8 recursion and dictionaries

Beyond Iteration and Indices

It's time to take your computational thinking up a notch.

And this is the chapter to do it: we've been happily coding along with an iterative style of programming—we've created data structures like lists and strings and ranges of numbers, and we've written code to compute by iterating over them. In this chapter we're going to look at the world differently, first in terms of computation, and then in terms of data structures. Computationally we'll look at a style of computing that involves writing code that *recurs*, or calls itself. We'll expand the kinds of data structures we can work with by looking at a dictionary-like data type that is more like an *associative map* than a list. We'll then put them together and cause all kinds of trouble. Be forewarned: these topics take a while to settle into your brain, but the effort is going to pay off in spades.

A different way to compute

It's time for some mind-bending activity—you've been thinking about the same, iterative style of programming for too long. So let's expose your brain to a totally different way of thinking about solving problems.

Before we get there, though, let's take a simple problem and think it through the way we have throughout this book. For instance, take a handy list of numbers you want to sum up; it could be any numbers, say the number of marbles you and each of your friends has in his or her pockets. Now, Python does have a `sum` function that can be used to sum a list of numbers:

We've got a list with each friend's count of marbles.

```
marbles = [10, 13, 39, 14, 41, 9, 3]

print('The total is', sum(marbles))
```

Here we use Python's built-in sum function to tally up the marbles.

But we're still learning about computation, so let's compute the sum the old-fashioned way (again, using what we've learned so far in this book) by writing code that uses iteration to tally the list. Like this:

Let's define a function to compute a sum of numbers.

```
def compute_sum(list):
    sum = 0

    for number in list:
        sum = sum + number

    return sum

print('The total is', compute_sum(marbles))
```

To compute the sum of a list of numbers, we start with a local variable, sum, set to zero, which will hold the running total.

Then we iterate through the list, and add each number to sum.

Finally, we return the sum.

Good, we got the same result.

Let's test this by calling compute_sum on our marbles list.

BRAIN POWER

Pretend the folks who developed the Python language decided to remove any form of iteration (like the `for` and `while` loops). But you still needed to sum a list of numbers, so could you do it without iteration?

No! You can't use the built-in sum function either.

And now the different way...

There's another approach that computer scientists (and some in-the-know coders) use to break down problems. At first, this approach may seem a little like magic (or sleight of hand), but let's get a feel for it by revisiting our problem of summing our marbles. Here's how the approach works: we come up with two cases for summing our list of numbers: a *base case*, and a *recursive case*.

base case

The base case is the simplest case you can think of. So what is the simplest list of numbers you can take the sum of? How about an empty list? What is its sum? Zero, of course!

An empty list

```
compute_sum([])
```

Here's the simplest case: if we have an empty list, then we know the sum is going to be 0.

recursive case

Now for the recursive case. With the recursive case we're going to solve a smaller version of the same problem. Here's how: we take the first item in the list, and add it to the sum of the rest of the list...

```
[10, 13, 39, 14, 41, 9, 3]
```

How can we reduce the problem a little? How about we just worry about the sum of a list that is one item smaller?

```
10 + compute_sum([13, 39, 14, 41, 9, 3])
```

We've made our problem a little smaller: to compute the sum of the list, we're going to add 10 to the sum of a slightly smaller list.

Now let's write some code for our two cases

Now that we have our base case and our recursive case, we're ready to code this new way of computing a sum. As we said up front, doing so is a little mind-bending for most, at least at first. So let's very slowly step through coding our new recursive sum function.

base case

For the base case our job is easy: we just need to see if the list is empty, and if so, return `0` as the sum of the list:

```
def recursive_compute_sum(list):
    if len(list) == 0:
        return 0
```

Here we check to see if the list is empty (in other words, if its size is 0), and if so, we return 0.

recursive case

The recursive case is less obvious. Let's take it a step at a time. We know that we're going to take the first item of the list and add it to the sum of the rest of the list. For clarity, let's first set up some variables to hold the first item and the remainder of the list (without the first item):

```
def recursive_compute_sum(list):
    if len(list) == 0:
        return 0
    else:
        first = list[0]
        rest = list[1:]
```

Here's our base case again.

Let's set a variable to the first item in the list, and another one to the rest of the list.

Remember your list notation? This returns a list starting at index 1 through the last element in the list.

