Udacity CS101: Building a Search Engine
Unit 6: How to Have Infinite Power

Infinite Power

Welcome to unit 6! After this unit you will have learned all of the technical aspects that you will be tested on in the final exam. Unit 7 will consist of field trips and interviews, which will put what you have learned in context.

The big idea that will be introduced in unit 6 is recursive definitions, which you will learn how to use as a method for increasing your page ranking – being able to find the best page to respond to the query. The real goal of this unit is to give you infinite power!

Recall, that in unit 2 when you learned about procedures, you were told that the **`if`** statement gave you enough to write every possible computer program, which is infinitely powerful. Then, you learned how to use the **`while`** loop to go on. If you were infinitely powerful just knowing the **`if`** statement then you should not have needed to learn the **`while`** loop. You should have been able to build it from the things you have already seen – and it turns out that you can!

In this unit you will learn how to build up your own powerful control structures without using anything other than procedures. You will see that you can build up these control structures, as powerful as the **`while`** loop, from nothing but the procedures, if, and arithmetic and comparison operations that you learned in unit 2.

The point of learning this is not to be able to replace procedures, but to learn a new way of thinking called recursive definitions, which is a very powerful tool for solving problems.

Long Words

Q6-1: Long Words

This is kind of a trick quiz. Don’t worry if you’re not a native English speaker. This quiz is just as hard for them as it is for you!

What's the longest word in the English language?

a. honorificabilitudinitatibus
b. antidisestablishmentarianism
c. hippopotomonstrosesquippedaliophobia
d. pneumonoultramicroscopicsilicovolcanoconiosis
e. None of the above

Answer to Q6-1

Counter

A word is something that has meaning that is understood by the speakers of that words language. A word could be defined as what is in a dictionary, but there are a lot of things that are words, but that are not in the dictionary.

There is a rule that says that for a word, you can make a new word by adding counter in front of the old one. The notation used is the BNF (Backus Naur Form) replacement grammar, which was introduced in unit 1. If you need a refresher, please see unit 1, sections 9-11. Recall that the basic property of a BNF grammar is to replace what is on the left by what is on the right.

Word → counter-Word

The meaning of the new word is something that goes against, or counter to the original word.

If you start with the word intelligence, (intelligence as in spycraft not smart), you can use the rule to replace intelligence with counter-intelligence, which means trying the thwart the intelligence of the enemy.

You can continue, replacing counter-intelligence with counter-counter-intelligence , which means trying to thwart the enemy's counter-intelligence. Repeating this again, you get a word that isn't used but that still has a sensible meaning – counter-counter-counter-intelligence.

Word → counter-Word
intelligence
counter-intelligence
counter-counter-intelligence
counter-counter-counter-intelligence

One of the long words in the first quiz, hippopotomonstrosesquippedaliophobia, means fear of long words. If you know what a word means, even if you've never seen counter in front of it, you can guess the meaning. Counter-hippopotomonstrosesquippedaliophobia means something that goes against the fear of long words. It could be a medication that can cure someone from the fear of long words.

If you add another counter in front of counter-hippopotomonstrosesquippedaliophobia, then you get counter-counter-hippopotomonstrosesquippedaliophobia, which is something that goes against counter-hippopotomonstrosesquippedaliophobia. Maybe coffee stops the medication from working and so coffee is a counter-counter-hippopotomonstrosesquippedaliophobia!

Q6-2: Counter

If the only rule we have for making words is this one:

Word → counter-Word

how many words can we make, starting from Word.

a. None
b. 1
c. 2
d. Infinitely Many

Answer to Q6-2

Q6-3: Expanding our Grammar

For this quiz, an extra rule is added.

How many different words can we make starting from Word using only these two rules:

Word → counter-Word
Word → hippopotomonstrosesquippedaliophobia

a) None
b) 1
c) 2
d) Infinitely Many

Answer to Q6-3

Recursive Definitions

Recursive definitions work for things other than words, but you're probably most familiar with them from language. You will learn how to use them in procedures and in later courses you will see how to use them to define data structures. A lot of things in computing are defined in terms of recursive definitions.

A recursive definition has two parts: the base case and the recursive case.

In the previous example, with respect to Word, the second rule:

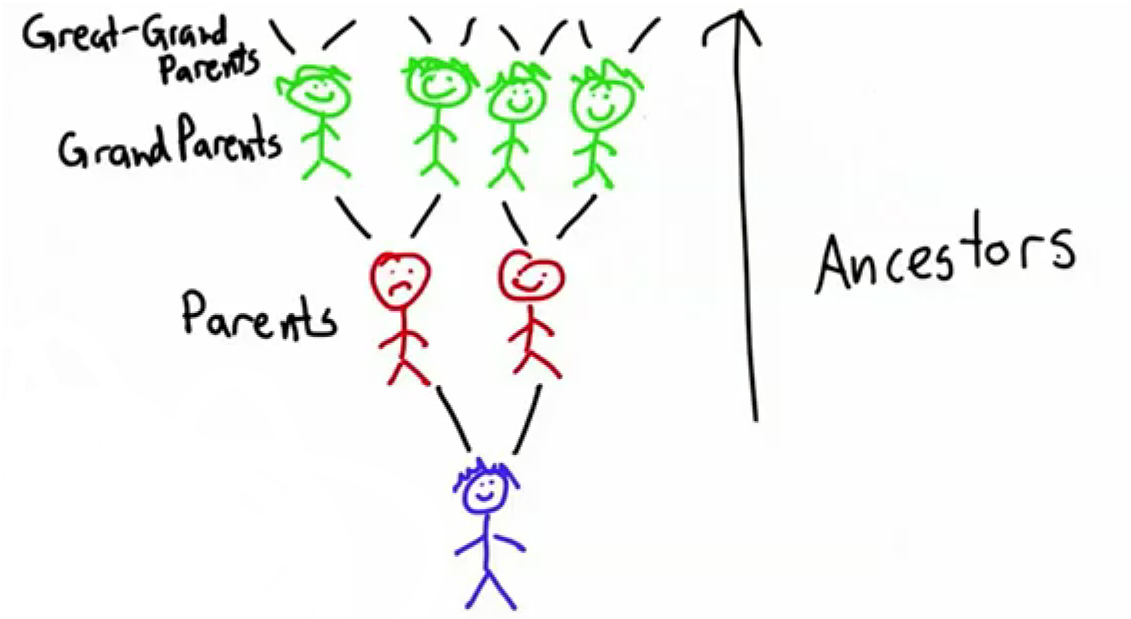
Word → hippopotomonstrosesquippedaliophobia

was the base case. It is a starting point and it is important that it is not defined in terms of itself. For programs it is usually going to be the smallest input, or the simplest input. The base case must be something that you already know how to define. You must already know the answer and not need to do anything to work it out.

The recursive case *is* defined in terms of itself, but not itself exactly. It is defined in terms of a smaller version of itself, as progress must be made towards the base case. You will see what this means in programs soon, but first another example – not in terms of a program – to get a better idea of how things can be defined recursively.

Ancestors

How can you define who your ancestors are?



Your parents are your ancestors, but they are not your only ancestors. Your parents have parents – your grandparents, who are also your ancestors. Your grandparents also have parents who are your ancestors too, and so on.

Q6-4: Ancestors

Which of these is the *best* definition of ancestors?

a. Ancestor → Parent of Ancestor

b. Ancestor → Parent
Ancestor → Parent of Ancestor

c. Ancestor → Parent
Ancestor → Parent of Parent
Ancestor → Parent of Parent of Ancestor

Answer to Q6-4

Recursive Procedures

You have seen how to use recursive definitions to make Words and define concepts like Ancestors. Now you'll see how to use recursive definitions to define a procedure. In unit 2, the factorial n was defined as the number of ways to arrange n items, which means that the input is n. This can be calculated as:

$$factorial(n) = n * (n-1) * (n-2) * \text{[?][?][?]} * 1$$

This equation is not a very precise mathematical definition because of the dot, dot, dot. Humans understand it correctly, but it's not precise mathematically. Using a recursive definition allows a factorial to be defined precisely. For this, a base case is needed. This should be the simplest input. It is something for which the answer is already known. For factorial and for many procedures involving numbers, the simplest input is 0. The factorial of 0 is defined as 1, that is:

$$factorial(0) = 1,$$

which is the base case.

In the imprecise definition:

$factorial(n) = n * (n-1) * (n-2) *$ [?][?][?] $* 1,$

the product:

$(n-1) * (n-2) *$ [?][?][?] $* 1$is just the factorial of $n-1,$

that is: $factorial(n-1).$

This means you can write the recursive case as:

for any integer $n>0$, $factorial(n) = n * factorial(n-1).$

It makes sense when you think of the way to arrange n items. There are n ways to pick the first item, and then there are $n-1$ items remaining. There are $factorial(n-1)$ ways to arrange these $n-1$ items.

Base case: $factorial(0) = 1$

Recursive case: $factorial(n) = n * factorial(n-1)$ for $n>0$

Q6-5: Recursive Factorial

Define a procedure, **`factorial`**, that takes a natural number as its input, and outputs the number of ways to arrange the input number of items.

You've already seen how to do this using a while loop. Your goal here is to define that procedure without using a while loop, that is, to define it using a recursive definition.

Note that a **natural number** is defined to be a positive whole number.

Answer to Q6-5

Palindromes

Here's another example of defining a recursive procedure. A **palindrome** is a string that reads the same way forwards and backwards.

For example: String 'level' is a palindrome (if you read 'level' forwards, you get 'level' and if you read it backwards, you get exactly the same string). Some other typical examples of palindromes include:

- Any single letter is a palindrome. For e.g. 'a' (If you read 'a' forwards, you get 'a' and if you read 'a' backwards, you again get 'a')
- An empty string is also a palindrome (i.e. ''). If you read empty string '' forwards, you get the empty string and if you read empty string backwards, you get an empty string.

Additionally:

- http://norvig.com/palindrome.html: This is an encyclopediac text on this topic by Prof. Peter Norvig.
- http://en.wikipedia.org/wiki/Palindrome#Long_palindromes: You might also want to include some interesting palindromes/facts from this Wikipedia page.

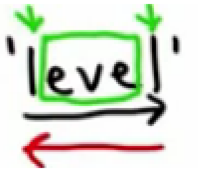
Q6-6: Palindromes

Define a procedure, **is_palidrome**, that takes as input a string, and outputs a Boolean to indicate if the input string is a palindrome.

