The following content is provided under a Creative Commons license. Your support will help MIT OpenCourseWare continue to offer high quality educational resources for free. To make a donation or view additional materials from hundreds of MIT courses, visit MIT OpenCourseWare at ocw.mit.edu.

PROFESSOR: Hey, everybody. You ready to learn some algorithms? Yeah!

Let's do it. I'm Eric Domain. You can call me Eric.

And the last class, we sort of jumped into things. We studied peak finding and looked at a bunch of algorithms for peak finding on your problem set. You've already seen a bunch more.

And in this class, we're going to do some more algorithms. Don't worry. That will be at the end. We're going to talk about another problem, document distance, which will be a running example for a bunch of topics that we cover in this class.

But before we go there, I wanted to take a step back and talk about, what actually is an algorithm? What is an algorithm allowed to do? And also deep philosophical questions like, what is time?

What is the running time of an algorithm? How do we measure it? And what are the rules the game?

For fun, I thought I would first mention where the word comes from, the word algorithm. It comes from this guy, a little hard to spell. Al-Khwarizmi, who is sort of the father of algebra.

He wrote this book called "The Compendious Book on Calculation by Completion and Balancing" back in the day. And it was in particular about how to solve linear and quadratic equations. So the beginning of algebra.

I don't think he invented those techniques. But he was sort of the textbook writer who wrote sort of how people solved them. And you can think of how to solve those

equations as early algorithms. First, you take this number. You multiply by this.

You add it or you reduce to squares, whatever. So that's where the word algebra comes from and also where the word algorithm comes from. There aren't very many words with these roots.

So there you go. Some fun history. What's an algorithm?

I'll start with sort of some informal definitions and then the point of this lecture. And the idea of a model of computation is to formally specify what an algorithm is. I don't want to get super technical and formal here, but I want to give you some grounding so when we write Python code, when we write pseudocode, we have some idea what things actually cost. This is a new lecture. We've never done this before in 006. But I think it's important.

So at a high level, you can think of an algorithm is just a-- I'm sure you've seen the definition before. It's a way to define computation or computational procedure for solving some problem. So whereas computer code, I mean, it could just be running in the background all the time doing whatever. An algorithm we think of as having some input and generating some output. Usually, it's to solve some problem.

You want to know is this number prime, whatever. Question?

AUDIENCE: Can you turn up the volume for your mic?

PROFESSOR: This microphone does not feed into the AV system. So I shall just talk louder, OK? And quiet the set, please.

OK, so that's an algorithm. You take some input. You run it through.

You compute some output. Of course, computer code can do this too. An algorithm is basically the mathematical analog of a computer program.

So if you want to reason about what computer programs do, you translate it into the world algorithms. And vice versa, you want to solve some problem-- first, you usually develop an algorithm using mathematics, using this class. And then you

convert it into computer code. And this class is about that transition from one to the other.

You can draw a picture of sort of analogs. So an algorithm is a mathematical analog of a computer program. A computer program is built on top of a programming language.

And it's written in a programming language. The mathematical analog of a programming language, what we write algorithms in, usually we write them in pseudocode, which is basically another fancy word for structured English, good English, whatever you want to say. Of course, you could use another natural language.

But the idea is, you need to express that algorithm in a way that people can understand and reason about formally. So that's the structured part. Pseudocode means lots of different things.

It's just sort of an abstract how you would write down formal specification without necessarily being able to actually run it on a computer. Though there's a particular pseudocode in your textbook which you probably could run on a computer. A lot of it, anyway.

But you don't have to use that version. It just makes sense to humans who do the mathematics. OK, and then ultimately, this program runs on a computer. You all have computers, probably in your pockets.

There's an analog of a computer in the mathematical world. And that is the model of computation. And that's sort of the focus of the first part of this lecture. Model of computation says what your computer is allowed to do, what it can do in constant time, basically? And that's what I want to talk about here.

