Intro to Computer Science

Lesson 04 Notes

Responding to Queries

Introduction

Welcome to unit 4! Unit 4 will be a little different from previous units. We are gonna start in right away by finishing the code for the search engine. What we have built so far is a web crawler. It can follow content on web pages, following links to find other pages, but it doesn't actually respond to queries. What we are going to build in this unit is a search engine, where we can pass in a keyword and get a list of the pages that contain that keyword. We are also going to learn how the Internet works and the World Wide Web. The main new computer science idea in this unit is the idea of how to use and build complex data structures. In order to respond to queries quickly, we want to be able to have a data structure where we can look up a keyword and quickly find the pages that match that keyword, without needing to scan all pages every time. To do this we are going to build a structure called an inverted index, or often just - an index, that allows us to have a mapping between keywords and pages that contain these keywords. Our index is really the same idea as an index in a book. I happen to have a book handy. That is one of my favorites. I might be a little biased, since I wrote it. But let's take a look and see if there is anything in the book that would help us understand index searching. We could try to read the whole book, see if there is anything about index search in here, or we could go to the index, and if we are lucky, the index will have what we have, it's got "index search". As each keyword in the index it has the page number, so we would go to the page number where that is found, and instead of needing to read the whole book to try to find what we are looking for, if we can find the right page, we will find the section on index search right away. So that's the goal of an index. For our search engine what we want is an index that provides a mapping from keywords to list of web pages where that keyword appears.

Programming Quiz: Data Structures

So now we're ready to think about how we should represent the index for our web content corpus. And we're going to have a quiz to see if you can think of a good way to represent our index. So I'm going to give you several options. And try to decide which one you think would be the best way to

represent our index. And I should note that several of these could work. We could build a search index using different data structures, and if we tried hard enough, we could make many of these possibilities work. But I want you to think hard about what's going to be the best one to use. So the first option is we could have a single list where we have a keyword followed by the URLs where that keyword appears followed by another keyword followed by the URLs where that keyword appears. The second option would be we would have a list containing as its elements lists. And each element would be a list of the keyword followed by the URLs where that keyword appears. The third option, we could have a list where each element of the list is a list where the first element is the URL, and the second element is a list of all the keywords that are found on that page. The fourth choice is a list where each element is a list and each element list is a keyword followed by a list of all the URLs that contain that keyword. So see if you can decide which one of these four representations will be the best one to use to represent the index for our web content corpus.

Answer:

So the answer that I think is best is choice four. Choice two is pretty good. But I think choice four is a little bit better. Choice one and choice three would be really difficult. So let's look at how these look to see why choice four is the best choice. So I need to shrink the choices to have a little more room for drawing. So here's what option one would look like. We have a single list and the elements of the list are keywords. Each keyword is a string and the keyword is followed by the URLs where that keyword appears. So for option one, we have a list, and it's a single list containing strings. Each string is either a keyword or a URL. And we have the keyword followed by the URLs where that keyword appears. This seems nice and simple. There's only one list. There are two real big problems with it. The first problem is it's hard to tell the keywords apart from the URLs. Maybe in this case we would say "well, anything that starts with HTTP is a URL" -- that's a well-formed web link-- and anything that doesn't is a keyword. But we could have keywords that start with HTTP as well. and then we couldn't tell which was which. The other really big problem with this representation is it's very hard to loop over the keywords. To find the next keyword, if we start from keyword one, we don't know which position to look at. There are different numbers of links that each keyword might have. And we'd have to search through every element to try to decide whether the next thing is a URL or the next keyword.

So this is not going to work very well. So option one is definitely a bad idea. So next let's look at option three, which was the other one that I said really would be very difficult. So with option three we have a list where each element of the list is itself a list. And the element lists are lists where the first element in the list is a URL. The second element in the list is a list of keywords. So that would be a list of the keywords that appear at that URL. So this has more structure than the first choice. We can tell apart where the URLs are-- all the URLs are the first elements of these lists-- and where the keywords are, which are all contained in the second element of the contained list. The problem with this approach is it's not going to make it easy to look up the pages where a keyword appears. This almost like having to scan all the pages over again. To look for a particular keyword, we've got to look at each

entry, look in the second part of that entry, scan it to see if that keyword appears. If it does, well then we want this URL in our result. If it doesn't, then that URL is not in the result. But to find particular keywords, we've got to look through each entry and look through all of the keywords in that entry. So this option is not going to work very well. Our goal in making the index was to be able to make lookups fast so we don't have to look through all the content of the pages, and this representation doesn't solve that problem. So we have two choices left, and I said both of these choices could work okay. But I think the fourth option is the best one. So let's look at why. So this is option B. We have a list where each element of the list is a list. And the element lists are themselves lists which contain the keyword followed by URLs where that keyword appears. This has a big advantage over the first two options. It means it's easier to tell the keywords from the URLs. The keyword is always the first element of the list. And unlike the previous option, it's also easy to go through the keywords.

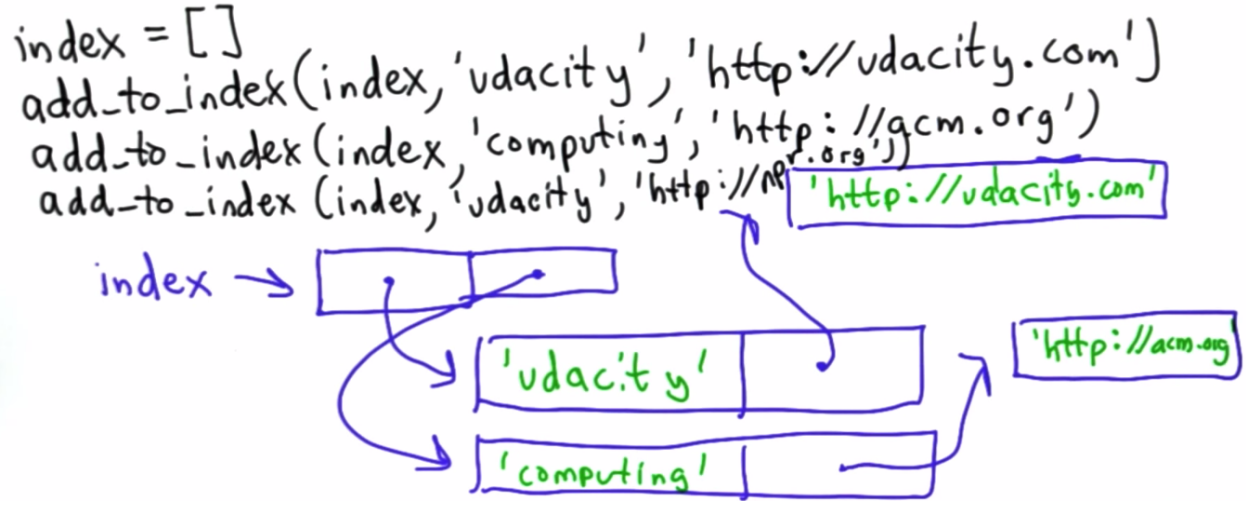
For each list contained, we just need to look at the first element to see if that's the keyword we're looking for. If not, we go onto the next one. We don't need to look through all the content in all the web pages. We just need to look at the first element in these lists to find the keyword we're looking for. So that's okay. The reason that I think option D is better, is that it makes a better separation between the keywords and the URLs. So the difference between this and option D is in option D instead of the inner lists being flat lists that are the keyword followed by the URLs, the inner list has just two elements. It has the keyword followed by a list of URLs. So that will be another list, and now we have a list of URLs. So this is option D. Each element of the main list is a list. And it's a list that just contains two elements. It contains the keyword--here the keyword "udacity"-- followed by a list containing all the URLs where that keyword appears. The reason I like this better than the second option is that it really makes a clear separation between the keyword and the list of URLs. It means if we decide we want to keep track of something else --say we want to keep track of the number of times someone searches for each keyword-- we could easily do that. We could have an extra element here that keeps track of the number of times someone searches for something. With option B, it wouldn't be that clear how to do that. Maybe we could add an extra value in here--we could add a number in here. We would need to change a lot of code that we already wrote. Instead of looking for URLs starting from position one, we'd have to start from position two now. Maybe we should put the number at the end to avoid that problem. Well then we have the problem of we've got to find where the number is. There are easy ways to find the last element of a list. But it's going to make things much more complicated. And if we start adding more and more things, we're going to have a hard time keeping track of everything. Option D gives us a lot more structure.

By keeping all the URLs in a list, we can treat that as a list and do things with it much more easily than if it's combined with a keyword. And we can also do things like add extra elements if we want to keep track of the number of times someone searches for that keyword or any other information we think of later that would be useful to add to our structure. So that's why I think option D is the best option. And deciding on data structures is one of the most important things we do when we build software. If you pick the right data structure, oftentimes the rest of the code is fairly easy to write. If you start

with the wrong data structure-- say we started with the structure that was choice A or choice C, well then it's going to be very very difficult, sometimes it's going to be impossible, to write the code in a way that performs well. So thinking about data structures is one of the most important things we do in computer science. We'll see that some more in this course. We'll also have following courses that really focus on this question of how to design and use data structures well.

Programming Quiz: Add To Index

Now that we've decided on our data structure, hopefully it will be easy to write the code for building our index. I think you actually know enough to be able to write the procedure that adds new entry to the index. Your goal for this quiz is to-- Define a procedure--we'll call it add to index-- and it takes 3 inputs. The 1st input is the index that we want to add the new keyword and URL to. Remember that we decided in the previous question-- that the way to represent index is as a list where each element of the list is itself a list and each of the element list is a keyword followed by a list of the URLs where that keyword appears. The 2nd input, is a keyword--that's just a string-- that's the word that we want to add to the index-- and the 3rd input is the URL-- which is a string that encodes the URL where that keyword appears. So, what add to index should do, depends on whether or not the keyword that's passed in is already in the index. If the keyword is already in the index, we don't want to create a new entry in the index for the keyword. We only want to have that keyword appear once-- what we want to do instead is add the URL to the list of URLs already associated with that keyword. So, if the keyword is not in the index, what we need to do is add a new entry to the index, and that new entry will be a list with the keyword, that's the new keyword, and the list the list of URLs where that keyword appears. This is the first one, since it wasn't already in an index, so, it will be a list containing just one URL.