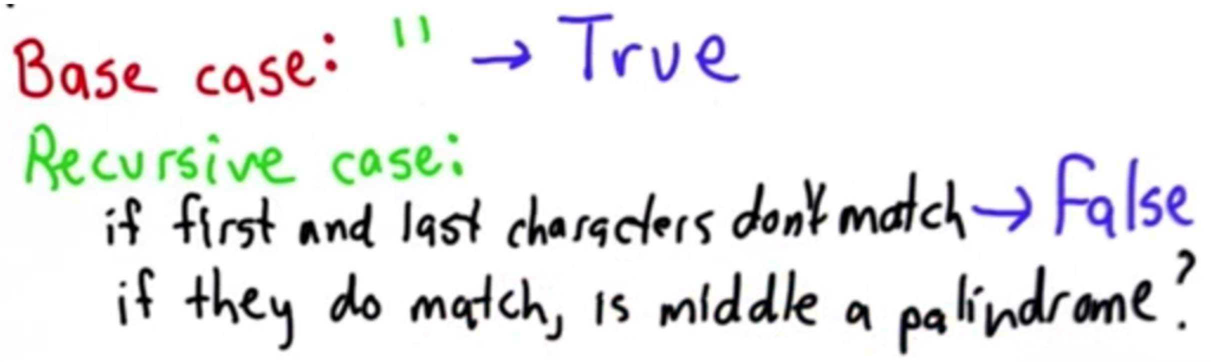
First, try to think on your own and define the procedure that tests whether an input string is a palindrome or not. This is a pretty tough question. There are easy ways to reverse the string and check to see if it is same as the original string in Python. But since you have not studied them yet in this course, write this procedure using what you already know about Python. Here are some hints for defining this procedure:

- You might have already noticed that there is one simple case (i.e. empty string '') where you know that the string is a palindrome. So, this will be your base case. If the input to the procedure, **is_palindrome** is an empty string, the result of **is_palindrome** is *True*.
  Note: When you write recursive procedures on numbers, the base case is often some small number (like 0 or 1). When you write recursive procedures on strings, the base case is more likely to be the simplest string (which is the empty string '').

- What will you do if the string is not an empty string? Well, one way you can solve this is by looking at the first and last letter of string. If these two are equal, then it might be a palindrome. It will be a palindrome only if all the letters left over in the middle are also a palindrome. In order to check whether remaining middle of string is a palindrome, you can recursively call **is_palindrome.**



This can be summarized as follows:



Base case: '' → True

Recursive case:

if first and last characters don't match → False

if they do match, is middle a palindrome?

[Answer to Q6-6](#)

Recursive v. Iterative

Any procedure that you write recursively, you can also write without using a recursive definition. Here is another way to define **`is_palindrome`**:

```
def iter_palindrome(s):
    for i in range(0, len(s) / 2):
        if s[i] != s[-(i + 1)]:
            return False
    return True
```

`iter_palindrome` is written using a **`for`** loop. You loop using variable **`i`** in range 0 to length of input string divided by 2 (i.e. this loop is going through halfway of string **`s`**). Inside the loop, there is an **`if`** test that checks if the character at position **`i`** is different from the character in position **`-(i + 1)`** (i.e. counting from back of string ith positions away). If those characters are different, you have bumped into a mismatch and we return *False*. If they are not different, continue going through the loop. If you reach the end of the **`for`** loop without finding any differences, you know that it is a palindrome and you return *True*.

This alternative way to define **is_palindrome** is more complicated to understand. If you want to test a very long palindrome, **iter_palindrome** (i.e. iterative version) will be much more efficient than **is_palindrome** (recursive version). There are a few reasons why:

1. Inside **is_palindrome**, when we call is_palindrome recursively:

   ```
   return is_palindrome(s[1:-1])
   ```

   This recursive version keeps making a new string every time you make a recursive call. This creates a new string and this step is pretty expensive.

2. Recursive calls themselves are fairly expensive themselves. There are languages which make recursive calls really cheap. Python is not one of them and Python is fairly expensive to do a recursive call. For most procedures, the recursive way is often the most elegant and easiest way to return a correct result. But if you are worried about performance and want your procedure to work on really large inputs, you are better off finding a non-recursive way to define that procedure.

Bunnies

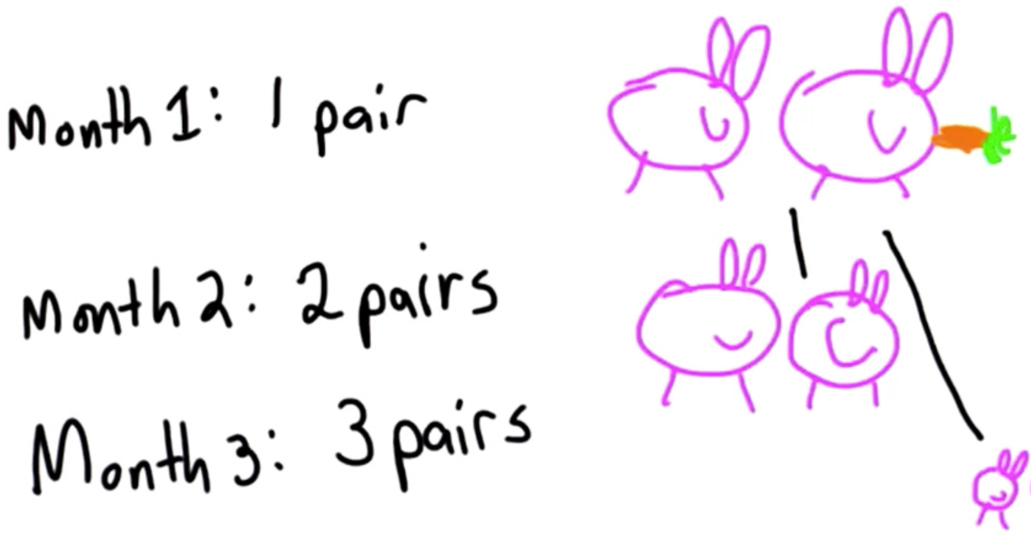
Fibonacci Numbers are one of the most interesting things in mathematics Once you know about them you will start to see them all over the place, both in nature and design.

The name comes from Leonardo da Pisa, who is also known as Fibonacci. In 1202 he published a book called, *Liber Abaci*. The root, abaci, is the same for the word abacus, the calculating machine. *Liber Abaci* is loosely translated as the "book of calculation." The book introduced Indian mathematics to the West, particularly, Arabic numerals. Arabic numerals soon replaced the Roman numeral system, which had been widely used. In his book, Fibonacci showed how much easier it is to do calculations using numbers in the decimal system where the position of the number indicates its value. He showed this by introducing problems and using calculation to solve them.

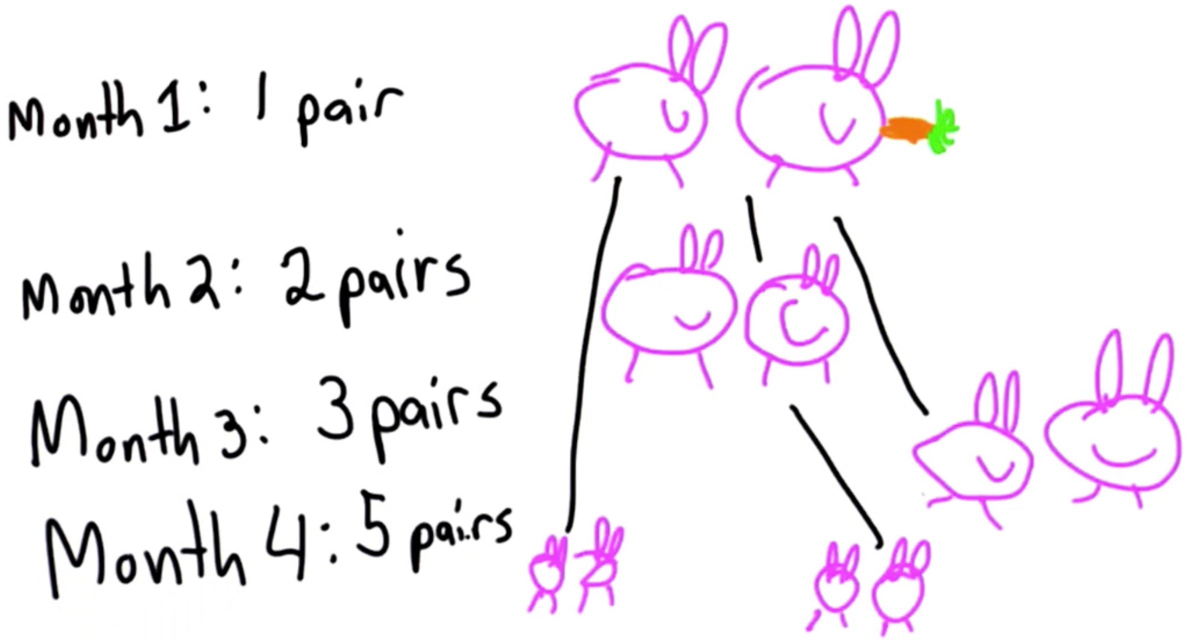
The problem that became known as the Fibonacci Numbers, was one of the problems in his book. He posed the problem like this:

In the beginning there is one pair of rabbits. It takes one month for a rabbit to mature, and one month for a rabbit to produce offspring.