So the model of computation specifies basically what operations you can do in an algorithm and how much they cost. This is the what is time. So for each operation, we're going to specify how much time it costs.

Then the algorithm does a bunch of operations. They're combined together with control flow, for loops, if statements, stuff like that which we're not going to worry about too much. But obviously, we'll use them a lot.

And what we count is how much do each of the operations cost. You add them up. That is the total cost of your algorithm. So in particular, we care mostly in this class about running time.

Each operation has a time cost. You add those up. That's running time of the algorithm.

OK, so let's-- I'm going to cover two models of computation which you can just think of as different ways of thinking. You've probably seen them in some sense as-- what you call them? Styles of programming.

Object oriented style of programming, more assembly style of programming. There's lots of different styles of programming languages which I'm not going to talk about here. But you've see analogs if you've seen those before.

And these models really give you a way of structuring your thinking about how you write an algorithm. So they are the random access machine and the pointer machine. So we'll start with random access machine, also known as the RAM. Can someone tell me what else RAM stands for?

AUDIENCE: Random Access Memory?

PROFESSOR: Random Access Memory. So this is both confusing but also convenience. Because RAM simultaneously stands for two things and they mean almost the same thing, but not quite. So I guess that's more confusing than useful. But there you go.

So we have random access memory. Oh, look at that. Fits perfectly. And so we're thinking, this is a real-- this is-- random access memory is over here in real computer land.

That's like, D-RAM SD-RAM, whatever-- the things you buy and stick into your motherboard, your GP, or whatever. And over here, the mathematical analog of-- so

here's, it's a RAM. Here, it's also a RAM.

Here, it's a random access machine. Here, it's a random access memory. It's technical detail.

But the idea is, if you look at RAM that's in your computer, it's basically a giant array, right? You can go from zero to, I don't know. A typical chip these days is like four gigs in one thing. So you can go from zero to four gigs. You can access anything in the middle there in constant time.

To access something, you need to know where it is. That's random access memory. So that's an array. So I'll just draw a big picture. Here's an array.

Now, RAM is usually organized by words. So these are a machine word, which we're going to put in this model. And then there's address zero, address one, address two.

This is the fifth word. And just keeps going. You can think of this as infinite. Or the amount that you use, that's the space of your algorithm, if you care about storage space.

So that's basically it. OK, now how do we-- this is the memory side of things. How do we actually compute with it?

It's very simple. We just say, in constant time, an algorithm can basically read in or load a constant number of words from memory, do a constant number of computations on them, and then write them out. It's usually called store.

OK, it needs to know where these words are. It accesses them by address. And so I guess I should write here you have a constant number of registers just hanging around.

So you load some words into registers. You can do some computations on those registers. And then you can write them back, storing them in locations that are specified by your registers.

So you've ever done assembly programming, this is what assembly programming is like. And it can be rather annoying to write algorithms in this model. But in some sense, it is reality.

This is how we think about computers. If you ignore things like caches, this is an accurate model of computation that loading, computing, and storing all take roughly the same amount of time. They all take constant time.

You can manipulate a whole word at a time. Now, what exactly is a word? You know, computers these days, it's like 32 bits or 64 bits.

But we like to be a little bit more abstract. A word is w bits. It's slightly annoying.

And most of this class, we won't really worry about what w is. We'll assume that we're given as input a bunch of things which are words. So for example, peak finding.

We're given a matrix of numbers. We didn't really say whether they're integers or floats or what. We don't worry about that. We just think of them as words and we assume that we can manipulate those words.

In particular, given two numbers, we can compare them. Which is bigger? And so we can determine, is this cell in the matrix a peak by comparing it with its neighbors in constant time.

We didn't say why it was constant time to do that. But now you kind of know. If those things are all words and you can manipulate a constant number of words in constant time, you can tell whether a number is a peak in constant time.

Some things like w should be at least log the size of memory. Because my word should be able to specify an index into this array. And we might use that someday. But basically, don't worry about it.

