So, the topic today is dynamic programming. The term programming in the name of this term doesn't refer to computer programming. OK, programming is an old word that means any tabular method for accomplishing something. So, you'll hear about linear programming and dynamic programming. Either of those, even though we now incorporate those algorithms in computer programs, originally computer programming, you were given a datasheet and you put one line per line of code as a tabular method for giving the machine instructions as to what to do. OK, so the term programming is older. Of course, and now conventionally when you see programming, you mean software, computer programming. But that wasn't always the case. And these terms continue in the literature. So, dynamic programming is a design technique like other design techniques we've seen such as divided and conquer. OK, so it's a way of solving a class of problems rather than a particular algorithm or something. So, we're going to work through this for the example of so-called longest common subsequence problem, sometimes called LCS, OK, which is a problem that comes up in a variety of contexts. And it's particularly important in computational biology, where you have long DNA strains, and you're trying to find commonalities between two strings, OK, one which may be a genome, and one may be various, when people do, what is that thing called when they do the evolutionary comparisons? The evolutionary trees, yeah, right, yeah, exactly, phylogenetic trees, there you go, OK, phylogenetic trees. Good, so here's the problem. So, you're given two sequences, x going from one to m, and y running from one to n. You want to find a longest sequence common to both. OK, and here I say a, not the, although it's common to talk about the longest common subsequence. Usually the longest comment subsequence isn't unique. There could be several different subsequences that tie for that. However, people tend to, it's one of the sloppinesses that people will say. I will try to say a, unless it's unique. But I may slip as well because it's just such a common thing to just talk about the, even though there might be multiple. So, here's an example. Suppose x is this sequence, and y is this sequence. So, what is a longest common subsequence of those two sequences? See if you can just eyeball it. AB: length two? Anybody have one longer? Excuse me? BDB, BDB. BDAB, BDAB, BDAB, anything longer? So, BDAB: that's the longest one. Is there another one that's the same length? Is there another one that ties? BCAB, BCAB, another one? BCBA, yeah, there are a bunch of them all of length four. There isn't one of length five. OK, we are actually going to come up with an algorithm that, if it's correct, we're going to show it's correct, guarantees that there isn't one of length five. So all those are, we can say, any one of these is the longest comment subsequence of x and y. We tend to use it this way using functional notation, but it's not a function that's really a relation. So, we'll say something is an LCS when really we only mean it's an element, if you will, of the set of longest common subsequences. Once again, it's classic abusive notation. As long as we know what we mean, it's OK to abuse notation. What we can't do is misuse it. But abuse, yeah! Make it so it's easy to deal with. But you have to know what's going on underneath. OK, so let's see, so there's a fairly simple brute force algorithm for solving this problem. And that is, let's just check every, maybe some of you did this in your heads, subsequence of x from one to m to see if it's also a subsequence of y of one to n. So, just take every subsequence that you can get here, check it to see if it's in there. So let's analyze that. So, to check, so if I give you a subsequence of x, how long does it take you to check whether it is, in fact, a subsequence of y? So, I give you something like BCAB. How long does it take me to check to see if it's a subsequence of y? Length of y, which is order n. And how do you do it? Yeah, you just scan. So as you hit the first character that matches, great. Now, if you will, recursively see whether the suffix of your string matches the suffix of x. OK, and so, you are just simply walking down the tree to see if it matches. You're walking down the string to see if it matches. OK, then the second thing is, then how many subsequences of x are there? Two to the n? x just goes from one to m, two to the m subsequences of x, OK, two to the m. Two to the m subsequences of x, OK, one way to see that, you say, well, how many subsequences are there of something there? If I consider a bit vector of length m, OK, that's one or zero, just every position where there's a one, I take out, that identifies an element that I'm going to take out. OK, then that gives me a mapping from each subsequence of x, from each bit vector to a different subsequence of x. Now, of course, you could have matching characters there, that in the worst case, all of the characters are different. OK, and so every one of those will be a unique subsequence. So, each bit vector of length m corresponds to a subsequence. That's a generally good trick to know. So, the worst-case running time of this method is order n times two to the m, which is, since m is in the exponent, is exponential time. And there's a technical term that we use when something is exponential time. Slow: good. OK, very good. OK, slow, OK, so this is really bad. This is taking a long time to crank out how long the longest common subsequence is because there's