6.046J Introduction to Algorithms, Fall 2005
Transcript – Lecture 14

And this is going to use some of the techniques we learned last time with respect to amortized analysis. And, what's neat about what we're going to talk about today is it's a way of comparing algorithms that are so-called online algorithms. And we're going to introduce this notion with a problem which is called self organizing lists. OK, and so the set up for this problem is that we have a list, L, of n elements. And, we have an operation. Woops, I've got to spell things right, access of x, which accesses item x in the list.

It could be by searching, or it could be however you want to do it. But basically, it goes and touches that element. And we were to say the cost of that operation is whatever the rank of x is in the list, which is just the distance of x from the head of the list. And the other thing that we can do that the algorithm can do, so this is what the user would do. He just simply runs a whole bunch of accesses on the list, OK, accessing one element after another in any order that he or she cares to. And then, L can be reordered, however, by transposing adjacent elements.

And the cost for that is one. So, for example, suppose the list is the following. OK, I missed something here. It doesn't matter. Well, I'll just make it be what I have so that it matches the online video. OK, so here we have a list. And so, if I do something like access element 14 here, the element of key 14, OK, that this costs me one, two, three, four. So here the cost is four to access. And so, we're going to have some sequence of accesses that the user is going to do. And obviously, if something is accessed more frequently, we'd like to move it up to the front of the list so that you don't have to search as far. OK, and to do that, if I want to transpose something, so, for example, if I transpose three and 50, that just costs me one.

OK, so then I would make this be 50, and make this be three. OK, sorry, normally you just do it by swapping pointers. OK, so those are the two operations. And, we are going to do this in what's called an online fashion. So, let's just define online. So, a sequence, S, of operations is provided one at a time for each operation. An online algorithm must execute the operation immediately without getting a chance to look at what else is coming in the sequence.

So, when you make your decision for the first element, you don't get to see ahead as to what the second, or third, or whatever is. And the second one you get, you get and you have to make your decision as to what to do and so forth. So, that's an online algorithm. Similarly, an off-line algorithm, OK, may see all of S in advance. OK, so you can see an off-line algorithm gets to see the whole sequence, and then decide what it wants to do about element one, element two, or whatever.

OK, so an off-line algorithm can look at the whole sequence and say, OK, I can see that item number 17 is being accessed a lot, or early on move him up closer to the front of the list, and then the accesses cost less for the off-line algorithm. The online algorithm doesn't get to see any of that. OK, so this is sort of like, if you're familiar with the game Tetris. OK, and Tetris, you get one shape after another that starts

coming down, and you have to twiddle it, and move it to the side, and drop it into place. And there, sometimes you get a one step look-ahead on some of them so you can see what the next shape is, but often it's purely online. You don't get to see that next shape or whatever, and you have to make a decision for each one. And you make a decision, and you realize that the next shape, ah, if you had made a different decision it would have been better. OK, so that's the kind of problem.

Off-line Tetris would be, I get to see the whole sequence of shapes. And now let me decide what I'm going to do with this one. OK, and so, in this, the goal is for any of the algorithms, either online or off-line is to minimize the total cost, which we'll denote by, I forgot to name this. This is algorithm A here. The total cost, C_A of S, OK, so the cost of algorithm A on the sequence, S. That's just the notation we'll use for what the total cost is. So, any questions about the setup to this problem? So, we have an online problem. We're going to get these things one at a time, OK, and we have to decide what to do.

So, let's do a worst-case analysis for this. OK, so if we're doing a worst-case analysis, we can view that we have an adversary that we are playing against who's going to provide the sequence. The user is going to be able to see what we do. And so, what's the adversary strategy? Thwart our plots, yes, that's his idea. And how is he going to thwart them, or she? Which is what for this problem? What's he going to do?

Yeah. No matter how we reorder elements using the transposes, he's going to look at every step and say, what's the last element? That's the one I'm going to access, right? So, the adversary always, always accesses the tail element of L. No matter what it is, no matter how we reorder things, OK, for each one, adversary just accesses the tail. So the cost of this, of any algorithm, then, is going to be omega size of S times n, OK, because you're always going to pay for every sequence. You're going to have to go in to pay a cost of n, OK, for every element in the sequence. OK, so not terribly in the worst-case.