Words are words. Words come in as inputs. You can manipulate them and you don't have to worry about it for the most part.

In unit four of this class, we're going to talk about, what if we have really giant integers that don't fit in a word? How do we manipulate them? How do we add them, multiply them?

So that's another topic. But most of this class, we'll just assume everything we're given is one word. And it's easy to compute on.

So this is a realistic model, more or less. And it's a powerful one. But a lot of the time, a lot of code just doesn't use arrays-- doesn't need it.

Sometimes we need arrays, sometimes we don't. Sometimes you feel like a nut, sometimes you don't. So it's useful to think about somewhat more abstract models that are not quite as powerful but offer a simpler way of thinking about things. For example, in this model there's no dynamic memory allocation.

You probably know you could implement dynamic memory allocation because real computers do it. But it's nice to think about a model where that's taken care of for you. It's kind of like a higher level programming abstraction.

So the one is useful in this class is the pointer machine. This basically corresponds to object oriented programming in a simple, very simple version. So we have dynamically allocated objects. And an object has a constant number of fields.

And a field is going to be either a word-- so you can think of this as, for example, storing an integer, one of the input objects or something you computed on it or a counter, all these sorts of things-- or a pointer. And that's where pointer machine gets its name. A pointer is something that points to another object or has a special value null, also known as nil, also known as none in Python.

OK, how many people have heard about pointers before? Who hasn't? Willing to admit it?

OK, only a few. That's good. You should have seen pointers.

You may have heard them called references. Modern languages these days don't call them pointers because pointers are scary. But there's a very subtle difference

between them.

And this model actually really is references. But for whatever reason, it's called a pointer machine. It doesn't matter.

The point is, you've seem linked lists I hope. And linked lists have a bunch of fields in each node. Maybe you've got a pointer to the previous element, a pointer to the next element, and some value. So here's a very simple linked list. This is what you'd call a doubly linked list because it has previous and next pointers.

So the next pointer points to this node. The previous pointer points to this node. Next pointer points to null.

The previous pointer points to null, let's say. So that's a two node doubly linked list. You presume we have a pointer to the head of the list, maybe a pointer to the tail of list, whatever.

So this is a structure in the pointer machine. It's a data structure. In Python, you might call this a named tuple, or it's just an object with three attributes, I guess, they're called in Python.

So here we have the value. That's a word like an integer. And then some things can be pointers that point to other nodes.

And you can create a new node. You can destroy a node. That's the dynamic memory allocation.

In this model, yeah, pointers are pointers. You can't touch them. Now, you can implement this model in a random access machine.

A pointer becomes an index into this giant table. And that's more like the pointers in C if you've ever written C programs. Because then you can take a pointer and you can add one to it and go to the next thing after that.

In this model, you can just follow a pointer. That's all you can do. OK, following a pointer costs constant time. Changing one of these fields costs constant time. All the

usual things you might imagine doing to these objects take constant time.

So it's actually a weaker model than this one. Because you could implement a pointer machine with a random access machine. But it offers a different way of thinking. A lot of data structures are built this way.

Cool. So that's the theory side. What I'd like to talk about next is actually in Python, what's a reasonable model of what's going on?

So these are old models. This goes back to the '80s. This one probably '80s or '70s.

So they've been around a long time. People have used them forever. Python is obviously much more recent, at least modern versions of Python.

And it's the model of computation in some sense that we use in this class. Because we're implementing everything in Python. And Python offers both a random access machine perspective because it has arrays, and it offers a pointer machine perspective because it has references, because it has pointers.

So you can do either one. But it also has a lot of operations. It doesn't just have load and store and follow pointer.

It's got things like sort and append and concatenation of two lists and lots of things. And each of those has a cost associated with them. So whereas the random access machine and pointer machine, they're theoretical models. They're designed to be super simple.