so many subsequences. OK, so we're going to now go through a process of developing a far more efficient algorithm for this problem. OK, and we're actually going to go through several stages. The first one is to go through simplification stage. OK, and what we're going to do is look at simply the length of the longest common sequence of x and y. And then what we'll do is extend the algorithm to find the longest common subsequence itself. OK, so we're going to look at the length. So, simplify the problem, if you will, to just try to compute the length. What's nice is the length is unique. OK, there's only going to be one length that's going to be the longest. OK, and what we'll do is just focus on the problem of computing the length. And then we'll do is we can back up from that and figure out what actually is the subsequence that realizes that length. OK, and that will be a big simplification because we don't have to keep track of a lot of different possibilities at every stage. We just have to keep track of the one number, which is the length. So, it's sort of reduces it to a numerical problem. We'll adopt the following notation. It's pretty standard notation, but I just want, if I put absolute values around the string or a sequence, it denotes the length of the sequence, S. OK, so that's the first thing. The second thing we're going to do is, actually, we're going to, which takes a lot more insight when you come up with a problem like this, and in some sense, ends up being the hardest part of designing a good dynamic programming algorithm from any problem, which is we're going to actually look not at all subsequences of x and y, but just prefixes. OK, we're just going to look at prefixes and we're going to show how we can express the length of the longest common subsequence of prefixes in terms of each other. In particular, we're going to define c of ij to be the length, the longest common subsequence of the prefix of x going from one to i, and y of going to one to j. And what we are going to do is we're going to calculate c[i,j] for all ij. And if we do that, how then do we solve the problem of the longest common of sequence of x and y? How do we solve the longest common subsequence? Suppose we've solved this for all I and j. How then do we compute the length of the longest common subsequence of x and y? Yeah, c[m,n], that's all, OK? So then, c of m, n is just equal to the longest common subsequence of x and y, because if I go from one to n, I'm done, OK? And so, it's going to turn out that what we want to do is figure out how to express to c[m,n], in general, c[i,j], in terms of other c[i,j]. So, let's see how we do that. OK, so our theorem is going to say that c[i,j] is just -- OK, it says that if the i'th character matches the j'th character, then i'th character of x matches the j'th character of y, then c of ij is just c of I minus one, j minus one plus one. And if they don't match, then it's either going to be the longer of c[i, j-1], and c[i-1, j], OK? So that's what we're going to prove. And that's going to give us a way of relating the calculation of a given c[i,j] to values that are strictly smaller, OK, that is at least one of the arguments is smaller of the two arguments. OK, and that's going to give us a way of being able, then, to understand how to calculate c[i,j]. So, let's prove this theorem. So, we'll start with a case x[i] equals y of j. And so, let's draw a picture here. So, we have x here. And here is y. OK, so here's my sequence, x, which I'm sort of drawing as this elongated box, sequence y, and I'm saying that x[i] and y[j], those are equal. OK, so let's see what that means. OK, so let's let z of one to k be, in fact, the longest common subsequence of x of one to i, y of one to j, where c of ij is equal to k. OK, so the longest common subsequence of x and y of one to I and y of one to j has some value. Let's call it k. And so, let's say that we have some sequence which realizes that. OK, we'll call it z. OK, so then, can somebody tell me what z of k is? What is z of k here? Yeah, it's actually equal to x of I, which is also equal to y of j? Why is that? Why couldn't it be some other value? Yeah, so you got the right idea. So, the idea is, suppose that the sequence didn't include this element here at the last element, the longest common subsequence. OK, so then it includes a bunch of values in here, and a bunch of values in here, same values. It doesn't include this or this. Well, then I could just tack on this extra character and make it be longer, make it k plus one because these two match. OK, so if the sequence ended before -- -- just extend it by tacking on x[i]. OK, it would be fairly simple to just tack on x[i]. OK, so if that's the case, then if I look at z going one up to k minus one, that's certainly a common sequence of x of 1 up to, excuse me, of up to i minus one. And, y of one up to j minus one, OK, because this is a longest common sequence. z is a longest common sequence is, from x of one to i, y of one to j. And, we know what the last character is. It's just x[i], or equivalently, y[j]. So therefore, everything except the last character must at least be a common sequence of x of one to i minus one, y of one to j minus one. Everybody with me? It must be a comment sequence. OK, now, what you also suspect? What do you also suspect about z of one to k? It's a common sequence of these two. Yeah? Yeah, it's a longest common sequence. So that's what we claim, z of one up to k minus one is in fact a longest common subsequence of x of one to i minus one, and