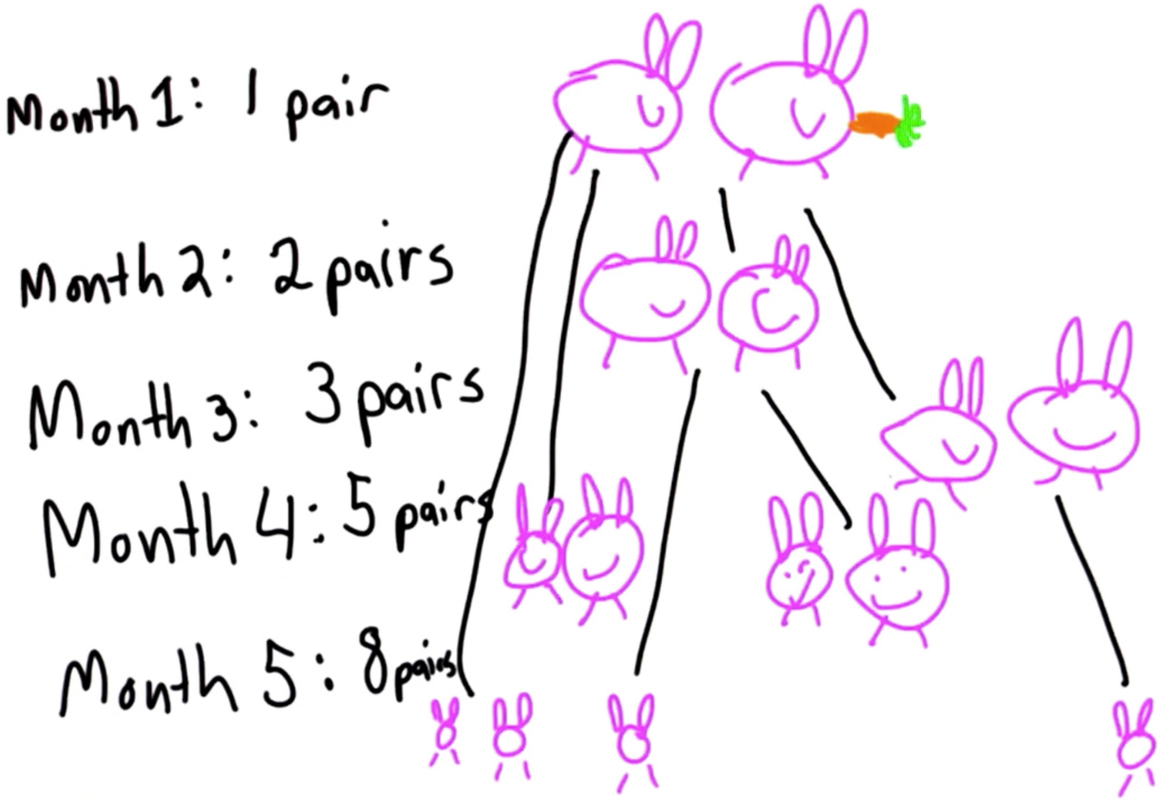
Every month a mature rabbit will produce a new pair of rabbits. Notice how in month three, since it takes a month for the rabbits to reach maturity (and only in maturity can the rabbits reproduce), there is only one set of offspring from the original pair of rabbits.



Assume that for the sake of this model, rabbits never die. In month four, one pair of baby rabbits will mature, while the two mature pairs of rabbits will reproduce. This makes a total of five pairs of rabbits.



This keeps going, since the model assumes rabbits never die, that every month a pair of mature rabbits produces a pair of rabbit babies and that it takes one month for rabbit babies to mature. Therefore, in month five the three mature pairs of rabbits will all produce a pair of offspring, making eight pairs of rabbits.



While this model is not realistic, it is an interesting mathematical model. This model can be written in a more formal way.

Each month the number of rabbits is the number of rabbit from the previous month, plus all of the rabbits that were mature, which is the number of rabbits from two months ago. After this observation you can predict the number of rabbit pairs there will be in month six using the same formula: the number of rabbit pairs in month five plus the number of rabbit pairs in month four. This makes 13 pairs of rabbits.

This is the model that Fibonacci developed. So, here is the question: can you figure out how many pairs of rabbits there will be at the end of month *n*, given any number *n*?

Here is how you can define this mathematically:

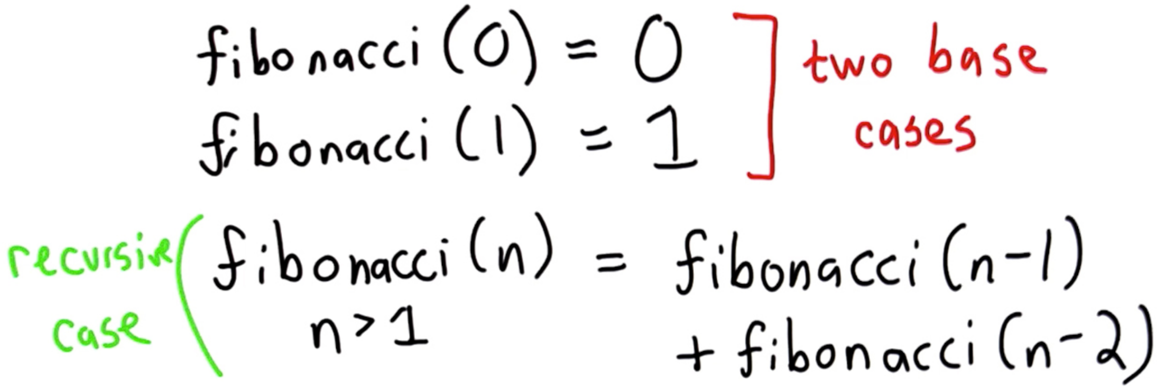
```
fibonacci (0) = 0
fibonacci (1) = 1
```

This is different from the other recursive definitions you've seen in that there are two base cases. From here you can define every other fibonacci number recursively, starting from these base cases. So, the fibonacci number *n*, where *n* is some whole number greater than one, is equivalent to the sum of all the babies from the previous month, **`fibonacci (n-1)`**, plus all of the new babies. The number of new babies is the number of rabbit pairs two months ago (all of the mature rabbit pairs), **`fibonacci (n-2)`**. You can write this:

$$n>1$$

$$fibonacci(n) = fibonacci\,(n-1) + fibonacci\,(n-2)$$

This equation defines every fibonacci number in terms of the two base cases and the one recursive case.



Q6-7: Bunnies

Define a procedure, **`fibonacci`**, that takes a natural number (any whole number zero or higher) as its input, and outputs the value of that fibonacci number.

Answer to Q6-7

Divide and Be Conquered

Here is the procedure you just defined as an answer to the previous quiz question:

```
def fibonacci(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    return fibonacci(n-1) + fibonacci(n-2)
print fibonacci(0) # print first base case
0
```

or

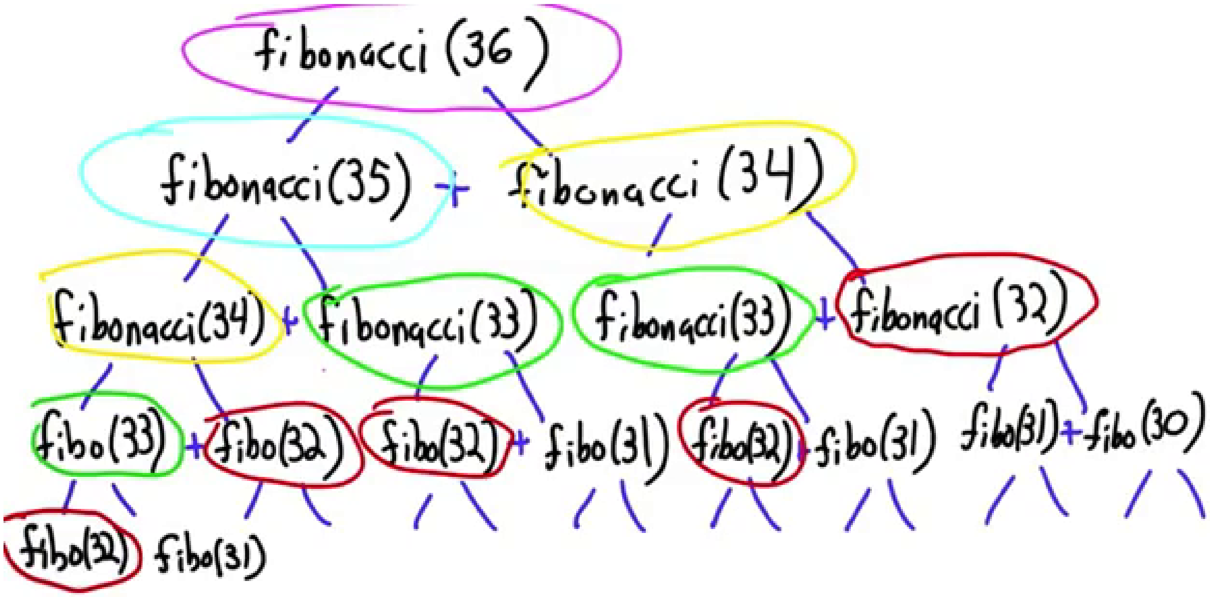
```
print fibonacci(1) # print second base case
1
```

and

```
print fibonacci(2) # plug 2 into final return statement
1
print fibonacci(3)
2
print fibonacci(4)
3
print fibonacci(5)
5
print fibonacci(10)
55
print fibonacci(24) # number of rabbits in two years
46368
print fibonacci(36) # number of rabbits in three years
Program times out in Professor’s/Udacity’s interpreter
```

If you try running the code above in your Python interpreter, you'll notice that the time to compute **`fibonacci`** this way gets longer as input numbers get larger. The reason for this is that you are making a lot of redundant computations. If you look at the procedure, you'll notice that **`fibonacci(n)`** recursively calls **`fibonacci(n - 1)`** and **`fibonacci(n - 2)`** every time the base case conditions are not satisfied.

At the start call **`fibonacci(36)`**, which is broken down into recursive calls to **`fibonacci(35)`** and **`fibonacci(34)`**. Now **`fibonacci(35)`** recursively calls **`fibonacci(34)`** and **`fibonacci(33)`**, while **`fibonacci(34)`** recursively calls **`fibonacci(33)`** and **`fibonacci(32)`**, and so on. See the tree below:



The procedure needs to do a lot of computations and it will take a long time to get calls to the base cases (**`fibonacci(0)`** and **`fibonacci(1)`**), which are the only places where the procedure stops making more recursive calls. If you look at the figure above, you'll notice that:

- We need to evaluate **`fibonacci(32)`** 5 times.
- We need to evaluate **`fibonacci(33)`** 3 times.
- We need to evaluate **`fibonacci(34)`** 2 times.
- We need to evaluate **`fibonacci(35)`** 1 time.
- We need to evaluate **`fibonacci(36)`** 1 time.

Do you see a pattern in the list above? Try to see if you can solve the next question related to this.

Q6-8: Counting Calls

How many times will you need to evaluate **`fibonacci(30)`** in evaluating **`fibonacci(36)`**?