So it's clear that everything you do takes constant time. In Python, some of the operations you can do take a lot of time. Some of the operations in Python take exponential time to do.

And you've got to know when you're writing your algorithms down either thinking in a Python model or your implementing your algorithms in actual Python, which operations are fast and which are slow. And that's what I'd like to spend the next few minutes on. There's a lot of operations.

I'm not going to cover all of them. But we'll cover more in recitation. And there's a whole bunch in my notes. I won't get to all of them.

So in Python, you can do random access style things. In Python, arrays are called lists, which is super confusing. But there you go.

A list in Python is an array in real world. It's a super cool array, of course? And you can think of it as a list. But in terms implementation, it's implemented as an array. Question?

AUDIENCE: I thought that [INAUDIBLE].

PROFESSOR: You thought Python links lists were linked lists. That's why it's so confusing. In fact, they are not.

In, say, scheme, back in the days when we taught scheme, lists are linked lists. And it's very different. So when you do-- I'll give an operation here.

You have a list L, and you do something like this. L is a list object. This takes constant time.

In a linked list, it would take linear time. Because we've got a scan to position I, scan to position J, add 5, and store. But conveniently in Python, this takes constant time. And that's important to know.

I know that the terminology is super confusing. But blame the benevolent dictator for life. On the other hand, you can do style two, pointer machine, using object oriented programming, obviously.

I'll just mention that I'm not really worrying about methods here. Because methods are just sort of a way of thinking about things, not super important from a cost standpoint. If your object has a constant number of attributes-- it can't have like a million attributes or can't have n executes-- then it fits into this pointer machine model.

So if you have an object that only has like three things or 10 things or whatever,

that's a pointer machine. You can think of manipulating that object as taking constant time. If you are screwing around the object's dictionary and doing lots of crazy things, then you have to be careful about whether this remains true.

But as long as you only have a reasonable number of attributes, this is all fair game. And so if you do something like, if you're implementing a linked list, Python I checked still does not have built-in linked lists. They're pretty easy to build, though.

You have a pointer. And you just say x equals x.next. That takes constant time because accessing this field in an object of constant size takes constant time. And we don't care what these constants are.

That's the beauty of algorithms. Because we only care about scalability with n. There's no n here.

This takes constant time. This takes constant time. No matter how big your linked list is or no matter how many objects you have, these are constant time.

OK, let's do some harder ones, though. In general, the idea is, if you take an operation like L.append-- so you have a list. And you want to append some item to the list. It's an array, though.

So think about it. The way to figure out how much does this cost is to think about how it's implemented in terms of these basic operations. So these are your sort of the core concept time things.

Most everything can be reduced to thinking about this. But sometimes, it's less obvious. L.apend is a little tricky to think about. Because basically, you have an array of some size. And now you want to make an array one larger.

And the obvious way to do that is to allocate a new array and copy all the elements. That would take linear time. Python doesn't do that.

What does it do? Stay tuned for lecture eight. It does something called table doubling.

It's a very simple idea. You can almost get guess it from the title. And if you go to lecture-- is it eight or nine? Nine, sorry.

You'll see how this can basically be done in constant time. There's a slight catch, but basically, think of it as a constant time operation. Once we have that, and so this is why you should take this class so you'll understand how Python works.

This is using an algorithmic concept that was invented, I don't know, decades ago, but is a simple thing that we need to do to solve lots of other problems. So it's cool. There's a lot of features in Python that use algorithms. And that's kind of why I'm telling you.

All right, so let's do another one. A little easier. What if I want to concatenate two lists?

You should know in Python this is a non-destructive operation. You basically take a copy of L1 and L2 and concatenate them. Of course, they're arrays.

The way to think about this is to re-implement it as Python code. This is the same thing as saying, well, L is initially empty. And then for every item x and L1, L.append(x).

And a lot of the times in documentation for Python, you see this sort of here's what it means, especially in the fancier features. They give sort of an equivalent simple Python, if you will. This doesn't use any fancy operations that we haven't seen already.

