Intro to Computer Science

Lesson 06 Notes

How to Have Infinite Power

Infinite Power

Welcome to Unit 6. The big idea in Unit 6 is recursive definitions and we'll see how to use a recursive definition of popularity to make the search engine respond with the best page for a given query instead of just giving us all the pages. This is really a key advance that made Google so successful. The real goal of this unit is to give you infinite power. I know what you're thinking, I have infinite power already, and you're right. You've had infinite power since Unit 2 when you learned about procedures and if. And that was enough to write every possible computer program. But we went on to introduce while, to use a while loop to find all the links in a page. And that really should not have been necessary. If you already had infinite power, you wouldn't have needed anything new to do that. What we're going to see in this unit is that you could do that without while, that all you really need is procedures, and by thinking about things recursively, you can solve lots of problems in interesting ways.

Quiz: Long Words

So, we'll start with a quiz. It's kind of a trick quiz. The question is, what's the longest word in the English language? If you're not a native English speaker, don't worry. This quiz is just as hard for the native English speakers as it is for you. And, here are the choices: honorificabilitudinitatibus, antidisestablishmentarianism, hippopotomonstrosesquippedaliophobia, pneumonoultramicroscopicsilicovolcanoconiosis, or none of the above.

[ ] honorificabilitudinitatibus
[ ] antidisestablishmentarianism
[ ] hippopotomonstrosesquipedaliophobia
[ ] pneumonoultramicroscopicsilicovolcanoconiosis
[x] None of the Above

Answer:

Copyright © 2014 Udacity, Inc. All Rights Reserved.

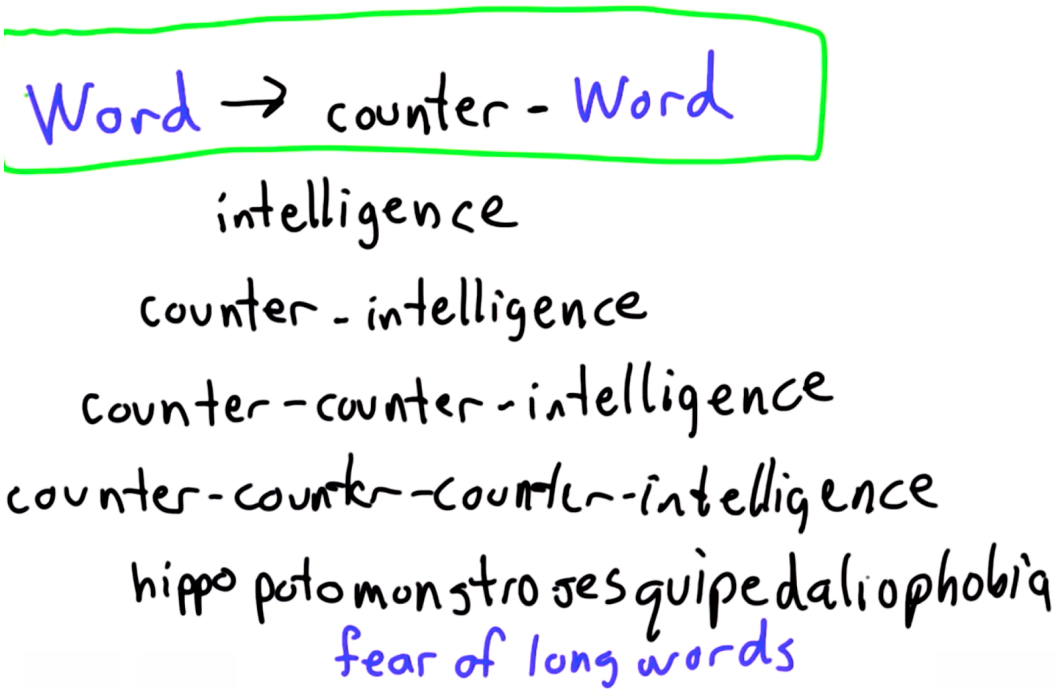
So the answer is none of the above and we'll see why in a minute. First I want to mention that all of these are sort of real words. It's not that well defined what it means for something to be a word. The first one is the longest word used by Shakespeare. This is 27 letters long and it means roughly, with honor. The second one is one letter longer. It means the movement against the division of church and state. The third one, which is 35 letters long, means fear of long words. You can see it's made up of phobia, which is fear and a really big sort of made up root about things being big and words. So it's fear of long words. The fourth one is the longest word in most large dictionaries. It's a kind of lung disease that you get from having contact with volcanic particles. So, the reason the answer is none of the above is that I'm very confident whatever you claim is the longest word, I can always make a bigger one. There is no such thing as the longest word in the English language.

Counter

So a word is just something that has meaning that speakers of a language will understand. We could define words as just what's in the dictionary, and then there would be a clear answer for that. But certainly there are lots of things that are words that aren't in the dictionary. And the important thing about words is that we have rules for making new words. For example we have a rule that says if we have a word, we can make a new word by adding counter in front of the old word. I'm using the same notation that we used back in unit one. This is a BNF replacement grammar. If you need a refresher on this, please go back to the video that introduced that in unit one. But the basic property is that we can replace what's on the left side with what's on the right side. So, anytime we have a word, we can replace it with counter-word. And the meaning of the new word is something that goes against the original word.

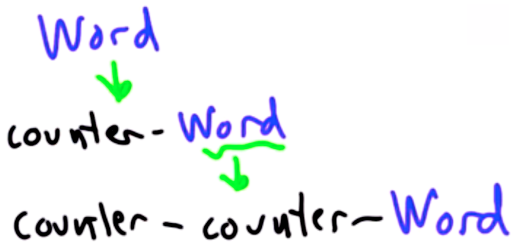
So, if we started with the word, intelligence, and I mean this in the sense of spy craft. Not in terms of smarts. Well then, we can use the rule. We can make the word, counter-intelligence. Intelligence was a word. We added counter in front of it to make counter-intelligence. And that means trying to thwart intelligence from the enemy. We can use the rule again, so now this is a word, and we can replace this word with counter in front of that word, and we get counter-counter-intelligence. And that would be trying to thwart the enemy's counter intelligence, that's preventing you from getting intelligence. And we could keep doing that. We could have counter-counter-counter-intelligence. And so forth, and these are words that, once we get up to at least three counters, it's not something that's used before. But it's still something that has a sensible meaning, and we could do this for other words.

Copyright © 2014 Udacity, Inc. All Rights Reserved.



So we saw that one of the very long words was this hippopotomonstrosesquippedaliophobia. And if you knew that this word meant fear of long words. Even if you've never seen the words counter in front of that before. So now we can make the new word counter-hippopotomonstrosesquippedaliophobia. Quite a mouthful. But even though you've never seen that word before. And I expect most of you have never seen that word before. Well, you could guess that this is something that goes against the fear of long words. Maybe, there's some new medication that cures people of this phobia. And you could also figure out, well, if there was a counter-counter-hippo and so forth phobia. Well that would be something that counteracts the counter-hippo, and that might be something where this new drug doesn't work if you have too much coffee. Then coffee would be a counter-counter-hippo so forth phobia. So the point is, because English has this rule that allows us to make new words from old words. There's no such thing as the longest word, we can always make a longer word, starting from any word that we have. If you remember how BNF grammars work, then you should be able to answer the next quiz.

Quiz: Counter Quiz



Copyright © 2014 Udacity, Inc. All Rights Reserved.

So the question is, if the only rule that we have for making words is this one, the one I should you before, where we have the non-terminal word, and that can be replaced by counter followed by another word, then the question is, how many words can we make starting from the non-terminal word? And the possible answers are none, one, two, or infinitely many.

[x] None
[ ] 2
[ ] 1
[ ] Infinitely Many

Answer:

So the answer is none, that we can't actually make any words with just this rule. And the reason for that is this is a circular definition, and it's a circular definition because we can never stop. If you remember the rules for BNF grammars, that we can start from a starting on terminal, and we keep doing replacements. We can only stop once we have all terminals. We are never going to get there if this is our only rule. We can keep adding more counters, but we can never replace the non-terminal word. Because every time we end up with the non-terminal word again, on the right side. So with just this rule, we can't actually make any words. All we can do is start from word. Now our only choice is to replace word, with counter-word well, we only have one rule, so our only choice is to replace the word here with counter-word. So now we end up with counter-counter-word, but since word is a nonterminal, we're never done. So we can never make any word this way.

Quiz: Expanding Our Grammar

So let's try adding one more rule. So now we'll ask the quiz again. But this time our grammar has two rules. So now we have two rules. We have the rule we had before, which allows us to replace word with counter-word. And we're going to add one more rule. It says we already know one word. We know the word hippopotomonstrosesquipedaliophobia. So now the question is, how many words can we make? The possibilities are none, one, two, or infinitely many.

[ ] None
[ ] 2
[ ] 1
[x] Infinitely Many

Copyright © 2014 Udacity, Inc. All Rights Reserved.

Answer:

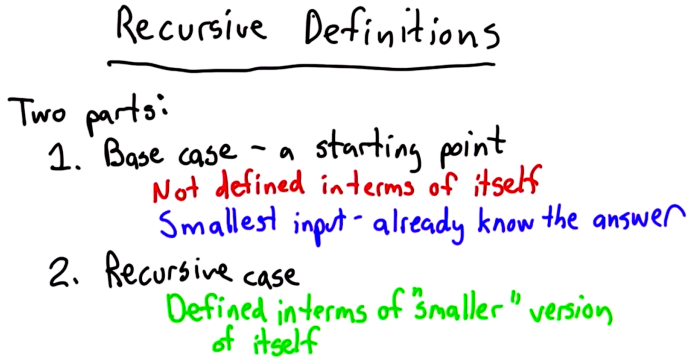
So now, the answer is infinitely many. All we needed was these two rules and we can make infinitely many words. This is the power that recursive definitions give us. And unlike the previous definition which was circular, what we have now is what we call a recursive definition. That means, we have defined word in terms of itself but that's not the only way we've defined word. We also have this other rule that allows us to have a starting point. That there's one word that we have that's defined not in terms of itself. So here's how we can make infinitely many words using these rules. So we can start with a non-terminal word and let's say we choose to use the first rule. We can replace word directly with our hippo word and we're done, but we have another option. We could have replaced word using the second rule, with counter word and if we replace this word, using the second rule, we'll end with that word counter hippo… Phobia. But that's not the only choice, right, we could've chosen to use the first rule again, and then we were to replace this word with counter-word and so now we have counter, counter, word. Again we have the choice of what to do with this word. We could use the second rule, replace it with the terminal. And then we'll have counter, counter hippo. Or we could replace it using the first rule. And then we'll have counter, counter, counter followed by word. So this can keep going as long as we want. We can produce all of these words with any number of counter, either zero, repetitions of counter one, two, three, four. As many as we want. That means we can produce infinitely many words. Some of them are going to be pretty hard to pronounce. Actually, they're all pretty hard to pronounce. But, there's no limit to the number of words that we can produce this way. So this is what's called the recursive definition and the important thing that it has is two parts. It has a base case which is here. That's the stopping condition. That's something that says we have at least one word that we can define already. That we don't need to define in terms of word. And it has the recursive case that says we can define a word in terms of another word. And if we combine those two, well now we have a definition that can make infinitely many words.

Recursive Definitions

So this works for things other than words. The place where you've seen it the most so far has been in language, but we're also going to to see how to use it to make procedures. And you'll also see how to use it in later classes, how to define data structures. There are lots and lots of different things in computing that are defined recursively. And a recursive definition has two parts. It has the base case. In our example with the word, that was the second rule. And the base case we can think of as a starting point. And the important thing about the base case, is it's not defined in terms of itself. It has to be something where we already know how to define it. So for programs, it's usually going to be the smallest input, or

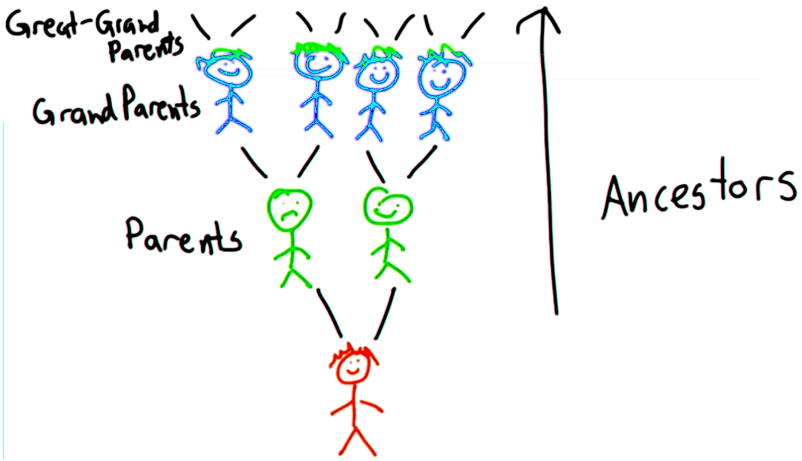
Copyright © 2014 Udacity, Inc. All Rights Reserved.

one of the simplest inputs. Something where we already know the answer. We don't have to do any computation to figure it out. The second part is the recursive case. And that is defined in terms of itself. But it shouldn't be defined in terms of itself exactly. It should be defined in terms of some sort of smaller version of itself. We need to make progress to eventually get to the base case. We'll see what that means in programs soon. First, we're going to try one other example not in terms of a program, just to get a better sense for how things can be defined recursively.