What is the value of rest if the list only has one item? It's the empty list.

Now we need to add the first item to the sum of the remainder of the list:

```
def recursive_compute_sum(list):
    if len(list) == 0:
        return 0
    else:
        first = list[0]
        rest = list[1:]
        sum = first + Sum of rest of list
```

We need to sum the rest of the list, but isn't that exactly what we're coding? A way to sum lists? It feels like a conundrum.

The sum is the first item plus the sum of the rest of the list.

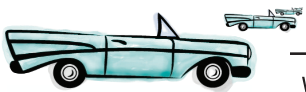
But how do we code this?

If only we knew how to compute the sum of the rest of the list, we'd be set. But how? Well, do you know of any good functions sitting around ready to compute the sum of a list? How about `recursive_compute_sum`?

```
def recursive_compute_sum(list):
    if len(list) == 0:
        return 0
    else:
        first = list[0]
        rest = list[1:]
        sum = first + recursive_compute_sum(rest)
        return sum
```

Let's not forget to return the sum after we've computed it!

Our assumption up front was that the recursive_compute_sum computes the sum of lists, so let's call it to finish the job on the slightly smaller list.



A Test Drive

Whether you believe this code will work or not, go ahead and get the `recursive_compute_sum` code (repeated below, including some test code) into a file called *sum.py*. Save your code and choose the **Run > Run Module** menu item. After that, head to the console to see the sum magically computed.

```
marbles = [10, 13, 39, 14, 41, 9, 3]

def recursive_compute_sum(list):
    if len(list) == 0:
        return 0
    else:
        first = list[0]
        rest = list[1:]
        sum = first + recursive_compute_sum(rest)
        return sum

sum = recursive_compute_sum(marbles)
print('The total is', sum)
```

We got the same result we did from iteration!

Python 3.6.0 Shell

```
The total is 129
>>>
```

No. Remember a function body is not evaluated until the function is called. So, in this code, the function `recursive_compute_sum` is first defined. Then, when the `recursive_compute_sum` function is called with:

```
sum = recursive_compute_sum(marbles)
```

the function's body is then evaluated and calls itself to recur. When that happens, the `recursive_compute_sum` function is already defined, so we are not violating that rule.

If you're finding this takes a bit to wrap your head around, that's normal. The trick is deliberate practice: write as many recursive functions as you can. Trace through the execution and understand how and why recursive functions work.

We'll be tracing through some recursive code in just a bit.

On that topic, let's get some more practice...

Let's get some more practice

Have no fear: getting your brain to think recursively takes a little extra effort, but it's well worth the blood, sweat, and tears (you think we're kidding). Now, we could stop at this point and analyze the `recursive_compute_sum` to death, but the best way to get your brain thinking more recursively is deliberate practice: take problems and solve them recursively, and, of course, write the code.

Let's practice on another problem. Remember those palindromes from Chapter 4? You'll recall that palindromes are words that read the same forward as they do backward, like "tacocat":

Reads the same forward...

tacocat

...as it does backward.

Want some more examples? How about "madam" or "radar" or "kayak," or there are even whole phrases (assuming you remove the punctutation and whitespace), like "a nut for a jar of tuna" or "a man, a plan, a canal: panama" or even more impressive, "a man, a plan, a cat, a ham, a yak, a yam, a hat, a canal: panama." Don't believe the last few? Try them; they're palindromes, alright.

Sharpen your pencil

Forget recursion for a bit, and think through how you might write a function to check if a word is a palindrome. Do that using the skills you learned in Chapters 1 through 7. Write some simple pseudocode to summarize your thoughts. Or, if you just had that cup of java and feel like writing some code, don't let us get in your way.

Using recursion to detect palindromes

So can we write a recursive function to detect palindromes? And if so, will we have gained anything? Let's give it a try and find out. Do you remember what to do? To write a recursive function we need a base case, and then we need a case that recurs by reducing the problem and then calling the same function recursively. Let's figure out the base and recursive cases:

base case

The base case is the simplest case we can think of. We can actually think of two simple cases. First, how about an empty string? Is the empty string a palindrome? It reads the same front and backward, so yes.

The empty string

```
is_palindrome('')
```

Here's the simplest case: if we have an empty string, then we know it's a palindrome.