Not terribly good. So, people in studying this problem: question? That analysis is for the online algorithm, right. The off-line algorithm, right, if you named those things, that's right. OK, so we're looking at trying to solve this in an off-line sense, sorry, in an online sense. OK, and so the point is that for the online algorithm, the adversary can be incredibly mean, OK, and just always access the thing at the end, OK? So, what sort of the history of this problem is, that people said, well, if I can't do well in the worst-case, maybe I should be looking at average case, OK, and look at, say, the different elements having some probability distribution.

OK, so the average case analysis, OK, let's suppose that element x is accessed with probability, P of x. OK, so suppose that we have some a priori distribution on the elements. OK, then the expected cost of the algorithm on a sequence, OK, so if I put all the elements into some order, OK, and don't try to reorder, but just simply look at, is there a static ordering that would work well for a distribution?

It's just going to be, by definition of expectation, the probability of x times, in this case, the cost, which is the rank of x in whatever that ordering is that I decide I'm going to use. OK, and this is minimized when? So, this is just the definition of expectations: the probability that I access x times the cost summed over all the elements. And the cost is just going to be its position in the list. So, when is this value, this summation, going to be minimized?

When the element is most likely as the lowest rank, and then what, what about the other element? OK, so what does that mean? Yeah, sort them, yeah, sort them on the basis of decreasing probability, OK? So, it's minimized when L is sorted, OK, in decreasing order with respect to P. OK, so just sort them with the most likely one at the front, and then just decreasing probability. That way, whenever I access something with some probability, OK, I'm going to access, it's more likely that I'm going to access.

And that's not too difficult to actually prove. You just look at, suppose there were two that were out of order, and show that if you swap them, you would improve this optimization function. OK, so if you didn't know it, this suggests the following heuristic, OK, which is simply keep account of the number of times each element is accessed, and maintain the list in order of decreasing count. OK, so whenever something is accessed, increment its count, OK, and that will move it, at most, one position, which only costs me one transposed to move it, perhaps, forward. OK, actually, I guess it could be more if you have a whole bunch of ties, right? Yeah. So, it could cost more. But the idea is, over time, the law of large numbers says that this is going to approach the probability distribution.

The frequency with which you access this, divided by the total number of accesses, will be the probability. And so, therefore you will get things in decreasing probability, OK, assuming that there is some distribution that all of these elements are chosen according to, or accessed according to. So, it doesn't seem like there's that much more you could really do here. And that's why I think this notion of competitive analysis is so persuasive, because it's really amazingly strong, OK? And it came about because of what people were doing in practice.

So practice, what people implement it was a so-called move to front heuristic. OK, and the basic idea was, after you access an element, just move it up to the front. OK, that only doubles the cost of accessing the element because I go and I access it, chasing it down paying the rank, and then I have to do rank number of transposes to bring it back to the front. So, it only cost me a factor of two, and now, if it happens to be a frequently accessed elements, over time you would hope that the most likely elements were near the front of that list.

So, after accessing x, move x, the head of the list using transposes, and the cost is just equal to twice the rank in L of x, OK, where the two here has two parts. One is the access, and the other is the transposes. OK, so that's sort of what they did. And one of the nice properties of this is that if it turns out that there is locality in the access pattern, if it's not just a static distribution, but rather once I've accessed something, if it's more likely I'm going to access it again, which tends to be the case for many input types of patterns, this responds well to locality because it's going to be up near the front if I access it very soon after I've accessed.

So, there is what's called temporal locality, meaning that in time, I tend to access, so it may be that I access some thing's very hot for awhile; then it gets very cold. This type of algorithm responds very well to the hotness of the accessing. OK, so it responds well to locality in S. So, this is sort of what was known up to the point that a very famous paper was written by Danny Sleator and Bob Tarjan, where they took a totally different approach to looking at this kind of problem. OK, and it's an approach that matter you see everywhere from analysis of caching and high-

performance processors to analyses of disk paging to just a huge number of applications of this basic technique.

And, that's the technique of competitive analysis. OK, so here's the definition. So, online algorithm A is alpha competitive. If there exists a constant, k, such that for any sequence, S, of operations, the cost of S using algorithm A is bounded by alpha times the cost of opt, where opt is the optimal offline algorithm. OK, so the optimal off-line, the one that knows the whole sequence and does the absolute best it could do on that sequence, OK, that's this cost here. This is sometimes called God's algorithm, OK, not to bring religion into the classroom, or to offend anybody, but that is what people sometimes call it.