Quiz: Ancestors

The question we want to think about is how you could define who your ancestors are. The way to think about ancestors. Well, here's you. Hopefully, you're smiling, because you're enjoying learning about recursive definitions. And you had some parents. Let's assume you had two. And we won't assume whether or not your parents are smiling, but let's hope at least one of them is. And your parents were your ancestors, but they're not all of your ancestors. Your parents had parents, as well. These are your grandparents. Grandparents are always happy. So, they're all smiling. And your grandparents, well, they also had parents. Those would be your great-grandparents, and so forth. And all of these, except yourself, are your ancestors. So our goal now is to define that precisely. Can we come up with a definition that describes exactly this same relationship, all the ancestors that you have? So the question is, which of these is the best definition of ancestors? And there are three choices, the first has one rule, ancestor is replaced by parent of ancestor. The second has two rules, ancestor is replaced by parent, and ancestor is replaced by parent of the ancestor. The third has three rules. Ancestor is replaced by parent. Ancestor is replaced by parent of parent. And ancestor's replaced by parent of parent of ancestor.



[ ] Ancestor > Parent of Ancestor
[x] Ancestor > Parent, Ancestor > Parent of Ancestor
[ ] Ancestor > Parent, Ancestor > Parent of Parent, Ancestor > Parent of Parent of Ancestor

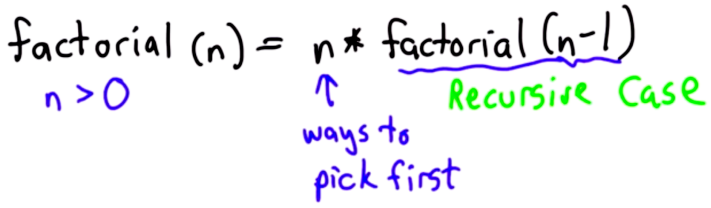
Copyright © 2014 Udacity, Inc. All Rights Reserved.

Answer:

So the best answer is the second choice. The first rule by itself is not sufficient. And the reason the first rule doesn't work is that there's no base case. We can keep producing parents of ancestor and parent of parent of parent of ancestor. But we can never stop because there's no rule that defines ancestor in terms of something other than itself. The second rule is a recursive definition that works; it says our parent is our ancestor. And the parent of an ancestor is also an ancestor. And so, this will give us exactly the set of ancestors that we showed before, that it will be the parent, the parent of the parent, the parent of the parent of the parent, the parents of the parents of the parents of the parents and so forth, covering all of our ancestors. The third choice will also produce that same set, but it has an unnecessary rule that we only need these two rules. Here we've said, parent, parent of parent, parent of parent of ancestor. We can still combine these rules to produce any word that is parent of parent of parent of any number of times ending in parent, but it's less elegant than the second answer, which only needs two rules to produce exactly the same set of ancestors.

Recursive Procedures

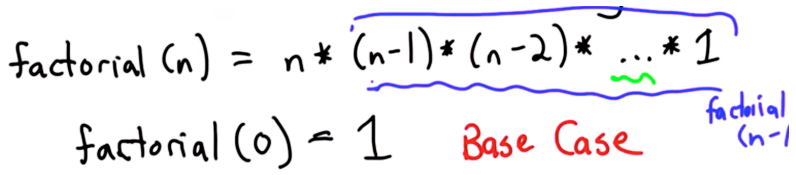
So we've seen how to use recursive definitions to make words, and to define concepts like ancestors. Now we're going to see how to use recursive definitions to define a procedure. And we're going to start with the procedure that we already defined in unit two. We defined the factorial procedure. And we defined factorial as the number of ways that we can order n items, and the input is the number n. And that could be calculated by multiplying n times n minus 1 times n minus 2, and so on until we get down to the 1. So that definition is not very precise mathematically, and the problem with this as a mathematical definition. Is, it's got this, dot dot dot in it, and humans sort of understand that correctly, what that dot dot dot means, but it's not a very precise mathematical definition. If we use a recursive definition, we can define factorial in a much more precise way. And we need to do that by giving a base case, so for the base case we want to think about the simplest input, something where we already know that answer. And for factorial, and for many procedures that involved numbers, the simplest input is the number 0. So if we try factorial defined as 1. So we know that result. That's going to be our base case.



Now all we need to do is define the recursive case, where we want to define what the meaning of factorial for any number n is. Where n is any integer greater than 0, and we can define that in terms of the factorial of the smaller numbers. So if you look at this definition with the dot dot dot, well we see

Copyright © 2014 Udacity, Inc. All Rights Reserved.

that the factorial of n is n times n minus 1 and so forth. Well, this what we have here, is actually the factorial of n minus 1. So that means that we can define the factorial then as n times the factorial of n minus 1. And that's our recursive case, and this definition matches our intuition well, if we think about factorial meaning the number of ways to arrange n objects. This corresponds exactly the way we think about this, that we have n ways to pick the first item. And once we've picked the first item, well, we have n items left, and factorial of n minus 1, gives us the number of ways to arrange the remaining n minus 1 items. So that's a way to define this mathematically. So now, I'll leave it to you as a quiz, to figure out how to define a procedure, that computes factorial without using a while loop.



Programming Quiz: Recursive Factorial

So your goal for this quiz, is to define a procedure, named factorial, that takes a number, which is a positive whole number as its input, and outputs the number of ways to arrange the input number of items. So, that's the mathematical definition of factorial. And, for your procedure, we've already seen how to do this using a while loop. Your goal here is to define that procedure without using a while loop, to define it as a recursive definition.

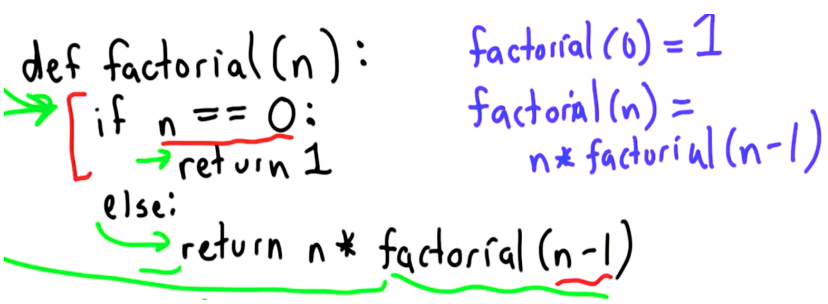
Answer:

So here's how we could define factorial. And, we should go back to our mathematical definition. That's what we want to turn into code. We had our base case where factorial is 0. The result should be 1. And we had our recursive case, where there is input is greater than 0. We want to have the result be n times factorial of n minus 1. So we can turn that fairly straightforwardly into code, so here's the code. We're going to define a procedure. Just like the previous definition, we have one input, it's a number. We'll call it n. Now we need to have the code, so we need to first check if we've reached the base case, so we'll use an if for that. And we're going to check if if n is equal to That means we've reached the base case. And we defined that value factorial as 0, is defined as we should do is return the value of 1. When it is not equal to 0, we'll use else for the case where n is not equal to 0. Well then we have the recursive case. And that was given by the definition, that the factorial of n, for numbers greater than 0, is equal to n times factorial of n minus 1. So that's exactly what we want to do in the Python code. We'll turn the new result, which is n times the result we get, calling

Copyright © 2014 Udacity, Inc. All Rights Reserved.

factorial passing in n minus 1. And this may seem strange that we're defining factorial using factorial. It seems like that's kind of circular, but the reason that it's not circular, is because we have this base case. That we have a case, where we do stop, we stop once we reach the case where n is equal to 0, and because every time we call factorial, instead of passing in the same value of n that we started with, we're passing in n minus 1. It's getting smaller. Eventually we're going to get to 0, assuming that we started with n as some positive whole number.

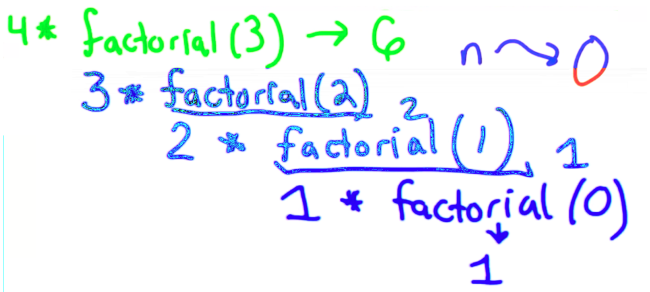
So we eventually stop, return one. And on the way there, we're going to be multiplying in all these values. So let's step through what happens when we run this code. So let's look at an example. Suppose we called factorial passing and three. So that means we're going to enter the procedure here. The value that n refers to is going to be 3 inside the body of factorial. We're going to try the comparison. It is not equal to 0, so we don't return 1. So we're going to go to the else. And that means we're going to return the value of n times factorial of n minus 1. So that means we're computing the value of n times factorial then and n minus That's what happened as a result in the first call. But we're not done, right. We still need to evaluate this, so now we're evaluating factorial of 2. That means we're entering factorial again. This time the value of n will refer to 2. We do the comparison, n is not equal to 0 so we go to the else clause. Now, we're going to return the result of n times factorial n minus 1. In this case, the value of n is 2. So the value of factorial of 2 will be 2 times the result we get by calling factorial, passing in the value of n minus 1, so calling factorial passing in 1. And remember that was part of the return, where we had 3 times factorial 2.



So we still have the 3 times. But now we're getting the result of factorial 2. It's 2 times something. We're calling factorial again. We still don't know what factorial of finished here. We're doing the call. We're going to re-enter factorial, but this time the value of n now refers to 1. And now again this test is false. So we go to the else clause. And now we're going to return n which is 1 times the result of factorial 0. So now, we're calling factorial 0. We're going back into the procedure. Now, the value of n refers to is 0. And, now, this test is actually true, n equals 0. So, we'll go to the return here. We return 1, that means the result of factorial 0 is 1, so to get the result of factorial 1, we had is 1, and now to get the result back from factorial 2, we had 2 times factorial 1, which we now know is 1. So we're going to have, the result here as 2. And to get the result for factorial has the value 2, will get the result, as 6. And note the way we've defined it, well if we tried instead,

Copyright © 2014 Udacity, Inc. All Rights Reserved.

factorial 4, if we started with factorial 4, what would have happened is the first time we go through, we get 4 times factorial 3. So factorial 4 would be 4 times factorial 3, well we had factorial 3 is 6, 4 times 6, would give us 24, which the result, is factorial 24.



Programming Quiz: Palindromes

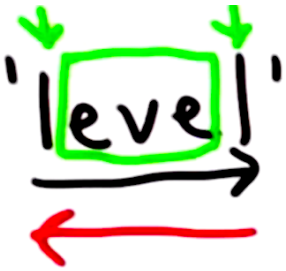
So, we can do another example of defining a recursive procedure. And the goal this time is to take in a string, and determine whether or not it's a palindrome. What a palindrome is, is a string that reads the same way, forwards and backwards. So, an example of a palindrome would be level. If we read level forwards, we get level If we read it backwards, we get exactly the same thing. Another example of a palindrome would be the word, the single letter a. If we read a forwards, we get a. If we read a backwards, we also get a. In fact, any single letter must be a palindrome. The empty string is also a palindrome. If we read the empty string forward, we have the empty string, if we read the string backward, we also have empty string. So, our goal is to define a procedure that will take any string as input and output true if that string is a palindrome. I am going to view, give you a few hints how to do this, but start thinking on your own, if you can think of a way to define a procedure that tests whether or not a string is a palindrome.

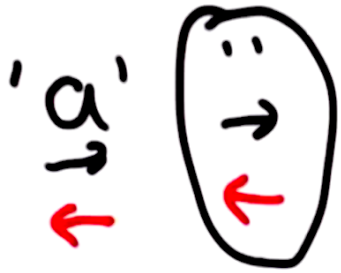
So, this is a pretty tough question. See how far you can get on your own, but I'm going to provide some hints before we make the quiz. So, the first hint is, we want to think of what it means to be a palindrome. To try to formalize that definition, if we knew easy ways to produce the reverse of the string, and check if it's equal. Well that would be an easy way to solve this, and in fact there are ways to do that in Python. But we haven't seen them yet. And I want to have you think about ways to do this palindrome that don't depend on that. So, that means we need to think about a way of defining a palindrome in terms of simpler things. So, the first thing to notice is, there's one simple case where we know right away whether a string's a palindrome. So, we should think that that might be our base case. When we do procedures on numbers our base case is often something that deals with a number like zero or one, a small number. When we do recursive procedures on strings, it's more likely that our base case is going to be the simpler string, which is the empty string. So, we know that if the input is an empty string, the result of this palindrome is true. The empty string is a palindrome. What if the input's not an empty string. Well then, one way to solve that would be to look at the first letter of the string, and look at the last letter

Copyright © 2014 Udacity, Inc. All Rights Reserved.

of the string. If those two are equal.

Well, then it might be a palindrome. It's a palindrome if all the ones left over in the middle are also a palindrome. So, this is how we could break the question of testing whether a string's a palindrome into smaller steps, that our recursive case is going to test the first and the last character of the string, see if those are equal, so if our recursive case. We're going to test the first and the last characters of the string. If those don't match, we know it's not a palindrome. That means we know the result is false. If they do match, we're not done. We need to check the rest of the string. And that means, we need to check the middle of the string, if that's a palindrome. And because we're able to define this recursively, remember that we're defining the procedure is_palindrome, but we can do this test assuming that we've already defined it. In order to check whether the middle of the string is a palindrome we can use the procedure that we're defining. This is like we were able to use factorial to define factorial in terms of a smaller number. In this case, we're defining a palindrome in terms of the smaller string. So, I hope this is enough for you to get started, so see if you can define the code for is_palindrome, keeping in mind that we can break it down into these two cases where if it's empty, we know it's a palindrome right away, if it's not empty, well, we need to look at that first and last characters. If they do match, we also need to look at the middle of the string.