But there's another really simple case to consider: the case of a single letter. Is a single letter a palindrome? It's the same forward and backward, so yes.

```
is_palindrome('a')
```

A single letter is a palindrome too; after all, it is the same read forward and backward.

recursive case

Now for the recursive case. This is where things always get interesting. Remember, we want to reduce the problem size a little before asking our `is_palindrome` function to finish the job for us. How about we compare the outer two characters and if they are the same, we can then check to see if the middle of the word (which is a little smaller) is a palindrome?

Check the outermost characters to see if they are the same.

```
'tacocat'
 'acoca'
```

And let our is_palindrome function worry about whether the middle is a palindrome.

Writing a recursive palindrome detector

We've got our base case and our recursive case, so once again we're ready to write our recursive code. As is typical, our base case is going to be fairly trivial to implement. Then we just need to wrap our minds around the recursive case. As with computing sums, the trick is always to reduce the problem a little and to rely on a recursive call to solve the problem.

base case

For the base case our job is easy: we just need to see if the word is the empty string or has one character:

```
def is_palindrome(word):
    if len(word) <= 1:
        return True
```

Let's check our base case to see if the word is the empty string (len is 0) or has one character (len is 1), and if so, return True.

recursive case

Now for the recursive case. First we're going reduce the problem by checking the outer two characters. If they match, we have a palindrome if all the rest of the letters (inside the two characters) make a word that is a palindrome. If not, we're going to return `False`:

```
def is_palindrome(word):
    if len(word) <= 1:
        return True
    else:
        if word[0] == word[-1]:


        else:
            return False
```

Here's our base case again.

Compare the first character to the last character to make sure they are equal. If not, we'll return False.

We'll think through the recursive call in the next step.

Now we need to finish the recursive case. At this point the code has determined the two outer characters are equal, so we have a palindrome *if the middle of the word is a palindrome*, and that's exactly what we need to code.

```
def is_palindrome(word):
    if len(word) <= 1:
        return True
    else:
        if word[0] == word[-1]:
            return is_palindrome(word[1:-1])
        else:
            return False
```

If the two ends match, then we need to see if the middle of the word is a palindrome. Good thing we have a function to do that—let's call it.

Note that we need to return the result of calling is_palindrome, which will ultimately return True or False.

A Test Drive

Go ahead and get the `is_palindrome` code (repeated below, including some test code) into a file called *palindrome.py*; save your code and choose the **Run > Run Module** menu item. After that head to the console to see if it is correctly detecting palindromes. Feel free to add your own palindrome candidates to the test as well.

```
def is_palindrome(word):
    if len(word) <= 1:
        return True
    else:
        if word[0] == word[-1]:
            return is_palindrome(word[1:-1])
        else:
            return False

words = ['tacocat', 'radar', 'yak', 'rader', 'kayjak']
for word in words:
    print(word, is_palindrome(word))
```

Take a look through the code again. Is this clearer than the iterative version? What do you think?

Looks like it works!

Python 3.6.0 Shell

```
tacocat True
radar True
yak False
rader False
kayjak False
>>>
```

there are no Dumb Questions

Q: How do you know a recursive function will ever end?

A: In other words, if a function keeps calling itself, over and over, how does it ever stop? That's where the base condition comes in. The base condition acts as a piece of the problem we know we can solve directly, without the help of recursively calling the function again. So, when we hit the base condition, we know we've reached the point where the recursive calls stop.

Q: Okay, but how do we know if we'll ever get to the base case?

A: Remember each time we call the recursive case, we make the problem a little smaller before calling the function again. So, if you designed your code correctly, you can see that at some point, by making the problem repeatedly smaller, you will eventually reach the base case.

Q: I kinda get how we could call a function from itself—after all, it is just like any other function call—but how do all the parameters not get messed up? That is, each time I recursively call the function, the parameters are reassigned to a new set of arguments, right?

A: This is a very good question. You are right; each time you call a function, the parameters are bound to a set of arguments. To make matters worse, if it is a recursive call, we're calling the *same function*, and so those parameters are going to get rebound to other arguments—you'd think the whole thing would go haywire when those parameter values get overridden, right? Ah, but that isn't what happens. You see, Python and all modern languages keep track of every call to a function along with its corresponding set of parameters (and local variables). Hang tight; we're going to look at this in a sec.