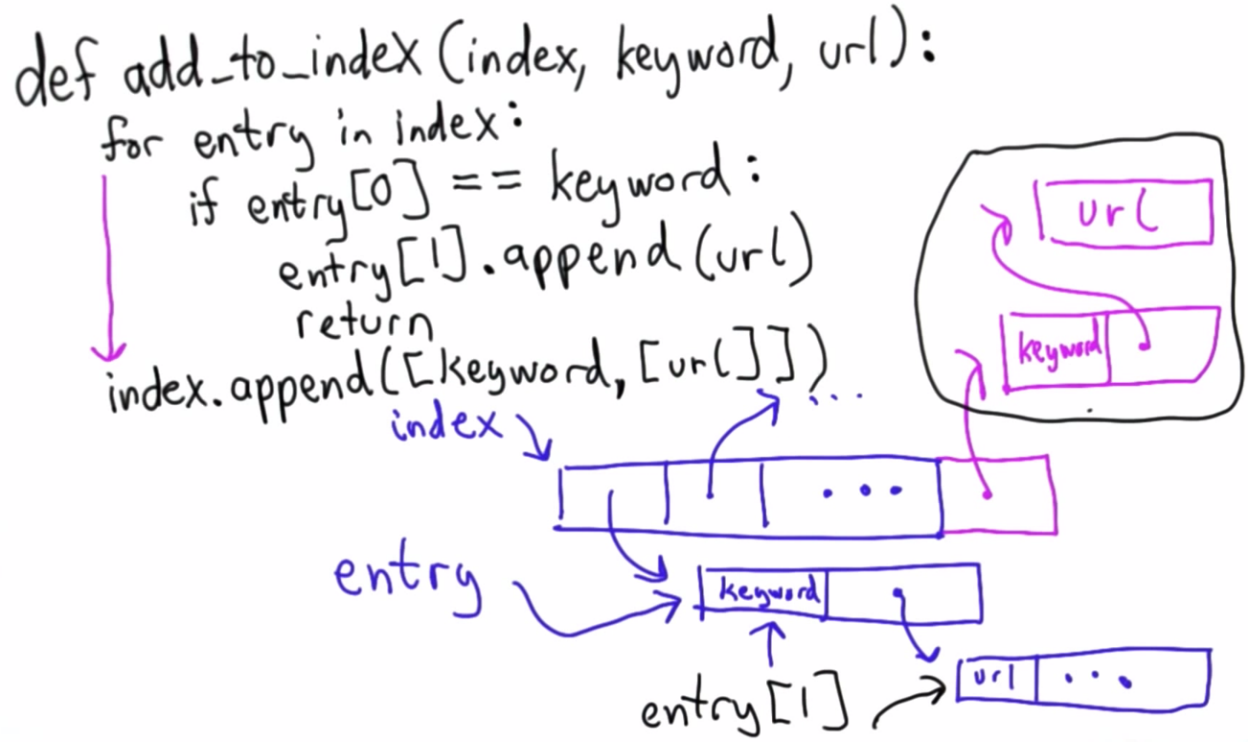
That's the list that we want to add to the index to represent that we found this keyword at this one URL so far. Let's work through an example-- Suppose we start by initializing an index to the empty list-- we have no entries in our index yet-- Then we'll call add to index, adding a new keyword, passing an index as the 1st input, for the 2nd input we need a keyword let's use the keyword udacity-- and for the 3rd input we need the URL where that keyword appears. So, here's what we should have-- then after

we call that index, we're going to add an element to the index list--- we've mutated index--now it contains element-- that element is a list containing the string udacity as the keyword-- the 2nd element of that list is itself a list containing one element-- which is the string "http://udacity.com" That's the new value of index. If we call out to index again, this time we're passing an index again-- it's the value that index refers to which is the structure-- now the keyword is computing, and we're passing in as the URL "http://acm.org" That's going to mutate index again. We're going to follow the rule here, the keyword is not in the index, so we should add a new entry. That means we are going to append a new entry to index. That new entry is going to have the keyword computing. As the second element we have a list, and this list will contain the URL we passed in, which is acm.org. I am going to show you one more example. In this case, we're going to pass in a keyword that is already in the index. We're calling out to index again--passing in the keyword udacity-- which is already in the index here-- and passing in a new URL, which is not in the index. What should happen this time-- we're going to mutate index, but instead of creating a new element-- we're going to mutate the element we already have-- we're going to look for the entry that matches udacity-- it's already in the index, so don't want to add a new one. What we're doing is following the 1st rule here-- that says if the first word is already in the index-- what we want to do is add new URL to the list of URLs associated with that keyword-- but we don't want to add any new entries to the index itself. Here's what that will look like-- we're adding to the list of URLs a new entry-- that's the new value of the URL. I hope it's clear now what add to index should do. I think you know enough to be able to define it yourself. That's the goal for this quiz, for you to define the procedure-- add to index that has this behavior- it takes the 3 inputs and adds a new keyword to the index.

Answer:

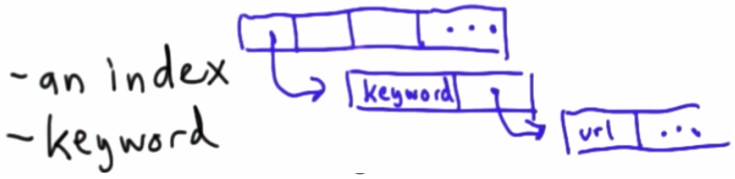
There are different ways that we can define add to index, here's one way that works. We're going to create a procedure, we'll call it add to index-- and it takes our 3 inputs--the index, the keyword, and the URL-- What we need to do in add to index is first find if the keyword already appears. To do that, we need to look through all of the entries in index. The natural way to do that is to use a 4-loop. We're going to define the procedure add to index-- and it takes our 3 inputs, index, keyword and URL-- to help keep track of what we are doing, I am going to draw a reminder of what the data structure of index is. Remember that it is a list of entries, and each entry is itself a list, where the 1st part is a keyword, and the 2nd part is a list of URLs. That's our data structure, that's going to help us figure out what to do to define add to index. The first thing we need to do is to check whether the keyword already exists in the index. If we can find it, well, then we want to modify that entry, rather than creating a new one. The natural way to do that is to use a 4-loop. We are going to loop through the elements of index. We will give each one the name entry to use in the block. This is what entry will be. The first time through the loop, the value of entry will be a reference to the list here-- which is the first element of the index list.

Now we need to find the keyword. The keyword is right here--that's the element at position zero of entry. We're going to test the value at position zero on entry identical to the keyword that's passed in we'll use the double-equal comparison to test that-- If it is equal, then we found a match-- this means we want to append the URL to the list of URLs associated with that entry. To get that list of URLs we want to find entry 1--that's the value at position 1 of entry-- and we want to append that the new URL. Here we've found an entry that matches the keyword we were looking for-- this means that the keyword is already in the index-- we've added the new URL to the URLs associated with that keyword-- so we're done, we have nothing else to do. What we want to make sure, is that we don't continue and and do anything else. One approach would be to use break---that would end the loop-- what we want to do instead is really end the whole procedure. If we did break, well, then we'd still have the problem of how do we deal with the case where the keyword wasn't found? Here we're just going to return-- we're all done with add to index, we've added the URL it belongs.

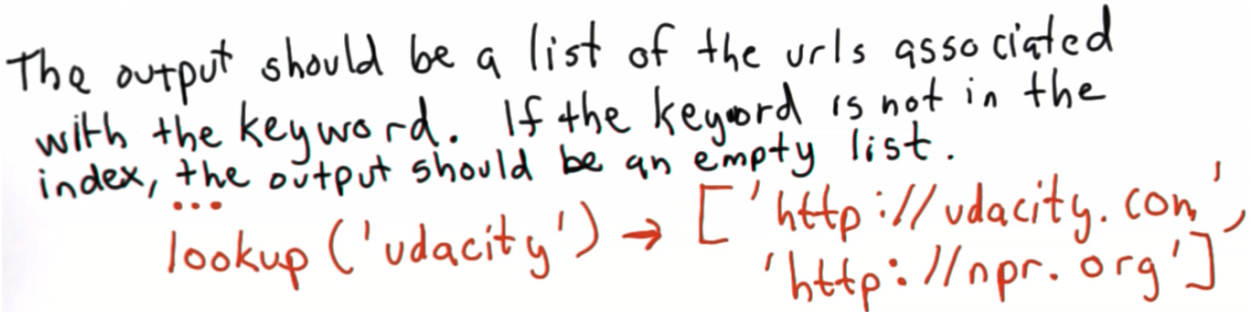


Now we need to think about what to do in the case where the keyword does not already exist in the index. If that's the case, then we get to the end of the loop without ever finding that entry. If we've got to the end of the loop, that means we did not find any entry in the index that matches the keyword, then what we want to do is add a new entry--- and that new entry is going to have, as its value, a list containing 2 elements, it will be the keyword-- and as the 2nd element we'll have a list containing the URLs that we found that have that keyword. So far we only have 1--the URL that was passed in to add to index. How do we do that? To add a new element to add to index we use append. We need something to pass in to append--that is the structure we want to add. This whole thing is what we want to add-- so, that's a list containing the value keyword as its 1st element. As the 2nd element, its a list containing just the single URL. That's what we want to append. in the case where we didn't already find the keyword in the index.

Programming Quiz: Lookup

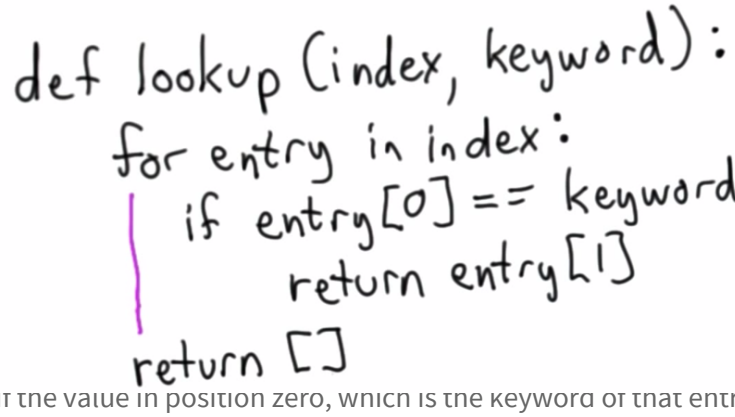


So now that we can build an index, we'll want to use it. We'll want to use it to respond to quer works, I think you'll be able to write the query c inputs. The first is an index, and that's the data each element of the list is a list containing a keyword and a list as its second element, and that list is a list of URLs where that keyword appears. The second input is the keyword to lookup. That's what we're searching for.