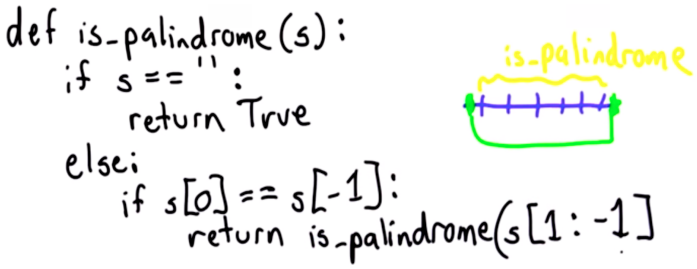


Answer:

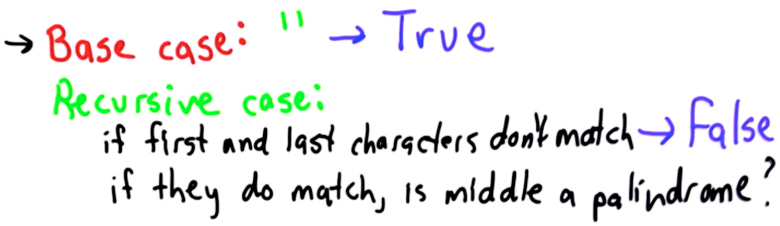
So here's a way to define is_palindrome. So we're taking a single string as an output, I will call it s, and we're first going to test the base case. So the base case was to see if the string is empty, we should return true right away. So we can do that with an if. We are going to check if s is equal to the empty string, and if it is, we return true, that's our base case. For the else, we have the reverse of case so now we need to do the test of the first and the last characters to see if they match. And we can do that using the string indexing operators, as 0 gets us the first character, s negative 1 gets us the last character. If they match, then we need to check the rest of the string. If they didn't match and let's finish the didn't match case first. So if they didn't match then we know it is a palindrome, because we found a place where the first and the last character did not match, so it should return false right away. If they did match, well then we have the harder problem. We need to do the recursive call to check all

Copyright © 2014 Udacity, Inc. All Rights Reserved.

the other letters in the strings still form a palindrome.



So this was our starting string. It had all these characters in it. We checked that this one matches this one, so now what we need to do is take the rest of the string and check if this is a palindrome. So that'll be a recursive call, so we're going to return the result, of calling is_palindrome. But instead of passing in s, what we want to do is pass in the string starting from position 1 of s, so removing the first character, and ending at position negative 1, removing the last character. And remember with our indexing, the last value here is not included. So by having the last index as negative the first thing to test is the base case. So we'll pass in the empty string. And the empty string is a palindrome, so it should give us the result true, which it does. Let's try the single letter string, a, that's also a palindrome. It's the same backwards and forwards. And so we also get the value true, if we try say string ab, which is not a palindrome. We get false. As a longer test, if we try a level, we should get true. Which we do. And, let's try one of the most famous palindromes, amanaplanacanalpanama. And we should get True which we do.



Recursive Vs Iterative

So any procedure that we could write recursively. We're going to also write without using a recursive definition. And I want to show you another way that we might have defined is palindrome. So here we're doing this with the for loop, and we're looping using the variable i and the range from 0 to the length of s divided by 2. So that's going through halfway of s. And inside the loop we have an if-test that checks if they character at position i is different at the position negative i plus 1. So that's going to be counting from the back of the string, i's positions away. If those are different, then we've found a mismatch and we return false. If they're not different, we're going to keep going through the loop. Once we get to the end of the loop without finding any differences, we know it's a palindrome and we return true. So this is also another way we could define is palindrome. I think this is more complicated to understand, and harder to get right. It took me three tries before this code worked correctly, whereas the recursive definition, I could get right the first time. If we wanted to test very long palindromes, this would be much more efficient, than the code that we had with the recursive definition. And there are a couple of reasons for that. One is that the recursive definition keeps making new strings, every time we do the recursive call, we have to create a new string, and that's pretty expensive. Another reason is that the recursive calls themselves are fairly expensive, and

Copyright © 2014 Udacity, Inc. All Rights Reserved.

there are languages that make recursive calls fairly cheap. Python is not one of them. In Python, it's fairly expensive to do a recursive call. So for most procedures, the recursive way is often the most elegant, and the easiest way to get correct. If we're really worried about performance, and we need procedures to work on really large inputs. We're often better off trying to find a non recursive way of defining that procedure.

Programming Quiz: Bunnies

So before we get to the problem of ranking web search results, I want to talk about one more interesting workers definition, the fibonacci series. And fibonacci numbers are one of the most interesting things in mathematics. Once you know about them, you'll see them all over the place, both in nature and in human designs. The name comes from Leonardo da Pisa who was also known as fibonacci. And back in is the same as the one for abacus the calculating machine. And this translates loosely as book of calculus. This was the book that introduced Indian mathematics to the west. In particular it introduced what we now know as Arabic numerals. This replace the Roman numeral system that was then widely used. And part of what fibonacci did in the book was show, how much easier it is to do calculations, using numbers in the decimal systems where the place, w as part of the book introduce these problems, and s problem that became known as the fibonacci numb posed a problem like this. So at the beginning, we h a rabbit to produce offspring, and every month a m so at month one, we had one pair of rabbits. At mo And now we have a new pair. We have two pairs of aren't yet ready to produce offspring. It takes a mo will produce new offspring each month. So we have were born in month two, well they've had a month had three pairs at the end of month three. So mont had, rabbits never die in this model, and the two pa produce new rabbits. So we have two new pairs of month three, well now those are mature. So now we have three pairs of mature rabbits, and two pairs of baby rabbits, going, so the model assumed that rabbits never die, that every month each pair of mature rabbits produces a new pair of rabbit babies. And it takes one month for a pair of rabbits that spawn, to become mature. So in month five, the three mature pairs of rabbits, will all produce new offspring. New offspring, new offspring, three new offspring. And the two that are a month old, that were born in month four, now become mature. So this isn't a very accurate model of how rabbits reproduce. It's good for us, if it was an accurate model, it would only take a few years for rabbits to control the entire planet. But it's an interesting mathematical model. And the model that this poses, we can write in a more formal way. So, the number of rabbits is the number of rabbits we had in the previous month, since those rabbits don't want to die. So in month five, we have the five pairs that we had in the

Copyright © 2014 Udacity, Inc. All Rights Reserved.

previous month, plus all of the rabbits that were mature, meaning, all of the rabbits that we had, two months ago, which was three if we're on month five while those reproduce. So we have three new pairs of rabbits, plus the five that we have in the previous month. And this keeps going.

So we could in month six, we're going to have the eight rabbits that we had at the end of month five, plus the five mature pairs, one, two, three, four, five, will each reproduce. So we'll have five new pairs, and that will give us 13 pairs of rabbits. So this was the model fibonacci developed. And the question is, can you figure out, at the end of month n, given any number n, how many rabbits will there be? So

the way we define this mathematically is a little different from the way, fibonacci posed the question. And that's because in modern mathematics, we usually like our series to start with a zero. If we are thinking of modeling rabbits, well that doesn't quite make sense to start with zero rabbits. But if we're thinking of it as a more general series, it does. So the way it's defined mathematically is that we say that the fibonacci number zero, is defined as zero. Fibonacci number one is defined as one, and those are our two base cases.

So this is different from the other recursive definitions we've seen in that the two base cases. Previously, all of our definitions just had one base case. And then we can define every other fibonacci number, but first we'll be starting from these base cases. And so, the fibonacci number n, where n is some whole number greater than one, is, well we have all the rabbits in the previous month. So, that's fibonacci of n minus 1, plus, all the new babies. And the number of new babies is the number of rabbits we had two months ago, those are all the mature rabbits. That gets added to the number of rabbits we had the previous month. So that's how we define fibonacci numbers. This defines every fibonacci number in terms of the two base cases, and in the one recursive case. So, your goal is to define a procedure called fibonacci. That takes a natural number as input. So, numbers starting from 0, any whole number 0 or higher, and outputs the value of that fibonacci number, defined using this recursive definition.

Answer:

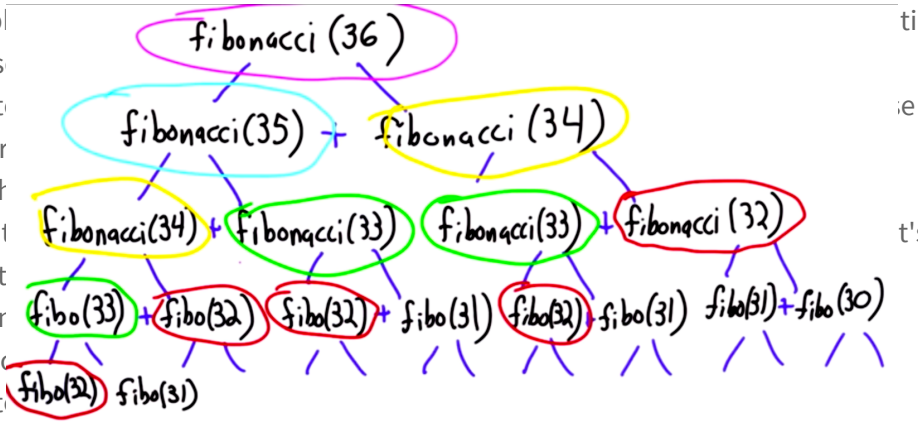
So here's one way to solve this, we're going to define a procedure Fibonacci and we'll call the input n. And now we need to write the code, and if we remember our definitions; so we said Fibonacci of 0 is

Copyright © 2014 Udacity, Inc. All Rights Reserved.

defined as 0, Fibonacci of 1 is defined as 1. And Fibonacci of any higher number is defined as Fibonacci of n minus 1 plus Fibonacci of n minus 2. So if you remember the definition, we have two base cases we need to consider. If the input value is 0, or the input value is 1, we need to do something special. So we could write those as separate if statements. So if n is equal to 0, what we want to do is return do is the recursive part of the definition. So we want to return the result of Fibonacci n minus 1, and we want to add that to Fibonacci n minus 2. So we could simplify this a little bit. Let's try this in the Python interpreter.

Divide And Be Conquered

We're going to define Fibonacci, so here's our definition, and let's try this out, so we'll print the result of fibonacci 0, first, we should get 0, which we do. We'll try fibonacci 1, we get 1, so we've seen our two base cases. Now, when we do fibonacci 2, what we should get is the result of. Fibonacci 1, which is 1, plus Fibonacci 2, minus 2, which is Fibonacci 0, which is 0. So we should get 1 again, which we do. And now let's try printing Fibonacci 3, we get 2. And if we print Fibonacci 4 we should get 1 plus 2; which is 3. Which we get and if we try 5, we should get 2 plus 3, which is 5. And let's try something a little bigger, let's try Fibonacci 10. We get 55. We'll try Fibonacci 25, so we'll try Fibonacci 24. So for counting months that



would be the number of rabl times out. So I'm going to try the s ones. But, the time it takes t e we're doing lots and lots of r time, we call Fibonacci n, wh Fibonacci n minus that was t t's going to be broken down int fibonacci, so the call to fibor that result to what we get fr that's going to be turned int to do lots, and lots, of calculations here. And we haven't got close to getting down, to fibo that we stop. If we look at the number of times we have to evaluate, fibo 32, let's pick that one, so Fibonacci 32,. So we need to evalu times we had to evaluate, Fibonacci 33, was 1, 2, 3, times The number of times we had to evaluate Fibonacci was 1 time, and the number of times we had to evaluate fibonacci 36 was 1 time. So, maybe you can see a pattern here. How many times do you think we're going to need to evaluate, fibonacci 31?

Quiz: Counting Calls

So, maybe you can see a pattern here. For the quiz, I'm going to see if you can figure out how many times we'll need to evaluate Fibonacci 30. Evaluate fibo 30 when we're evaluating fibo whole thing

Copyright © 2014 Udacity, Inc. All Rights Reserved.

and counting it, that's going to be a lot of work. But if you think about what we've seen so far, maybe you see a pattern here, and you can figure out the result for how many calls there would be to fibo 30 without doing all that work. So see if you can figure out the answer without working out every step.

Answer:

So the answer is 13. And the reason for this is quite interesting. If you look at the answers we were getting so far, this is the number of calls for each number. So when we had fibo 36, the number of calls was also 1. When the input was 34, the number of calls was 2. When the input was 33, the number of calls was 3. When the input was 32, the number of calls was 5. We should start to notice a pattern. This is exactly the Fibonacci series that we're computing here. Every time we decrease n by 1, so to get the number of calls for 32, we added the number of calls for 1 before. And the number of calls for of this, it follows from the same rule that we had; thinking about the number of rabbits. Reproducing. That everyone that we have on the previous level, leads to two more. And that means as we do this addition, we have all the previous calls.

So we had, 1 call the 34 at the previous level. We're going to produce two new calls but the inputs are different. But, if we look at the way the inputs are distributed, we have 2 plus have 2 plus 3. We have 5. For 31, we're going to have eight calls, and for 30, we're going to have 5 plus 8, which is 13. And this will keep going. These numbers will get. Quite big very fast, and the number of calls we need, every time we evaluate fibo 36. So the evaluate the number of calls, when we started with 36 that's going to be the result of fibo calls we'll need to fibo 2. And evaluating fibo 36 will be fibo 33 calls, and we don’t know what that number is yet, because when we tried to evaluate fibonacci 36, using our recursive definition, our evaluator timed out, so we're in big trouble. If we want to figure out how many calls there are, we need a more efficient way of computing Fibonacci numbers.

