possible to convert iteration to recursion, and vice versa.
- Recursion is often the most natural way of thinking about a problem.
  - Some computations are very difficult to perform without recursion.

Recursion in programming

- The idea behind recursion in programming:
  - Break down a complex problem into a simpler version of *the same problem*.
  - Implemented by functions *that call themselves*.
    - **Recursive functions**.
  - The same computation recurs (occurs repeatedly).
    - This is not the same as iteration (looping)!
    - But it is possible to convert iteration to recursion, and vice versa.
- Recursion is often the most natural way of thinking about a problem.
  - Some computations are very difficult to perform without recursion.

Thinking recursively

- Suppose we want to write a function that prints a triangle of stars.
  `print_triangle(4)`
  →
  ```
  *
  * *
  * * *
  * * * *
  ```
- We could use nested loops, but let's try using recursion instead.

Thinking recursively

- Suppose we want to write a function that prints a triangle of stars.
  `print_triangle(4)`
  ```
  →*
   * *
   * * *
   * * * *
  ```
- We could use nested loops, but let's try using recursion instead.
  - Pretend someone else has already written a function to print a triangle of size 3.

Thinking recursively

- Suppose we want to write a function that prints a triangle of stars.
  `print_triangle(4)`

  ```
  →*
   * *
   * * *
   * * * *
  ```

- We could use nested loops, but let's try using recursion instead.
  - Pretend someone else has already written a function to print a triangle of size 3. How would you print a triangle of size 4?

Thinking recursively

- Suppose we want to write a function that prints a triangle of stars.
  `print_triangle(4)`
  ```
  →*
   * *
   * * *
   * * * *
  ```
- We could use nested loops, but let's try using recursion instead.
  - Pretend someone else has already written a function to print a triangle of size 3. How would you print a triangle of size 4?
    - First call that function.
    - Then print a row of four stars.

Thinking recursively

- Suppose we want to write a function that prints a triangle of stars.
  `print_triangle(4)`
  ```
  →*
    * *
    * * *
    * * * *
  ```
- We could use nested loops, but let's try using recursion instead.
  - Pretend someone else has already written a function to print a triangle of size 3. How would you print a triangle of size 4?
    - First call that function.
    - Then print a row of four stars.
  - What about size 5?
    - Print a triangle of size 4.
    - Then print a row of five stars.

Thinking recursively

- Suppose we want to write a function that prints a triangle of stars.
  `print_triangle(4)`

  ```
  →*
   * *
   * * *
   * * * *
  ```
- We could use nested loops, but let's try using recursion instead.
  - Pretend someone else has already written a function to print a triangle of size 3. How would you print a triangle of size 4?
    - First call that function.
    - Then print a row of four stars.
  - What about size 5?
    - Print a triangle of size 4.
    - Then print a row of five stars.
  - Recursion: Use the solution to a simpler version of the same problem!

Thinking recursively

- Suppose we want to write a function that prints a triangle of stars.
  `print_triangle(4)`

  ```
  →*
   * *
   * * *
   * * * *
  ```

- We could use nested loops, but let's try using recursion instead.
  - Pretend someone else has already written a function to print a triangle of size 3. How would you print a triangle of size 4?
    - ★ First call that function.
    - ★ Then print a row of four stars.
  - What about size 5?
    - ★ Print a triangle of size 4.
    - ★ Then print a row of five stars.
  - Recursion: Use the solution to a simpler version of the same problem!

A (broken) recursive function

```
def print_triangle(side_len):
    # First solve a simpler version of the problem.
    print_triangle(side_len - 1)
```

A (broken) recursive function

```
def print_triangle(side_len):
    # First solve a simpler version of the problem.
    print_triangle(side_len - 1)
    # Now turn it into the solution to this problem.
    # (by drawing the last line).
    print("* " * side_len)
    print()
```

A (broken) recursive function

```
def print_triangle(side_len):
    # First solve a simpler version of the problem.
    print_triangle(side_len - 1)
    # Now turn it into the solution to this problem.
    # (by drawing the last line).
    print("* " * side_len)
    print()
```

- But there's one small problem. . .