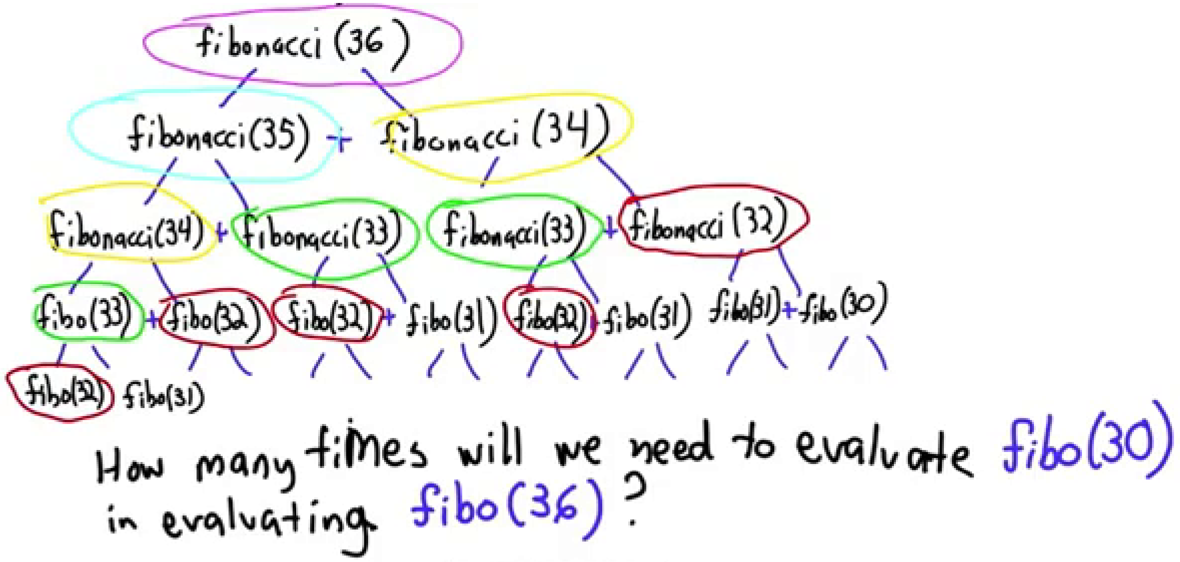

fibonacci (36)
fibonacci(35) + fibonacci (34)
fibonacci(34) + fibonacci(33) fibonacci(33) + fibonacci (32)
fibo(33) + fibo(32) fibo(32) + fibo(31) fibo(32) + fibo(31) fibo(31) + fibo(30)
fibo(32) fibo(31)
How many times will we need to evaluate fibo(30) in evaluating fibo(36)?

Figure this out without drawing the whole tree. Think about what you read in last section, do you notice a pattern in the figure above that could help you to answer this question.

Answer to Q6-8

Q6-9: Faster Fibonacci

Define a faster **`fibonacci`** procedure that will enable you to compute **`fibonacci(36)`**.

Your procedure should estimate the number of rabbits after 36 months, according to Fibonacci's model.

Hint: You'll need a **`while`** loop where you use variables to keep track of the previous two numbers, **`fibonacci(n-1)`** and **`fibonacci(n-2)`**, so you can compute the next one by adding those. You'll also need to figure out how to keep the variables up-to-date to maintain the previous two numbers each time you go through the loop.

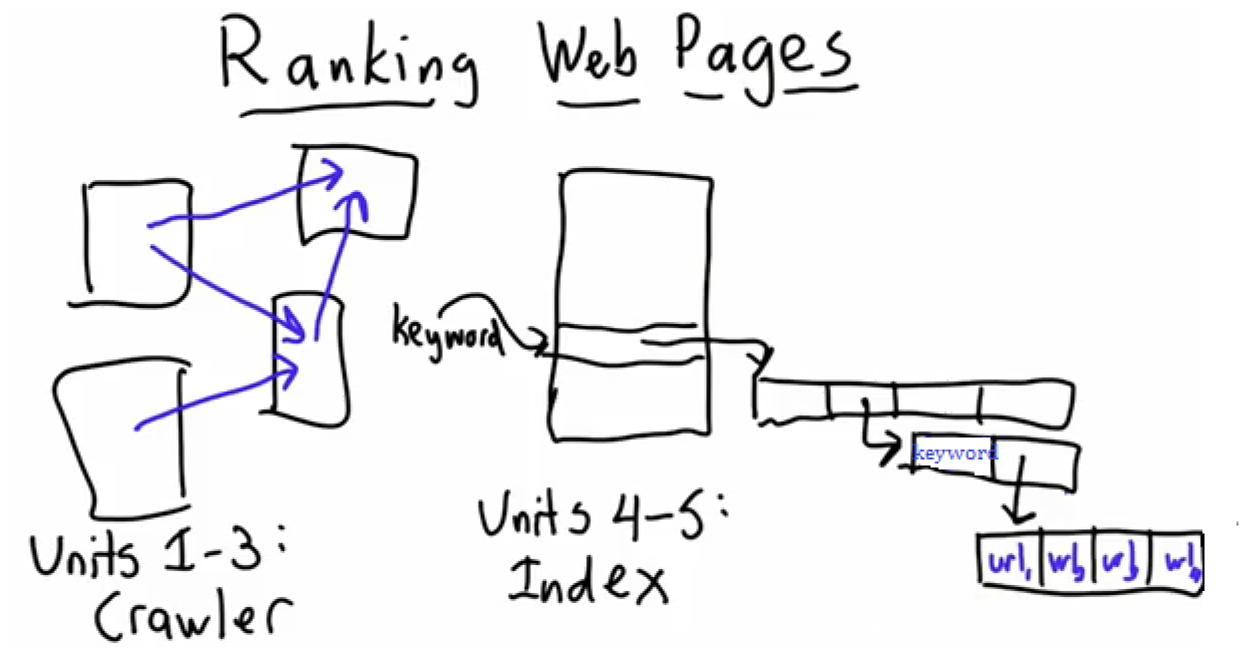
Test your code on small numbers before trying **`fibonacci(36)`**. If you do it this way, you should be able to compute values of the Fibonacci Numbers for much higher input thanx you could with a recursive definition.

Answer to Q6-9

Ranking Web Pages

Having survived the bunny uprising, you're ready to move on to the main goal of the class, which is to return the best page that matches the search query rather than returning all the pages. It's important to do this well. This is something that really distinguished Google from earlier search engines. They had a much smarter way of ranking pages. Often the first or second item in the returned search was what the user was looking for.

To recap from earlier units; first, you learned to build a crawler (units 1-3). The crawler followed all the links in the web pages and built an index. After units 4-5, you had an index, which was a hash table, where you could look up a keyword. You could find the entry where the keyword appears, and find the list of all the urls of all the pages that contain that keyword.

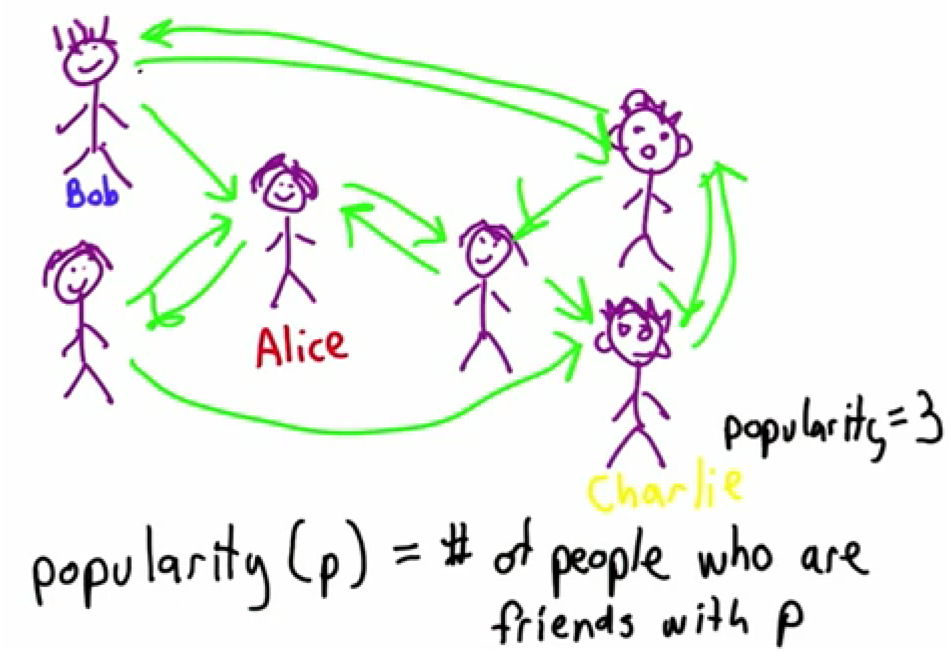


The order in which the pages appear in the list of urls associated with a keyword is the order the pages were crawled. This process says nothing about which pages are best. In the early days of the web, when there weren't many pages, this maybe wasn't too much of a problem since only a few pages might match a given keyword. Those days are long gone and now there could be thousands, if not millions of pages containing a given keyword. A good search engine ranks the pages so that the one at the front of the list is the one the user most likely wants.

The problem of deciding how to rank the pages leads to the question of how to decide popularity, which is the topic of the next section.

Popularity

Consider a typical group of friends in middle school. One way to decide popularity is to look at friendship links. Friendship links are go in one direction. Just because Bob is friends with Alice does not mean Alice is friends with Bob.



Is having a lot of friends enough to make you popular? No, it's not. You have to have the right sort of friends. It's no good to have lots of friends with no friends, you have to have friends who are popular.

Popularity is about having lots of friends who have lots of friends

Initially, you can define popularity as the number of friends a person has. From the diagram above, it's the number of arrows pointing towards a person.

$$popularity(p) = \#\ of\ people\ who\ are\ friends\ with\ p$$

This isn't quite right though because it doesn't take into consideration the number of friends of those friends. To do this, you could sum all the popularities of all the friends of a person. In mathematical notation:

$$popularity(p) = \sum_{f \in friends\, of\, p} popularity(f)$$

The notation $\sum$, is the summation sign, which tells you to sum up the popularity(f). The text under the symbol tells you what values of f to include in the sum. It tells you to sum the popularity of each friend, f of p. If you're unfamiliar with the mathematical notation, here's the same thing in Python pseudocode.
Pseudocode is an outline of the code which is written for human readability rather than for a computer. Read more on pseudocode.

In the code below, you know that friends **p** means the friends of **p**, but it's not actually defined anywhere so the computer will not be able to run it.

```
def popularity(p):
    score = 0
    for f in friends(p):
        score = score + popularity(f)
    return score
```

Q6-10: Good Definitions

Is this a good recursive definition? For something to be a good definition it has to provide a meaningful answer for all possible inputs.

a. Yes
b. No

Answer to Q6-10

Circular Definitions

How can this problem be fixed? With all the other recursive definitions you had a base case – a way to stop.

For the recursive factorial definition, you predefined the value at 0 to be 1, that is, **`factorial(0)`** →**`1`**.

For the palindromes, you defined an empty string to be a palindrome, that is, **`palindrome('')`**→ **`True`**.