So let's see if we can do that. And the reason this was so inefficient was because we're doing all this redundant work, right? We saw that to compute fibonacci 36, well we had to compute fibonacci 35, and we had to compute fibonacci 34, all of this work computing fibonacci 34. We did the same exact thing over here, right? This is producing the same output. There's tons and tons of redundant computation going on. So if we're going to compute this more efficiently, we don't want to duplicate all that work. We need to do it in a way, where we don't need to keep recomputing the same thing. So the solution to this, is instead of using a recursive procedure to compute Fibonacci. We're going to compute fibonacci using a while loop. Anything that we can define recursively, we can also define without using a recursive definition. It's often much easier and cleaner to think about things, with a recursive definition, but it's often not the fastest way to calculate things, and certainly, in this case. Because of all the redundant computation, it's a very, very inefficient way to calculate fibonacci. So let's try to do this with a while loop instead and we'll make that a quiz.

Copyright © 2014 Udacity, Inc. All Rights Reserved.

Quiz: Faster Fibonacci

So the goal is to define a faster Fibonacci procedure, fast enough to enable us to compute Fibonacci 36. To estimate, at least according to Fibonacci's model, how many rabbits there will be in three months. And I'm going to give you a little hint for how to do this. So we're going to want a while loop, and the loop. It's going to go up to n. But within the loop you are going to need to keep track of two things. You are going to need to keep track of the two previous fibonacci numbers. And instead of going backwards the recursive definition did. We started with our base cases, we started with our base cases with zero and one and then we defined every previous case by adding the previous two. So to compute this with the while loop. If we keep track of the previous two, invariables then you can compute the next one by adding those. And then what you gotta figure out is how to keep up to date the variables, to know what the previous two are each time you go through the loop. So see if you can figure out how to define fibonacci yourself. Test it on some of the smaller numbers before trying it on fibonacci 36. But if you define it this way, you should be able to compute Fibonacci numbers for much higher inputs than we could with our recursive definition.

Answer:

So here's a way to define Fibonacci iteratively. We're going to avoid all the redundant computation by keeping track as we go. And we're going to have two variables. And I'm going to do this in a slightly strange way, and the reason for this will become clear soon. I want to make it so we can get the right answer when n is 0 and when n is 1, without having special cases. So instead of keeping track of the previous two, I'm going to keep track of the current one and the imaginary one that's going to be after that. And we know that the first two Fibonacci numbers are 0 and 1, so we'll use current is So that's the one after the one that we're currently doing, and now we have a loop, so we're going to go from i in the range from 0 to n.

So we're looking for Fibonacci number n, that means we want to start at 0. The current value is the value for Fibonacci 0 and after is the value for Fibonacci updating those. And we want to update them by following the recursive rule and so that means that the new value of current, is the current value of after. And the new value of after, is the sum of those two lists; current plus after. We can do that with a multiple assignment, that'll save us from needing a temporary variable. We can assign current and after, to their new values. The new value of current, is the current value of after, and the new value of after is current plus after.

So, this is the place where a multiple assignment comes in handy. If we didn't use a multiple assignment, we'd have to use a temporary value to keep track of one of these while we do the assignments. But with multiple assignment, we get both of these values first, and then we assign them to the two variables on the left side. So, that's all we need. And then after the loop, we should return the value of current, which is the current Fibonacci number if we're looking for Fibonacci N. So

Copyright © 2014 Udacity, Inc. All Rights Reserved.

let's try that. So we should be able to see Fibonacci 0, and the result should be 0. And that's what we get, and, because that's the value of current, when the range is from 0 to 0, we don't go through the loop at all. So we get the value 1. Let's check Fibonacci 1, and we run this, we get the value 1, which is also what we expect. And we got that because we went through the loop once, assigning the value of after, which started as 1 to current and that's what we return as the value of current. And we can keep going, we'll try Fibonacci 2. And that's also 1, as we expect and Fibonacci 3 should be 1 plus 1, gets us 2 and so forth. Okay, so this looks like it's working. We've tried a few simple ones. Let's try Fibonacci 33. So we estimated in the earlier quiz, in order to compute fibonacci 36, we would need Fibonacci 33 calls, using the previous recursive definition.

So, why did it take so long for that code to run? So what's the value of Fibonacci 33? And that's what it is, it's 3 and a half million calls. And so even with a processor that's doing a billion instructions a second, doing 3 and a half million recursive calls takes quite a while. Each time through the call, is many more than just one instruction it's many thousands of instructions. So this starts to take enough time that we didn't see the result. And, it wasn't only those Fibonacci 33 calls to Fibonacci 2, we had all the other things that we had to do to get Fibonacci 36. But let's see that now we have our faster, internet definition of Fibonacci that isn't doing all that duplicate work, that we can compute Fibonacci 36. And so that gives us this value, so indicating that there would be about 15 million rabbits after 3 years using Fibonacci's model. Let's try what we'd have after 5 years, passing in 60 months, and we get this starting to be quite a huge number. To try to relate to this, let's look at how long it would take for the mass of all the rabbits that are born to exceed the mass of the Earth. So the mass of the Earth is 5.9722 times 10 to the 24, and that's in kilograms, and I'm using the times time notation. This gives us a power, so this is 10 to the power 24.

So that's one way to write 5.9 times 10 to the 24 kilograms, just to demonstrate the power notation, this is is 1,024. That's what we get by multiplying 2 times 2 times 2 times 2 ... 10 times. Here we're multiplying 10 by itself 24 times and that's a good estimate for the mass of the earth. So now to find out how many months it takes before the mass of the rabbits exceeds the mass of the earth, we're going to have a for loop. We're going to loop from Fibonacci numbers, until we get to a number that exceeds the mass of the Earth. We also need to decide what a mass of a rabbit is, and I'm going to assume that a rabbit weighs about 2 kilograms. And, that's a pretty good guess for how heavy a rabbit is. That's assuming of course, a well fed rabbit like we have today, not if the rabbits spread as fast as Fibonacci's model would suggest that they do. So, we'll write a loop to see when the mass of the rabbits exceeds the mass of the earth. We'll start with n equals 1, and we're going to keep going until Fibonacci n exceeds the mass of the earth.

So, we'll go while Fibonacci n times the mass of the rabbit. So Fibonacci n gives us the numbers of the rabbits in month end, times the mass of the rabbit and as long as that is less than the mass of the Earth, the Earth is still safe for humanity, or at least there's some space left for humans. And every time through the loop, we'll increase n by 1. And at the end of the loop, we'll print out the value of n, we'll see where we got, and let's also print out the value of Fibonacci n, to see how big the Fibonacci

Copyright © 2014 Udacity, Inc. All Rights Reserved.

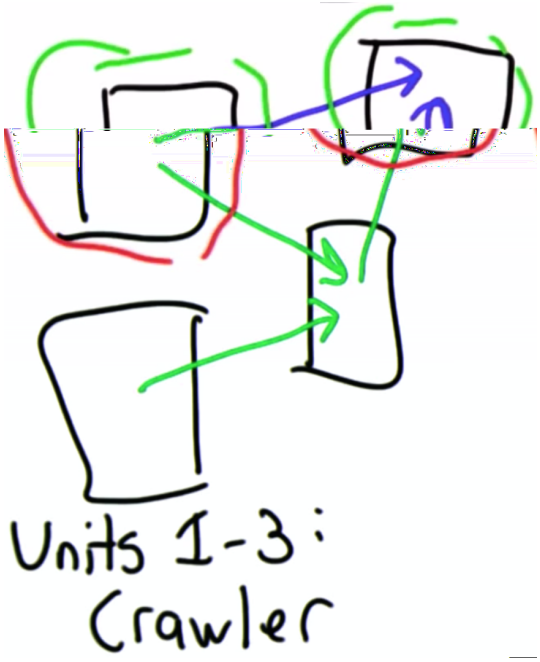
number of that n is. So we'll keep going through the loop, as long as the Fibonacci of n times the mass of the rabbit is less than the mass of the Earth. And once we stop the loop that means we've exceeded the mass of the Earth and we'll see what the results is. So let's try running that, and, we get this result. The value of n is 119, so it'll only take 119 months, or just less than 10 years, until the mass of the rabbits exceeds the mass of the Earth. And this is the number of rabbits we would have then. A pretty big number, you should be very afraid of all these rabbits. The good news is that Fibonacci's model, is not actually correct. That this was a mathematical abstraction for rabbit reproduction. Real rabbits actually die off after some point, and if there're too many rabbits, they don't have enough food, so they don't keep growing like the Fibonacci numbers and take over the entire planet. So we should be very afraid if Fibonacci's model is correct. It would only take 10 years for the rabbits to take over the entire planet, and weigh more than the earth does itself. The good news is, it's not a very accurate model of how rabbits reproduce, that they don't live forever, and once there're too many rabbits, they start to run out of food. So they stop reproducing, and stop surviving.

Ranking Web Pages

So now that you've survived the bunny uprising, we're ready to get to the main goal of this class, as far as building the search engine. Our goal is to improve the results by finding the web page instead of just returning a list of all the web pages that match a query. As the web has grown, it's become more and more important for search engines to do this ranking well. That what really distinguished Google from previous search engines was they had a much smarter way of ranking pages that produced more useful results, where the first one or two results in response to a search query were often the very thing that the user was searching for. So now we're ready to start thinking about the problem of how to rank web pages. Let's start by recapping how our search engine works. So we started by building a crawler. And what the crawler did, and this is what we did in units one, two, and three, what the crawler did was follow all the links in the web pages. Following those links, building up an index. And the end result of the crawler, after units four and five was we had an index. By the end of unit five, it was a table where we could look up a key word, and we would find the entry where that key word might appear. And we'd follow, and we could look through each of those entries to find the one that matched, and that would match the key word that we were looking for. And as it's value, it would have a list of all the urls where that keyword appears. And the order of those urls in that list was determined just by how we added them to the crawl. Every time we encountered new page, we

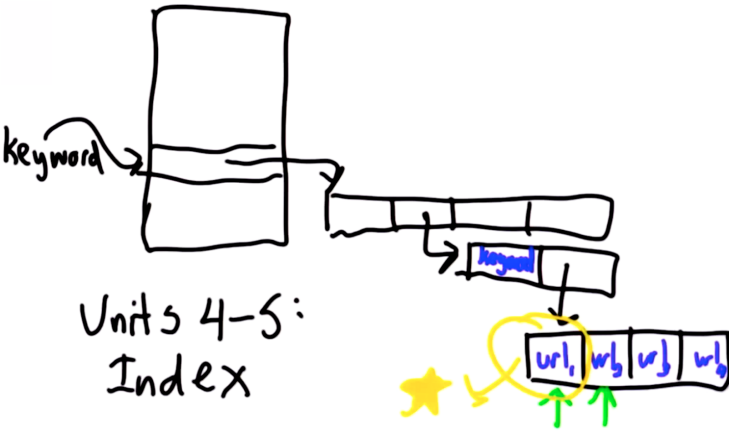
Copyright © 2014 Udacity, Inc. All Rights Reserved.

indexed that page, and we added a url for that keyword. So the one that's first in this list is just the one that we happen to find first. So say it's the c page. The one that's second would be the one that we find next. And it's this page. So, this doesn't tell us anything about which page is best. So the order of the URLs in the list and what we were getting as our output just depends on the order that things happen to go in the crawl. When the web was really small, which was quite a while ago now, this was sort of okay. That there were only a few pages that might match a given keyword, and you could look through them all and decide which one you wanted. With the web today, this doesn't work at all. There are thousands of pages that match any interesting keyword. Maybe millions. Certainly many more than you want to look through by hand. So the most important thing that a good search engine does is to figure out how to rank these pages so the one that's at the front of the list is the one the user wants. So that's our goal for the rest of this unit, to figure out how to rank pages. Before we do this for web pages, we're going to do something very similar, but perhaps easier to relate to. We're going to talk about how we decide who's popular.