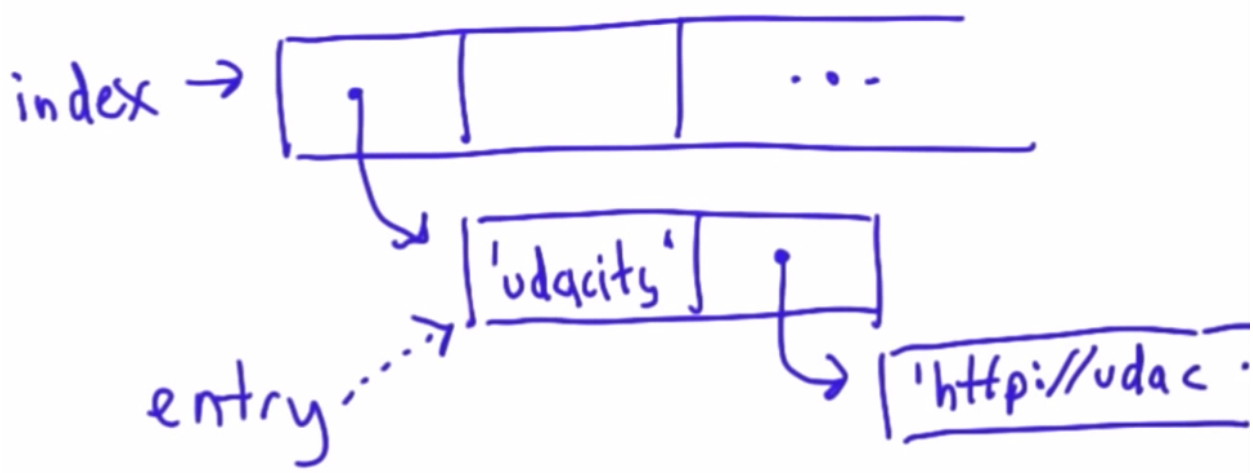
So the output of lookup should be a list of all the URLs associated with the input keyword. So as an example, we constructed the index here where we've added 2 URLs with the keyword udacity. The result from doing a lookup on the keyword udacity should be a list containing those 2 URLs. We should be concerned what happens if the keyword does not exists. If the keyword doesn't exists, well there are no URLs associated with that keyword. So we would just return an empty list. So see if you can define the lookup procedure.

Lookup Solution



So here's a way we could define the lookup procedure. It takes 2 inputs, th structure of the index here. It's a list, e Where the first part of the list is a keyw fine lookup is very similar to what we c i, ™s write that again. That's going to be a for loop, and it's going to be exact through the elements in the index eacl those elements. We're going to check if the value in position zero, which is the keyword of that entry, matches the keyword we're looking for. When we we're defining add to index, what we did when we found a match wa with that keyword. What we want to do for lookup is different.

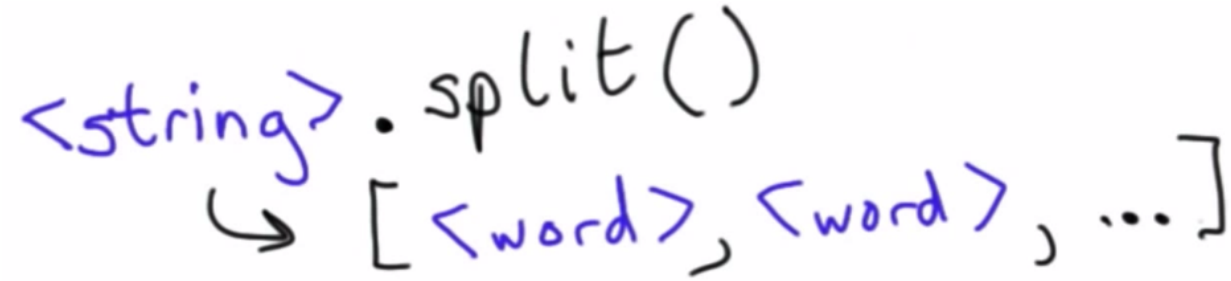
the dex to try to find 1 of



All we need to do for lookup now is return the list of URLs. And we can find that by looking in position 1 of entry. If we get to the end of the list without finding that keyword. Well then we still need to return something and the way we describe lookup, what it should return is the empty list. That means there are no URLs that we've found so far that contain that keyword.

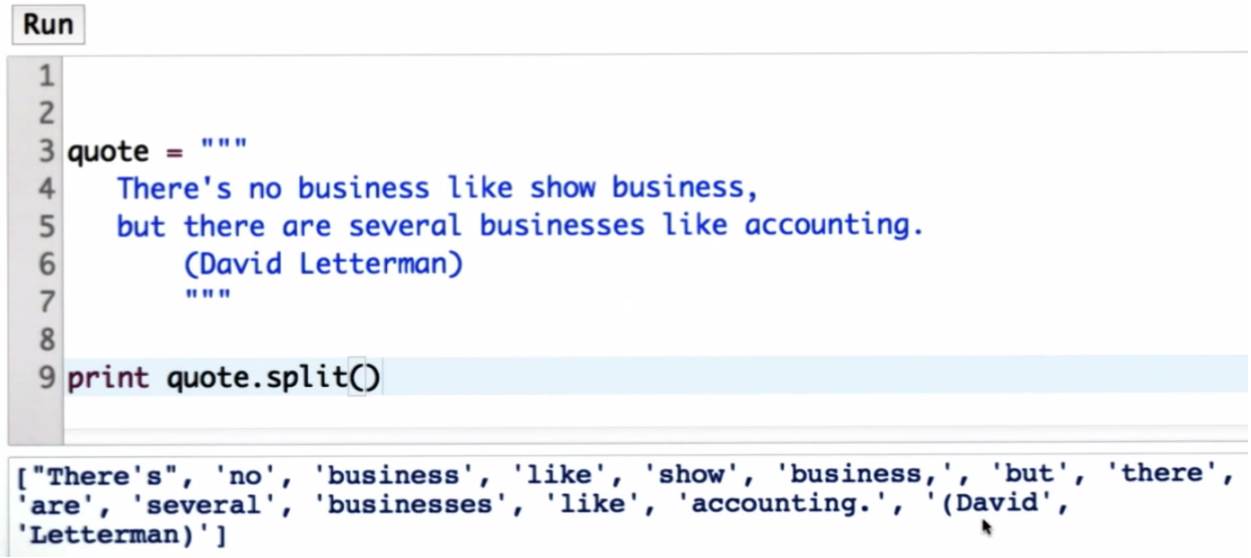
Building The Web Index

So now we're ready to build the Web index. We're going to use the split operation to do this. So we can certainly build a way to separate all the words in a Web page ourselves using the operation we've already seen. We can use indexing to go through the letters in the string We can identify characters that we think of as separating words such as a space or a comma. And we can collect those all as separate strings.



The good news is Python provides a built-in operation that does exactly what we need. It's called split. We invoke split on a string and what it outputs is a list of the words in the string. For now we're not going to worry too much about the details of how split decides how to separate a string into words, but let's look at a few examples to see that it does something pretty close to what we want for building our Web index. So we've defined quote as a string which is this quote from Robert Reich. "In Washington it's dog eat dog. In academia, it's exactly the opposite." And now we'll use split to divide the quote into its words. And you can see the result. We have a list where the elements of the list are the words from the quote. And split does a pretty good job of dividing the string into the words that we want for our Web index. It's not perfect. We can see that Washington, which was followed by a

comma in the quote, ends up as the string Washington including the comma. That's a different value than the keyword Washington. So if we search for Washington and we built our index this way, we're not going to actually find this occurrence. So this isn't perfect, but it's going to be good enough for what we use for now. Later on we can think about ways to divide the Web page into its component words in a way that will be more accurate. Figuring out exactly how to decide when something's a new word is a fairly tough problem, though. As a second example to get a feel for how split works, let's try another quote. Here we have a quote from David Letterman. "There's no business like show business, but there are several businesses like accounting." I've introduced a new Python constrict. We're using the triple quote here.



The nice thing about triple quotes is that we can divide them over multiple lines. When we tried to enter the previous quote using just the double quotes, it fell off the edge of the screen. Using triple quotes, we can define one string spread over multiple lines. And now when we print out the result of splitting the quote, we see it's divided into words. It's fine that there are new lines. It still separating it into the words we have. We still have some issues, like the parentheses is included as part of David. So this quote wouldn't show up for the keyword David. I'll leave it for you to decide at the end of the class whether computing is a business more like show business or business more like accounting. I'm going with show business, though. So now let's see if you can write the code to build an index using the URLs collected from our Web problem.

Programming Quiz: Add Page To Index

For this quiz, your goal is to define the procedure add page to index that takes 3 inputs. The 1st is the index, and I've drawn a picture of index to remind you of its structure-- and it's a list where each entry in the list is a list containing a key, and the list of the URLs where that keyword is found. The 2nd input is the URL-- so that's the location of the page, where the content came from-- And the 3rd input is the content. That's the entire text of the page at that location where URL is. The results should be updating the index to include all word occurrences found in the page content by adding the URL

to the words associated URL list. I'll show you a few examples so it is clear what add to index should do, and then you'll have a chance to define it yourself. We're going to start with an empty index-- and we'll call add page to index-- and let's print out what index looks like after this-- it's a little hard to see like that, so let's see what index position zero is.

We can see that the keyword this appears at the URL "fake test" if we print index at position 1 we'll see the same thing, but this time for the keyword is. We have an entry, we have an index, we have 4 words in our index this is a and test and for each word, the list of URLs where each word appears is fake.test which is what we passed in as the URL here. Let's try adding some other things to our index. This time, we'll add a page called real.test. and have the content of the page be this is not a test. Now, when we run this, we see this index if we look at a particular element--let's look at the element at position 2-- it has 2 entries in its URL list-- since both of the pages we passed in contain is-- and if you look at index position 4-- we see that for the word not--that only appears on the real.test page-- so there's only 1 entry in it's URL list. We already defined the procedure for respond to new query. So, that should work on this index. Let's try. Look up takes the index and the keyword and returns the list of URLs at that keyword. Delete the previous tests, and see what happens when we look up is, we see that it occurs on both pages-- if we look up udacity--it doesn't occur anywhere so far. I hope it's clear now what add to index page should do-- see if you can write the code to define add page to index.