Behind the Scenes

How is Python handling recursion and keeping track of all those calls to the same function? Let's take a look behind the scenes and see how `is_palindrome` is being computed by the Python interpreter.

```
def is_palindrome(word): 1
    if len(word) <= 1: 2
        return True
    else:
        first = word[0] 3
        last = word[-1]
        middle = word[1:-1]
        if first == last: 4
            return is_palindrome(middle) 5
        else:
            return False
```

Here's the code again. Notice that we added some local variables that make the code a bit clearer. It also allows us to see how the variables work behind the scenes.

Let's evaluate this statement.

```
is_palindrome('radar')
```

1 The first thing Python (or practically any language) does when it sees a function call is to create a data structure to hold its parameters and local variables. This is typically called a *frame*. Python first puts the value for the parameter `word` in the frame.

```
frame 1
  word = 'radar'
```

2 Next we see if the word has a length of 1 or less, which it doesn't.

3 Next we have three local variables that are created and set to the first, last, and middle portions of the word passed in. These values are added to the frame as well.

```
frame 1
  word = 'radar'
  first = 'r'
  last = 'r'
  middle = 'ada'
```

4 Next we check to make sure the first and last characters are equal, which they are, so we then recursively call `is_palindrome`:

```
return is_palindrome(middle)
```

Referring to frame 1, middle is 'ada'.

---

1 We're back to another function call, so we need a new frame to hold the parameters and local variables. Python stores the multiple frames like a stack of plates, putting one on top of the other. We refer to the set of frames as a stack or *call stack*. That name kinda makes sense, doesn't it?

```
frame 2
  word = 'ada'
frame 1
  word = 'radar'
  first = 'r'
  last = 'r'
  middle = 'ada'
```

2 Okay, next we can see this word is not <= 1 characters, so we move on to the `else` statement.

❸ Once again, we compute our local variables and add them to the frame.

❹ And we can see that the first and last characters are equal.

So, all that remains is to call `is_palindrome` again.

```
return is_palindrome(middle)
```

Referring to frame 2, middle is 'd'.

```
frame 2
  word = 'ada'
  first = 'a'
  last = 'a'
  middle = 'd'
frame 1
  word = 'radar'
  first = 'r'
  last = 'r'
  middle = 'ada'
```

---

❶ We're back to another function call, so we need a new frame to hold the parameters and local variables. At this point the `word` parameter is just the string `'d'`.

❷ Finally, the length of the `word` parameter is finally less than or equal to 1, meaning we return `True`. When the call returns, we remove, or *pop*, the top frame off the stack.

```
frame 3
  word = 'd'
frame 2
  word = 'ada'
  first = 'a'
  last = 'a'
  middle = 'd'
frame 1
  word = 'radar'
  first = 'r'
  last = 'r'
  middle = 'ada'
```

When we return from the function its frame is popped off the stack.

---

❺ Now we need to return the result of calling `is_palindrome`, which is `True`. So we pop another frame off the stack as we return `True`.

```
frame 2
  word = 'ada'
  first = 'a'
  last = 'a'
  middle = 'd'
frame 1
  word = 'radar'
  first = 'r'
  last = 'r'
  middle = 'ada'
```

When we return from the function its frame is popped off the stack.

---

❺ Again, now we need to return the result of calling `is_palindrome`, which is `True`. Note this is the only remaining frame that resulted from calls to `is_palindrome`, so we're done (again, with a result of `True`).

```
frame 1
  word = 'radar'
  first = 'r'
  last = 'r'
  middle = 'ada'
```

Again, we pop a frame off the stack.

When we return from the initial call to `is_palindrome`, we return the value `True`.

The stack is now clear of all calls to is_palindrome.

```
is_palindrome('radar')
```

Evaluates to True!

Sharpen your pencil

Try evaluating some recursive code yourself. How about using our `recursive_compute_sum` function?

```
def recursive_compute_sum(list):
    if len(list) == 0:
        return 0
    else:
        first = list[0]
        rest = list[1:]
        sum = first + recursive_compute_sum(rest)
        return sum

recursive_compute_sum([1, 2, 3])
```

Here's the code again.

And here we're calling the function.

`recursive_compute_sum([1, 2, 3])`

frame 1
list = [1, 2, 3]
first = 1
rest = [2, 3]

We did the first one for you. The parameter list is bound to the list [1,2,3] and then the local variables first and rest get computed and added to the frame.