For the Fibonacci sequence, you had two base cases.

For all of these definitions you had a starting point that was not defined in terms of the thing you're defining. That is why they were good recursive definitions. You had a base case. Maybe inventing a base case will solve the popularity problem.

Assume Alice has popularity 1, and try that as a base case. For the mathematical definition, this is:

$$
\begin{aligned}
&popularity([?][?][?]Alice[?][?][?]) = 1 \\
&popularity(p) = \sum_{f \in friends\,of\,p} popularity(f).
\end{aligned}
$$

For the code, you need to add the base case, which is an **if** statement, to see if the person you're checking the popularity of is Alice.

```
def popularity(p):
    if p == 'Alice':
        return 1
    score = 0
    for f in friends(p):
        score = score + popularity(f)
    return score
```

Q6-11: Circular Definitions

Would this definition work?

a. Only if everyone is friends with 'Alice'.
b. Only if no one is friends with 'Alice'.
c. Only if there is a friendship path from everyone to 'Alice'. (There is some way to follow links from every person in the graph to get to 'Alice')
d. Only if there are no cycles in the graph. (There is no way to start from one person and end up back at the same person by following friendship links.)
e. No.

Answer to Q6-11

Relaxation Algorithm (Relaxation)

There was no sensible base case that provides a good recursive definition. Instead, an algorithm called the **relaxation algorithm** can be used. The basic idea is simple. Start with a guess and then loop where you do something to improve the guess. There isn't a good stopping place yet, or a clear starting place, like setting the popularity of Alice. Each time you go through the loop, the guess will be refined, and at some point you'll stop and take that to be the result you want. In summary:

```
# start with a guess
while not done:
    make the guess better
```

The procedure will have an extra parameter, which is a time step:

```
popularity(<time step> ,<person>) → score
```

The base case will be to set the popularity for everyone at time 0 to 1. For the recursive step, the popularity of each of their friends at the *previous* time step, $t-1$is summed. In mathematical terms this is:

Base case: $popularity(0, p) = 1$

Recursive step: $popularity(t, p) = \sum_{f \in friends(p)} popularity(t-1, p)$ for $t > 0.$

In Python code:

```
def popularity(t,p):
    if t == 0:          # base case, at time step 0
        return 1        # the score is always 1
    else:
        score = 0
        for f in friends(p):                    # summing over the friends
            score = score + popularity(t-1,f)   # adding the popularity at
                                                # the time step before
        return score
```

So, now you have a new definition written in both mathematical notation and in Python code.

Q6-12: Relaxation

Is this a good recursive definition? (By this, it is not meant whether this is a good definition of popularity. For the way in which popularity is defined, for all possible inputs, that is, all possible values for **`t`** and **`p`**, does it give a result?)

a. yes
b. only if people can't befriend themselves
c. only is everyone has at least one friend
d. only if everyone is more popular than 'Alice'

Answer to Q6-12

Page Rank

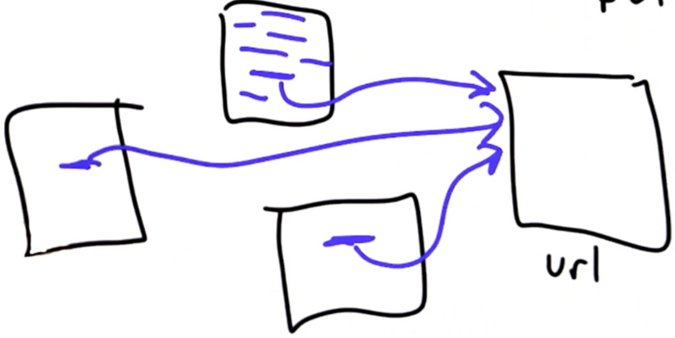
Ranking web pages is the same as measuring popularity for people. Links on the web are analogous to friendships in that model. And links from some pages count for more than links from others.

This model is a random web surfer who starts at a random page and then follows the links at random. The popularity of a page is the probability that the random surfer reaches a particular page.

The rank function is defined recursively over time. At time 0, the rank of a url is 1. At time **t**, the rank of a url is the sum of the ranks for all pages that link to that url.

$$\mathrm{rank}(\overset{\text{timestep}}{0}, \overset{\text{page}}{\mathrm{url}}) \rightarrow 1$$

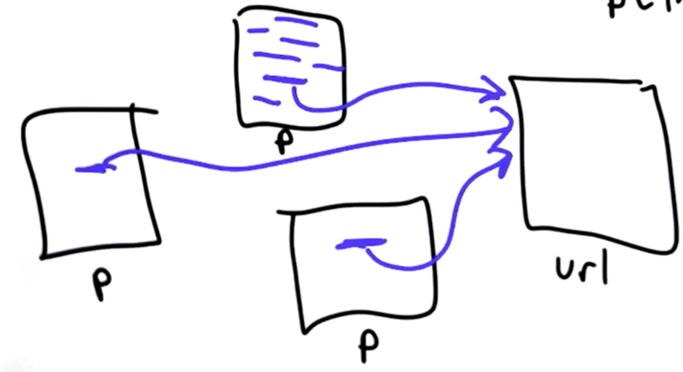
$$\mathrm{rank}(t, \mathrm{url}) \rightarrow \sum_{p \in \mathrm{inlinks}[\mathrm{url}]} \mathrm{rank}(t-1, p)$$



In addition, the rank contributed by each page is inversely weighted by the number of outlinks from that page. So, divide each rank in the sum by the number of outlinks from that page.

$$\text{rank}(\overset{\text{timestep}}{0}, \overset{\text{page}}{\text{url}}) \rightarrow 1$$

$$\text{rank}(t, \text{url}) \rightarrow \sum_{p \in \text{inlinks}[\text{url}]} \text{rank}(t-1, p) / \text{outlinks}[p]$$



Altavista

On this model, pages with no links have a rank of 0, which makes it very hard to start a new page. So, instead each page will have some starting rank greater than 0.

Since the model represents the probability that a random surfer reached a given page, the ranks should be a probability distribution. This means that the ranks for all pages will sum up to 1. And so at time 0, instead of 1, the rank is 1/N for each page, where N is the number of pages.

A **damping constant** is used to diminish the raw values of the ranking algorithm. In the model, this represents the probability that a page was reached via following a link. Set the damping constant, d, to 0.8 for now.

At time `t`, add (1-d)/N to each rank that will represent the probability that the page was not reached via following a link. Multiply the first term, the sum, by d. This will make the ranks a probability distribution at any given time step.

Q6-13: Altavista

What is AltaVista?

a. The view from the Udacity headquarters.
b. The most popular web search engine in 1998.
c. Spanish for "You're Terminated, Baby!"
d. A small town in Virginia.

Answer to Q6-13

Urank

Since PageRank is a registered trademark of Google, the algorithm will be called URank instead.

URank needs to keep track of which pages link to which pages, so you'll need a data structure to keep track of which pages link to which other pages. You'll use a directed graph. A **directed graph** is a data structure where nodes are linked to other nodes, and the links only go one way. (see http://en.wikipedia.org/wiki/Directed_graph)

So **`crawl_web`** will produce a graph in addition to an index, where the graph gives a mapping from each page to all the pages it links to. The graph will be a dictionary, since it is a mapping from individual URLs to lists of URLs.

Also, add the variable **`outlinks`** which stores the return value of **`get_all_links(content)`** so it can be used for both **`tocrawl`** and **`graph`**. Adding the line to update the graph will be a quiz.

Q6-14: Implementing Urank

For this quiz, add one line of code which will update the graph for each page crawled.

```
def crawl_web(seed): # returns index, graph of inlinks
    tocrawl = [seed]
    crawled = []
    graph = {}  # <url>, [list of pages it links to]
    index = {}
    while tocrawl:
        page = tocrawl.pop()
        if page not in crawled:
            content = get_page(page)
            add_page_to_index(index, page, content)
            outlinks = get_all_links(content)

            #Insert Code Here
            union(tocrawl, outlinks)
            crawled.append(page)
    return index, graph
```

Answer to Q6-14

Computing Page Rank

Much like the Fibonacci solution before, you can use an iterative solution to implement the recursive definition, instead of a recursive solution. Use a loop to update the page ranks for each time step.

The output of **`compute_ranks`** is a dictionary mapping each URL to its rank, which is a number.

For the homework, the function **`lookup_best`** will find the highest-ranking page for a given keyword, using both the index and the page ranks.

Formal Calculations

Remember how the ranking function was defined; **npages** refers to the number of pages:

```
rank(0, url) = 1/npages
rank(t, url) = (1-d)/npages +
  sum([for each page 'p' that links to URL,
    d*rank(t-1, p)/(number of outlinks from p)])
```

Since the ranks should not depend on the order that the pages were examined by the algorithm, you need to keep track of the ranks at the last time step. Keep two separate dictionaries, **ranks** and **newranks**, where **newranks** is the working space for each time step. This is similar to the trick used earlier for the iterative Fibonacci solution.

Computer Rank

The code for this section is shown below under the quiz. **d** is the damping factor. **numloops** is the number of times to do our "relaxation". Changing the number of loops can give different results. **npages** is the number of pages in the graph, which is given by **len(graph)**.

Initially, all ranks are set to **1.0/npages** (remember: the decimal point to use floating point arithmetic), matching the base case of the recursive definition.

Then, the algorithm loops through **numloops** times, updating the rank for each page in the graph. **newrank** is initialized to **(1-d)/npages**, and then the quiz will be to update **newrank** with the sum of the inlink ranks. Then **newrank** is stored in the **newranks** dictionary.

When the **for** loop is finished, assign **newranks** to **ranks**, since the calculations are finished for that time step. At the end of the function, return **ranks**.

Q6-15: Finishing Urank

Update **newrank** based on the values of the previous iteration, **ranks**, and the incoming links from **graph**.

```
def compute_ranks(graph):
    d = 0.8 # damping factor
    numloops = 10

    ranks = {}
    npages = len(graph)
    for page in graph:
        ranks[page] = 1.0 / npages

    for i in range(0, numloops):
        newranks = {}
        for page in graph:
            newrank = (1 - d) / npages

            #Insert Code Here

            newranks[page] = newrank
        ranks = newranks
    return ranks
```

Answer to Q6-15

Search Engine

Congratulations! You have now built a search engine! You learned how to collect a corpus using a web crawler, how to build an index and how to make it faster. Most recently you learned how to rank the results. Your search engine does page ranking better than any search engines before 1998.

Now, there is one thing left to learn – how to use the ranks.