Answer:

Here's one way to define add page to index. We're defining a procedure that takes in 3 inputs-- the index, the URL and the content at that location-- We use the split method to divide the content into its component words. We store that into the words variable. Then what we want to do is go through all of the words, adding each word into the index. We can do that using a 4-loop. We're going through all of the words, we'll use word as our variable. For each word, we call add to index, pass in to index the word and the URL. Note that if a word occurs more than once on the same page, we're going to keep adding it to the index each time. There might be more than one occurrence of the same URL in the list of URLs associated with a word.

Depending on what we want our search engine to do, and how we are going to decide on how to respond to queries, this might be a good thing or a bad thing. We'll talk more about that in a later class. Let's try this in the Python interpreter. Let's recap the code we already wrote-- we have the add to index procedure-- that takes in the index, the keyword and the URL. It goes through the entries in the index, and it appends the URLs to the entries that matches the keyword. If it's not found, it adds a new entry to the index, that's the list of the keyword, and the single URL. We defined the lookup procedure that takes the index and the keyword, finds the entry in the index that corresponds to that keyword, and returns the URLs associated with that keyword. So, now we're going to define the add page to index procedure.

So, here it is, we're splitting the content into it's component words, we're looping through the words, and we're adding each word to the index-- note that we don't need to return anything-- the point of this procedure is to modify the index. Let's try add page to index with a few simple examples. Here we have the test we tried before-- we see that we get the expected result-- if we're adding the first quote to the index, we see that there is the 4 words in the quote each has the list of pages associated with it--the list of URLs contains fake.test. Then we added the 2nd page, the 2nd print we see, now for most of the words we have both fake.test and not.test except for not, which only has not.test. So, to be really convinced that this works, let's try a few more interesting examples. Now we're going to add page to index that might be on the dilbert.com page-- this quote from Scott Adams about time management-- "Another strategy is to ignore the fact that you're slowly killing yourself by not sleeping and exercising enough that frees up several hours a day, the only down side is that you get fat and die." Hopefully that is not happening to anyone in this class. I know that it's a lot of work, but you need to manage your time so that you're still getting enough sleep and exercise. The second quote we'll add to the page index and we'll give the URL randy.pousch for that. And it's a quote that says "Good judgement comes from experience, experience comes from bad judgement. If things aren't going well, it probably means you are learning a lot and things will go better later." So, hopefully things are going well in this class, but if they're not, that means you're learning a lot and hopefully things will go better later.

Let's see what the index looks like after adding these 2 pages. You see that it is pretty big, there are a lot of words in those 2 quotes. We can see that some of them only appear at dilbert.com and from the quote from Scott Adams on time management-- others appear on both pages-- We'll see more if we try a query. Let's try a query. We'll look up in our index to see where the word you appears. And when we run that, we see it appears in both documents. It was there in the dilbert quote twice, it was listed 2 times at the URL dilbert.com and once and the Randy Pousch URL. Just to confirm, the 2 occurrences in this quote, there is one here--and there's one here. And there's one in the second quote right here. If we look up another word, we see good did not occur in any of the quotes, but bad appeared in the 2nd one. So, it looks like all the code that we have seems to be working. We can look up words that are indexed and get the list of URLs where they were found. We can add pages to our index. And we can record all of the words in that page at the location they occur. We're really close to having a search engine. The one thing that we have left to do is to connect the code we already wrote for crawling the web to the code that we have for indexing documents and looking up the keywords. It's been awhile since we looked at the code for crawling the web. Let's look at that again and see if we can think how to connect the two.

Programming Quiz: Finishing The Web Crawler

So let's remember the code we had at the end of unit 2 for crawling the web. So we used 2 variables. We initialized "tocrawl" to the seed, a list containing just the seed, and we're going to use "tocrawl" to keep track of the pages to crawl. We initialized "crawled" to the empty list, and we're keeping track

of the pages we found using "crawled." Then we had a loop that would continue as long as there were pages left to crawl. We'd pop the last page off the "tocrawl" list. If it's not already crawled, then we'll union into "tocrawl" all the links that we can find on that page, and then we'll add that page to the list of pages we've already crawled. So now we want to figure out how to change this so instead of just finding all the URLs, we're building up our index. We're looking at the actual content of the pages, and we're adding it to our index. So the first change to make, we're updating the index, and we're going to change the return result. So instead of returning "crawled" what we want to return at the end is the index. `If we wanted to keep track of all the URLs crawled, we could still return "crawled" and return both "crawled" and "index," but let's keep things simple and just return "index." That's what we really want for being able to respond to search queries.

So now we have one other change to make, and this is the important one. We need to find a way to update the index to reflect all the words that are found on the page that we've just crawled. I'm going to make one change before we do that. Since both "get<u>all</u>links" and what we need to do to add the words to the index depend on the page, let's introduce a new variable and store the content of the page in that variable. This will save us from having to call "get_page" twice. "Get_page" is fairly expensive. It requires a web request to get the content of the page. It makes a lot more sense to store that in a new variable, and that will simplify this code. So now we just need to pass in content. So we have one missing statement, and I'll leave it to you to see if you can figure out statement we need there to finish the web crawler. When it's done, the result of "crawl-web," what we return as "index" should be an index of all the content we find starting from the seed.

Answer:

So the answer is we should use the "add<u>page</u>to_index" procedure we just defined, and we should pass in the index. We should pass in the page, that's the URL that identifies the location, and we should pass in the content. And that's all we need. So we're done with our web crawler. From a seed, we can find a set of pages. Following that seed, following all the links that we find on the pages that we find starting from that seed, for each page, we're going to add the content that we find on that page to an index, and we're going to return that index. And we've already written a code that given the index, can do a lookup. So for any word we want to look up, we'll find the list of URLs for the pages that contain that word.

Startup

DE: Congratulations, you've now got a working search engine. You can crawl the Web, finding content from pages, building an index, and then respond to search queries with the pages that contain that content.

P: This is really exciting. I think I'm ready to start my search company. Cool. And so I was looking at the competitors out there, and the key is that you've got to have a catchy name. So Google, Duck Duck Go. I've got to come up with something.

DE: This is the really hard part.

P: Yeah. I'm ready to announce -- I'm ready to share with you my idea. I've been putting a lot of thought into this. How about Search With Peter dot Info?

DE: Oooh.

P: Yeah. So there's a lot of work to do. I've got to buy servers. I've got to get employees. Investors. I've got to -- Oh, man -- I should probably start now.

DE: Okay. Well, good luck to Peter. There are some more things that we want to learn in this class. And there's some things we need to do to really improve the search engine to make it useful. In Unit 5 we'll work on making the search engine faster. And in Unit 6 we'll work on making it find the best page for a given query, not just all the pages. So for the rest of this unit we're going to look at understanding more how the Internet works and what happens when we request a page on the World Wide Web.

The Internet

So now that we've got our basic search engine working, for the rest of the unit, we're going to learn more about how we actually get web pages on the internet. So far we've been using the magic get_page function. And we provided this function that takes as its input a URL, which is a resource locator on the web, and it
produces as output the
contents of that page. I can
show you the Python code
that we use for get_page. It

won't really help very much in terms of understanding what's going on, on the internet. But I don't want you to think there's anything hidden here. So the main thing that we're doing in get_page is we're returning the result of calling URL open, passing in the URL. So that opens the web page that was requested. And then reading that page. And that returns the output of that page as a string. This is provided by the Python library urllib, so we needed to import that library. And so that's what's going on. The rest of this is an exception handler. So we call this a try block. That we are going to try these things. They might not always work. And this is where we use try. There might be an error. Maybe the URL is bad. Maybe we don't actually get a page back. Maybe the internet request times out, doesn't always work even if its in the URL. So that's why you put this inside a try block. And if something fails, execution will jump to the except block.

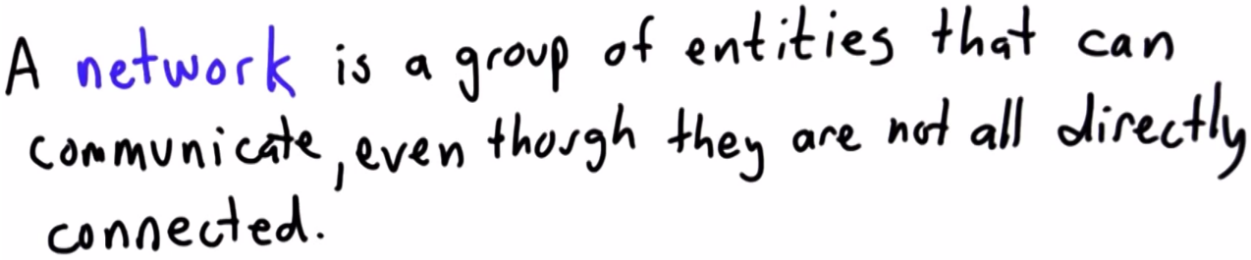
Run

```
1
2
3 def get_page(url):
4     try:
5         import urllib
6         return urllib.urlopen(url).read()
7     except:
8         return ""
9
```

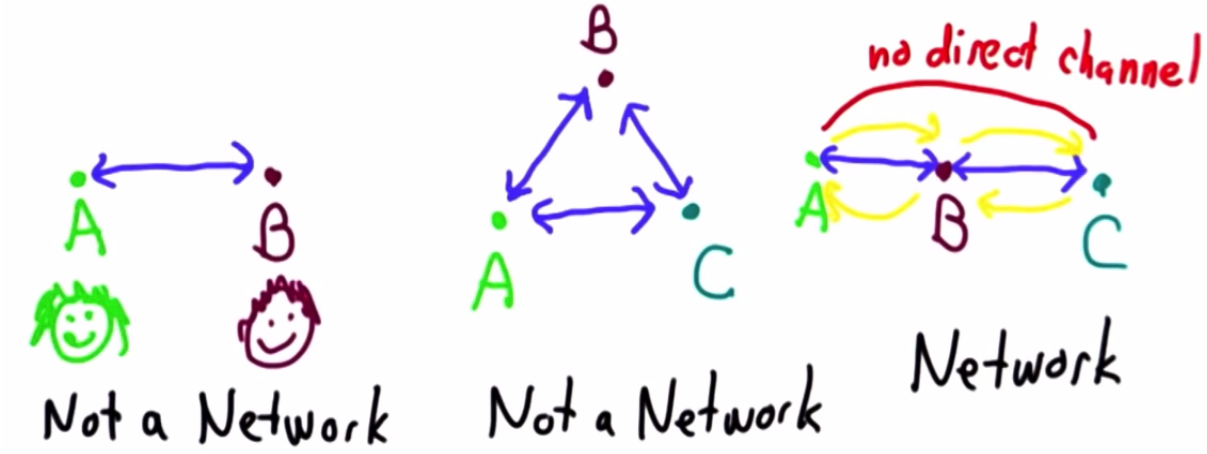
This is called the exception handler, and in this case we just return an empty string. So that means if we request a URL that can't be loaded instead of getting an error it will just return an empty string, and for the use in our web crawler this is a good thing. If we didn't do this then the WebCrawler would have to deal with a case where the page doesn't exist. In our case we just get an empty string, so there's no content on that page. So looking at the actual code for get_page doesn't really help us understand much about what's going on when we request a webpage. All the actual work is hidden in this Python library function. So in order to understand more about what's going on, we need to understand more about how the internet works.