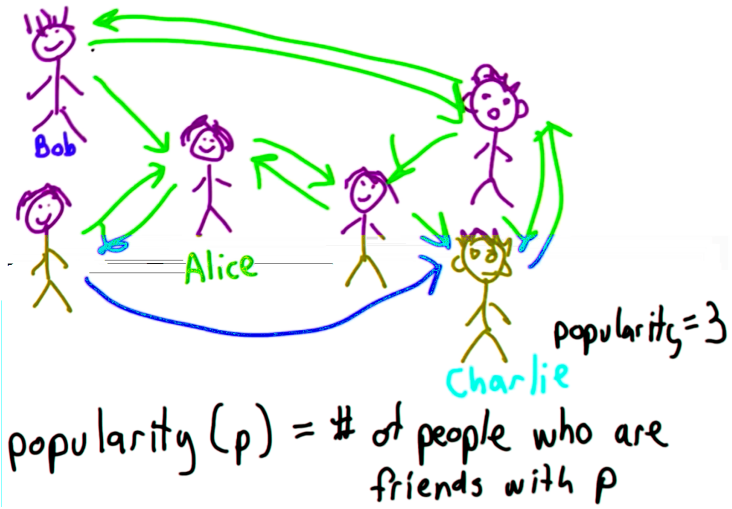
Popularity

So here's a typical group of friends in middle school. And some of the people are popular, some of them might not be. The first step to deciding who's popular, is looking at who has a lot of friends. So let's draw in some links, that show who's a friend. And friendship links are one directional. Just because Alice's friends with Bob, doesn't mean that Bob is friend with Alice. So we'll draw our links as arrows, so this means this person, we'll call him Bob, is friends with Alice. And let's say, Alice has many friends. And let's say Bob is also friends with this person, they're friends with each other. We have lots of friendship links. Some of them are bi-directional but not all of them. So, we have lots of



Copyright © 2014 Udacity, Inc. All Rights Reserved.

friendships. So, is this enough to decide who's popular. So, if you went to a school like I did, it's not. Just having a lot of friends is not enough to make you popular, you have to have the right friends. You have to be friends with the popular people. So, it's not enough to have lots of geeky friends say in high school, you've gotta have lots of friends that are popular. So that means the definition of popular, isn't just about having lots of friends, it's about having lots of friends, who also have lots of friends. That's what make some one popular, so we can define popularity of a person is the number of people who are friends with p. This means the number of links from someone else to that person is their popularity score. So here is Charlie, so there are one, two, three links into Charlie, so Charlie's popularity score would be three. Alice also has three links, so her popularity score would also be three. Bob only has one arrow going to Bob, so his popularity score would be one. So, this isn't a bad way to define popularity, but it's not quite right. So, the definition of popularity doesn't just depend on the number of friends you have, it depends on, both, the number, and the popularity of your friends. So, we can change the definition. Let's instead define the popularity score of a person p. Now it's going to be the sum of the popularities of all of their friends.



So we can write that in a mathematical way, so using the sigma means to sum up. We're going to take each friend, that is in the friends of p. And we're going to sum up the popularity score of the friends. If the mathematical notation is unfamiliar to you, we could also write this as pseudo Python code. We're thinking of the popularity of a person p. Let's assume we have a function that gives us the friends. So we're going to start with a score of zero. We're going to loop through the friends. And for each friend, we're going to add to p score, the popularity score of the friend. And we'll return the score as the result. So now, you've seen a mathematical definition of popularity, you've seen the same thing as code. I'm going to ask you a very important quiz question next. Its an easy one to get right if you try it twice because there's only two answers, but think about it carefully. See if you can get it right the first time.

Quiz: Good Definitions

The question is, is this a good recursive definition? The answers are yes, or no. And for something to be a good definition, well it has to provide a meaningful answer, for all possible inputs.

[ ] Yes

Copyright © 2014 Udacity, Inc. All Rights Reserved.

[x] No

Answer:

So the answer is no. That it does not and the key reason why it does not is there's no base case. Without a base case, it's not a recursive definition. What it is, is a circular definition. And that means that it actually doesn't give us any answers. Because we'll never finish here right. The only way that we've defined popularity whether you look at that mathematical expression or the python code. Always involves calling popularity again. And we are never going to get to a point where we can stop. That we are going to keep calling popularity for different people. If we think of doing this for our friend graph, well here's what would happen. If we wanted to know the popularity of Charlie. Well, we have to look at all the people who are friends with Charlie. So, that means that the popularity of Diana, the popularity of Edgar, and the popularity of Fred. But, for each of those, we need to compute their popularities to the popularity of Diana. That's going to be equal to the popularity of Alice. So, now we're here, but we need keep going. And, the popularity of Alice is the popularity of Edgar, and the popularity of Bob. But we need to giv, keep going. We still need the popularity of Edgar well, that's the popularity of Fred. And we need the popularity of Bob, which is the popularity of Fred. And we need the popularity of Fred, which is the popularity of Charlie, plus. The popularity of Bob.

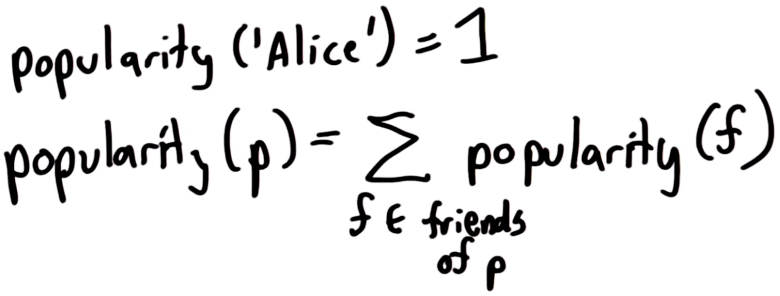
So, this is not okay, right? The problem is, we started trying to figure out the popularity of Charlie, we did all this work following these links backward. And now, to solve it, we need to know the popularity of Charlie. But that's not what we were trying to solve. So this is not a recursive definition. To be a good recursive definition, we have to end up with a simpler version of the problem. And never go back to the one that we started with. The way we've defined it, we get back to the one that we started with. We haven't made any progress. We're never going to get an answer. Charlie will be very unhappy because we'll never know how popular he is.

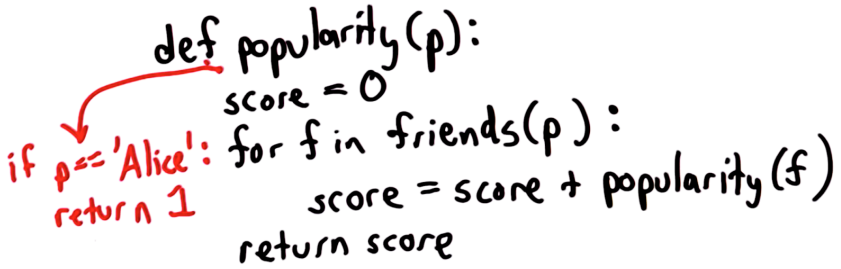
Quiz: Circular Definitions

So how can we fix this problem? Well, the first thing we should think about is, well, can we give a base case. All right. All of the recursive definitions we had. We had a way of stopping. So, we had a base case. Right. With, factorial, we said, we are going to pre-define, that we know the value of factorial when the input is 0. We know that the value is 1. We are not going to define it, in terms of factorial. We are going to note it's value. We did this for palindrome we said, palindrome, we have a base case, when the input string is an empty string, it's predefined as a palindrome We don't have to do anything else. And we did this for fibonacci, where

Copyright © 2014 Udacity, Inc. All Rights Reserved.

we had two bases cases? But for all these definitions, we had some starting point, that was not defined in terms of thing we are defining. And that's why it was good recursive definition, because we had the base case. We don't have one here. So let's try to invent one. Let's suppose that we made our base case. So if we're going to fix this, what we need to do is invent a base case. Maybe that will solve our problem. So let's try and add a base case. So. Suppose we assume we know the popularity of Alice and sadly Alice is not very popular. Her popularity score is a 1. So that looks like a base case, right we define the base case for factorial for 0 for palindrome for space. Let's pick Alice as our base case now. And. That works like this for the mathematical definition. For the python code, what we would need to do is add the base case, as in a statement. So we would insert a line here that says that if p is Alice, return Alice's popularity score which is our base case which is 1. So this looks more like the recursive definitions we've seen. Now we have a question. See if this actually works. So the question is, would this definition work? The possible answers are, only if everyone is friends with Alice, only if no one is friends with Alice, only if, from every person in the network. There's some way that you can follow links that eventually reaches Alice. Only if there are no cycles in the graph. So there's no way to start from one person and end up at the same person by following friendship's, by following friendship links. And the final choice is no, that there's really no situation where this works well.





[ ] Only if everyone is friends with ‘Alice’
[ ] Only if no one is friends with ‘Alice’
[ ] Only if there is a friendship path from everyone to ‘Alice’
[ ] Only if there are no cycles in the graph
[x] No

Answer:

So the answer is no, that even with any of these restrictions, we still don't have a good definition. So, let's consider all the restrictions. So, the first one was, if everyone is friends with Alice. So that would only work if they don't have any other friends. Let's say this is Alice. We've got Bob and Charlie. They're both friends with Alice. But Bob is also friends with Charlie and Charlie is friends with Bob. That means that to figure out the popularity of Bob we need to know the popularity of Charlie. To

Copyright © 2014 Udacity, Inc. All Rights Reserved.

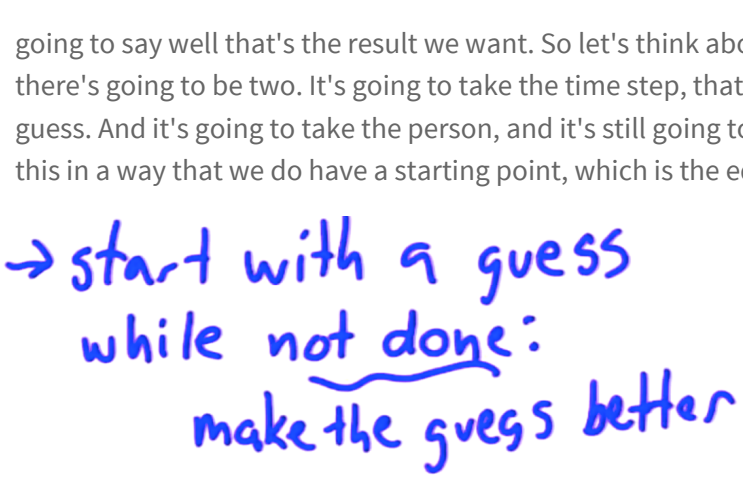
figure out the popularity of Charlie we need to know the popularity of Alice as well as the popularity of Bob. So we're never going to get to a solution. We are going to keep bouncing back and forth between Bob and Charlie doing this. The second choice only if no one is friends with Alice. Well, if no one was friends with Alice, that would remove these links. It doesn't solve our problem. We're still not going to be able to give a popularity score for Bob and Charlie. The third choice only if there is a friendship path from everyone in the graph that eventually reaches Alice. So adding this link would provide that property, but it still doesn't solve our problem. It doesn't give us a way to figure out the popularity of Charlie, because to know that, we need to know the popularity of Bob, which we need to know the popularity of Charlie for. We still end up in this cycle. The final choice seems, possibly more promising. It says there are no cycles in the graph, so if we want to remove this cycle. We could do this. In this case, we'd be okay, right? We could figure out the popularity of Bob, by figuring out the popularity of Charlie, which depends on the popularity of Alice. Where we're not okay, is if Bob has another friend. Let's say Bob is friends with Diana. Well then, to figure out the popularity of Diana, we need to know the popularity of Bob. Where it breaks down is, suppose we also have Diana and Edgar, and Diana's friends with Edgar. To figure out the popularity score of Diana, we need to know the popularity score of Edgar. We don't have a cycle, but we don't have an answer either. To figure out the popularity of Edgar,. We are going to go through Edgar's friends, and the way the Python code is written. This could actually work, right? Because if we define popularity when you have no friends. Well, if the friends of p is empty, when we go through this loop, the score is going to be 0. So, if you answered there are no cycles. That's at least worth credit for this, that could be correct. In terms of the mathematical definition, it doesn't make very good sense. We still needed a way to know the popularity of Edgar. We've sort of defined things in this case to say, if you have no friends, your popularity score is zero. And the Python code will work for that. But it's not a good way to define popularity. So its very arbitrary to say we're going to make Alice the one whose popularity score is predefined as one. There is nothing Alice could do to make herself more popular. That's not very fair to Alice and it doesn't give us meaningful scores.

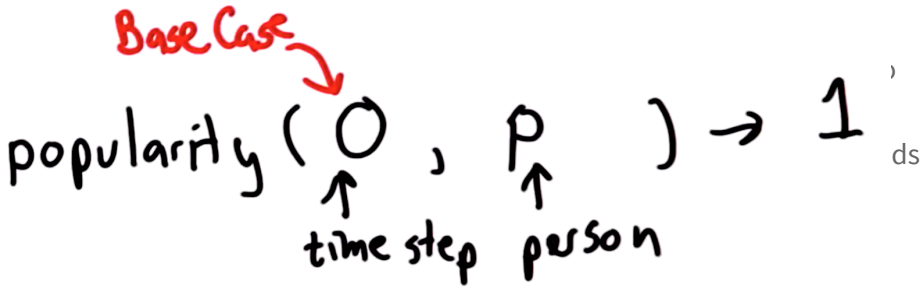
Quiz: Relaxation

So we need to figure out something different. There's no sensible base case that gives us a good recursive definition. What we're do, going to do instead is what's called a relaxation algorithm. And the basic idea is pretty simple. We're going to start with a guess and then we're going to have some loop where we keep going. We'll just say, while we're not done, we'll figure out how to decide when we're done. We're going to do something that makes a guess better. So, that's the basic idea. That v don't have a clear starting point of saying Alice's popularit everything in that. What we're going to do instead is we're be, well, let's assume everyone's initial popularity is one. guess. We're going to have some process that updates the time we do this, we're going to get better and better. And

Copyright © 2014 Udacity, Inc. All Rights Reserved.