For your homework you are asked to use the ranks to get the best result. To find the best result, just using the dictionary of ranks is not enough. Instead, you need to use the dictionary to find the result that matches the query with the best rank.

There are still a few problems left to solve before you can build a search engine to compete with Google. Finding a name for your search engine is probably the hardest problem. Yoogle? DuckDuckFind?

Another problem to solve is actually getting your search engine on the web so that other people can send queries to it. You can learn how to do this in the upcoming Web Applications Course .

In unit 7, you will get prepared for the final exam and you will see some interesting examples of using computing in context.

Answer Key

A6-1: Long Words

The answer is e, none of the above. All of these are sort of real words. It's not that well defined what it means for something to be a word.
a. honorificabilitudinitatibus (27 letters) is the longest word used in the works of Shakespeare. It means roughly "with honor".
b. antidisestablishmentarianism (28 letters) means the movement against the division of Church and State.
c. hippopotomonstrosesquippedaliophobia (36 letters) means the fear of long words. It ends with phobia which means fear.
d. pneumonoultramicroscopicsilicovolcanoconiosis (45 letters) is a kind of lung disease you get from contact with volcanic particles. It's the longest word in most large dictionaries.

The reason the answer is e, None of the above is because, given a word of any length, it's always possible to construct a longer one. You'll see how in the next section.

A6-2: Counter

The correct answer is a, None. You can't actually make any words with the definition as it's a circular definition. If you recall from BNF grammar, you can only stop when you reach a terminal word. The only choice here, is to replace Word with counter-Word.

Word → counter-Word
↓
counter-counter-Word
↓
counter-counter-counter-Word

You can never stop and so can never make a word this way.

A6-3: Expanding our Grammar

The correct answer is d) Infinitely Many. All that is needed is the two rules to make infinitely many words. This is the power of recursive definitions. Unlike the previous quiz where the definition was circular, this definition is recursive. That means that Word is defined in terms of itself, but that's not the only way it's defined. There is also another rule which is a starting point where Word is not defined in terms of itself.

Word → counter-Word
Word → hippopotomonstrosesquippedaliophobia

First, you could choose to use the second rule to replace Word with hippopotomonstrosesquippedaliophobia, but that's not the only thing you could do. You could also choose to replace Word with counter-Word, and so.

Word → hippopotomonstrosesquippedaliophobia

↓

counter-Word → counter-hippopotomonstrosesquippedaliophobia

↓

counter-counter-Word →counter-counter-hippopotomonstrosesquippedaliophobia

↓

. . .

There is no limit to the number of words which can be produced this way.

This is a recursive definition. It has a **base case**,

Word → hippopotomonstrosesquippedaliophobia

which is the stopping condition that means you have at least one word that is not defined in terms of another word. It also has a **recursive case** which defines a word in terms of another word.

Word → counter-Word

Combining the two gives a definition which can be used to make infinitely many words.

A6-4: Ancestors

The best answer is:

b. Ancestor → Parent
  Ancestor → Parent of Ancestor

These two rules provide a recursive definition, which work. They say that the Parent is an Ancestor, and so is the Parent of the Parent of the Ancestor and so on, covering all of the ancestors.

Here is why the others are incorrect:

a. Ancestor → Parent of Ancestor

This first rule by itself is not enough because there is no base case; there is no rule that defines Ancestor in terms of something that is not itself. You can keep producing parent of parent of parent of ... of Ancestor but you can never stop.

c. Ancestor → Parent
  Ancestor → Parent of Parent
  Ancestor → Parent of Parent of Ancestor

These three rules also give a recursive definition, which works. They produce the same set of ancestors as part b. The three rules can be combined to give any Parent of Parent of ... of Ancestor, but there are more rules than are necessary.

A6-5: Recursive Factorial

Going back to the mathematical definition,

$$factorial(0) = 1$$
$$factorial(n) = n * factorial(n-1) \text{ for } n > 0,$$

you need to turn this into code. It's fairly straight forwards. First, you need to define if you've reached the base case, which is when n=0, so you need to check for that. Remember you need to use double equals == for comparison. When n is 0, the code should return the value 1 since the base case is $factorial(0) = 1.$ `This is done using the following piece of code.`

```
def factorial(n):
    if n == 0:
        return 1
```

Following on from there, you need to deal with the recursive case, where

$$factorial(n) = n * factorial(n-1).$$

You need to return the new result which is $n * factorial(n-1),$ which completes the procedure.

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```

It might seem like circular to define factorial using factorial, but it's not because of the base case when `n==0` which will be reached when any positive whole number is passed in since it is decreased by 1 for each new call. A walkthrough of the code for `factorial(3)` is below. First, there is an explanation of the steps, and then a diagram showing how the `factorial` procedure is called and the final answer constructed as the returns are fed back. The initial input to the procedure factorial is 3, and then the procedure calls itself several times.

first call: `factorial(3)`
`n` → `3`
`n == 0` is `False` so don't return 1
go to `else:`
`return 3 * factorial(2)`

second call: `factorial(2)`
`n` → `2`
`n == 0` is `False` so don't return 1
go to `else:`
`return 2 * factorial(1)`

second call: `factorial(1)`
`n` → `1`
`n == 0` is `False` so don't return 1
go to `else:`
`return 1 * factorial(0)`

final function call: `factorial(0)`
`n` → `1`
`n == 0` is `True` so `return 1`

This is shown in the flowchart below. The left hand column shows how factorial is called with decreasing input values until the base case is reached. When the input value is 0, the base case is reached and 1 is returned. That value is then fed back to the previous call and then the next value going back up on the right is calculated is fed back to the previous call to that and so. (Follow the arrows down the left hand column and up the right hand column.)

```
factorial(3)                                        6 so factorial(3) → 6
   ↓                                           ↑ giving the final answer of 6
 3 * factorial(2)                                                  3 * 2  = 6
     ↓                                    ↑and 2 replaces factorial(2)
    2 * factorial(1)              2 * 1 = 2
       ↓                         ↑and now 1 replaces factorial(1)
         1 * factorial(0)       1 * 1 = 1
           ↓                    ↑ which feeds back to factorial(0) in the line above
              1      →         returns 1
```

A6-6: Palindromes

The **is_palindrome** procedure will take a single string as input. Write this procedure in such a way that it first checks for the base case. The base case was an empty string (in which case we return *True*). This can be done as follows:

```
def is_palindrome(s):
    if s == '':
        return True
```

Now you need to test if the first and last characters match. You can do this using Python's string indexing operators. The code looks like:

```
def is_palindrome(s):
    if s == '':
        return True
    else:
        if s[0] == s[-1]:
            return is_palindrome(s[1:-1])
        else:
            return False
```

`s[0]` and **`s[-1]`** get us the first and last characters respectively. If they match, then we recursively call **`is_palindrome`** (passing in the middle string using Python's slice notation). If they do not match, then you know it's not a palindrome and the procedure returns false.

You can test **`is_palindrome`** using following test cases:

- When you test **`is_palindrome`** with base case (i.e. empty string **`''`**), you get the result *True* (since empty string is a palindrome).
- When you test **`is_palindrome`** with any single letter string (e.g. **`'a'`**), you get the result *True* (since every single letter string is a palindrome).
- If you try string **`'ab'`** (which is not a palindrome), you get the result *False*.
- As a large string test, you can try the string **`'level'`** (a palindrome). You get *True* in the result.
- You can also test for the most famous palindrome string which is “A man, a plan, a canal, Panama.” In order to input this into the procedure, you need to remove the spaces, commas and capitals as the code doesn’t remove them and they aren’t considered in palindromes. That leaves you with the string **`'amanaplanacanalpanama'`**. When you test this string, you get *True* as the result.

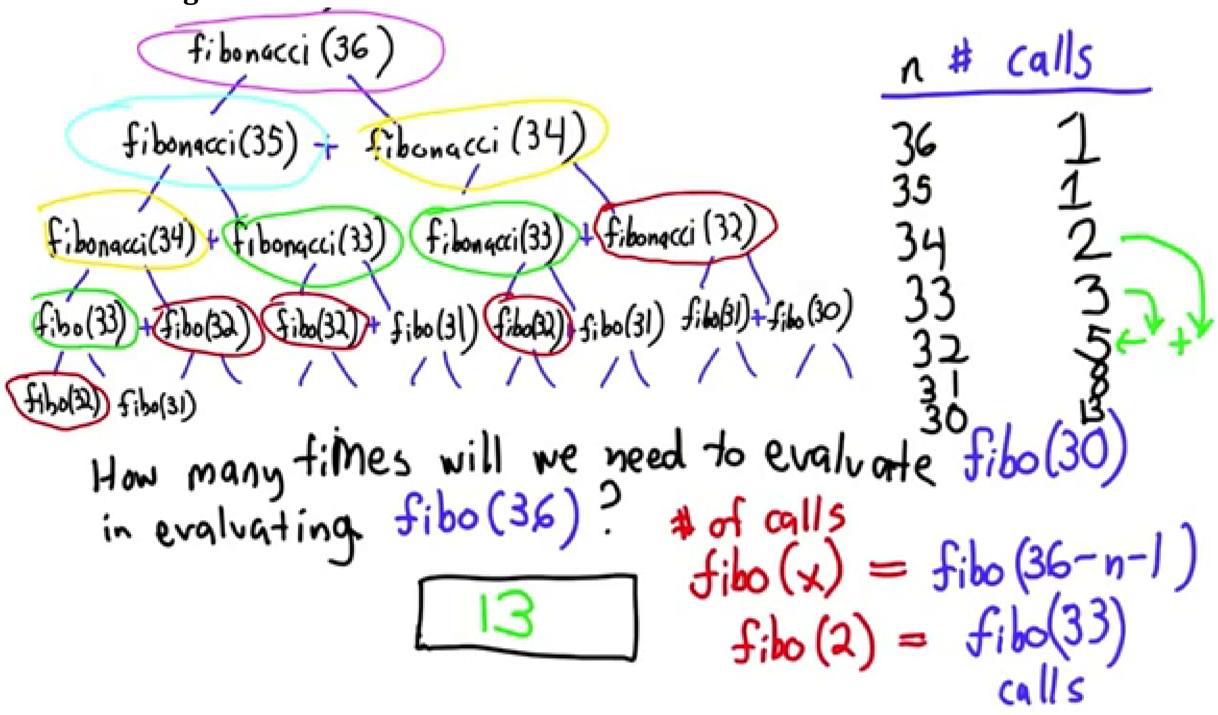
A6-7: Bunnies

Here is one way to answer this question:

```
def fibonacci(n):
    if n == 0: # account for base case
        return 0
    if n == 1 # account for base case
        return 1
    return fibonacci(n-1) + fibonacci(n-2) # otherwise do recursive part
                                           # of the definition
```

In the next lecture you will learn how to simplify this and write it in the Python interpreter.