y of one to j minus one, OK? So, let's prove that claim. So, we'll just have a little diversion to prove the claim. OK, so suppose that w is a longer comment sequence, that is, that the length, the w, is bigger than k minus one. OK, so suppose we have a longer comment sequence than z of one to k minus one. So, it's got to have length that's bigger than k minus one if it's longer. OK, and now what we do is we use a classic argument you're going to see multiple times, not just this week, which it will be important for this week, but through several lectures. Hence, it's called a cut and paste argument. So, the idea is let's take a look at w, concatenate it with that last character, z of k. so, this is string, OK, so that's just my terminology for string concatenation. OK, so I take whatever I claimed was a longer comment subsequence, and I concatenate z of k to it. OK, so that is certainly a common sequence of x of one to I minus one, and y of one to j. And it has length bigger than k because it's basically, what is its length? The length of w is bigger than k minus one. I add one character. So, this combination here, now, has length bigger that k. OK, and that's a contradiction, thereby proving the claim. So, I'm simply saying, I claim this. Suppose you have a longer one. Well, let me show, if I had a longer common sequence for the prefixes where we dropped the character from both strings if it was longer there, but we would have made the whole thing longer. So that can't be. So, therefore, this must be a longest common subsequence, OK? Questions? Because you are going to need to be able to do this kind of proof ad nauseam, almost. So, if there any questions, let them at me, people. OK, so now what we have established is that z one through k is a longest common subsequence of the two prefixes when we drop the last character. So, thus, we have c of i minus one, j minus one is equal to what? What's c of i minus one, j minus one? k minus one; thank you. Let's move on with the class, right, OK, which implies that c of ij is just equal to c of I minus one, j minus one plus one. So, it's fairly straightforward if you think about what's going on there. It's not always as straightforward in some problems as it is for longest common subsequence. The idea is, so I'm not going to go through the other cases. They are similar. But, in fact, we've hit on one of the two hallmarks of dynamic programming. So, by hallmarks, I mean when you see this kind of structure in a problem, there's a good chance that dynamic programming is going to work as a strategy. The dynamic programming hallmark is the following. This is number one. And that is the property of optimal substructure. OK, what that says is an optimal solution to a problem, and by this, we really mean problem instance. But it's tedious to keep saying problem instance. A problem is generally, in computer science, viewed as having an infinite number of instances typically, OK, so sorting is a problem. A sorting instance is a particular input. OK, so we're really talking about problem instances, but I'm just going to say problem, OK? So, when you have an optimal solution to a problem, contains optimal solutions to subproblems. OK, and that's worth drawing a box around because it's so important. OK, so here, for example, if z is a longest common subsequence of x and y, OK, then any prefix of z is a longest common subsequence of a prefix of x, and a prefix of y, OK? So, this is basically what it says. I look at the problem, and I can see that there is optimal substructure going on. OK, in this case, and the idea is that almost always, it means that there's a cut and paste argument you could do to demonstrate that, OK, that if the substructure were not optimal, then you'd be able to find a better solution to the overall problem using cut and paste. OK, so this theorem, now, gives us a strategy for being able to compute longest comment subsequence. Here's the code; oh wait. OK, so going to ignore base cases in this, if -- And we will return the value of the longest common subsequence. It's basically just implementing this theorem. OK, so it's either the longest comment subsequence if they match. It's the longest comment subsequence of one of the prefixes where you drop that character for both strengths and add one because that's the matching one. Or, you drop a character from x, and it's the longest comment subsequence of that. Or you drop a character from y, whichever one of those is longer. That ends up being the longest comment subsequence. OK, so what's the worst case for this program? What's going to happen in the worst case? Which of these two clauses is going to cause us more headache? The second clause: why the second clause? Yeah, you're doing two LCS sub-calculations here. Here, you're only doing one. Not only that, but you get to decrement both indices, whereas here you've basically got to, you only get to decrement one index, and you've got to calculate two of them. So that's going to generate the tree. So, and the worst case, x of i is not equal to x of j for all i and j. So, let's draw a recursion tree for this program to sort of get an understanding as to what is going on to help us. And, I'm going to do it with m equals seven, and n equals six. OK, so we start up the top with my two indices being seven and six. And then, in the worst case, we had to execute these. So, this is going to end up being six, six, and seven, five for indices after the first call. And then, this guy is going to split. And he's going to produce five, six