going to say well that's the result we want. So let's think about how to do that for popularity scores. So, instead of just there's going to be two. It's going to take the time step, that's the number of times we've tried to guess. And it's going to take the person, and it's still going to output a score. And now we can define this in a way that we do have a starting point, which is the equivalent to having a stopping point in a recursive definition. We'll define what the value is at step zero. For any person, we'll say that the sco no clear ordering of the people. We had a graph, we had lo was no clear starting point. We're going to add an extra pa and that's going to be our base case. And if we think of ou time step zero is our base case. And at step zero, we're go is one. And now we can have our update rule, which says, step t, where t is greater than 0 for any person, and we're going to define that as the sum over all their friends. So we're friends. So we're going to go through each friend summing up the popularity score of the friend. But instead of at time t, we're going to use time t minus popularity of the friend in the previous step and we're using that to compute the popularity of the person they're friends with in this step. So let me write out that more in pseudocode. If we think of this as the Python procedure, it's going to be something where we're taking in two imports now, the time and the person. And now we have the base case, if the time is If the time is not 0, well, we can compute the score similar to how we did before. We're going to have the score at 0, we're going to sum over the friends. We're summing over the friends, updating the score, by adding the friends' popularity score at the previous step. And when we're done, the value of score is the result. So now we have our new definition. We have it written in mathematical notation at the top, and in the Pyth And I don't mean to ask w ask is, for all possible inpu popularity give us a result with themselves. It's good only a good definition if th





ds

[x] Yes
[ ] Only if people can't befriend themselves
[ ] Only if everyone has at least one friend
[ ] Only if everyone is more popular than 'Alice'

Answer:

So the answer, is yes, we have a good recursive definition, no matter what we pass in for t and p, we'll eventually get a result, and the reason for that is because every time we do a recursive crawl, the value that's passed in for t, is one less than the previous value, and eventually, if we start with an integer. And we keep reducing it, eventually we're going to get to the case where t is equal to 0, and then we have a base case. We can return the value one, without using the popularity function

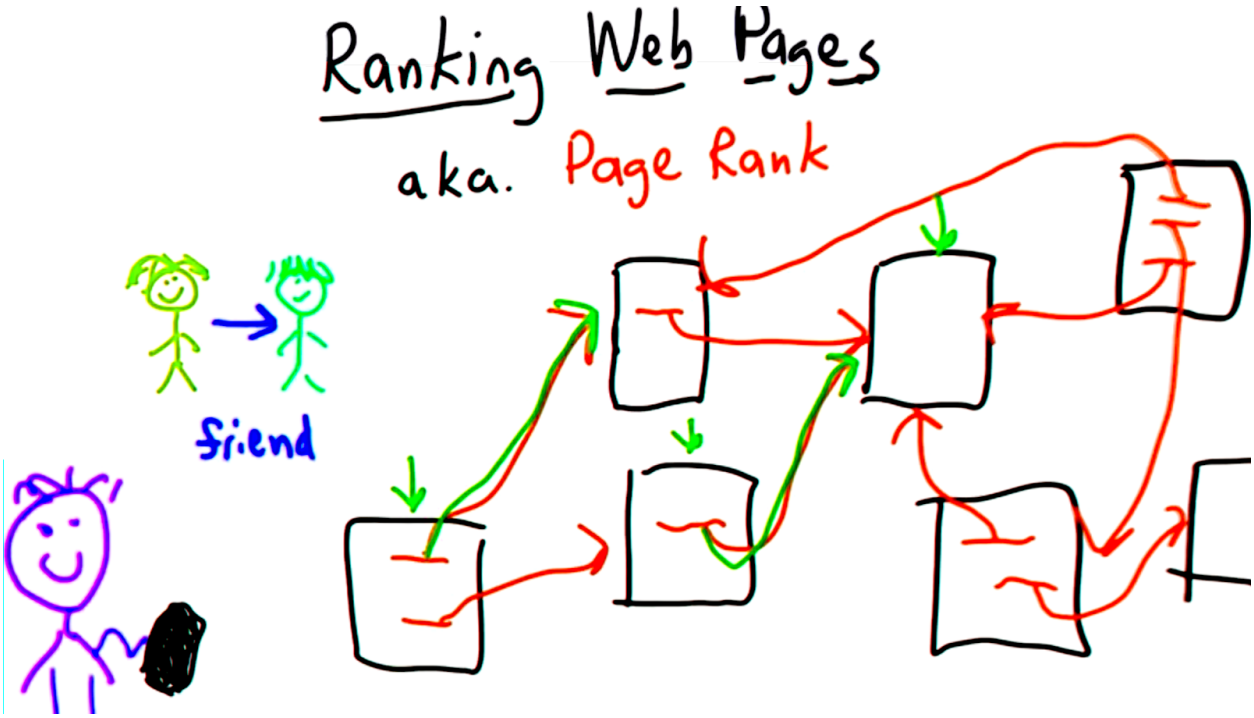
Copyright © 2014 Udacity, Inc. All Rights Reserved.

again. So this is a good recursive definition. It will always produce a result. It might not produce a meaningful measure of popularity, but if this is how we define popularity, this code will always produce a result.

Page Rank

So, that idea for how to rank web pages is the same idea as how we measure popularity of people. But instead of thinking about friendships as the way to measure popularity, what we're measuring is links on the web. When one page has a link to another page, well that indicates it's more likely that this other page is popular, just like when someone is a friend, it indicates that the person they are friends with is more likely to be popular. So, the goal in ranking web pages is to get a measure of how popular are pages, based on the other pages that link to it but we have the same issue with popularity then not all links are the same. That a link from a page that's really important counts for a lot more than a link from a page that is not important. So, if the New York Times has a link to your page, that counts for a lot more than if your mom sets up a web page and puts a link to your page in it. Unless your mom is Lady Gaga, in which case her link probably counts for more. So, another way of thinking about this is what we're trying to model is a random web surfer. So, our random web surfer has some set of pages that they know about. And those pages have links to other pages. Some pages might have a lot of links. Some pages might just have one link. Some pages might have no links. So, one way to think about this is that we're trying to model a random web surfer. The web surfer starts knowing about some pages. And she picks one page at random, let's say she picks this page. And then, when she's on that page, she picks a random link and follows that links. Whoops, this was a bad starting page. It actually has no out links.

Copyright © 2014 Udacity, Inc. All Rights Reserved.



So, then, she picks another random page. Let's say she picks this one. She follows the link from that page, and now she got to the page with no links again. Let's say she picked a better starting point. Let's say she randomly picked this one. Now she's got two links to follow. She randomly picks one of those. She follows it. She gets to a new page. She randomly picks a link from that page, in this case, it only had one, and in this case, it seems we have a bit of a problem, because all of the starting pages eventually lead into this one, which has no outgoing links. So, we'll think about how to solve that problem later, but we can think of what our random web surfer is doing, is picking random pages, following links, and what we want to measure is the popularity of a page. And that's the probability that she reaches that given page, starting from these random pages. So, if you did this over and over again, and you counted the number of times you reached each page, that would give you a measure of that page's popularity. So, this is very similar to the popularity function. We're going to define a function that we'll call the rank of the page. And, like the way we defined the popularity function, it's going to have two inputs. It's going to have a time stamp, and it's going to have the page which we'll use a URL for. And the output of rank will be some number. Except for we'll define for time step zero. This is our base case, and we're going to find all the ranks have value one. We'll actually change that shortly, but we'll start out think of it, all the ranks having value one, like we did with the popularity scores. And then we'll define the value of the rank at time step t for any given URL. Just like we defined the popularity score. It's going to be the sum of all the pages that are friends with this page, and what it means for our webpage to be friends with another page is that it has a link to it.

So, this is going to be for all the pages that exist that have some link to that URL, or its friends. And so, we're going to go through each of those pages. We'll call them inlinks instead of friends. We're

Copyright © 2014 Udacity, Inc. All Rights Reserved.

going to go through those and we're going to sum up the ranks that we got for those pages at time t minus one. So, this is our first model of popularity on, of web pages. This is exactly the same as the model we had of popularity for people. It's not going to work that well yet and one of the reasons it's not going to work that well is some pages might have lots of links. And if a page has lots of links, the value of each one of it's links should be diminished. It shouldn't have the same value as the page that only has one link that links to this URL. Maybe that should be the same case for a friend's popularity, if someone has lots of friends, each friend is less valuable. Whereas someone who only has a small number of friends has lots of time for each friend.

So this is the way we're going to model web popularity. We don't want to just give the same score to each link. We're going to change this by dividing by the number of outlinks. So, if a page has many outgoing links, the value to the pages that it links to is less for each page. So, a page that's just a long list of lots of links won't have that much influence on the rankings. If a page only has a few outgoing links, well then, they are worth more. So, what are going to do, is divide this by the number of out links from p. So, each of the p pages, right? These will be the values of p, as they go through the inlinks of URL. We are going to sum up the rank that we got on the previous time step and divide that by the number of out links.

Quiz: Altavista

So the final change we're going to make, is thinking about our random surfer model. If this was our model, if a page has no incoming links, well, it's popularity ranking would be 0. We don't want to have that be the case. If we think of our random web surfer, well if she randomly starts at a page, she could randomly start at a page that has no links. If we made all the page scores 0, for pages with no links, well, then it would be very hard for a new page to get started. So, we don't want those to be 0, so we're going to have some random probability that you reach a page even when it has no links.

So that means we're going to add something to this sum so it's not zero when there are no incoming links. And the other thing we're going to do is we're going to scale this summation a little bit. And we're going to scale that with what is called the damping function. And that just means, if we think of our random web surfer again, even if she's following a path. That does have more links. At some point, she might decide to get tired and give up and start again with a new random page. So, the dampening function determines how frequently, we think our random web surfer will pick a random link, versus starting over again on a new random page. So that's call the dampening value and we are just going to use the dampening value to scale this number. So, we'll call that d, that's the dampening constant. A typical value for that is something less than one, and a good value is something like 0.8. So now we're going to change our rank function to take into account that. We're going to have some initial value and if we want to keep the values in a reasonable range, instead of starting from one, we're going to divide the values by a total number of pages. And the reason for that is just to keep the ranks in some meaningful range. This will keep the ranks so the total of the ranks when we start the sum of

Copyright © 2014 Udacity, Inc. All Rights Reserved.

the ranks is 1, that gives them a little more meaning than if the sum of the ranks was the number of pages. So n is going to be the number of pages, the total number in our corpus and d is the damping constant, and so now we're going to change our initial values to divide them by n. This means, the sum of the paid ranks at the beginning will be 1, and, we're going to change the way we compute the rank by adding to it, a value that gives us the sense of starting over.

So, the value that we're going to use for this is 1 minus d, divided by n and, that gives the notion that. That times when we don't decide to do a new page. So, 0.8 of the time. We decide to follow a new page. So we're going to multiply this by D. The times when we don't follow a new page, that's one minus D. We're going to start over, and we're going to divide that by N, because N is the number of pages that the probability that this is the page that would be picked when we start over. So now we've got our recursive definition of pagerank. We start by initializing the rank n times at 0 to one minus, divided by N, and then for as many times up, as we want, we're going to keep improving our results. By using this formula. For each link that links to us, we're going to take it's popularity on the last step, divide that by the number of out links so it's dividing it by the number of other pages it links to. Multiply that by our damping constant and that gives the probability that this page was selected by the Random Web Server, and then we're going to add the term that takes into account the Random Web Server might have started over from scratch and picked a new random page. So, before we try to change our Web search engine to actually program this, it's time for a little quiz. And the quiz is, What is AltaVista?

And if you don't know this, feel free to use Google, or DuckDuckGo, to find out. These are the choices so, it's the view from the Udacity headquarters in Palo Alto. It's the most popular web search engine in 1998. It is Spanish for, You're Terminated Baby and, it's a small town in Virginia. And feel free to Google the answer if you don't know.

[ ] The view from the Udacity Headquarters
[x] The most popular web search engine in 1998
[ ] Spanish for “You’re Terminated, Baby!”
[ ] A small town in Virginia

Answer:

And answer is the second choice. It is also true that there's a small town in Virginia named Alta Vista. But the relevant answer is the second choice. Back in 1998, the most popular search engine was probably the search engine called AltaVista. They certainly had the biggest web index at the time. And they also had great technology for responding to queries quickly. 1998 is the year that Google was founded, and most people probably have not heard of AltaVista today, and, might have had a hard time answering this quiz. And the reason for that is Google figured out a better way to do page ranking. And because of this, search engines that didn't have good ways of ranking pages, like AltaVista, quickly become irrelevant. And the algorithm that I've described, is the algorithm that

Copyright © 2014 Udacity, Inc. All Rights Reserved.

launched Google. It's called PageRank. And you may think the Page stands for webpage. The Page actually stands for Larry Page, who was the co-founder with Sergey Brin, of Google. So, that's the algorithm that we've described. And that's what allowed Google to produce so much better search results, than other search engines at the time.

I hope you remember Anna Patterson from Unit Three, and are still following her advice to be polite on the web. One of the reasons AltaVista started to work so badly, was because websites were not polite. They tried to find ways to game the rankings. Certainly this is still something websites still do today. And the way AltaVista ranks pages, it was very easy to game them, so the site that would be ranked was not the best site about something, but a site that was best at gaming the rankings. So I asked Anna to explain why Google's page ranking algorithm was so important to the success of Google.

So a pagerank on the web, was a huge step forward. I'm old enough to actually remember the time that Google launched and you know, it's advance over AltaVista. Was AltaVista was very, very prone to keyword stuffing spam. So if you said restaurant 32 times, well you were better than a page that said restaurant twice. So, PageRank came out and actually gave a measure of popularity or traffic, and therefore, the things that said restaurant twice, but were awesome. They wound up going to the top.