A6-8: Counting Calls


fibonacci (36)
fibonacci(35) + fibonacci (34)
fibonacci(34) + fibonacci(33) fibonacci(33) + fibonacci (32)
fibo(33) + fibo(32) fibo(32) + fibo(31) fibo(32) + fibo(31) fibo(31) + fibo(30)
fibo(32) fibo(31)
n # calls
36 1
35 1
34 2
33 3
32 5
31 8
30 13
+
How many times will we need to evaluate fibo(30) in evaluating fibo(36) ?
13
# of calls fibo(x) = fibo(36-n-1)
fibo(2) = fibo(33) calls

The answer is 13. The reason for this is quite interesting. Look at the numbers you got so far in the tree:

n (input to **`fibonacci`** procedure)	Number of calls
36	1
35	1
34	2
33	3
32	5
31	8
30	13

You might have noticed a pattern in number of calls. This is exactly the fibonacci series.

Every time you decrease the number by 1, you add the number of calls for last and last-but-one number of calls. If you look at the structure of the tree, you'll notice that it follows the same rule you read about with respect to the number of rabbits reproducing. Every pair of rabbits you have on a previous level reproduces two more. This means that when you do the addition, you are redundantly calling all previous calls.

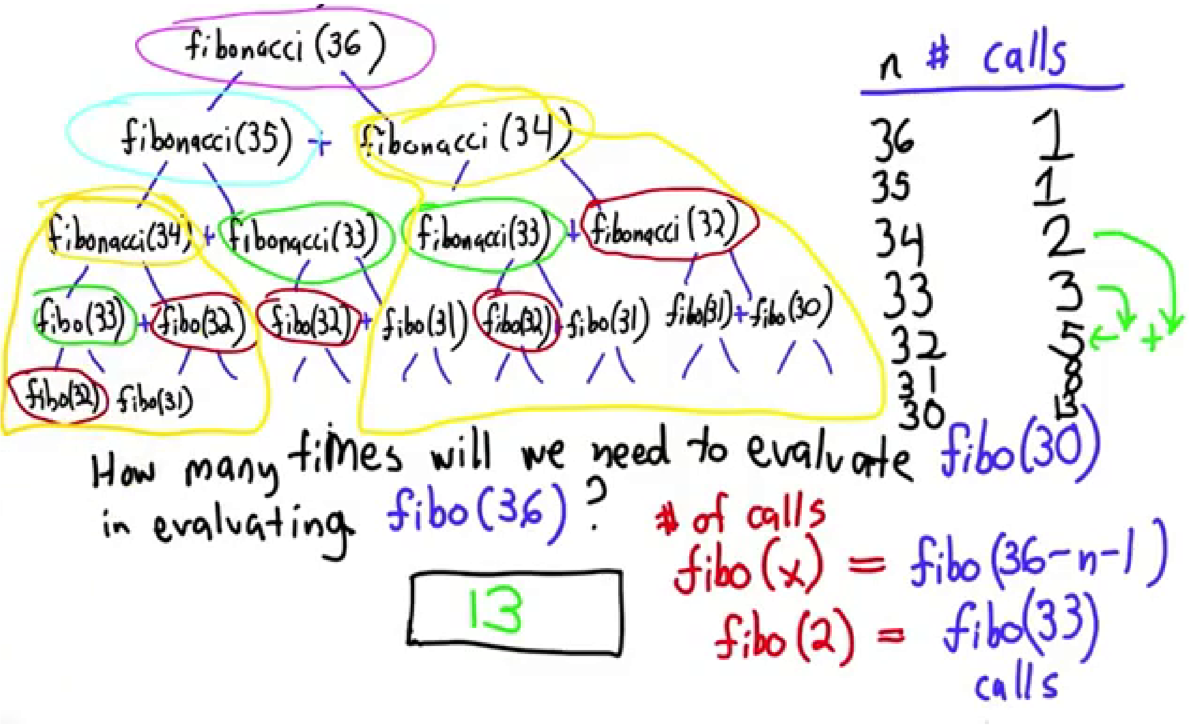
As you can see, there is one call to **`fibonacci(34)`** at a previous level. This produces two new calls to **`fibonacci`** where the inputs are different. But if you look at the way the inputs are distributed, you'll see 3 (2 + 1) calls to **`fibonacci(33)`**, 5 (3 + 2) calls to **`fibonacci(32)`** and 8 (5 + 3) calls to **`fibonacci(31)`**.

Therefore, you get 13 (8 + 5) calls for **`fibonacci(30)`**. This pattern will continue and these numbers will increase very fast. The number of calls, **`fibonacci(x)`** that you need every time you evaluate **`fibonacci(36)`** is given by:

Number of **`fibonacci(x)`** calls when evaluating **`fibonacci(36)`** = fibonacci(36 - n - 1)

When you try to evaluate **`fibonacci(x)`** for larger values of x (using the recursive definition), it takes large amount of time. You need to find a more efficient way to compute fibonacci numbers (You will study this in next section).

The recursive procedure is inefficient because the procedure is making redundant computations. For example, to compute **`fibonacci(36)`**, you had to compute **`fibonacci(34)`** and **`fibonacci(35)`**. All the work required to compute **`fibonacci(34)`**–the yellow circled section on right side in the diagram below – was redundantly computed again by call to **`fibonacci(34)`** –the yellow circles section on left side in the diagram.


fibonacci (36)
fibonacci(35) + fibonacci (34)
fibonacci(34) + fibonacci(33)
fibonacci(33) + fibonacci (32)
fibo(33) + fibo(32)
fibo(32) + fibo(31)
fibo(31) fibo(32) + fibo(31)
fibo(31) + fibo (30)
fibo(32) fibo(31)
+

n	# calls
36	1
35	1
34	2
33	3
32	5
31	8
30	13

How many times will we need to evaluate fibo(30) in evaluating fibo(36)?

13

# of calls fibo(x) = fibo(36-n-1)

fibo(2) = fibo(33) calls

So, the solution to this is to use an iterative procedure (i.e. use a **`while`** loop) instead of a recursive procedure. Every recursive procedure can also be defined without using recursive definition. Though it is often easier and cleaner to think, read and write a recursive procedure, it is often not the fastest way to compute (e.g. redundant calls to **`fibonacci`**) in this example.

A6-9: Faster Fibonacci

You can start off with base cases:

```
   def fibonacci(n):
       # base cases
       if n == 0:
           return 0
       if n == 1:
           return 1
```

Then add recursive case. For that, you will need three variables:

- res: variable storing the result to be returned
- before2: auxiliary variable storing value from two steps before
- before1: auxiliary variable storing value from one step before

```
       # final result
       res = 0
       # two steps before ( fibonacci(0) = 0 )
       before2 = 0
       # one step before ( fibonacci(1) = 1 )
       before1 = 1
```

And, obviously, some loop will come in handy as well. Iterate over all the numbers from **`0`** to **`n`**, which means **`n+1`** iterations in total. However, the first two numbers are handled by the base cases, which will leave you with **`n+1-2`**, i.e. **`n-1`** iterations (i.e. **`i=2`** instead of **`0`** and **`while i <= n`** instead of **`while i < n`**).

```
       i = 2
       while i <= n:
```

Note: It would be an error to count only **n** iterations at the beginning of the previous paragraph. Since you want to include both **0** and **n**, it actually makes **n+1** iterations instead of just **n**. This is one version of a very frequent, yet very hard-to-find error. This error is in fact so frequent and ugly, that it even has a name: ***off-by-one error***. You can find out more about it here: http://en.wikipedia.org/wiki/Off_by_one.

The body of the loop will then directly mimic the definition of fibonacci numbers:

```
# result for current i is sum of two previous ones
res = before2 + before1
# i is going to be one more, thus before1 and before2
# variables must be updated by one step as well
before2 = before1
before1 = res
# it is easy to forget this line, thus it is the last
# to be well visible
i = i + 1
```

And in the end, return the result:

```
    return res
```

So you end up with fully functional and correct solution:

```
def fibonacci(n):
    # base cases
    if n == 0:
        return 0
    if n == 1:
        return 1

    # final result
    res = 0
    # two steps before ( fibonacci(0) = 0 )
    before2 = 0
    # one step before ( fibonacci(1) = 1 )
    before1 = 1

    i = 2
    while i <= n:
        # result for current i is sum of two previous ones
        res = before2 + before1
        # i is going to be one more, thus before1 and before2
        # variables must be updated by one step as well
        before2 = before1
        before1 = res
        # it is easy to forget this line, thus it is the last
        # to be well visible
        i = i + 1
    return res
```

However, there are few details that can be improved. First, the usage of the **while** loop instead of **for** loop and **range**:

```
def fibonacci(n):
    # base cases
    if n == 0:
        return 0
    if n == 1:
        return 1

    # final result
    res = 0
    # two steps before ( fibonacci(0) = 0 )
    before2 = 0
    # one step before ( fibonacci(1) = 1 )
    before1 = 1

    # This goes out:
    #i = 2
    #while i <= n:
    # This goes in:
    for i in range(2,n+1)  # beware of off-by-one error!
        # result for current i is sum of two previous ones
        res = before2 + before1
        # i is going to be one more, thus before1 and before2
        # variables must be updated by one step as well
        before2 = before1
        before1 = res
        # This goes out:
        ## it is easy to forget this line, thus it is the last
        ## to be well visible
        #i = i + 1
    return res
```

Then, notice that you can write the **`for`** loop in a way that will not lure you into an off-by-one trap but only if you replace saving the past two values by holding the current and proceeding one. As a bonus, first two **`if`** statements defining base cases will be rendered redundant and can be removed as well:

```
def fibonacci(n):
    # This goes out:
    ## base cases
    #if n == 0:
    #    return 0
    #if n == 1:
    #    return 1

    # This goes out:
    ## final result
    #res = 0
    ## two steps before ( fibonacci(0) = 0 )
    #before2 = 0
    ## one step before ( fibonacci(1) = 1 )
    #before1 = 1

    # This goes in:
    current = 0  # fibonacci(0) at the bginning
    after = 1  # fibonacci(1) at the beginning

    # This goes out:
    #for i in range(2,n+1)  # beware of off-by-one error!
    # This goes in:
    for i in range(0, n)  # no off-by-one trap
        # This goes out:
        ## result for current i is sum of two previous ones
        #res = before2 + before1
        ## i is going to be one more, thus before1 and before2
        ## variables must be updated by one step as well
        #before2 = before1
        #before1 = res
        # This goes in:
        temp = current  # save old current value
        current = after
        after = temp + current  # use saved old current value
    # This goes out:
    #return res
    # This goes in:
    return current
```