So now we know this takes constant time. The append, this append, takes constant time. And so the amount of time here is basically order the length of L1. And the time here is order the length of L2.

And so in total, it's order-- I'm going to be careful and say 1 plus length of L1 plus length of L2. The 1 plus is just in case these are both 0. It still takes constant time to build an initial list.

OK, so there are a bunch of operations that are written in these notes. I'm not going

to go through all of them because they're tedious. But a lot of you, could just expand out code like this.

And it's very easy to analyze. Whereas you just look at plus, you think, oh, plus is constant time. And plus is constant time if this is a word and this is a word.

But these are entire data structures. And so it's not constant time. All right. There are more subtle fun ones to think about. Like, if I want to know is x in the list, how does that happen? Any guesses?

There's an operator in Python called in-- x in L. How long do you think this takes? Altogether?

Linear, yeah. Linear time. In the worst case, you're going to have to scan through the whole list.

Lists aren't necessarily sorted. We don't know anything about them. So you've got to just scan through and test for every item. Is x equal to that item?

And it's even worse if equal equals costs a lot. So if x is some really complicated thing, you have to take that into account. OK, blah, blah, blah.

OK, another fun one. This is like a pop quiz. How long's it take to compute the length of a list? Constant.

Yeah, luckily, if you didn't know anything, you'd have to scan through the list and count the items. But in Python, lists are implemented with a counter built in.

It always stores the list at the beginning-- stores the length of the list at the beginning. So you just look it up. This is instantaneous.

It's important, though. That can matter. All right. Let's do some more.

What if I want to sort a list? How long does that take? N log n where n is the length of the list. Technically times the time to compare two items, which usually we're just sorting words. And so this is constant time.

If you look at Python sorting algorithm, it uses a comparison sort. This is the topic of lectures three and four and seven. But in particular, the very next lecture, we will see how this is done in n log n time.

And that is using algorithms. All right, let's go to dictionaries. Python called dicts. And these let you do things. They're a generalization of lists in some sense. Instead of putting just an index here, an integer between 0 and the length minus 1, you can put an arbitrary key and store a value, for example.

How long does this take? I'm not going to ask you because, it's not obvious. In fact, this is one of the most important data structures in all of computer science. It's called a hash table.

And it is the topic of lectures eight through 10. So stay tuned for how to do this in constant time, how to be able to store an arbitrary key, get it back out in constant time. This is assuming the key is a single word. Yeah.

AUDIENCE: Does it first check to see whether the key is already in the dictionary?

PROFESSOR: Yeah, it will clobber any existing key. There's also, you know, you can test whether a key is in the dictionary. That also takes constant time.

You can delete something from the dictionary. All the usual-- dealing with a single key in dictionaries, obviously dictionary.update, that involves a lot of keys. That doesn't take some time. How long does it take? Well, you write out a for loop and count them.

AUDIENCE: But how can you see whether [INAUDIBLE] dictionary in constant time?

PROFESSOR: How do you do this in constant time? Come to lecture eight through 10. I should say a slight catch, which is this is constant time with high probability.

It's a randomized algorithm. It doesn't always take constant time. It's always correct. But sometimes, very rarely, it takes a little more than constant time.

And I'm going to abbreviate this WHP. And we'll see more what that means mostly,

actually, in 6046. But we'll see a fair amount in 6006 on how this works and how it's possible. It's a big area of research.

A lot of people work on hashing. It's very cool and it's super useful. If you write any code these days, you use a dictionary. It's the way to solve problems.

I'm basically using Python is a platform to advertise the rest of the class you may have noticed.

Not every topic we cover in this class is already in Python, but a lot of them are. So we've got table doubling. We've got dictionaries.

We've got sorting. Another one is longs, which are long integers in Python through version two. And this is the topic of lecture 11. And so for fun, if I have two integers x and y, and let's say one of them is this many words long and the other one is this many words long, how long do you think it takes to add them? Guesses?