`recursive_compute_sum([2, 3])`

frame 2
list =
first =
rest =
frame 1
list = [1, 2, 3]
first = 1
rest = [2, 3]

Trace through the rest of the computation and fill in the stack details.

`recursive_compute_sum([3])`

frame 3
list =
first =
rest =
frame 2
list =
first =
rest =
frame 1
list = [1, 2, 3]
first = 1
rest = [2, 3]

`recursive_compute_sum([])`

frame 4
```
list =
```
frame 3
```
list =
first =
rest =
```
frame 2
```
list =
first =
rest =
```
frame 1
```
list = [1, 2, 3]
first = 1
rest = [2, 3]
```

`recursive_compute_sum([3])`

frame 3
```
list =
first =
rest =
sum =
```
frame 2
```
list =
first =
rest =
```
frame 1
```
list = [1, 2, 3]
first = 1
rest = [2, 3]
```

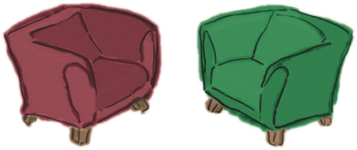
`recursive_compute_sum([2, 3])`

frame 2
```
list =
first =
rest =
sum =
```
frame 1
```
list = [1, 2, 3]
first = 1
rest = [2, 3]
```

`recursive_compute_sum([1, 2, 3])`

frame 1
```
list = [1, 2, 3]
first = 1
rest = [2, 3]
sum =
```

Fireside Chats



Tonight's talk: **Iteration and Recursion answer the question "Who's better?"**

Iteration: To know I'm better, all you have to do is look at how many times coders use iteration over recursion.

Recursion: I think that depends on the language you're talking about.

Iteration: What do you mean? Any modern language supports recursion, and yet coders opt to use me.

Recursion: Take a language like LISP or Scheme or Clojure, for instance—way more recursion is used than iteration.

Iteration: Last time I looked, this book was in Python.

Recursion: That's not the point. The point is, some programmers know and understand recursion very well, and see the beauty and efficiency of using it.

Iteration: Hah! Efficient? Ever heard of a call stack?

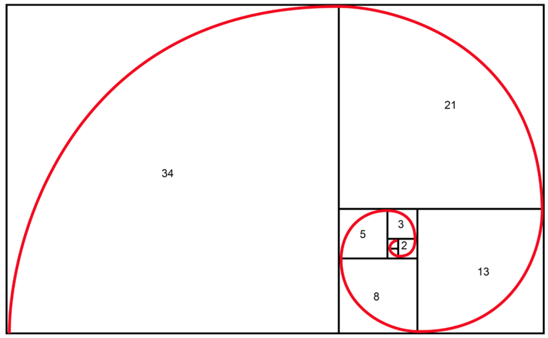
Recursion: Well, yes, and so have the readers, but please, do educate us.

Iteration: Every time a function calls itself, the Python interpreter has to create a little data structure to hold all the parameters and local variables of the current function. As the function gets called recursively, it has to maintain a whole stack of those data structures, which goes on and on as you keep calling the function over and over. Call it enough times and that adds up to a lot of memory, and then BOOM, your program goes bye bye.

Iteration	Recursion
	That's actually the way any modern (or ancient, for that matter) language, including Python, works. Anytime you call a function, that is happening.
Right, but as I said, when you do it recursively, it's like abusing the system, and sooner or later there's going to be trouble.	
	Not true. For many recursive algorithms, that isn't an issue and there are techniques for dealing with that, anyway. The point is, look at the clarity of using a recursive solution. Palindromes were a good example; look how ugly and unclear the iterative code was.
Hasn't stopped millions of coders from writing palindromes iteratively.	
	My point is, for some algorithms the recursive one is easier to think about and code.
Sure, for those brainiacs who get recursion.	
	Oh please, as we've seen it just takes a little practice.
You have to admit, for a lot of problems, iterative solutions are better.	
	I wouldn't say better, I'd say more natural, but I'd also say for some problems recursion is more natural.
I say why bother for a little clarity?	
	It's not just that the code is more readable, it's that there are algorithms that are downright hard to code iteratively, but that work out easily and naturally with recursion.
Oh, you mean for those Earth-Shattering-Grand-Challenge-type problems like finding palindromes?	
	Of course not; however, maybe we'll see one before the end of this book. By the way, you don't find the fact we're talking about the book *in the book* slightly recursive? Recursion is everywhere.
You think talking about talking about the book in the book is...oh dear.	