Quiz: Networks

First I'm going to talk about networks in general. The internet is just a special kind of network. There are lots of different ways to define a network. We are going to use quite a precise definition. So we are going to define a network as a group of entities. And entities could be people, they could be computers, they could be organizations, they could be governments, that can communicate, even though they're not all directly connected.



So here's an example to show what that definition means. So we have Alice. And we have Bob. And Alice and Bob know each other, and can talk to each other. The way they talk, we can call the channel. That could be just talking in person. That could be talking over a telephone. We're going to say that just this is not enough to be a network. This is just two people having a conversation. If we had the second scenario, we'll say we have Alice, Bob and Charlie. And Alice can communicate with Bob. Bob can communicate with Charlie. And Charlie can communicate with Alice. But they can do all this directly. They're all directly connected. If Alice wants to talk to Charlie, well, she has a direct link to Charlie. She doesn't need to use Bob. There's no way to communicate other than across these direct links. So even in this case where there are three people who can all communicate we're still going to say that's not a network. Many people would call both of these networks. We're going to use a somewhat more precise definition where neither of these will be considered a network. What will be considered a network is this third example and in the third example we have three nodes. We have Alice, Bob, and Charlie let's say. They're the three people but now Alice is not directly connected to Charlie. But there's a way that Alice can send a message to Charlie.



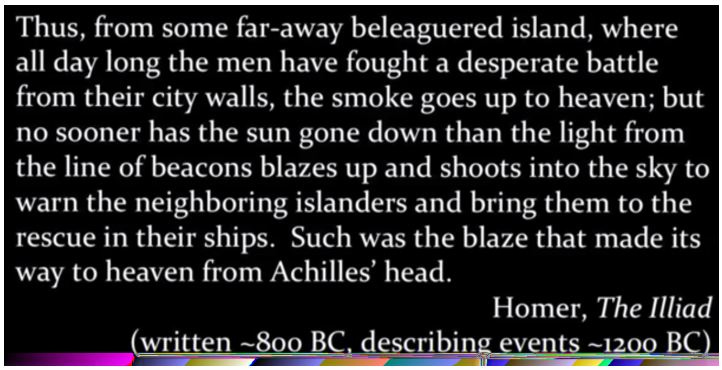
And she can send the message, because Bob will forward the message for her. Once the message goes through two hops, to make it so two people are not directly connected, can communicate. And maybe Charlie can also send messages back to Alice. Then we call it a network. So that's going to be our definition. There has to be at least three entities and there has to be a way that entities that are not

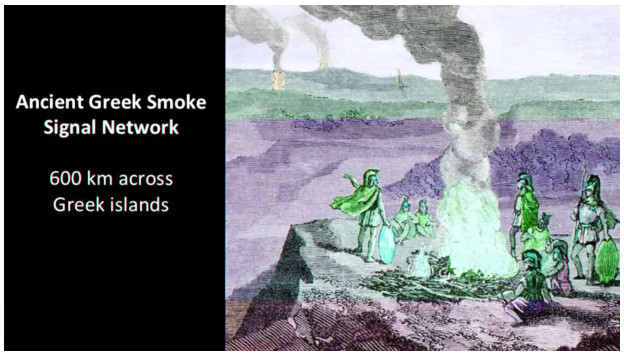
directly connected can still communicate. So to see that you understand that definition of a network and maybe also know a little bit of world history we're going to have a quiz. So here are the possible answers. Ten years old, 30 years old, 100 years old, 1000 years old, or over 3000 years old.

[ ] 10 Years Old
[ ] 30 Years Old
[ ] 100 Years Old
[ ] 1000 Years Old
[x] Over 3000 Years Old

Networks Solution

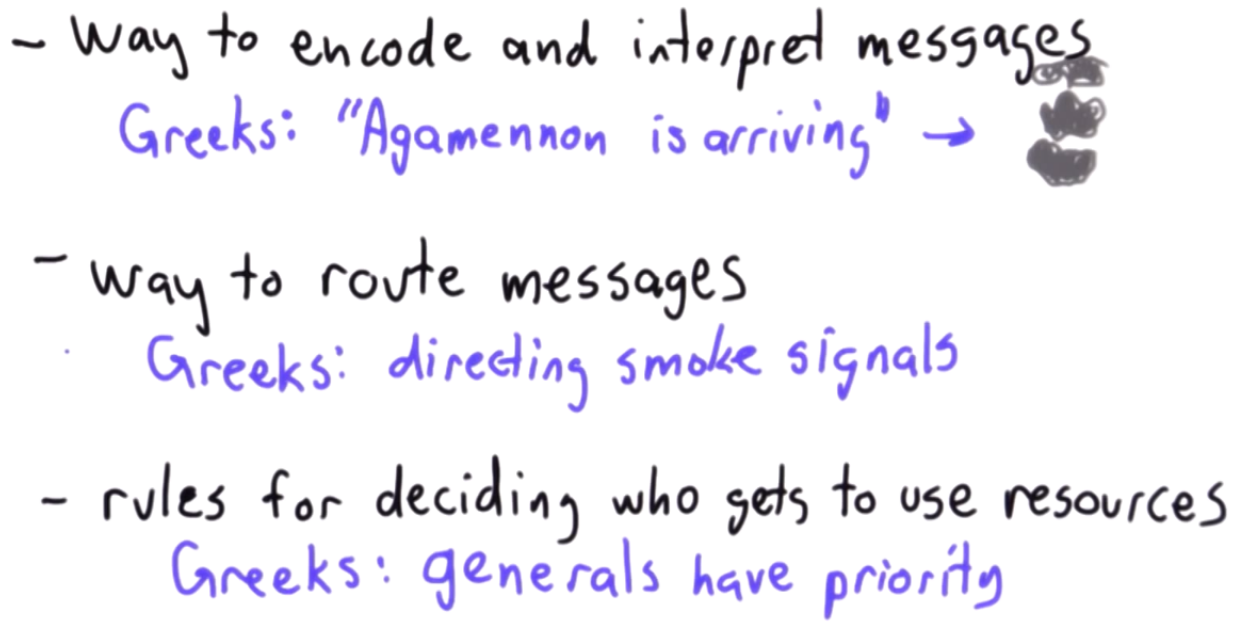
So the answer is over 3,000 years old. Humans have been using networks like this since at least as long as we can record. They might have just been networks where people talk to each other, but even networks where they were using technology to communicate, in this way, go back at least 3,000 years. So to give you an idea of how sophisticated networks could be over 3,000 years ago, here's a quote from Homer's The Iliad, where he's talking about the network the ancient Greeks had. And they had a line of beacons spread over the Greek islands. They could use that to send messages, to warn the neighboring islands when there was an enemy attacking, and that would tell them to bring in their ships, that trouble's about to occur. And Homer was writing this around 800 BC. But what he was describing are wars that happened over 3,000 years ago. And Homer wasn't just making this up. The Greeks actually had such a network. They had a network where soldiers could light a fire on one island and soldiers on another island would see that fire. Light their own fire that would send a signal to the next island. The next island would see that and light its own fire. And they could send messages over great distances this way. They could send a message over 600 kilometers across the Greek islands. So what do you need to make a network like this?





Smoke Signals

So now I really went to art school. I thank Amy for the Greece drawing. So what does it take to make a network like this work? So let's suppose we wanted to send a message from Rhodes to Sparta, and we have many points along the way where
we can send smoke signals. So we'd start by sending a smoke signal from Rhodes. This would be visible from a few other points. And to reach Sparta, the one at Noxos would need to resend it, would need to also submit it as smoke signal that would be visible at some other points. The one at Melos would need to see it and resend it. And maybe that's already visible at Sparta. So we need several different things in order to make a network like this work. The first thing we need is a way to encode messages. We need to be able to convert the message that we want to send, say it's a message that says the Trojans are arriving. We needed to be able to send that message and encode that in a smoke signal. We need a way to do routing. We need to figure out where the message should go. If we want to send it to Sparta, we need to know that it should be forwarded by these nodes. The node that sees it from Miletus does not need to resend it. Probably the way the ancient Greek network worked, all the nodes resent every message. They didn't have a way to do routing selectively. But that's very wasteful. What we'd like is a way to only send the message to the destination it goes to. For the kinds of messages the ancient Greeks were sending that says things like the enemy's arriving, well, it was okay. You wanted to send it everywhere, that's called flooding the net. But if we want to send messages just between two points, we've got to know which nodes should resend the message or which direction to send it in. A
is, we need to decide who gets to use the network. So it might be that there are more than one enemy arriving at once or maybe, the enemy is arriving, it might be at the same time. Achilles, say Achilles is in Mellitus, and he wants to send a message back to Medea in Delphi.