A (broken) recursive function

```
def print_triangle(side_len):
    # First solve a simpler version of the problem.
    print_triangle(side_len - 1)
    # Now turn it into the solution to this problem.
    # (by drawing the last line).
    print("* " * side_len)
    print()
```

- But there's one small problem. . .
  - It will never end!
  - To print a triangle of size 1, first print a triangle of size 0.

A (broken) recursive function

```
def print_triangle(side_len):
    # First solve a simpler version of the problem.
    print_triangle(side_len - 1)
    # Now turn it into the solution to this problem.
    # (by drawing the last line).
    print("* " * side_len)
    print()
```

- But there's one small problem...
  - It will never end!
  - To print a triangle of size 1, first print a triangle of size 0.
  - To do that, first print a triangle of size -1...

A (broken) recursive function

```
def print_triangle(side_len):
    # First solve a simpler version of the problem.
    print_triangle(side_len - 1)
    # Now turn it into the solution to this problem.
    # (by drawing the last line).
    print("* " * side_len)
    print()
```

- But there's one small problem...
  - It will never end!
  - To print a triangle of size 1, first print a triangle of size 0.
  - To do that, first print a triangle of size -1...
    - What?

A (broken) recursive function

```
def print_triangle(side_len):
    # First solve a simpler version of the problem.
    print_triangle(side_len - 1)
    # Now turn it into the solution to this problem.
    # (by drawing the last line).
    print("* " * side_len)
    print()
```

- But there's one small problem...
  - It will never end!
  - To print a triangle of size 1, first print a triangle of size 0.
  - To do that, first print a triangle of size -1...
    - What?

The base case

- Every recursion must end somewhere.
  - At some point the problem is so simple we can solve it directly.

The base case

- Every recursion must end somewhere.
  - At some point the problem is so simple we can solve it directly.
  - Usually that is when the size of the problem is zero or one.

The base case

- Every recursion must end somewhere.
  - At some point the problem is so simple we can solve it directly.
  - Usually that is when the size of the problem is zero or one.
  - We call this the **base case** or **termination condition**.

The base case

- Every recursion must end somewhere.
  - At some point the problem is so simple we can solve it directly.
  - Usually that is when the size of the problem is zero or one.
  - We call this the **base case** or **termination condition**.
  - How do we print a triangle with size zero?

The base case

- Every recursion must end somewhere.
  - At some point the problem is so simple we can solve it directly.
  - Usually that is when the size of the problem is zero or one.
  - We call this the **base case** or **termination condition**.
  - How do we print a triangle with size zero?
    - By doing nothing!

The base case

- Every recursion must end somewhere.
  - At some point the problem is so simple we can solve it directly.
  - Usually that is when the size of the problem is zero or one.
  - We call this the **base case** or **termination condition**.
  - How do we print a triangle with size zero?
    - By doing nothing!

```
def print_triangle(side_len):
    if side_len <= 0 # Base case.
        pass # Do nothing.
    else:  # Recursive case
        print_triangle(side_len - 1)
        print("* " * side_len)
        print()
```

The base case

- Every recursion must end somewhere.
  - At some point the problem is so simple we can solve it directly.
  - Usually that is when the size of the problem is zero or one.
  - We call this the **base case** or **termination condition**.
  - How do we print a triangle with size zero?
    - By doing nothing!

```
def print_triangle(side_len):
    if side_len <= 0 # Base case.
        pass # Do nothing.
    else:  # Recursive case
        print_triangle(side_len - 1)
        print("* " * side_len)
        print()
```

Rules for recursion

There are three key requirements for a recursive function to work correctly.

Rules for recursion

There are three key requirements for a recursive function to work correctly.

1. **Base case**: There must be a special case to handle the simplest versions of the problem directly, without recursion.
   - Does *not* call the function again.