RECURSION LAB

Today we're testing the code for a recursive algorithm that computes the *Fibonacci sequence*. The sequence produces a set of numbers that appear often in nature and can describe shapes, like the pattern of seeds in a sunflower or the shape of galaxies.



The Fibonacci sequence is related to the Golden Ratio, which appears often in nature and is considered by many artists to be related to good design.

It works like this:

fibonacci(0) = 0
fibonacci(1) = 1

If you evaluate the function with 0 you get 0, and if you evaluate it with 1 you get 1.

fibonacci(n) = fibonacci(n-1) + fibonacci(n-2)

And for any other number, n, we produce the Fibonacci number by adding fibonacci(n–1) to fibonacci(n–2).

Here are a few values from the sequence:

fibonacci(0) is 0
fibonacci(1) is 1
fibonacci(2) is 1
fibonacci(3) is 2
fibonacci(4) is 3
fibonacci(5) is 5
fibonacci(6) is 8

Every number in the sequence is computed by adding the two Fibonacci numbers before it.

and continuing from there... 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181...and so on.

In the lab we've developed an algorithm to compute Fibonacci numbers. Let's take a look:

Working from the definition above...

```
def fibonacci(n):
    if n == 0:          # base case
        return 0
    elif n == 1:
        return 1
    else:               # recursive case
        return fibonacci(n-1) + fibonacci(n-2)
```

If n is 0 or 1, we just return that number.

Otherwise, we return the sum of the two previous Fibonacci numbers in the sequence, by recursively calling fibonacci.

Did you notice the recursive case calls fibonacci not once but twice!

Now it's time to test the code. Here in the Recursion Lab we need it to be correct and fast. To do that we've developed a little test code using a new module, `time`, which is going to help us time our code's execution.

```
import time
```

We're going to use Python's time module to time our code execution; see below.

```
def fibonacci(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

Here's the recursive Fibonacci code.

Test code.

```
for i in range(20, 55, 5):
    start = time.time()
    result = fibonacci(i)
    end = time.time()
    duration = end - start
    print(i, result, duration)
```

As a test we're going to compute the Fibonacci numbers 20 through 50, counting by fives. If that goes well, we'll compute all 100.

Start timer.

Compute the Fibonacci.

End timer.

Compute duration.

Print results.

We're also going to time each computation of Fibonacci. To do that we're going to use a module called time. See Appendix A for more on date and time modules.

You job is to get this code entered and to perform the test run. When you get the data, record it below, including the value of `n`, the Fibonacci number, and how long it took to compute, in seconds. <u>**For this code to be used in production, it has to compute the first 100 Fibonacci numbers in less than 5 seconds. Based on this test run, would we pass?**</u>

If this program is taking too long to execute, you can always stop it by closing the shell window it's running in.

Here's what we got for the first test, n=20. Your timings may differ depending on the speed of your computer.

Fibonacci Test Data

Number	Answer	Time to compute
20	6765	.002 seconds

So far it looks nice and fast!

Our results are on the next page; compare them with yours!

RECURSION LAB FAIL

To meet Recusion Lab standards, this code has to compute the first 100 Fibonacci numbers in less than 5 seconds. How did you do? What? You had lunch and they are still computing? No worries—we went ahead and computed the results. Our numbers are below, but they don't look encouraging at all.

The code is working great in that we're getting the right answers.

Fibonacci Test Data

Number	Answer	Time to compute
20	6765	0.002 seconds
25	75025	0.04 seconds
30	832040	0.4 seconds
35	9227465	4.8 seconds
40	102334155	56.7 seconds
45	1134903170	10.5 minutes
50	12586269025	1.85 hours

Your own results may vary depending on the speed of your machine.

But while the execution time started very fast, it is getting slower and slower the larger n is. At 50 we're taking almost 111 minutes to compute just that one Fibonacci number!

Uh oh. It doesn't look good. We're hoping we could really nail this Fibonacci code so that we could compute the first 100 numbers in the sequence in less than 5 seconds, but our sample test run shows the 50th number on its own takes over an hour!

Are we doomed? What on earth is taking so long? Give it some thought, and we'll come back to this after learning about an interesting data structure (maybe it will help us?).