So maybe Achilles wants to send a message that would go like this at the same time as the army needing to send this important message that's saying the Trojans are arriving. So we need to figure out who gets to use the network and what they get to send. So, these are all things the ancient Greek network would have had to deal with if it was sophisticated enough to worry about them. If it was just sending the message that says the troops are arriving, well, that's one smoke signal and it didn't necessarily need that. All of these are the same things that we need to worry about if we want to send messages on the internet. So these are the things that we need to make any network. We need some way to take the messages that we want to send and encode them in a way that they can be transmitted, and interpret them when they get to that other end. For the Greeks, the message was something like, Agamemnon is arriving, get the ships to a safe place. And they needed to encode that in a smoke signal. We need a way to route messages. We need a way for the receiver of this smoke signal to know the next place to send it. And for the Ancient Greek network, since smoke signals are not directional, they just go up in the air, there wasn't much to do here. Everyone could see the message and they wanted to send the message everywhere. If we want to send messages just between two points, this becomes a much tougher problem. We need rules for deciding who gets to use resources. If there are two messages that want to go through the same place, well, only one of those can be sent at a time. We need some way to decide when. For the Greeks, well, I don't know what their rules were, but let's assume their rule was, if you're a general, your message has priority. We needed all these same things for the Internet. Before we talk about how they are done on the Internet, I am going to introduce the two main ways that we measure networks.

Latency

I want to talk about the two main ways to measure a network. Those are called Latency and Bandwidth. What they mean is very different. They're often confused. Often people say bandwidth when they mean latency, or say latency when they mean bandwidth. They're quite different things. So let me explain what they are. So latency is the time that it takes for a message to get from the source to the destination. And that's for the start of the message. So, we can measure latency by timing when you start sending to the time that the receiver starts receiving. So this is the unit of time. It will be measured in something like seconds. For a fast network today, it's more often measured in milliseconds and there are 1000 milliseconds in one second. So now, you understand about latency. And when I go back to the Greek signaling network.

So, suppose now that Zeus, who is all powerful, wanted to send a message from Rhodes to Sparta. And he thought that the latency of the smoke signal network as it was currently set up was too high, that it takes too long for the message that he sends starting from Rhodes to reach Sparta. So the question is, how could Zeus, and remember that in ancient Greece, Zeus was all powerful, reduce the latency between Rhodes and Sparta. So here are the choices. It makes the signalling nodes further apart. So instead of going from Rhodes to Naxos to Millios to Sparta, maybe it would have to go one hop. There would be a new island in the middle here, and it could go from Rhodes to the new island

and then to Sparta. He could threaten the soldiers at all the signalling points and scare them into working harder, so they start the fires more quickly, when they need to send a message. He could find a way to make it so instead of just sending one color smoke, you could send different colors smoke. That would mean that with the same amount of smoke, you could send more different messages. Or he could increase the speed of light. That would mean that the the soldiers at each signaling point would see the previous smoke signal more quickly than they do now.

[x] Make the signalling nodes further apart, so it takes fewer hops
[x] Threaten the soldiers to make the signalling fires quicker
[ ] Add colors to the smoke so there are more messages per signal
[x] Increase the speed of light

Answer:

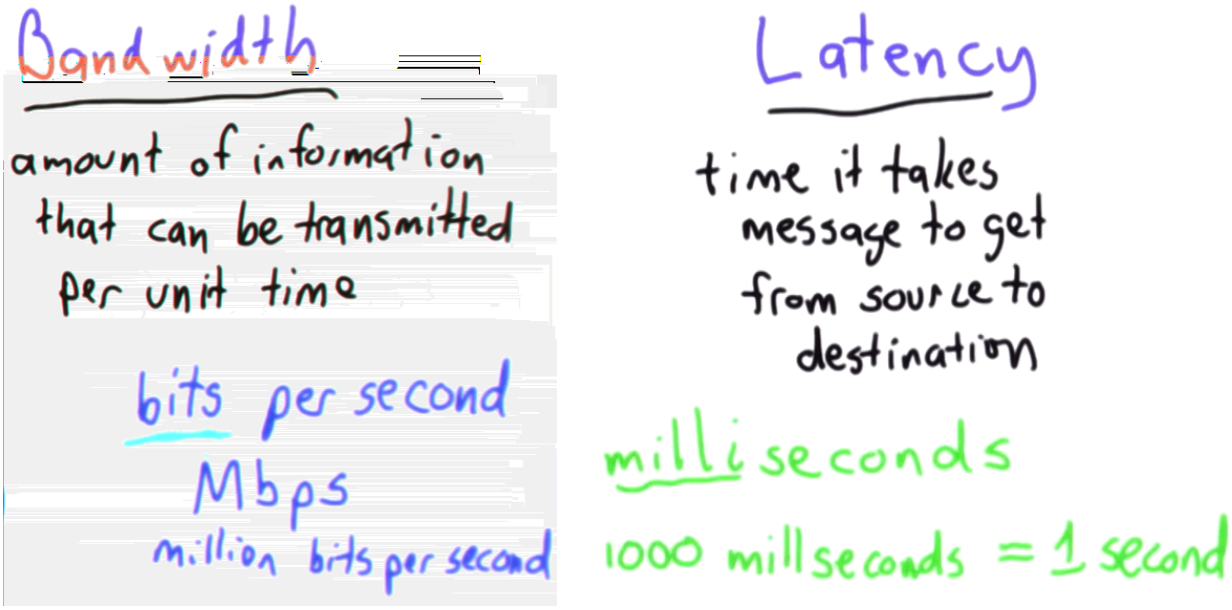
So three of these would reduce latency. The first one is correct. That would reduce latency because every time you go through a hop, well there's some time delay there. That you need the time for the troops at the hop to see the message that came in and then to start their fire to send the same message on to the next hop. So if you have fewer hops its going to take less time for the message to get across. You could reduce the time it takes to go through each hop. So, if you can make the soldiers at each node work faster, it would take less time from the time that they see the message to the time that they send it on, which is the time when the next person will see it. So, the total time to get across the network would also be reduced.

The third choice does not reduce latency. So adding colors would allow you to send more messages, but it doesn't reduce the time it takes for a message to get across. And we'll see next that what adding colors would do would actually increase the bandwidth of the network. But it doesn't improve the latency at all.

And finally, because Zeus can do things that most people can't, other than possibly physicists in Switzerland. If you could increase the speed of light, well that would make the network faster. Not a very realistic option and it wouldn't make it much faster because of the transit time even with the regular speed of light, the time it takes for the light to travel between the points is infinitesimally small compared to the time for all the other things like starting fires. But all three of these would in theory reduce the latency. The adding colors would not reduce the latency, but it would increase the amount of information that Zeus could send between those points in the same amount of time. And that's what we call bandwidth.

Bandwidth

So adding colors wouldn't increase the latency, but it would increase the bandwidth. What bandwidth means, is the amount of information that can be transmitted per unit time. And it doesn't matter if there's a start-up time. Right, the start-up time is the latency. That's how long it takes to start sending a message across. That's how long it takes for the start of the message to be received. The bandwidth is, once you've got some part of the message across, what's the rate that you can send information? So, this is going to be measured in terms of units of information divided by units of time. So bandwidth could be measured in terms of bits per second. On the Internet this is often measured In Mbps, which is megabits per second or million bits per second. I haven't yet explained what a bit is, and in order to understand information, it's important to understand what a bit is.



Quiz: Bits

So, what's a bit? And a bit is the smallest unit of information. So suppose I tell you that there are two boxes. There's a green box and a blue box, and in one of those two boxes, there's a gold star. And it's equally likely to be in either the green box or the blue box. One bit is the answer to one yes or no question. So if you ask one yes or no question, you ask, is the gold star in the green box, then you get one bit back as the response. You get an answer that's either yes or no. And when you learn which one it is, that's one bit of information that allows you to go from having two choices down to one choice and then you know, well, if the answer was yes, the gold star's in the green box, that's the one you should open. So when we think about bits in computing, we don't usually call them yes or no. We call them 0 and 1. 0 more readily maps to no, and 1 to yes.

So I switched blue and green here. But we could pick any two things. We could use green and blue. We could use yes or no. If we have two things And we can choose one of them. Knowing which one to choose is one bit of information. So that's what one bit can do. It allows us to decide between two things. For the ancient Greeks network, maybe that's all it could transmit. There was either the no

bit, which said there weren't any enemies arriving. Or if there was a smoke signal, that was one bit saying, yes there is an enemy arriving. So it's possible if it was only able to transmit that can do is send bits, we can send either a 0 or a 1, or a yes or a no. But let's think of it as 0 and 1s now. Can we send anything more interesting? Can we just send one thing? And it turns out that we can actually send everything that we want, just using bits.

So instead of just picking between two boxes, now let's suppose there are four boxes. So let's say, there's, a purple box, and there's a dark red box. So There are four boxes. There's still only one gold star. So can we figure out which box the gold star is in with just yes, no questions? Well one way to do that would be to ask for a question, right? We could say is it in the green box? We could ask is it in the purple box? We could ask is it in the blue box and as long as we there is at least one gold star. We don't really need to ask if it's in the dark red box, because if we got news for all three of these well then we know the answer to that next question, if we asked if it was in the red box, would be yes. So here we needed up to three questions, to figure out which box the gold star was in. Can we do better? So the question is, how many bits do we need to find the gold star? And remember each bit is the answer to one, yes or no question. And we have four possible boxes and the gold star could be any anyone of those four.

[ ] 1
[x] 2
[ ] 3

Answer:

So we saw that if we ask these three questions. Well we needed three questions to be sure that we will find the gold star, by opening box, but we don't need three questions. If we are little smarter about our questions, we only need 2. And the way to do it in 2 questions is to ask for questions that gives us more information. So the problem with asking is that is it in the green box question first? Is that three 3 4 of the time? The answer is no. And if we're trying to use yes, no questions. Or we're trying to use bits to get as much information as possible, what we want to do is ask questions where the answer is equally likely to be yes as it is to be no. We want a question where half the time the answer is yes and half the time the answer is no. If the question isn't like that, so if it's a question where it's no more often than it's yes, well then we could guess the answer is no and be right more than half the time. So we're not getting a full bit of information in the result, because we had a good chance of being right without getting that answer.