OK, so the fully omniscient knows absolutely the best thing that could be done, sees into the future, the whole works, OK? It gets to apply that. That's what opts algorithm is. And, what we're saying is that the cost is basically whatever this alpha factor is. It could be a function of things, or it could be a constant, OK, times whatever the best algorithm is. Plus, there's a potential for a constant out here.

OK, so for example, if alpha is two, and we say it's two competitive, that means you're going to do, at worst, twice the algorithm that has all the information. But you're doing it online, for example. OK, it's a really pretty powerful notion. And what's interesting about this, it's not even clear these things should exist to my mind. OK, what's pretty remarkable about this, I think, is that there is no assumption of distribution, of probability distribution or anything. It's whatever the sequence is that you give it. You are within a factor of alpha, essentially, of the best algorithm, OK, which is pretty remarkable.

OK, and so, we're going to prove the following theorem, which is the one that Sleator and Tarjan proved. And that is that MTF is four competitive for self organizing lists. OK, so the idea here is that suppose the adversary says, oh, I'm always going to access the thing at the end of the list like we said in the beginning. So, the adversary says, I'm always going to access the thing there. I'm going to make MTF work really bad, because you're going to go and move that thing all the way up to the front. And I'm just going to access the thing way at the end again. OK, well it turns out, yeah, that's a bad sequence for move to front, OK, and it will take a long time.

But it turns out God couldn't have done better, OK, by more than a factor of four no matter how long the list is. OK, that's pretty amazing. OK, so that's a bad sequence. But, if there's a way that the sequence exhibits any kind of locality or anything that can be taken advantage of, if you could see the whole thing, MTF takes advantage of it too, OK, within a factor of four. OK, it's a pretty remarkable theorem, and it's the basis of many types of analysis of online algorithms. Almost all online algorithms today are analyzed using some kind of competitive analysis. OK, not always. Sometimes you do probabilistic analysis, or whatever, but the dominant thing is too competitive analysis because then you don't have to make any statistical assumptions.

OK, just prove that it works well no matter what. This is remarkable, I think. Isn't it remarkable? So, let's prove this theorem, we're just going to spend the rest of the lecture on this proof. OK, and the proof in some ways is not hard, but it's also not necessarily completely intuitive. So, you will have to pay attention. OK, so let's get some notation down. Let's let L_i be MTF's list after the i'th access. And, let's let L be opt's list after the i'th access.

OK, so generally what I'll do is I'll put a star if we are talking about opt, and have nothing if we're talking about MTF. OK, so that's going to be the list. So, we can say, what's the list? So, we're going to set it up where we have one in operation that transforms list i minus one into list i. OK, that's what the i'th operation does. OK, and move to front does it by moving whatever the thing that was accessed to the front. And opt does whatever opt thinks is the best thing to do. We don't know. So, we're going to let c_i be MTF's cost for the I'th operation.

And that's just twice the rank in L_i minus one of x if the operation accesses x, OK, two times the rank in L_i minus one because we're going to be accessing it in L_i minus one and transforming it into L_i. And similarly, we'll let c_i star be opt's cost for the i'th operation. And that's just equal to, well, to access it, it's going to be the rank in L_i minus one star, whatever its list is of x at that step, because it's got to access it.

And then, some number of transposes, t_i if opt forms t_i transposes. OK, so we have the setup where we have two different lists that are being managed, and we have different costs in the list. And, we're interested in is comparing in some way MTF's list with opt's list at any point in time. And, how do you think we're going to do that? What technique do you think we should use to compare these two lists, general technique from last lecture? Well, it is going to be amortized, but what?

How are we going to compare them? What technique did we learn last time? Potential function, good. OK, we're going to define a potential function, OK, that measures how far apart these two lists are. OK, and the idea is, if, let's define that and then we'll take a look at it. So, we're going to define the potential function phi mapping the set of MTF's lists into the real numbers by the following.

phi of L_i is going to be twice the cardinality of this set. OK, so this is the precedes-operation and list, i. So, we can define a relationship between any two elements that says that x precedes y in L_i if, as I'm accessing it from the head, I hit x first. OK, so what I'm interested in, here, are in some sense the disagreements between the two lists. This is where x precedes y in MTF's list, but y precedes x in opt's list. They disagree, OK? And, what we're interested in is the cardinality of the set. And we're going to multiply it by two. OK, so that's equal to two times; so there is a name for this type of thing. We saw that when we were doing sorting.