Rules for recursion

There are three key requirements for a recursive function to work correctly.

1. **Base case**: There must be a special case to handle the simplest versions of the problem directly, without recursion.
   - Does *not* call the function again.
2. **Recursive case**: There must be a case where the function does call itself.

Rules for recursion

There are three key requirements for a recursive function to work correctly.

1. **Base case**: There must be a special case to handle the simplest versions of the problem directly, without recursion.
   - Does *not* call the function again.
2. **Recursive case**: There must be a case where the function does call itself.
3. **Simplification**: The recursive call must be on a simpler version of the problem.

Rules for recursion

There are three key requirements for a recursive function to work correctly.

1. **Base case**: There must be a special case to handle the simplest versions of the problem directly, without recursion.
   - Does *not* call the function again.
2. **Recursive case**: There must be a case where the function does call itself.
3. **Simplification**: The recursive call must be on a simpler version of the problem. That is, it must reduce the size of the problem, bringing you *closer to the base case*.

Rules for recursion

There are three key requirements for a recursive function to work correctly.

1. **Base case**: There must be a special case to handle the simplest versions of the problem directly, without recursion.
   - Does *not* call the function again.
2. **Recursive case**: There must be a case where the function does call itself.
3. **Simplification**: The recursive call must be on a simpler version of the problem. That is, it must reduce the size of the problem, bringing you *closer to the base case*.
   - That means the arguments must be changed from the parameters.
   - If not, you have **infinite recursion**

Rules for recursion

There are three key requirements for a recursive function to work correctly.

1. **Base case**: There must be a special case to handle the simplest versions of the problem directly, without recursion.
   - Does *not* call the function again.
2. **Recursive case**: There must be a case where the function does call itself.
3. **Simplification**: The recursive call must be on a simpler version of the problem. That is, it must reduce the size of the problem, bringing you *closer to the base case*.
   - That means the arguments must be changed from the parameters.
   - If not, you have **infinite recursion**

And a few related guidelines:

- You should check for the base case **first**.
  - ...before making any recursive calls.

Rules for recursion

There are three key requirements for a recursive function to work correctly.

1. **Base case**: There must be a special case to handle the simplest versions of the problem directly, without recursion.
   - Does *not* call the function again.
2. **Recursive case**: There must be a case where the function does call itself.
3. **Simplification**: The recursive call must be on a simpler version of the problem. That is, it must reduce the size of the problem, bringing you *closer to the base case*.
   - That means the arguments must be changed from the parameters.
   - If not, you have **infinite recursion**

And a few related guidelines:

- You should check for the base case **first**.
  - ...before making any recursive calls.
- The base case is usually, but not always, a problem of size 0 or 1.

Rules for recursion

There are three key requirements for a recursive function to work correctly.

1. **Base case**: There must be a special case to handle the simplest versions of the problem directly, without recursion.
   - Does *not* call the function again.
2. **Recursive case**: There must be a case where the function does call itself.
3. **Simplification**: The recursive call must be on a simpler version of the problem. That is, it must reduce the size of the problem, bringing you *closer to the base case*.
   - That means the arguments must be changed from the parameters.
   - If not, you have **infinite recursion**

And a few related guidelines:

- You should check for the base case **first**.
  - ...before making any recursive calls.
- The base case is usually, but not always, a problem of size 0 or 1.

About the rules

- You can have multiple base cases, as long as there is at least one.

About the rules

- You can have multiple base cases, as long as there is at least one.
- Sometimes the base case does nothing.
  - You could write this using `pass` ("do nothing")

About the rules

- You can have multiple base cases, as long as there is at least one.
- Sometimes the base case does nothing.
  - You could write this using `pass` ("do nothing")
  - Or you could put the recursive case in an `if`.
    - ("If it's not the base case, then do something.")