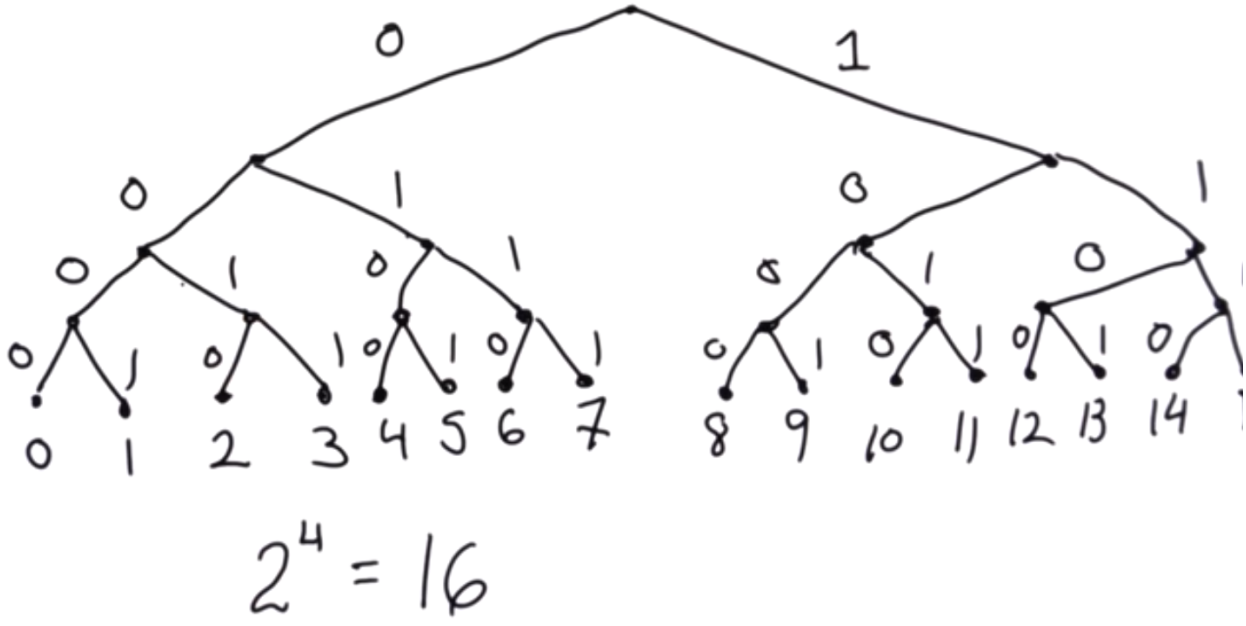
So we want to figure out questions where the answer is equally likely to be yes. As it is to be no. So we could ask that as our first question. So we're going to ask, is the gold star in either the green or the purple box? So the answer is yes if it's in one of those two boxes. The answer is no if it's in either the blue or the red box. So, there are four boxes. For two of the boxes the answer is going to be yes, so

half the time the answer is yes. So that means, we're gaining one full bit of information. If the answer is yes, we've learned that it's in either the green or the pop-, purple box. If the answer is no, we've learned that it's in either the blue or the red box. And so if the answer is yes, well now we need to ask one more question. To figure out which box it's in. And that could be, is it in the green box? And so, if we ask that question, if the answer is yes, well then we know it's in the green box. If the answer is no, then we know it's in the purple box. We only needed two questions to get to the right box. And that's the same, if the answer was no to this.

Well then we can ask is it in the blue box? That's a yes no question. If the answer is yes, well then we know it's in the blue box. If the answer is no, then we know that it's in the red box. So now, we can cover all four possible boxes. We didn't need three questions like we did when we asked, is it in the green box, is it in the purple box, is it in the blue box? We might have needed three questions to determine the box, here we only needed two.

Buckets Of Bits

So now you've seen that you can encode four things in bits. You can encode any number, we just need more bits. So if we have more bits, we can encode more different things. So, here I've got four bits. That means I have four yes, no choices. And I can encode sixteen different numbers. And the reason for that is because 2 to the 4 is 16. Every time I add one more bit, I double the number of things. Because I have one more yes, no question. So for just this part of the graph, I had eight things, I had three bits. I could distinguish between the numbers zero through seven. Adding one more bit, doubles the number of things I can distinguish. So that's a way to measure information. And once we can distinguish any number of things. Well, we can send any kind of data we want. Anything that's discreet, we can turn into a number.



So we could turn letters into numbers. If we have a long string, we could have a sequence of numbers, which we can turn into one really big number. So, once we can send bits, we can send anything we want. So that's what bandwidth is. It's a measure of the amount of information that we can send, and we're measuring the number of bits. Each bit is one yes or no decision. In computing, we usually think of that as either a 0 or 1. So that long stream of encoding a string of text like a dozen of web page. It could be encoding an image, it could be encoding a number. And to measure our bandwidth, we need to know how many bits can we send per second.

Programming Quiz: What is your Bandwidth

So there are lots of different ways you can try to measure the bandwidth that you're getting over your Internet connection. I'm going to show you one that's provided by CNET. This is not necessarily that accurate and it'll depend where you are. But you can go to the CNET site, Internet Speed Test with hyphens between Internet, speed, and test. Pick your location. I'll deal with work, and it will try sending some messages to figure out what your bandwidth is. And, you can see that I'm getting 44,000 kilobits per second, it says kbps here.

So that's if we get the same result. And we try it a second time, we get 31 megabits per second, or 31.7. We can try again and we get 62. So we can see that it is varying each time we try it. And bandwidth does vary. You're sharing connections on the network with other people, they may be doing different things. There are lots of reasons why the bandwidth that you measure varies, but if you try this test you'll see what you get. And we're going to have a quiz. This is more of a survey than a quiz. But, we're curious to see what bandwidth you're getting. So, try a bandwidth test and see what the result is, and the quiz will ask you to report that to us. So, this isn't really a quiz. It's more of a survey. But the question is, what is your bandwidth? And you can use one of the tools we provided links to, to figure out what your bandwidth is. And the boxes give you a choice of ranges. And if you can get greater than than 200 megabits per second were you live, I'm very jealous.

Answer:

So there's no correct answer to this quiz, of course. What I'm getting here, is usually between 40 and 60, sometimes it was between 60 and 100, never much above know some of you are, unfortunately, having much less bandwidth from that, and I hope you're still able to get the videos okay. If you have less than one kilobit per second, unfortunately, you're probably having a really hard time. And some of you probably live in countries that have much better bandwidth than the United States, and are fortunate enough to have over 200 megabits per second.

Traceroute

[Narrator] We can also learn a lot about the internet by measuring our latency to different destinations, and in fact we can also see the hops along the way. I showed you on the map of Greece the hops along the way. We can see the same thing for destinations on the internet, and we can do that using an application called traceroute. If you're using a Mac you can just create a shell and run traceroute directly. If you're using some other operating system you might need to do something a little different, and we'll have directions for that on the site, but you can do traceroute and after traceroute you pick where you want the destination to be. First we'll try tracing the route to Udacity.com, and what traceroute is doing is sending packets over the internet, looking at all the intermediate hops to figure out the route it takes to get a packet from where I am now to the Udacity.com site, and when we run that we see the route. You can see each hop, so from where I am now to reach Udacity.com took 15 steps. The total time was about 39 milliseconds. You can see there are several different times in doing multiple tests and the time might vary each test, but the time is about 39 milliseconds, and you can also see the steps. The first thing you see is that our site is actually being served on a server run by Google. That's what the Udacity.com site resolves to, and you can see all the hops along the way. The 192.168.1.1 that's a special internet address that means your local machine, you always start there, and then you can see all the other hops that we go through, and the time it took to get there. There are different hops that took different amounts of time. You can see the time does vary. We can see that the time to get to the Comcast site here in Santa Clara varied from 11 to 37 milliseconds and you can see all the other hops along the way. That each hop took a millisecond or 2 between that hop. To get 15 hops took us about 40 milliseconds. If we try to go somewhere further away, let's try tracing the route to MIT.edu.

Well that server is running from Boston. I'm here in California. To get to Udacity.com we didn't have to go very far geographically. To get to MIT we've got to go across the country, now we're going to Boston. We started where we are now, we went through Santa Clara, went through San Jose. Sometimes from the host names you can guess where they are, sometimes you can't, but here it's pretty clear we're going from San Jose, and then we're going across the country to New York, and you can see that the big time difference was when we went from San Jose to New York. There's no hops, so unlike the Greeks where the distance between hops was very limited, in the internet the distance between hops can be thousands of miles. There's some fiber optic cable probably between those 2 points, and there's no need for any routing directions.

Once you got to this point in San Jose you end up in New York without going through any other decision points, and then we get to Boston, and it takes about 100 milliseconds before we get to MIT, and you can see we're not actually getting the final destination. We're getting some stars; we're not getting a response from the web server at MIT. We could do that if we tried a few more traceroutes and set the time outs differently, but to get across the country it took about 100 milliseconds, and we can go somewhere further away. About the furthest away you can get from where I am now is Madagascar. I'm going to try tracing a route to a server in Madagascar.

If there are any students in Madagascar please contact me, and we try that traceroute, and we can see that it starts again going through San Jose, goes through Dallas, and now it's going through a lot of servers run by the same country, and that's getting across the ocean, getting towards Madagascar, and you can see it's starting to take quite a long time. It took 100 milliseconds to get to MIT. Here we have taking 195 milliseconds already, and we haven't yet reached Madagascar. The time it takes to do the traceroute is much longer than the time it takes to get a request from these pages because it's sending many requests trying to find all the points along the way, but this is a good way you can see what's going on in the internet and get a better understanding of how packets are getting from where you are to where you're making a request.

Programming Quiz: Travelling Data

So we observed that the latency between where I am now, which is Palo Alto, California, and Cambridge, Massachusetts is 100 milliseconds, and that's what we measured using traceroute. And the distance between those two locations is fraction of the speed of light which is between Palo Alto and Cambridge.