here, decrement the first index, I. And then, if I keep going down here, we're going to get four, six and five, five. And these guys keep extending here. I get six five, five five, six four, OK? Over here, I'm going to get decrement the first index, six five, and I get five five, six four, and these guys keep going down. And over here, I get seven four. And then we get six four, seven three, and those keep going down. So, we keep just building this tree out. OK, so what's the height of this tree? Not of this one for the particular value of m and n, but in terms of m and n. What's the height of this tree? It's the max of m and n. You've got the right, it's theta of the max. It's not the max. Max would be, in this case, you're saying it has height seven. But, I think you can sort of see, for example, along a path like this that, in fact, I've only, after going three levels, reduced m plus n, good, very good, m plus n. So, height here is m plus n. OK, and its binary. So, the height: that implies the work is exponential in m and n. All that work, and are we any better off than the brute force algorithm? Not really. And, our technical term for this is slow. OK, and we like speed. OK, we like fast. OK, but I'm sure that some of you have observed something interesting about this tree. Yeah, there's a lot of repeated work here. Right, there's a lot of repeated work. In particular, this whole subtree, and this whole subtree, OK, they are the same. That's the same subtree, the same subproblem that you are solving. OK, you can even see over here, there is even similarity between this whole subtree and this whole subtree. OK, so there's lots of repeated work. OK, and one thing is, if you want to do things fast, don't keep doing the same thing. OK, don't keep doing the same thing. When you find you are repeating something, figure out a way of not doing it. So, that brings up our second hallmark for dynamic programming. And that's a property called overlapping subproblems, OK? OK, recursive solution contains many, excuse me, contains a small number of distinct subproblems repeated many times. And once again, this is important enough to put a box around. I don't put boxes around too many things. Maybe I should put our boxes around things. This is definitely one to put a box around, OK? So, for example, so here we have a recursive solution. This tree is exponential in size. It's two to the m plus n in height, in size, in the total number of problems if I actually implemented like that. But how many distinct subproblems are there? m times n, OK? So, the longest comment subsequence, the subproblem space contains m times n, distinct subproblems. OK, and then this is a small number compared with two to the m plus n, or two to the n, or two to the m, or whatever. OK, this is small, OK, because for each subproblem, it's characterized by an I and a j. An I goes from one to m, and j goes from one to n, OK? There aren't that many different subproblems. It's just the product of the two. So, here's an improved algorithm, which is often a good way to solve it. It's an algorithm called a memo-ization algorithm. And, this is memo-ization, not memorization because what you're going to do is make a little memo whenever you solve a subproblem. Make a little memo that says I solved this already. And if ever you are asked for it rather than recalculating it, say, oh, I see that. I did that before. Here's the answer, OK? So, here's the code. It's very similar to that code. So, it basically keeps a table around of c[i,j]. It says, what we do is we check. If the entry for c[i,j] is nil, we haven't computed it, then we compute it. And, how do we compute it? Just the same way we did before. OK, so this whole part here, OK, is exactly what we have had before. It's the same as before. And then, we just return c[i,j]. If we don't bother to keep recalculating, OK, so if it's nil, we calculate it. Otherwise, we just return it. It's not calculated, calculate and return it. Otherwise, just return it: OK, pretty straightforward code. OK. OK, now the tricky thing is how much time does it take to execute this? This takes a little bit of thinking. Yeah? Yeah, it takes order MN. OK, why is that? Yeah, but I have to look up c[i,j]. I might call c[i,j] a bunch of times. When I'm doing this, I'm still calling it recursively. Yeah, so you have to, so each recursive call is going to look at, and the worst-case, say, is going to look at the max of these two things. Well, this is going to involve a recursive call, and a lookup. So, this might take a fair amount of effort to calculate. I mean, you're right, and your intuition is right. Let's see if we can get a more precise argument, why this is taking order m times n. What's going on here? Because not every time I call this is it going to just take me a constant amount of work to do this. Sometimes it's going to take me a lot of work. Sometimes I get lucky, and I return it. So, your intuition is dead on. It's dead on. We just need a little bit more articulate explanation, so that everybody is on board. Try again? Good, at most three times, yeah. OK, so that's one way to look at it. Yeah. There is another way to look at it that's kind of what you are expressing there is an amortized, a bookkeeping, way of looking at this. What's the amortized cost? You could say what the amortized cost of calculating one of these, where basically whenever I call it, I'm going to charge a constant amount for looking up. And so, I could get to look up whatever is in here to call the things. But if it, in fact, so in