AUDIENCE: [INAUDIBLE].

PROFESSOR: Plus? Times? Plus is the answer.

You can do it in that much time. If you think about the grade school algorithm for adding really big multi-digit numbers, it'll only take that much time. Multiplication is a little bit harder, though.

If you look at the grade school algorithm, it's going to be x times y-- it's quadratic time not so good. The algorithm that's implemented in Python is x plus y to the log base 2 of 3.

By the way, I always write LG to mean log base 2. Because it only has two letters, so OK, this is 2. Log base 2 of 3 is about 1.6.

So while the straightforward algorithm is basically x plus y squared, this one is x plus y to the 1.6 power, a little better than quadratic. And the Python developers found that was faster than grade school multiplication. And so that's what they implemented.

And that is something we will cover in lecture 11, how to do that. It's pretty cool. There are faster algorithms, but this is one that works quite practically.

One more. Heap queue, this is in the Python standard library and implements something called the heap, which will be in lecture four. So, coming soon to a classroom near you.

All right, enough advertisement. That gives you some idea of the model of computation. There's a whole bunch more in these notes which are online. Go check them out. And some of them, we'll cover in recitation tomorrow.

I'd like to-- now that we are sort of comfortable for what costs what in Python, I want to do a real example. So last time, we did peak finding. We're going to have another example which is called document distance. So let's do that. Any questions before we go on?

All right. So document distance problem is, I give you two documents. I'll call them D1 D2. And I want to compute the distance between them. And the first question is, what does that mean? What is this distance function?

Let me first tell you some motivations for computing document distance. Let's say you're Google and you're cataloging the entire web. You'd like to know when two web pages are basically identical.

Because then you store less and because you present it differently to the user. You say, well, there's this page. And there's lots of extra copies. But you just need-- here's the canonical one.

Or you're Wikipedia. And I don't know if you've ever looked at Wikipedia. There's a list of all mirrors of Wikipedia.

There's like millions of them. And they find them by hand. But you could do that using document distance. Say, are these basically identical other than like some stuff at the-- junk at the beginning or the end?

Or if you're teaching this class and you want to detect, are two problem sets cheating? Are they identical? We do this a lot.

I'm not going to tell you what distance function we use. Because that would defeat the point. It's not the one we cover in class.

But we use automated tests for whether you're cheating. I've got some more. Web search.

Let's say you're Google again. And you want to implement searching. Like, I give you three words.

I'm searching for introduction to algorithms. You can think of introduction to algorithms as a very short document. And you want to test whether that document is similar to all the other documents on the web.

And the one that's most similar, the one that has the small distance, that's maybe what you want to put at the top. That's obviously not what Google does. But it's part of what it does.

So that's why you might care. It's partly also just a toy problem. It lets us illustrate a lot of the techniques that we develop in this class.

All right, I'm going to think of a document as a sequence of words. Just to be a little bit more formal, what do I mean by document? And a word is just going to be a string of alphanumeric characters-- A through Z and zero through nine.

OK, so if I have a document which you also think of as a string and you basically delete all the white space and punctuation all the other junk that's in there. This Everything in between those, those are the words. That's a simple definition of decomposing documents into words.

And now we can think of about what-- I want to know whether D1 and D2 are similar. And I've thought about my document as a collection of words. Maybe they're similar if they share a lot of words in common.

So that's the idea. Look at shared words and use that to define document distance. This is obviously only one way to define distance.

It'll be the way we do it in this class. But there are lots of other possibilities. So I'm going to think of a document.

It's a sequence of words. But I could also think of it as a vector. So if I have a document D and I have a word W, this D of W is going to be the number of times that word occurs in the document. So, number of recurrences W in the document D.

So it's a number. It's an integer. Non-negative integer. Could be 0. Could be one. Could be a million.