Answer:

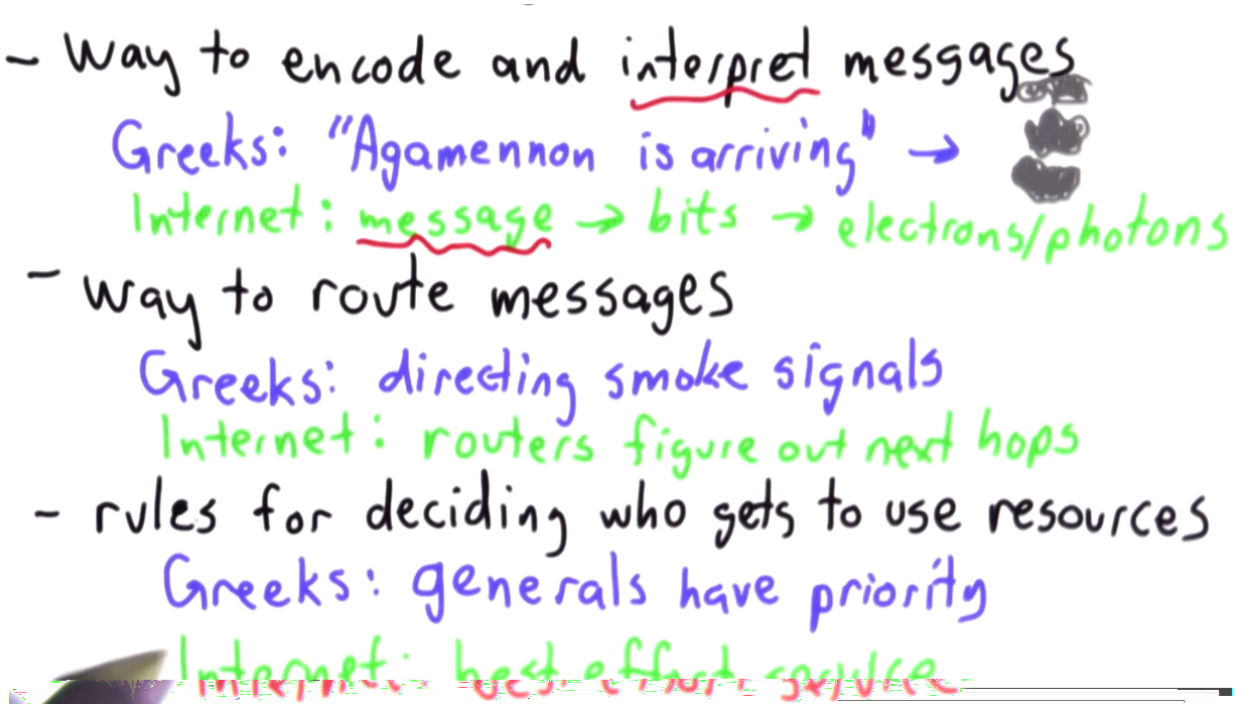
So the answer is 1 seventh, which is pretty good. There aren't many things that we deal with where we can think of them in terms of fractions of the speed of light and fractions above 10%. So our data was traveling between California and Massachusetts at a speed of about 43,000 kilometers per second. And most of that time is not the time on the wire. While the data's traveling on the wires, it's traveling pretty close to the speed of light. The speed of light through Optical FireWire is about most of the time is all the routers that it had to go through. And we saw that, from the traceroute, that it went through about 20 routers. So each of those routers had to take a packet in, figure out where to send it, and do all that in the time that it had to travel across the country, which was a 100 milliseconds total, so the average speed going across the country was one seventh the speed of light, I can show you the Python code to compute that.

So we have the distance which is 4300, that's the number of kilometers between Palo Alto and Cambridge. We have the speed of light, which is approximately 300,000 kilometers per second. And we have the time it took, which, since we're using seconds as our unit for the speed of light, we should put in seconds. And so it as 100 milliseconds, which is 0.1 seconds. And so I can compute the time it took light to travel that distance. By dividing the distance by the speed of light and let's see what that is. So if it was travelling at the speed of light it would take 0.014 seconds to get across the country, which is if we divide the time it actually took by the time it would take light to travel, we get 6.97. So, it's taking almost seven times as long as it would take if it was just light traveling in a vacuum for that distance. If you had a vacuum between Palo Alto and Cambridge and no routers and nothing else

along the way, well then you could get your packets across about seven times faster than the internet actually does. Not many people can afford setting up a dedicated vacuum between the two points they want to communicate.

Making A Network

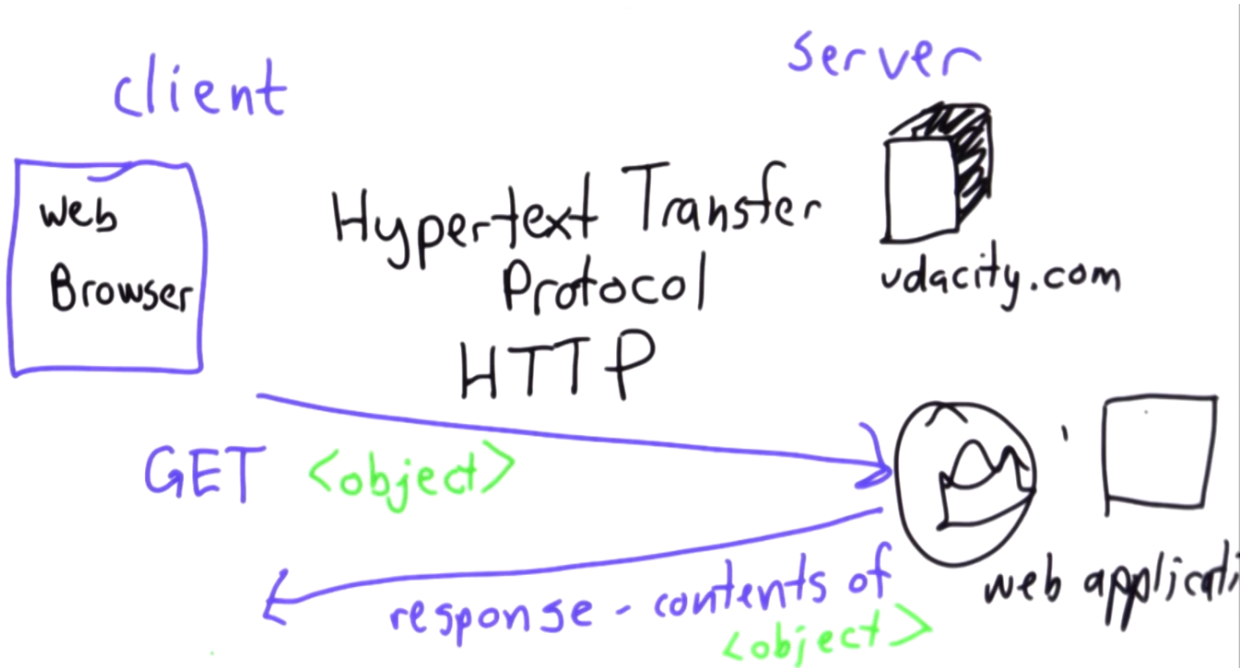
So now let's go back to the things that we need to operate a network. So we need a way to encode and interpret messages. So we saw that we can encode any message that we want in bits. And then we can encode the bits on the wire. How that encoding actually works is pretty complicated, it's not something that we're going to talk about in this class, but there are lots of different ways to do it. But we're figuring out ways for each zero and one to encode that as something that we send along a wire, or it could be wireless. What we haven't talked about, which we're, we are going to talk about soon, is what these high level messages are. Right? We need ways to send messages that can be interpreted and understood at the other side. And so, what we'll talk about next is how that works on the internet. As far as the way to route messages. For the internet, all of the routers along that path. Well, a message comes in. It has a desination. The routers have to figure out the next destination to go to. And that's also quite a challenging problem. It's something we're not going to get into more detail in this class. But you can imagine different ways you might do this. A router might have a table that says, well, if you're in California and you want to send a message to Boston, you should first send it to Nevada, because that's going in the right direction. That's not quite the way things work, because we saw that there was actually just one message. That really, what you want to do is send it to San Jose, we've got a big strong pipe that goes all the way to Boston. Finally you need ways to decide who gets resources.



And on our Greek network we assume that the general could decide. The internet is much more of a wild west in that in the internet there aren't any real rules for who gets resources. Everywhere along the network gets to decide on it's own how to do that. And what we really get on the Internet is just what we call best effort service. If your message needs to go over the same link as some other message, the router can only send one message at a time. It's up to the router to decide what to do. There are different policies that different routers follow. There's no general rule that is enforced on the whole internet. And this means that sometimes your packet might just get dropped. There's no guarantee, when you send a message on the internet, that it actually gets to where you want it to. So we're not going to talk more about these two in this class. I'll encourage you to take a future networking class that will get into those details. We are going to talk a little bit about how the messages work for the web.

Protocols

And what we need to make a network work is a Protocol. And what a protocol is, a set of rules that people agree to, that tell you how two entities can talk to each other. So, for the web, the protocol gives rules about how a client and a server talk to each other. The client is the web browser and the server, is the web server. So that might be udacity dot com. The web browser is what you're running at home. And what the protocol says is, if you want to get the server to do something, the client has to send a message in a particular way. The protocol that we use on the web, is called hypertext transfer proto, protocol. Which is abbreviated as HTTP. When you look in your browser, almost all the URLs that you use start with HTTP. That indicates that the protocol, that you should use to talk to server server, that you're requesting a document from, is this protocol, called hypertext transfer protocol. An it's a very simple protocol. There aren't, too many messages. There's actually only two main messages and there is only one, we'll talk about. That's the message called GET. The client can send a message to the server, where the message says get, and then the name of that object that you want to get. So that's all the client does. It sense the message like this, and if you remember the python code for get page. Well, lets call in some library function that actually does this. That's sending the get message to the server. The server will get that message. It will do some, run some code on it. It will find a file that was requested. It might run some more code to get the result.



Take the web application course to understand more about what the server does, but what matters to the client, and what matters to us in using this, is what happens after that. The server sends back a response, which is the contents of the requested object. So that's the whole protocol, that's what's going on whether you send a web request using your browser, by clicking on a link. Well then the browsers doing a lot of things to figure out what you requested and then its sending a get message to the right web server. That was the server specified by the URL, to know which server it is, and then its getting a response and its doing processing on that response to render it. If you want to understand more about what the web browser is doing, take the programming languages class that focuses on how to build a web browser.

Conclusion

So we've reach the end of unit four. I hope you feel like you understand at a high level what's going on in a web browser, what's going on when you request a page over the internet. There's certainly lots of details, we're not going to cover them all in this class. But I think you should have a pretty clear picture of everything that's going on. There's nothing that's magic here, everything can be understood. Everything's done by sending messages across the internet and getting responses back and the responses are just text that are processed either by your web browser or now by your very own web crawler that you've written, and now you've got a search engine that can respond to queries. So, congratulations for making it to the end of Unit Four. There'll be a homework to do now, and in Unit Five we're going to look at the problem of how to actually make our search engine scale. The search engine that we've built in unit four works, it gives us responses to queries, it doesn't do that in a very fast or smart way. That's what we're going to improve in unit five. We're going to make it so that we can respond to queries much faster than we can with the code that we've written so far. And then

in unit six we're going to look at the very interesting question of how do you find the best response for a given query. We don't usually want to get all the pages that contain the search word we're looking for. What we want to do is get the best page. And that's what we'll look at in unit six.