some sense, this charge here, of calling it and returning it, etc., I charged that to my caller. OK, so I charged these lines and this line to the caller. And I charge the rest of these lines to the c[i,j] element. And then, the point is that every caller basically only ends up being charged for a constant amount of stuff. OK, to calculate one c[i,j], it's only an amortized constant amount of stuff that I'm charging to that calculation of i and j, that calculation of i and j. OK, so you can view it in terms of amortized analysis doing a bookkeeping argument that just says, let me charge enough to calculate my own, do all my own local things plus enough to look up the value in the next level and get it returned. OK, and then if it has to go off and calculate, well, that's OK because that's all been charged to a different ij at that point. So, every cell only costs me a constant amount of time that order MN cells total of order MN. OK: constant work per entry. OK, and you can sort of use an amortized analysis to argue that. How much space does it take? We haven't usually looked at space, but here we are going to start looking at space. That turns out, for some of these algorithms, to be really important. How much space do I need, storage space? Yeah, also m times n, OK, to store the c[i,j] table. OK, the rest, storing x and y, OK, that's just m plus n. So, that's negligible, but mostly I need the space m times n. So, this memo-ization type algorithm is a really good strategy in programming for many things where, when you have the same parameters, you're going to get the same results. It doesn't work in programs where you have a side effect, necessarily, that is, when the calculation for a given set of parameters might be different on each call. But for something which is essentially like a functional programming type of environment, then if you've calculated it once, you can look it up. And, so this is very helpful. But, it takes a fair amount of space, and it also doesn't proceed in a very orderly way. So, there is another strategy for doing exactly the same calculation in a bottom-up way. And that's what we call dynamic programming. OK, the idea is to compute the table bottom-up. I think I'm going to get rid of, I think what we'll do is we'll just use, actually I think what I'm going to do is use this board. OK, so here's the idea. What we're going to do is look at the c[i,j] table and realize that there's actually an orderly way of filling in the table. This is sort of a top-down with memo-ization. OK, but there's actually a way we can do it bottom up. So, here's the idea. So, let's make our table. OK, so there's x. And then, there's y. And, I'm going to initialize the empty string. I didn't cover the base cases for c[i,j], but c of zero meaning a prefix with no elements in it. The prefix of that with anything else, the length is zero. OK, so that's basically how I'm going to bound the borders here. And now, what I can do is just use my formula, which I've conveniently erased up there, OK, to compute what is the longest common subsequence, length of the longest comment subsequence from this character in y, and this character in x up to this character. So here, for example, they don't match. So, it's the maximum of these two values. Here, they do match. OK, so it says it's one plus the value here. And, I'm going to draw a line. Whenever I'm going to get a match, I'm going to draw a line like that, indicating that I had that first case, the case where they had a good match. And so, all I'm doing is applying that recursive formula from the theorem that we proved. So here, it's basically they don't match. So, it's the maximum of those two. Here, they match. So, it's one plus that guy. Here, they don't match. So, it's basically the maximum of these two. Here, they don't match. So it's the maximum. So, it's one plus that guy. So, everybody understand how I filled out that first row? OK, well that you guys can help. OK, so this one is what? Just call it out. Zero, good. One, because it's the maximum, one, two, right. This one, now, gets from there, two, two. OK, here, zero, one, because it's the maximum of those two. Two, two, two, good. One, one, two, two, two, three, three. One, two, three, get that line, three, four, OK. One there, three, three, four, good, four. OK, and our answer: four. So this is blindingly fast code if you code this up, OK, because it gets to use the fact that modern machines in particular do very well on regular strides through memory. So, if you're just plowing through memory across like this, OK, and your two-dimensional array is stored in that order, which it is, otherwise you go this way, stored in that order. This can really fly in terms of the speed of the calculation. So, how much time did it take us to do this? Yeah, order MN, theta MN. Yeah? We'll talk about space in just a minute. OK, so hold that question. Good question, good question, already, wow, good, OK, how do I now figure out, remember, we had the simplification. We were going to just calculate the length. OK, it turns out I can now figure out a particular sequence that matches it. And basically, I do that. I can reconstruct the longest common subsequence by tracing backwards. So essentially I start here. Here I have a choice because this one was dependent on, since it doesn't have a bar here, it was dependent on one of these two. So, let me go this way. OK, and now I have a diagonal element here. So what I'll do is simply mark the character that appeared in those positions as I go this way. I have