I think of this as a giant vector. A vector is indexed by all words. And for each of them, there's some frequency.

Of lot of them are zero. And then some of them have some positive number occurrences. You could think of every document is as being one of these plots in this common axis.

There's infinitely many words down here. So it's kind of a big axis. But it's one way to draw the picture.

OK, so for example, take two very important documents. Everybody likes cats and dogs. So these are two word documents.

And so we can draw them. Because there's only three different words here, we can draw them in three dimensional space. Beyond that, it's a little hard to draw.

So we have, let's say, which one's the-- let's say this one's the-- makes it easier to draw. So there's going to be just zero here and one. For each of the axes, let's say this is dog and this is cat.

OK, so the cat has won the-- it has one cat and no dog. So it's here. It's a vector pointing out there.

The dog you've got basically pointing there. OK, so these are two vectors. So how do I measure how different two vectors are? Any suggestions from vector calculus?

AUDIENCE: Inner product?

PROFESSOR: Inner product? Yeah, that's good suggestion. Any others.

OK, we'll go with inner product. I like inner product, also known as dot product. Just define that quickly.

So we could-- I'm going to call this D prime because it's not what we're going to end up with. We could think of this as the dot product of D1 and D2, also known as the sum over all words of D1 of W times D2 of W. So for example, you take the dot product of these two guys.

Those match. So you get one point there, cat and dog multiplied by zero. So you don't get much there.

So this is some measure of distance. But it's a measure of, actually, of commonality. So it would be sort of inverse distance, sorry.

If you have a high dot product, you have a lot of things in common. Because a lot of these things didn't be-- wasn't zero times something. It's actually a positive number times some positive number. If you have a lot of shared words, than that looks good.

The trouble of this is if I have a long document-- say, a million words-- and it's 99% in common with another document that's a million words long, it's still-- it looks super similar. Actually, I need to do it the other way around. Let's say it's a million words long and half of the words are in common.

So not that many, but a fair number. Then I have a score of like 500,000. And then I have two documents which are, say, 100 words long. And they're identical.

Their score is maybe only 100. So even though they're identical, it's not quite scale invariant. So it's not quite a perfect measure.

Any suggestions for how to fix this? This, I think, is a little trickier. Yeah?

AUDIENCE: Divide by the length of the vectors?

PROFESSOR: Divide by the length of the vectors. I think that's worth a pillow. Haven't done any pillows yet. Sorry about that. So, divide by the length of vector. That's good.

I'm going to call this D double prime. Still not quite the right answer. Or not-- no, it's pretty good. It's pretty good.

So here, the length of the vectors is the number of words that occur in them This is pretty cool. But does anyone recognize this formula? Angle, yeah.

It's a lot like the angle between the two vectors. It's just off by an arc cos. This is the cosine of the angle between the two vectors.

And I'm a geometer. I like geometry. So if you take arc cos of that thing, that's a well established distance metric. This goes back to '75, if you can believe it, back when people-- early days of document, information retrieval, way before the web, people were still working on this stuff.

So it's a natural measure of the angle between the two vectors. If it's 0, they're basically identical. If it's 90 degrees, they're really, really different. And so that gives you a nice way to compute document distance.

The question is, how do we actually compute that measure? Now that we've come up with something that's reasonable, how do I actually find this value? I need to compute these vectors-- the number of recurrences of each word in the document.

And I need you compute the dot product. And then I need to divide. That's really easy.

So, dot product-- and I also need to decompose a document to a list of words. So there are three things I need to do. Let me write them down.

So a sort of algorithm. There's one, split a document into words. Second is compute

word frequencies, how many times each word appears. This is the document vectors .

And then the third step is to compute the dot product. Let me tell you a little bit about how each of those is done. Some of these will be covered more in future lectures. I want to give you an overview.