About the rules

- You can have multiple base cases, as long as there is at least one.
- Sometimes the base case does nothing.
  - You could write this using `pass` ("do nothing")
  - Or you could put the recursive case in an `if`.
    - ("If it's not the base case, then do something.")
- If the function returns something, it should use the value of the recursive call.

About the rules

- You can have multiple base cases, as long as there is at least one.
- Sometimes the base case does nothing.
  - You could write this using `pass` ("do nothing")
  - Or you could put the recursive case in an `if`.
    - ("If it's not the base case, then do something.")
- If the function returns something, it should use the value of the recursive call.
- The changes you make to the recursive arguments can be anything:
  - Often subtraction, division, or shortening a list.

About the rules

- You can have multiple base cases, as long as there is at least one.
- Sometimes the base case does nothing.
  - You could write this using `pass` ("do nothing")
  - Or you could put the recursive case in an `if`.
    - ("If it's not the base case, then do something.")
- If the function returns something, it should use the value of the recursive call.
- The changes you make to the recursive arguments can be anything:
  - Often subtraction, division, or shortening a list.
  - But in some situations, addition or other operations.
  - The important thing is that it gets closer to a base case.

About the rules

- You can have multiple base cases, as long as there is at least one.
- Sometimes the base case does nothing.
  - You could write this using `pass` ("do nothing")
  - Or you could put the recursive case in an `if`.
    - ("If it's not the base case, then do something.")
- If the function returns something, it should use the value of the recursive call.
- The changes you make to the recursive arguments can be anything:
  - Often subtraction, division, or shortening a list.
  - But in some situations, addition or other operations.
  - The important thing is that it gets closer to a base case.
- The order of recursive calls matters!
  - What happens if we move the `print_triangle` call after the `print`?

About the rules

- You can have multiple base cases, as long as there is at least one.
- Sometimes the base case does nothing.
  - You could write this using `pass` ("do nothing")
  - Or you could put the recursive case in an `if`.
    - ("If it's not the base case, then do something.")
- If the function returns something, it should use the value of the recursive call.
- The changes you make to the recursive arguments can be anything:
  - Often subtraction, division, or shortening a list.
  - But in some situations, addition or other operations.
  - The important thing is that it gets closer to a base case.
- The order of recursive calls matters!
  - What happens if we move the `print_triangle` call after the `print`?
  - The triangle is upside-down!

About the rules

- You can have multiple base cases, as long as there is at least one.
- Sometimes the base case does nothing.
  - You could write this using `pass` ("do nothing")
  - Or you could put the recursive case in an `if`.
    - ("If it's not the base case, then do something.")
- If the function returns something, it should use the value of the recursive call.
- The changes you make to the recursive arguments can be anything:
  - Often subtraction, division, or shortening a list.
  - But in some situations, addition or other operations.
  - The important thing is that it gets closer to a base case.
- The order of recursive calls matters!
  - What happens if we move the `print_triangle` call after the `print`?
  - The triangle is upside-down!

Infinite recursion

What happens if you break one of the rules?

Infinite recursion

What happens if you break one of the rules?

- You may get an **infinite recursion**.
- Meaning the function just keeps calling itself "forever".

Infinite recursion

What happens if you break one of the rules?

- You may get an **infinite recursion**.
- Meaning the function just keeps calling itself "forever".
- Even worse than an infinite loop!
  - Every recursive call uses a little bit of memory:
    - Parameters, return address...
  - Where are these stored?

Infinite recursion

What happens if you break one of the rules?

- You may get an **infinite recursion**.
- Meaning the function just keeps calling itself "forever".
- Even worse than an infinite loop!
  - Every recursive call uses a little bit of memory:
    - Parameters, return address. . .
  - Where are these stored? The call stack!

Infinite recursion

What happens if you break one of the rules?

- You may get an **infinite recursion**.
- Meaning the function just keeps calling itself “forever”.
- Even worse than an infinite loop!
  - Every recursive call uses a little bit of memory:
    - Parameters, return address...
  - Where are these stored? The call stack!
  - So eventually an infinite recursion will run out of memory.
    - At least crashing your program.