The last small detail, is to remove temporary variable temp using multiple assignment:

```
def fibonacci(n):
    current = 0  # fibonacci(0) at the bginning
    after = 1  # fibonacci(1) at the beginning

    for i in range(0, n)
        # This goes out:
        #temp = current  # save old current value
```

```
            #current = after
            #after = temp + current  # use saved old current value
            # This goes in:
        current, after = after, current + after
        return current
```

The final result is shown below. All the redundant calculations are avoided by keeping track of two variables. To avoid the need for special cases for **n = 0** and **n = 1**, the code keeps track of the current value and the number following it instead of keeping track of the previous two numbers.

```
def fibonacci(n):
    current=0 # fibonacci(0)
    after=1   # fibonacci(1)
    for i in range(0, n): # start at 0, where current is fibonacci(0),
                          # and after is fibonacci(1)
        current, after = after, current + after # see note below
    return current
```

The values of **current** and **after** are updated according to the recursive definition. This is done using multiple assignment:

```
      current, after = after, current + after
```

This means that **current** is set to **after**, and **after** is set to the sum of the old **current** and the old **after** values simultaneously. Without using multiple assignment, you would need to use a temporary variable to save one of the values.

When **n=0**, the code doesn't go through the loop at all

```
print fibonacci(0)
0
```

When **n=1**, it goes through the loop once.

```
print fibonacci(1)
1
print fibonacci(2)
1
print fibonacci(3) # this should be 1 + 1 = 2, which it is
2
print fibonacci(5)
5
```

In the answer to quiz 6-8 (Counting Calls) you saw that the number of calls to **fibonacci(2)** when calculating **fibonacci(36)** using the recursive definition was equal to **fibonacci(33)** but you couldn't calculate it then because the recursive code was too slow. Why did it take that code so long to run?

```
print fibonacci(33)
3524578
```

That's a little over three and a half million calls! Even with a processor doing a billion instructions a second, doing 3.5 million recursive calls is going to take a while. Each time through a call is not just one instruction, but many thousands of instructions, so it takes quite a long time to run. It wasn't just the **fibonacci(33)** calls to **fibonacci(2)** that needed to be done, but also all the other elements leading up to get to **fibonacci(36)**. With this faster, iterative code, you now can calculate **fibonacci(36)**.

```
print fibonacci(36)
14930352
```

So, after three years, there would be nearly 15 million rabbits using Fibonacci's model. How many would there be after 5 years?

```
print fibonacci(60)
1548008755920
```

After five years there would be over one and a half *trillion* rabbits! To try to relate to this, consider how many months it would take for the mass of the rabbits to overtake the mass of the earth.

A well fed rabbit weighs approximately 2 kg, although if they were spreading as fast as this model suggests, there wouldn't be enough food for them to all be well fed. The mass of the earth is$5.9722 * 10^{24}$kg. In Python, 10^{24}, which is a 1 with 24 zeroes after it, is written as **10**24**. To demonstrate the power notation,

```
print 2**10
1024
```

So **2**10** is 1024 because 2*2*2*2*2*2*2*2*2*2 = 1024.

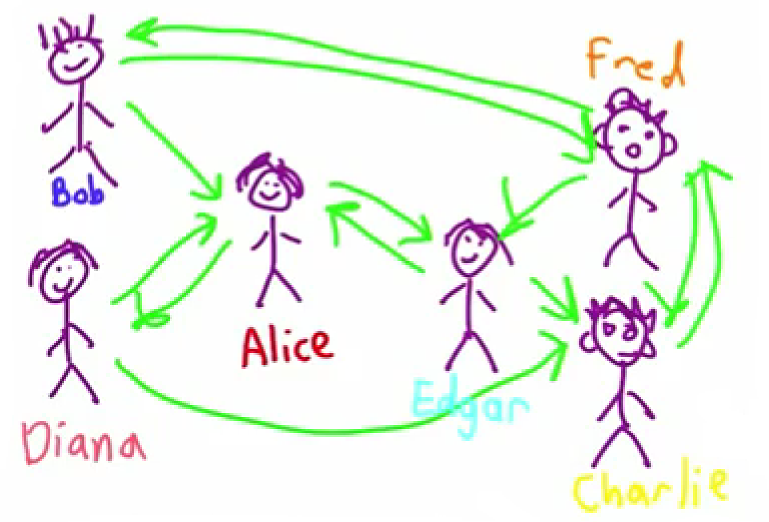
```
mass_of_earth = 5.9722 * 10**24 # kilograms
mass_of_rabbit = 2 # 2 kilograms per rabbit

n = 1
while fibonacci(n) * mass_of_rabbit < mass_of_earth:
    n = n + 1
print n, fibonacci(n)
119 3311648143516982017180081
```

Only 119 months, which is less than 10 years for the rabbits to take over the earth! In less than 10 years, using this model, the mass of rabbits would be greater than the mass of the earth. There would be over 3 trillion trillion ($3 * 10^{24}$) of them! Be afraid, be very afraid of all these rabbits!

The good news is that the model is not very accurate as rabbits do die, and when there is too little food, their numbers won't grow as fast as the model suggests. The model was created to illustrate mathematical principles rather than to accurately represent the population growth of bunnies.

A6-10: Good Definitions

The answer is b, no. It is not a good recursive definition because it has no base case. It’s a circular definition. The definition requires calling popularity over and over again, but it will never get to a point where it can stop.



Consider the popularity of Charlie:

$$?][?][?]Charlie[?][?][?]) = popularity([?][?][?]Diana[?][?][?]) + popularity([?][?][?]Edgar[?][?][?]) + popularity([?][?][?$$

Now you need to work out the popularity of each of Diana, Edgar and Fred.

$$popularity([?][?][?]Diana[?][?][?]) = popularity([?][?][?]Alice[?][?][?])$$

As you continue to try to work out the popularity of Diana, you need to compute the popularity of Alice.

$$([?][?][?]Alice[?][?][?]) = popularity([?][?][?]Diana[?][?][?]) + popularity([?][?][?]Edgar[?][?][?]) + popularity([?][?][?$$

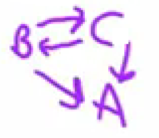
Now to work out the popularity of Alice, you need the popularity of Diana, but to work out Diana’s popularity you need to work out Alice’s popularity!

$$popularity([?][?][?]Diana[?][?][?]) = popularity([?][?][?]Alice[?][?][?])$$

It's circular! It just gets worse and worse and never stops.

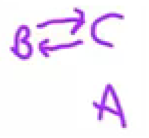
A6-11: Circular Definitions
The correct answer is e) No.

a. Only if everyone is friends with 'Alice'.



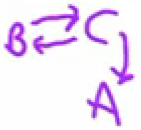
To figure out the popularity of Bob, you need to know the popularity of Charlie and to figure out the popularity of Charlie you need to figure out the popularity of Bob as well as of Alice. You'll keep bouncing back and forth between Bob and Charlie.

b. Only if no one is friends with 'Alice'.



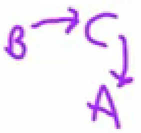
Removing the links between Alice and the others doesn't solve the problem. You'll still be bouncing back and forth between Bob and Charlie.

c. Only if there is a friendship path from everyone to 'Alice'.

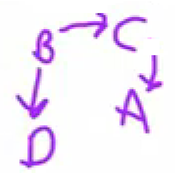


This still doesn't remove the bouncing back and forth between Bob and Charlie problem.

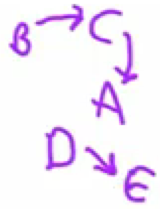
d. Only if there are no cycles in the graph.



Here it would be ok, because you could work out the popularity of Bob by working out the popularity of Charlie, and to work out the popularity of Charlie, you'd use the popularity of Alice.



This also works, because you only need to work out the popularity of Bob to work out the popularity of Diane, and from above, that's possible.



There isn't a cycle, but how can you work out the popularity of Edgar? The way the Python code is written, this could work, as the popularity of someone with no friends is 0, so you're due some credit if you chose this option. On the other hand, it doesn't make sense from the mathematical definition as you don't have a way to find the popularity of Edgar. However, even though the Python code might work, it doesn't give you meaningful popularity scores.

This is not a good way to define popularity. It was arbitrary to choose Alice to give popularity 1. Poor Alice! There is also nothing she can do to make herself more popular which isn't very fair.

A6-12: Relaxation

The answer is yes. No matter what is passed in for **t** and **p**, you'll eventually get a result. Every time you do a recursive call, the value of **t** is one less than before since

```
      score = score + popularity(t-1,f)
```

If you start with an integer for **t**, and keep reducing it by one, you'll eventually get to **t=0** which returns 1 without referring back to the popularity procedure again.

```
if t == 0:
    return 1
```

It might not produce a meaningful definition of popularity, but it will produce a result.

A6-13: Altavista

The answer is b, the most popular web search engine in 1998.

AltaVista is also a small town in Virginia, but that's not important right now.

The reason Google outcompeted AltaVista was because of superior page ranking. The algorithm is called PageRank, which is named after Larry Page, the co-founder of Google (with Sergey Brin). (To read more about how the original Google search engine worked, check out this link, which is student paper describing its inner workings: http://infolab.stanford.edu/~backrub/google.html)

The problem with search engines like AltaVista is they were too easy to game; the highest-ranking pages were often those that were best at gaming the rankings.

A6-14: Implementing Urank

The one line of code that needs to be added is:

```
graph[page] = outlinks
```

This adds the list of outlinks to the **graph** dictionary, using **page** as the key.

The next step is to use the graph to calculate page ranks.

A6-15: Finishing Urank
The code to be inserted is:

```
            for node in graph:
                if page in graph[node]:
                    newrank += ranks[node] * d / len(graph[node])
```

node is used instead of **page** because **page** is already used to refer to something. Loop through each node in the graph, checking whether that node links to **page**. If so, update **newrank** according to the formula from the recursive definition.

So now, **compute_ranks** returns a dictionary which gives a rank for each page in the index.

In the example input, kathleen has a relatively high rank even though it only has 2 incoming links. This is because the incoming links are from very popular pages.