three here. And now, let me keep going, three here, and now I have another one. So that means this character gets selected. And then I go up to here, OK, and then up to here. And now I go diagonally again, which means that this character is selected. And I go to here, and then I go here. And then, I go up here and this character is selected. So here is my longest common subsequence. And this was just one path back. I could have gone a path like this and gotten a different longest common subsequence. OK, so that simplification of just saying, look, let me just run backwards and figure it out, that's actually pretty good because it means that by just calculating the value, then figuring out these back pointers to let me reconstruct it is a fairly simple process. OK, if I had to think about that to begin with, it would have been a much bigger mess. OK, so the space, I just mentioned, was order MN because we still need the table. So, you can actually do the min of m and n. OK, to get to your question, how do you do the min of m and n? Diagonal stripes won't give you min of m and n. That'll give you the sum of m and n. So, going in stripes, maybe I'm not quite sure I know what you mean. So, you're saying, so what's the order I would do here? So, I would start. I would do this one first. Then which one would I do? This one and this one? And then, this one, this one, this one, like this? That's a perfectly good order. OK, and so you're saying, then, so I'm keeping the diagonal there all the time. So, you're saying the length of the diagonal is the min of m and n? I think that's right. OK, there is another way you can do it that's a little bit more straightforward, which is you compare m to n. Whichever is smaller, well, first of all, let's just do this existing algorithm. If I just simply did row by row, I don't need more than a previous row. OK, I just need one row at a time. So, I can go ahead and compute just one row because once I computed the succeeding row, the first row is unimportant. And in fact, I don't even need the whole row. All I need is just the current row that I'm on, plus one or two elements of the previous row, plus the end of the previous row. So, I use a prefix of this row, and an extra two elements, and the suffix of this row. So, it's actually, you can do it with one row, plus order one element. And then, I could do it either running vertically or running horizontally, whichever one gives me the smaller space. OK, and it might be that your diagonal trick would work there too. I'd have to think about that. Yeah? Ooh, that's a good question. So, you can do the calculation of the length, and run row plus order one elements. OK, and our exercise, and this is a hard exercise. OK, so that a good one to do is to do small space and allow you to reconstruct the LCS because the naÃ”ve way that we were just doing it, it's not clear how you would go backwards from that because you've lost the information. OK, so this is actually a very interesting and tricky problem. And, it turns out it succumbs of all things to divide and conquer, OK, rather than some more straightforward tabular thing. OK: so very good practice, for example, for the upcoming take home quiz, OK, which is all design and cleverness type quiz. OK, so this is a good one for people to take on. So, this is basically the tabular method that's called dynamic programming. OK, memo-ization is not dynamic programming, even though it's related. It's memo-ization. And, we're going to see a whole bunch of other problems that succumb to dynamic programming approaches. It's a very cool method, and on the homework, so let me just mention the homework again. On the homework, we're going to look at a problem called the edit distance problem. Edit distance is you are given two strings. And you can imagine that you're typing in a keyboard with one of the strings there. And what you have to do is by doing inserts, and deletes, and replaces, and moving the cursor around, you've got to transform one string to the next. And, each of those operations has a cost. And your job is to minimize the cost of transforming the one string into the other. This actually turns out also to be useful for computational biology applications. And, in fact, there have been editors, screen editors, text editors, that implement algorithms of this nature in order to minimize the number of characters that have to be sent as IO in and out of the system. So, the warning is, you better get going on your programming on problem one on the homework today if at all possible because whenever I assign programming, since we don't do that as sort of a routine thing, I'm just concerned for some people that there will not be able to get things like the input and output to work, and so forth. We have example problems, and such, on the website. And we also have, you can write it in any language you want, including Matlab, Python, whatever your favorite, the solutions will be written in Java and Python. OK, so the fastest solutions are likely to be written in c. OK, you can also do it in assembly language if you care to. You laugh. I used to be in assembly language programmer back in the days of yore. OK, so I do encourage people to get started on this because let me mention, the other thing is that this particular problem on this problem set is an absolutely mandatory problem. OK, all the problems are mandatory, but as you know you can skip them and it doesn't hurt you too much if you only skip one or two. This one, you skip, hurts big time: one letter grade. It must be done.