Infinite recursion

What happens if you break one of the rules?

- You may get an **infinite recursion**.
- Meaning the function just keeps calling itself “forever”.
- Even worse than an infinite loop!
  - Every recursive call uses a little bit of memory:
    - Parameters, return address...
  - Where are these stored? The call stack!
  - So eventually an infinite recursion will run out of memory.
    - At least crashing your program.
    - And possibly the whole operating system!

Infinite recursion

What happens if you break one of the rules?

- You may get an **infinite recursion**.
- Meaning the function just keeps calling itself "forever".
- Even worse than an infinite loop!
  - Every recursive call uses a little bit of memory:
    - Parameters, return address...
  - Where are these stored? The call stack!
  - So eventually an infinite recursion will run out of memory.
    - At least crashing your program.
    - And possibly the whole operating system!
- Python has built-in checks to avoid crashing the OS with recursion.
  - When there is too much recursion, it raises an exception: `RuntimeError("Maximum recursion depth exceeded...")`
  - So the program crashes before the OS does.

Infinite recursion

What happens if you break one of the rules?

- You may get an **infinite recursion**.
- Meaning the function just keeps calling itself “forever”.
- Even worse than an infinite loop!
  - Every recursive call uses a little bit of memory:
    - Parameters, return address. . .
  - Where are these stored? The call stack!
  - So eventually an infinite recursion will run out of memory.
    - At least crashing your program.
    - And possibly the whole operating system!
- Python has built-in checks to avoid crashing the OS with recursion.
  - When there is too much recursion, it raises an exception: `RuntimeError("Maximum recursion depth exceeded...")`
  - So the program crashes before the OS does.
  - You can change the limit with `sys.setrecursionlimit(1000)`
    - But then you risk crashing more than just your program!

Infinite recursion

What happens if you break one of the rules?

- You may get an **infinite recursion**.
- Meaning the function just keeps calling itself "forever".
- Even worse than an infinite loop!
  - Every recursive call uses a little bit of memory:
    - Parameters, return address...
  - Where are these stored? The call stack!
  - So eventually an infinite recursion will run out of memory.
    - At least crashing your program.
    - And possibly the whole operating system!
- Python has built-in checks to avoid crashing the OS with recursion.
  - When there is too much recursion, it raises an exception: `RuntimeError("Maximum recursion depth exceeded...")`
  - So the program crashes before the OS does.
  - You can change the limit with `sys.setrecursionlimit(1000)`
    - But then you risk crashing more than just your program!

Recursive definitions

When solving a problem recursively, it helps to write out the definition of the problem recursively.

Recursive definitions

When solving a problem recursively, it helps to write out the definition of the problem recursively. This is usually the hard part.

Recursive definitions

When solving a problem recursively, it helps to write out the definition of the problem recursively. This is usually the hard part.

Consider the Fibonacci sequence:

Fib(0) = 1, Fib(1) = 1, Fib(2) = 2,
3, 5, 8, 13, 21, 34, ...

What's the pattern?

Recursive definitions

When solving a problem recursively, it helps to write out the definition of the problem recursively. This is usually the hard part.

Consider the Fibonacci sequence:
Fib(0) = 1, Fib(1) = 1, Fib(2) = 2,
3, 5, 8, 13, 21, 34, ...

What's the pattern?

- **Recursive case**: $\text{Fib}(n) = \text{Fib}(n-1) + \text{Fib}(n-2)$

Recursive definitions

When solving a problem recursively, it helps to write out the definition of the problem recursively. This is usually the hard part.

Consider the Fibonacci sequence:
$\text{Fib}(0) = 1$, $\text{Fib}(1) = 1$, $\text{Fib}(2) = 2$,
3, 5, 8, 13, 21, 34, ...

What's the pattern?

- **Recursive case**: $\text{Fib}(n) = \text{Fib}(n-1) + \text{Fib}(n-2)$
  - When does this definition work?