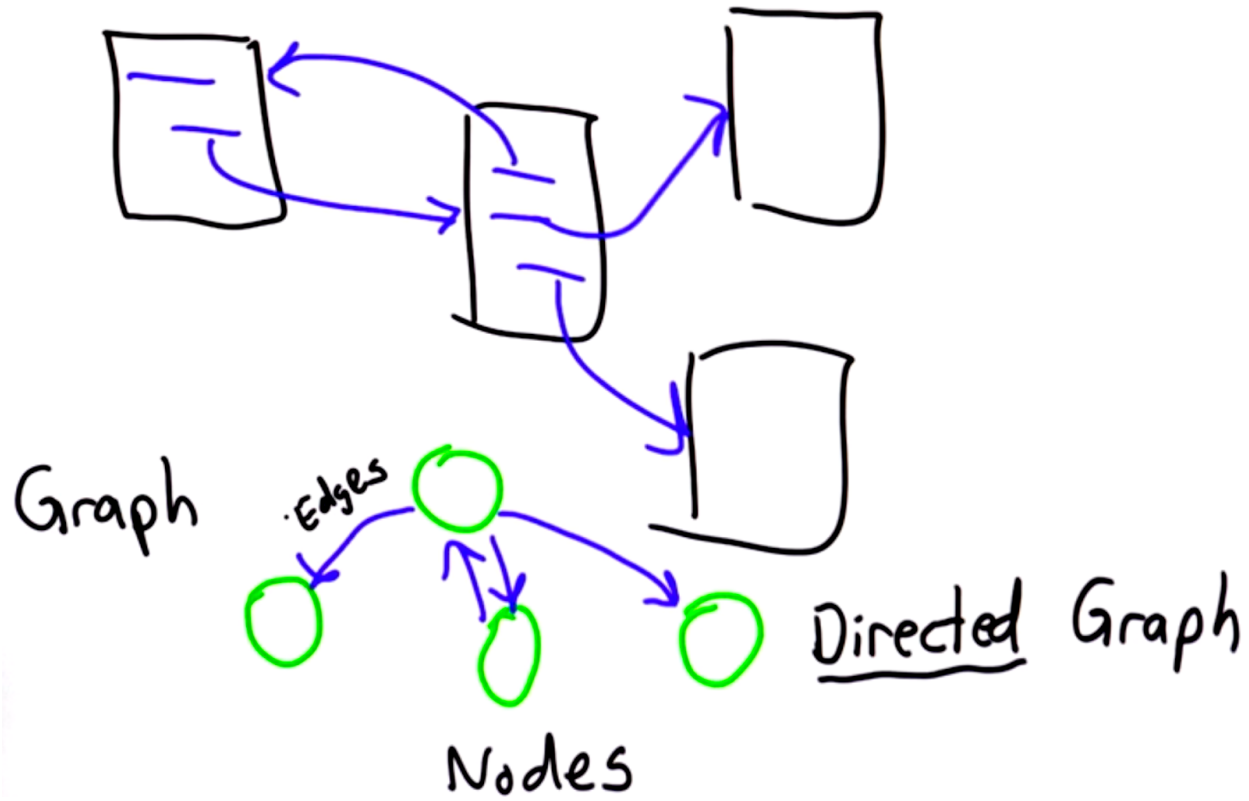
Certainly things have got more complicated since then, and there's more things that go into the ranking algorithm, than just what we've shown you. But that's the basic idea, and the next thing we're going to do, is change our search engine to implement this algorithm.

Urank

So the goal for the rest of this unit is to modify our search engine code to implement the PageRank algorithm. We have one little problem. PageRank is a registered trademark of Google. So we're not going to call our algorithm PageRank, even though it will do the same thing. We'll call it URank. The first thing we need to be able to do, to implement this ranking algorithm, is keep track of the link graph. So our popularity of pages depends on the link structure. So that means, we need to keep track of, what pages link to what pages. So, for each link, there is a connection between pages and we can think of that, as a graph. Abstractly, a graph is just a set of nodes, we'll draw those as circles, with edges between the nodes. And because our edges go one way, just like links in a page, we call this a directed graph. So, in order to represent our web link structure we need to build the directed graph. The pages in the graph are the nodes. For each page, we need to keep track of the edges that connect that node to other nodes. And so the way that we're going to do this is to keep a dictionary. So we're going to have a dictionary where the entries in the dictionary are the node, which is the URL, that's the page. And for each URL, we'll have a list of all the pages that it links to. So if this was say node a, and these were nodes b, c, and d, our entry for node a would contain the list b, c, d. And our entry for node

Copyright © 2014 Udacity, Inc. All Rights Reserved.

b, well, there are no edges out of b, so it would be an empty list. And finishing the example, C has an out link to one node, and D has no out links. So that's our goal. We want to build a structure like this that shows the structure of the webpages that we crawl, and we see that structure because we're following the links in our crawler. So our goal is to modify the crawl web procedure that we defined at the end of Unit Five. And to modify it so that instead of just producing an index, it also produces a graph.



So we're going to modify crawl web. It's going to still take a seed page as its start. But what it's going to produce now is both an index and a graph. And the graph is a structure that gives a mapping from each node to the pages that it links to. So let's look at the code that we had at the end of Unit Five and see how we need to change that. So here's the code, that we had at the end of unit five, for crawling the web. And as a reminder, we're keeping track of the pages left to crawl in the list to crawl starting with the seed page, and we're building up the index as a dictionary. And as long as there are pages left to crawl, we go through the while loop, which finds a page to crawl, popping the list of pages to crawl. As long as this one, we haven't crawled before, it gets the content from that page. It adds it to the index. It finds all the links, using get_all_links, passing in the content on the page and unions those with to crawl to update the to crawl list and then it appends this page to the list of pages, that have already been crawled. So to change this to build a graph, we're going to keep most of the code the same. In addition to producing just index, we're going to produce a graph. And the graph is also going to be a dictionary. And the reason the graph is a dictionary is that the mapping from nodes, which are urls, to the list of edges that go out from that node. So we'll create the graph as an empty dictionary. And as we find new pages, we're going to add them to the graph. And we're going to also change the

Copyright © 2014 Udacity, Inc. All Rights Reserved.

return to return both the index and the graph. I'm going to make one more change before we give you a quiz. And the change I'm going to make is, instead of calling get_ all_links here, Since both the graph building and the tocrawl list depend on knowing all the links, we're going to create a new variable. And we'll assign the result of get_all_links(content) to that variable. That means we can use those links as the input to content. But we can also use them to build the graph. And I'm going to leave the line that we need to build a graph for you to complete. So we'll make that a quiz, to finish this code. Write the line that we need to update the graph.

Programming Quiz: Implementing Urank

So your goal for this quiz, is to finish modifying the crawl_web procedure, so that instead of returning just the index, it returns both an index and a graph. And the graph gives the link structure. The graph should be a dictionary, where the entries in the graph are a URL, which is the target page, and a list. Which is the pages that link to that target. So to test your graph making code, we've provided a sample site at this URL. So if that's the link that you pass in as the seed for the crawl_web. It's going to crawl in the site we set up. And there's an index page that looks like this, it's got five links on it. And, those links go to different web pages. So the first one goes to my favorite hummus recipe.

So, there's a link from index.html. And it goes to the page hummus.html. And if we follow that link, we get to a page that has no links on it. This second link. Goes to page arsenic.html. And that page includes a link the Nickel Chef page. And the Nickel Chef page includes a link to kathleen.html. And that page has no links on it. And the third link on the index page, also goes to the kathleen.html page. And there's two more links on the start page. The first one goes to the Nickel Chef page, which we've already seen. And the last link, goes to the Zinc Chef. And from the Zinc Chef page, we have two more links, we have one to the Nickel page. And we have one to the Arsenic Chef's famous hummus recipe page. So this is a pretty complicated link structure, although it's very simple compared to the things that we actually find on the web. But your goal is to be able to produce this graph, by modifying the code that we have for crawl_web. And then once we have the page ranks, we'll be able to use PageRank to find the best page. To find a hummus recipe. So for this site, the graph that your crawl_web should produce, should have entries that show these connections, so there should be an entry where the URL is the urank/index.html, I'm leaving out the beginning of the URL. And the entries should be a list of all the pages that, that links to there.

So they're going to be five pages. All five of these should be in the list here. There should be the Hummus page, there should be the Arsenic page, and the other three pages. The order that. Links appear in this list doesn't matter, it's correct as long as you have all five of the links that you can reach from the index.html page. The kathleen.html page, doesn't have any outgoing links, so the entry for kathleen.html should be an empty list. So see if you can figure out how to change the code for crawl_web, to produce as its second output, a graph that shows the structure of the web pages that we crawled.

Copyright © 2014 Udacity, Inc. All Rights Reserved.

Answer:

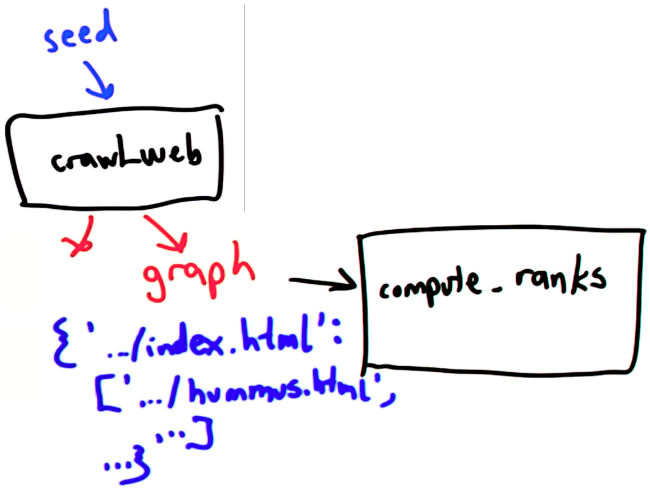
So the answer is, we only need to add one line. We need to add the line at line 13. And the line that we need, is to update the graph. So we're going to have a new entry in the graph, that is the page that we're crawling. So that's going to be the key for the entry that we want to associate with that page, a list of the links. That you can reach from that page, all the outgoing links. Well that's exactly what we stored in the variable outlinks. So that's the only line of code that we need to add, to produce the graph. We needed to initialize the graph as an empty dictionary, we needed to change the return it. But the only substantive change, was adding this line. And, we'll run that, we'll call crawl_web. Passing in the test URL. And this time we have two variables on the left side. So we're going to assign the index to index. And we'll assign the graph to graph. And let's check that the graph has what it should. So we'll print out to get graph entry for the index page. And, what we get, we see a list of five URLs. Hummus, arsenic, kathleen. Nickel and zinc. And this corresponds to the picture here. We had five outgoing links from the index page to those five other pages. Let's look at another one. We can print the graph for the Kathleen page, and that had no outgoing links, so what we see here is the empty list. And, we can see the whole graph, we have each URL. With a list of all the pages that, that page links to. So both the Hummus and the Kathleen page don't link to anything. The Zinc page links to the Nickel page, and the Arsenic page, and so on. So now we have our graph. The next step is to figure out how to use that graph, to do the page ranks.

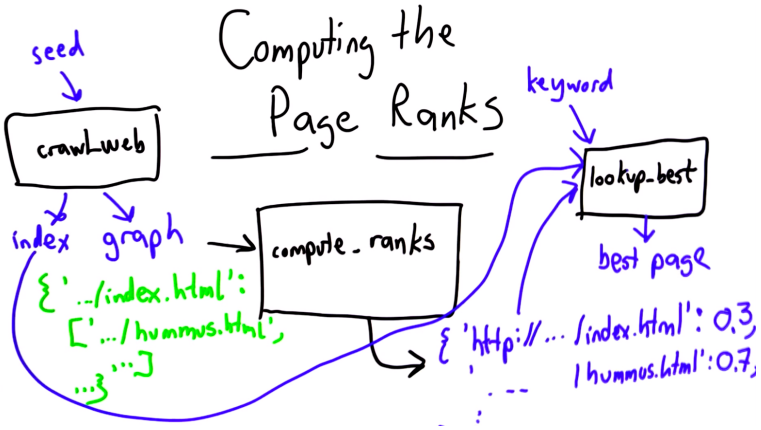
Computing Page Rank

So our goal is to write code now, that computes the formula that we worked out earlier. Then we want to compute the rank, for each page. The thing that we're going to do, instead of using this recursive definition. We're going to write a loop, that goes through the time steps, and we're going to figure out how many time steps we want. The more time steps, the more accurate rankings we'll have, but the longer it will take. So we'll just pick a value for the maximum number of time steps, and we'll make our loops go through those steps, computing these equations. So let's work out the code to do that. So now that we've got the graph. All that's left to do is figure out how to use it, to compute the page ranks. So to do that, we'll make a procedure called compute_ranks. It takes as input, a graph that gives the link structure. So, as an example we have our graph that has the URL's, and each URL, followed by the link of pages that it

Copyright © 2014 Udacity, Inc. All Rights Reserved.

links to. And this is the graph that was produced by our crawler, and crawl_web produced as outputs, a graph as well as an index. We're going to use the graph as the input to compute_ranks. So the output from compute_ranks, will be a dictionary, giving for each URL, Q rank, the ranking. That we compute using our formula. And, the higher the rank, the better. And, what we're going to do once we have those ranks, and this last part is going to be left as a homework assignment for you, we're going to instead of having just look up, we're going to have lookup_best. And lookup_best will take three inputs. It will take the keyword we're looking for. It will also take the index that came from the crawl, and it will also take the ranks, and from those three inputs, the index will give us all the pages that contain the keyword, and then what lookup_best will do is, use the ranks to find the highest ranked page. And give that as the output. So once we've done that, you'll have a complete search engine. You'll have a crawler that starts from a seed, finds pages, produces an index, produces a graph. That graph is the input to compute_ranks, compute_ranks uses our page ranking algorithm to figure out the pages that are most popular, and then lookup_best takes the graph as input. The index as input, and the keyword, finds the pages that contain that keyword, and identifies the one that's best, using the ranks. So that's what you'll have after the homework of this unit. We're going to finish compute_ranks now, and leave it up to you to define lookup_best as a homework question.