There's a lot of ways to do each of these steps. If you look at the-- next to the lecture notes for this lecture two, there's a bunch of code and a bunch of data examples of documents-- big corpuses of text. And you can run, I think, there are eight different algorithms for it.

And let me give you-- actually, why don't I cut to the chase a little bit and tell you about the run times of these different implementations of this same algorithms. There are lots of sort of versions of this algorithm. We implement it a whole bunch.

Every semester I teach this, I change them a little bit more, add a few more variants. So version one, on a particular pair of documents which is like a megabyte-- not very much text-- it takes 228.1 seconds-- super slow. Pathetic.

Then we do a little bit of algorithmic tweaking. We get down to 164 seconds. Then we get to 123 seconds.

Then we get down to 71 seconds. Then we get down to 18.3 seconds. And then we get to 11.5 seconds.

Then we get to 1.8 seconds. Then we get to 0.2 seconds. So factor of 1,000. This is just in Python.

2/10 of a second to process a megabytes. It's all right. It's getting reasonable.

This is not so reasonable. Some of these improvements are algorithmic. Some of them are just better coding.

So there's improving the constant factors. But if you look at larger and larger texts, this will become even more dramatic. Because a lot of these were improvements

from quadratic time algorithms to linear and log n algorithms.

And so for a megabyte, yeah, it's a reasonable improvement. But if you look at a gigabyte, it'll be a huge improvement. There will be no comparison.

In fact, there will be no comparison. Because this one will never finish. So the reason I ran such a small example so I could have patience to wait for this one. But this one you could run on the bigger examples.

All right, so where do I want to go from here? Five minutes. I want to tell you about some of those improvements and some of the algorithms here.

Let's start with this very simple one. How would you split a document into words in Python? Yeah?

AUDIENCE: [INAUDIBLE]. Iterate through the document, [INAUDIBLE] the dictionary?

PROFESSOR: Iterate through the-- that's actually how we do number two. OK, we can talk about that one. Iterate through the words in the document and put it in a dictionary.

Let's say, count of word plus equals 1. This would work if count is something called a count dictionary if you're super Pythonista. Otherwise, you have to check, is the word in the dictionary?

If not, set it to one. If it is there, add one to it. But I think you know what this means.

This will count the number of words-- this will count the frequency of each word in the dictionary. And becomes dictionaries run in constant time with high probability-- with high probability-- this will take order-- well, cheating a little bit. Because words can be really long.

And so to reduce a word down to a machine word could take order the length of the word time. To a little more precise, this is going to be the sum of the lengths of the words in the document, which is also known as a length of the document, basically. So this is good. This is linear time with high probability.

OK, that's a good algorithm. That is introduced in algorithm four. So we got a significant boost.

There are other ways to do this. For example, you could sort the words and then run through the sorted list and count, how many do you get in a row for each one? If it's sorted, you can count-- I mean, all the identical words are put right next to each other.

So it's easy to count them. And that'll run almost as fast. That was one of these algorithms.

OK, so that's a couple different ways to do that. Let's go back to this first step. How would you split a document into words in the first place? Yeah?

AUDIENCE: Search circulated spaces and then [INAUDIBLE].

PROFESSOR: Run through though the string. And every time you see anything that's not alphanumeric, start a new word. OK, that would run in linear time.

That's a good answer. So it's not hard. If you're a fancy Pythonista, you might do it like this.

Remember my Reg Exes. This will find all the words in a document. Trouble is, in general, re takes exponential time.

So if you think about algorithms, be very careful. Unless you know how re is implemented, this probably will run in linear time. But it's not obvious at all.

Do anything fancy with regular expressions. If you don't know what this means, don't worry about it. Don't use it.

If you know about it, be very careful in this class when you use re. Because it's not always linear time. But there is an easy algorithm for this, which is just scan through and look for alpha numerics.

String them together. It's good. There's a few other algorithms here in the notes.

You should check them out. And for fun, look at this code and see how small differences make dramatic difference in performance. Next class will be about sorting.