Recursive definitions

When solving a problem recursively, it helps to write out the definition of the problem recursively. This is usually the hard part.

Consider the Fibonacci sequence:
$\mathrm{Fib}(0) = 1$, $\mathrm{Fib}(1) = 1$, $\mathrm{Fib}(2) = 2$,
3, 5, 8, 13, 21, 34, ...

What's the pattern?

- **Recursive case**: $\mathrm{Fib}(n) = \mathrm{Fib}(n-1) + \mathrm{Fib}(n-2)$
  - When does this definition work? When $n \geq 2$.

Recursive definitions

When solving a problem recursively, it helps to write out the definition of the problem recursively. This is usually the hard part.

Consider the Fibonacci sequence:
$\text{Fib}(0) = 1$, $\text{Fib}(1) = 1$, $\text{Fib}(2) = 2$,
3, 5, 8, 13, 21, 34, ...

What's the pattern?

- **Recursive case**: $\text{Fib}(n) = \text{Fib}(n-1) + \text{Fib}(n-2)$
  - When does this definition work? When $n \geq 2$.
- **Base case**:

Recursive definitions

When solving a problem recursively, it helps to write out the definition of the problem recursively. This is usually the hard part.

Consider the Fibonacci sequence:
$\text{Fib}(0) = 1$, $\text{Fib}(1) = 1$, $\text{Fib}(2) = 2$,
3, 5, 8, 13, 21, 34, ...

What's the pattern?

- **Recursive case**: $\text{Fib}(n) = \text{Fib}(n-1) + \text{Fib}(n-2)$
  - When does this definition work? When $n \geq 2$.
- **Base case**: actually, there are two!

Recursive definitions

When solving a problem recursively, it helps to write out the definition of the problem recursively. This is usually the hard part.

Consider the Fibonacci sequence:
$\text{Fib}(0) = 1$, $\text{Fib}(1) = 1$, $\text{Fib}(2) = 2$,
3, 5, 8, 13, 21, 34, ...

What's the pattern?

- **Recursive case**: $\text{Fib}(n) = \text{Fib}(n-1) + \text{Fib}(n-2)$
  - When does this definition work? When $n \geq 2$.
- **Base case**: actually, there are two!
  - $\text{Fib}(0) = 1$
  - $\text{Fib}(1) = 1$

Recursive definitions

When solving a problem recursively, it helps to write out the definition of the problem recursively. This is usually the hard part.

Consider the Fibonacci sequence:
$\mathrm{Fib}(0) = 1$, $\mathrm{Fib}(1) = 1$, $\mathrm{Fib}(2) = 2$,
3, 5, 8, 13, 21, 34, . . .

What's the pattern?

- **Recursive case**: $\mathrm{Fib}(n) = \mathrm{Fib}(n-1) + \mathrm{Fib}(n-2)$
  - When does this definition work? When $n \geq 2$.
- **Base case**: actually, there are two!
  - $\mathrm{Fib}(0) = 1$
  - $\mathrm{Fib}(1) = 1$
- Each recursive call brings us closer to the base cases.
  - As long as n isn't negative, anyway.

Recursive definitions

When solving a problem recursively, it helps to write out the definition of the problem recursively. This is usually the hard part.

Consider the Fibonacci sequence:
$\mathrm{Fib}(0) = 1$, $\mathrm{Fib}(1) = 1$, $\mathrm{Fib}(2) = 2$,
3, 5, 8, 13, 21, 34, ...

What's the pattern?

- **Recursive case**: $\mathrm{Fib}(n) = \mathrm{Fib}(n-1) + \mathrm{Fib}(n-2)$
  - When does this definition work? When $n \geq 2$.
- **Base case**: actually, there are two!
  - $\mathrm{Fib}(0) = 1$
  - $\mathrm{Fib}(1) = 1$
- Each recursive call brings us closer to the base cases.
  - As long as n isn't negative, anyway.
- Now that we have a recursive definition, writing the code is easy.