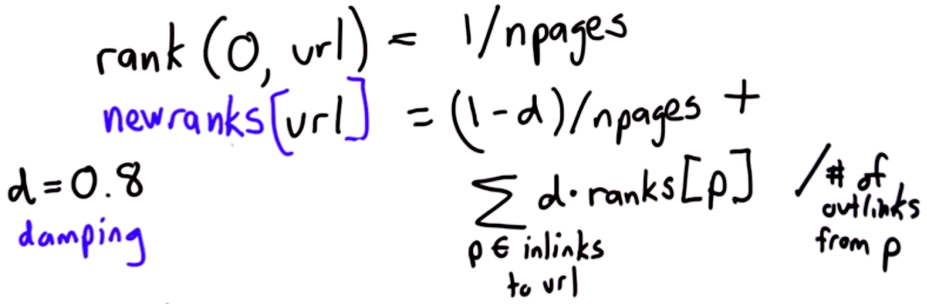


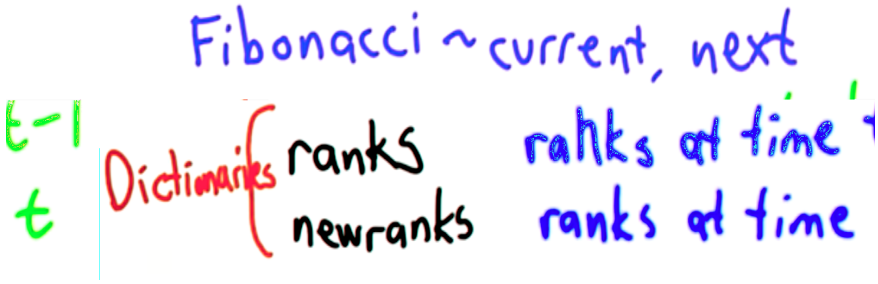
Formal Calculations

So, let's remember how we defined our ranking function. So, we said we're going to have a time step. We're going to keep going through steps to get more accurate rankings. But initially, we're going to give every rank 1 divided by the number of pages. Before I called that capital N. I'm going to use npages as a more Pythonic variable name for that. And we updated the rank at time step t by adding the probability that the random surfer starts over and randomly picks that page. So, that's 1 minus d, is the probability of starting over, divided by npages, how many pages there are. And remember than d is our damping constant, and we'll decide that d is 0.8. That's the damping constant. We add to that

Copyright © 2014 Udacity, Inc. All Rights Reserved.

the sum that we get of all the ranks, all the pages that link to this URL. We add up d times the previous iterations. So, it's going to be d times rank on step t minus the number of out links from p. So, it's starting to look more like Python code. We want to simplify it a little bit before we get to Python code. And the first thing we're going to do to simplify it, is observe that well, we have this t parameter. But we only ever use the very previous one. So, we don't really need to keep track of the rank value for all of the different t values. We just need to keep track of the previous one. When we did Fibonacci, we kept track of current and next in variables. And that allowed us to keep track of the previous and the previous previous Fibonacci numbers. What we want to do with ranks is something similar. So, we're going to use the variable ranks to keep track of the current ranks. And as we compute the new ranks, we're going to use the variable newranks. So, we'll use ranks. This corresponds to the ranks at time t minus 1. The ranks of the previous iteration, and we'll have a variable newranks, which is the ranks at time t. And the reason we need both of those. As we update the rank from each page. So we're going to go through the pages. Recomputing its rank. We can't lose the previous one. We still need to get rank at the previous step for that page. If we use the one that we got this time, that would distort the results. It would mean that the order that we update the ranks for changes the results. So, we don't want the results to depend on the order that we go through the pages. That's why we need to keep track of both the previous values, which will be used for ranks, t minus 1 and the newranks. Once we have done that, we don't need the t parameter anymore. What we are going to do is change this rank, the rank at time t is going to be the value of newrank and the value of rank at time t minus 1. That's the value of ranks. The other change I slipped into this. Before we had parentheses here and I've changed them to square brackets. And square, square brackets should give you the idea that well, this might be a data structure that we're indexing. And that's correct. We want to index the data structure. And we want to index it by the URL that we're looking for. And the Python dictionary provides a great way to do that. At the end of this will be a Python dictionary. And that's exactly what we want. So, I think we're ready to start looking at the code. We've written this formula. It's still a mix of math and Python. But we're going to turn this into the code for computing the page ranks.





Copyright © 2014 Udacity, Inc. All Rights Reserved.

Computer Ranks

So, I'm going to provide a start on this code and now we'll leave the crucial part of it for you to do as a quiz. You should feel free at any point to stop and try to figure out more of the code on your own. There are lots of different ways to do this and we'll show you one. And you'll finish that as a quiz. So, the first thing we're going to do is define two constants. So, we're going to use d as the damping factor. And I'll use 0.8 as my damping factor. That's the probability, thinking about our random web surfer, that she selects the link on the current page, rather than starting over with a new random page. The other constant I'm going to define here we'll call numloops. That's the number of times we're going to go through the relaxation. What we're computing is the value of rank at some time step. The number of times we go through that is going to determine the accuracy of our ranks. We'll use ten as the number of loops. You can experiment with changing that, and one of the questions in the homework assignment will ask you to think about what happens when you change that. So, now we need to start, we said initially the rank of each url is 1 divided by the number of pages and so the dictionary ranks, we want to initialize with those values.

So, we are going to create an empty dictionary. We'll call it ranks. The number of pages, the number of pages, we can get from the graph. The graph is a dictionary of nodes and len of graph will tell us the number of entries in that dictionary. So, that's the number of nodes in the graph which is the number of pages that we've crawled. And now we want to go through the pages initializing each page with the value one divided by npage, and I'm remembering to use 1.0 here to make sure the division is done as floating point division and we get an accurate number rather than integer division. So, now, we've initialized the ranks. We have a dictionary that maps each page to its current rank, which is one divided by the number of pages. So, now, we get to the interesting part. We need a loop that's going to go through the number of times of numloops. Each time through this loop, what we want to do is update the new ranks based on this formula, using the old ranks and then, at the end of the loop, we are going to make the variable ranks hold what was previous new ranks and that way, we can keep going, each time we are going to get a new value from newranks. At the end of doing all updates, we are going to update ranks to refer to whatever newranks is. So, that means each time through this loop we are going to create a new dictionary called newranks that starts as empty, and we're going to add all the pages to newranks as we update their rank. So, to do that we need to go through the pages in the graph and for each page what we want to do is compute the new rank for that page. And the first thing we'll do is this part. The new rank is 1 minus d divided by npages plus this summation.

So, the first thing we will do is introduce a new variable, we will call it newrank and we will assign it to this value. Then we are going to update it as we go through the pages that link into this page. So, we'll start by initializing newrank as 1 minus d divided by the number of pages. So, then what we need to do is update for this summation. And I'm going to leave this blank. And I'm going to skip that for now. This is going to be left as a quiz. We'll finish the rest of the code. And then your quiz

Copyright © 2014 Udacity, Inc. All Rights Reserved.

will be to finish this part of the code, which is really the most interesting part of computing the page ranks. Once we've done that, so we've used newrank as a variable to keep track of the rank for this page. Well, we want to update our dictionary, so we're going to add an entry, newranks. We're still within the for loop, you're going to put your code that sums up all the links here. Once we've done that, we've got the value of newrank that reflects both the probability of starting from that page, and the popularity from all the inlinks. And so, we'll update this to be newrank. We've added that to our dictionary, so once we've finished looping through all the pages in the graph, well now, we're ready for the next step. So, that means, we want to make the var, variable ranks refer to the newranks, so we've changed the time step to the next time step, and now we're going to go back through this loop, and we go through this loop, number of loops times, each time we're updating the ranks, and when we're done, what we want to return is ranks. That's the dictionary that maps each page to its rank.

Programming Quiz: Finishing Urank

So, I've left the most interesting and challenging part of this for you to finish as a quiz. And that's how to compute the new rank based on the values of the previous iteration, which are now stored in the variable ranks. We want to up, compute the new ranks based on all the pages that link to this one. And if you think about the graph structure and how to go through the nodes in the graph, I think you can figure out how to do this.

Answer:

So here's what we want to do, we want to go through all the pages in the graph. So, we'll call them nodes this time, we can't use page again, because we already used page up here. So we're going to go through each page in the graph, giving it the name node, and for each node, now what we need to do, is check whether that page links to this one. So we're going to look in graph index node, that's going to get us the list of all the pages. That node links to. If page is in that list, well that means that node links to page. So that means that we should add to newrank, based on the rank of this node. So that's what we're going to do. We're going to add to newrank, and the new value is going to be the old value. We summing into the value of newrank. We're going to multiply by d, that's our damping factor. And, the value that we're going to use here, is the page rank of the node. Remember, it's the node that links to this node that we care about, so we're getting the rank on the page. What we want to do, is divide that by the number of out links from that page, and that's important, that means that each, a page with many links, the value of each link is less. We need to divide that by the length. Of that list, which is what we get by looking at graph index node. So, that's all we need. We are going to update the newrank. We are going through all the nodes in the graph. We are finding all the nodes that link to this one, and we are updating our rank using the formula to get that, and at the end, we're returning the ranks.

Copyright © 2014 Udacity, Inc. All Rights Reserved.

So let's try that in the Python Interpreter. We have the code that we just wrote for compute_ranks. We're going to use crawl_web passing in the example site. Assigning the outputs to the variable index, which is the index and graph. And then we're passing in graph to compute_ranks, storing the result in the variable ranks. And we can print out the result in ranks, to see the page ranks. So, here's what we get. So we get a dictionary, for each URL that we crawled. We have an entry where we have the URL, followed by each page rank as the value. So let's see if the ranks that we get make sense. So, going back to our example site, we have the page Kathleen that has two links going into it. We have the page Nickel. That has three li, links going into it. If we just did simple link counting, well then, Nickel should be more popular than Kathleen. If we look at the page ranks, well the page rank for Nickel is 0.97. The page rank for Kathleen, is 0.11. So the page rank is actually higher, for the page that only has two incoming links. And the reason for that is, well, even though it only has two incoming links, the links that are coming into the page, are from popular pages. It's coming in from the index page, and it's coming in from the very popular Nickel page. So that's why the page rank of Kathleen is actually higher than the page rank of Nickel.

Search Engine


So congratulations--you've now built a search engine. You've got all of the components that you need for a search engine. You've got a way to collect a corpus, using a Web crawler that you built in the first 3 Units. In Units 4 and 5, you figured out--and hopefully understood--how to build the index and then how to make it faster in Unit 5. And in Unit 6--what we've just finished-- you figured out how to rank the results. There's one little bit left: we haven't used those ranks, so one of the questions on your homework will be using the ranks to get the best result. If we want to get the best result, just having the dictionary ranks is not enough--we need to use that to find the result that matches the query that has the best rank. And that's one of the questions that you'll do on the homework for this Unit. There are a lot of other hard problems. So you have a few problems left to solve before you can build a search engine that will compete with Google. Probably the hardest one: you've got to come up with a name. This is really hard. You could try "Yoogle", You could try "DuckDuck Find". None of those really work--you'll have to come up with a better name, and this is a pretty tough problem. Make sure to also talk to your Trademark lawyers. Another thing you'd like to be able to do is actually get your search engine on the Web so other people can send queries to it. That's not something we're going to cover in this class, but if you take the Web Applications course that will start shortly after this class finishes, you'll learn a lot about how to build Web applications and be able to do that. So congratulations--you've reached the end of Unit 6. And this is actually the last technical content that will be in this course. Unit 7 will get you ready for the final exam and will give you some interesting examples of using Computing in Context.

Copyright © 2014 Udacity, Inc. All Rights Reserved.

DE: So congratulations. You've now finished building a search engine and it actually does Page Ranking better than any search engine that existed before 1998.

ST: That's quite amazing. I first want to also congratulate David for getting you to this point. I think that's quite amazing and, honestly, he's put a lot of sleepless nights into this. But also, i want to tell you: you guys are my heroes. You got to a point where you reached something, I think, really significant in this class--you actually learned how to-- And you programmed your own search engine. Now, I have to say--I'm a little bit jealous. When I was a student at college, there was no search engines, and there wasn't even and ability for me to learn all this stuff. As for you, you've been afforded the ability to learn all this stuff. But then you really got an amazing speed here. That's something really, really significant. I hope--I mean, if you keep up that speed and keep improving yourself with that same speed and learn new things-- maybe in 3 or 4 years you are the next Google. That would be really amazing.

DE: Well, I think you're definitely on track to doing that, and I hope you'll take some of the courses that we'll start offering, as 200 available courses and more advanced courses later. This is the end of Unit 6--we're not going to have any more technical content to the course. Unit 7 is going to be all field trips and interviews, giving you some idea of how the things that you've learned in class fit into Context--and that should be a lot of fun. The other thing that you'll have, as part of Unit 7, is the final exam. And I hope everyone enjoys Homework 6 and will be ready for the final next week.

ST: And I hope you enjoy the final exam.

Copyright © 2014 Udacity, Inc. All Rights Reserved.