Recursive definitions

When solving a problem recursively, it helps to write out the definition of the problem recursively. This is usually the hard part.

Consider the Fibonacci sequence:
$\text{Fib}(0) = 1$, $\text{Fib}(1) = 1$, $\text{Fib}(2) = 2$,
3, 5, 8, 13, 21, 34, ...

What's the pattern?

- **Recursive case**: $\text{Fib}(n) = \text{Fib}(n-1) + \text{Fib}(n-2)$
  - When does this definition work? When $n \geq 2$.
- **Base case**: actually, there are two!
  - $\text{Fib}(0) = 1$
  - $\text{Fib}(1) = 1$
- Each recursive call brings us closer to the base cases.
  - As long as n isn't negative, anyway.
- Now that we have a recursive definition, writing the code is easy.

The Fibonacci sequence in code

$$
\begin{aligned}
\mathsf{Fib}(0) &= 1 \\
\mathsf{Fib}(1) &= 1 \\
\mathsf{Fib}(n) &= \mathsf{Fib}(n-1) + \mathsf{Fib}(n-2) \qquad \text{where } n > 1
\end{aligned}
$$

The Fibonacci sequence in code

$$
\begin{aligned}
\mathsf{Fib}(0) &= 1 \\
\mathsf{Fib}(1) &= 1 \\
\mathsf{Fib}(n) &= \mathsf{Fib}(n-1) + \mathsf{Fib}(n-2) \qquad \text{where } n > 1
\end{aligned}
$$

```
def fibonacci(n):
    # Base cases.
    if n == 0 or n == 1:
        result = 1
```

The Fibonacci sequence in code

$$\mathrm{Fib}(0) = 1$$
$$\mathrm{Fib}(1) = 1$$
$$\mathrm{Fib}(n) = \mathrm{Fib}(n-1) + \mathrm{Fib}(n-2) \qquad \text{where } n > 1$$

```
def fibonacci(n):
    # Base cases.
    if n == 0 or n == 1:
        result = 1
    else:
        # Recursive case.
        result = fibonacci(n - 1) + fibonacci(n - 2)
    return result
```

The Fibonacci sequence in code

$$\begin{aligned} \mathrm{Fib}(0) &= 1 \\ \mathrm{Fib}(1) &= 1 \\ \mathrm{Fib}(n) &= \mathrm{Fib}(n-1) + \mathrm{Fib}(n-2) \qquad \text{where } n > 1 \end{aligned}$$

```
def fibonacci(n):
    # Base cases.
    if n == 0 or n == 1:
        result = 1
    else:
        # Recursive case.
        result = fibonacci(n - 1) + fibonacci(n - 2)
    return result
```

Recursion and the call stack

- Every recursive call adds a new entry to the call stack.
  - When that recursive call returns, the entry is removed.

Recursion and the call stack

- Every recursive call adds a new entry to the call stack.
  - When that recursive call returns, the entry is removed.
- So you'll have the same function on the call stack many times.

Recursion and the call stack

- Every recursive call adds a new entry to the call stack.
  - When that recursive call returns, the entry is removed.
- So you'll have the same function on the call stack many times.
  - Each **instance** of the function has its own parameters, local variables, and return value.

Recursion and the call stack

- Every recursive call adds a new entry to the call stack.
  - When that recursive call returns, the entry is removed.
- So you'll have the same function on the call stack many times.
  - Each **instance** of the function has its own parameters, local variables, and return value.
  - Variables are local to **one call** to the function.
- Let's observe the call stack in a recursive program using the debugger.

Recursion and the call stack

- Every recursive call adds a new entry to the call stack.
  - When that recursive call returns, the entry is removed.
- So you'll have the same function on the call stack many times.
  - Each **instance** of the function has its own parameters, local variables, and return value.
  - Variables are local to **one call** to the function.
- Let's observe the call stack in a recursive program using the debugger.