Anybody remember the name? It was very briefly. I don't expect anybody to remember, but somebody might. Inversions: good, OK, twice the number of inversions. So, let's just do an example. So, let's say L_i is the list with five elements. OK, I'll use characters for the order just to keep things simple. So, in this case phi of L_i is going to be twice the cardinality of the set. So what we want to do is see which things are out of order. So here, I look at E and C are in this order, but C and E in that order. So, those are out of order. So, that counts as one of my elements, EC, and then, E and A, A and E.

OK, so those are out of order, and then ED, DE, out of order, and then EB, BE, those are out of order. And now, I go C, A, C, A. Those are in order, so it doesn't count. CD, CD, CB, CB, so, nothing with C. Then, A, D, A, D, those are in order, A, B, A, B, those are in order. So then, DB, BD, so BD. And that's the last one. So, that's my potential function, which is equal to, therefore, ten, because the cardinality of the set is five.

I have five inversions, OK, between the two lists. OK, so let's just check some properties of this potential function. The first one is notice that phi of L_i is greater than or equal to zero for all i. The number of inversions might be zero, but is never less than zero. OK, it's always at least zero. So, that's one of the properties that we normally have we are dealing with potential functions. And, the other thing is, well, what about phi of L0? Is that equal to zero? Well, it depends upon what list they start with. OK, so what's the initial ordering?

So, it's zero if they start with the same list. Then there are no inversions. But, they might start with different lists. We'll talk about different lists later on, but let's say for now that it's zero because they start with the same list. That seems like a fair comparison. OK, so we have this potential function now that's counting up, how different are these two lists? Intuitively, we're going to do is the more differences there are in the list, the more we are going to be able to have more stored up work than we can pay for it. That's the basic idea. So, the more that opt changes the list, so it's not the same as ours, in some sense the more we are going to be in a position as MTF to take advantage of that difference in delivering up work for us to do.

And we'll see how that plays out. So, let's first also make another observation. So, how much does phi change from one transpose? How much does phi change from one transpose? So, basically that's asking, if you do a transpose, what happens to the number of inversions? So, what happens when a transposing is done? What's going to happen to phi? What's going to happen to the number of inversions?

So, if I change, it is less than n minus one, yes, if n is sufficiently large, yes. OK, if I change, so you can think about it here. Suppose I switch two of these elements here. How much are things going to change? Yeah, it's basically one or minus one, OK, because a transpose creates or destroys one inversion. So, if you think about it, what if I change, for example, C and A, the relationship of C and A to everything else in the list is going to stay the same.

The only thing, possibly, that happens is that if they are in the same order when I transpose them, I've created an inversion. Or, if they were in the wrong order when I transpose them, now they're in the right order. So therefore, the change to the potential function is going to be plus or minus two because we're counting twice the number of inversions. OK, any questions about that? So, transposes don't change the potential very much, just by one. It either goes up by two or down by two, just by one inversion. So now, let's take a look at how these two algorithms operate.

OK, so what happens when op i accesses x in the two lists? What's going to be going on? To do that, let's define the following sets. Why do I keep doing that? OK, so we're going to look at the, when we access x, we are going to look at the two lists, and see what the relationship is, so, based on things that come before and after. So, I think a picture is very helpful to understand what's going on here. OK, so let's let, so here's L_i minus one, and we have our list, which I'll draw like this. And somewhere in there, we have x.

OK, and then we have L_i minus one star, which is opt's list, OK, and he's got x somewhere else, or she. OK, and so, what is this set? This is the set of Y that come before x. So, that basically sets A and B. OK, those things that come before x in both. And, some of them, the A's come before it in x, but come after it in, come before it in A, but come after it in B. OK, and similarly down here, what's this set?

That's A union C, good. And this one? Duh. Yeah, it better be C union D because I've got A union B over there, and I've got x. So that better be everything else. OK, and here is B union D. OK, so those are the four sets that we're going to care about. We're actually mostly going to care about these two sets. OK, and we also know something about the r here. The position of x is going to be the rank in L_i minus one of x.

And here, this is our star. It's just to the rank in L_i minus one star of x. So, we know what these ranks are. And what we're going to be interested in is, in fact, characterizing the rank in terms of the sets. OK, so what's the position of this? Well, the rank, we have that r is equal to the size of A. What's the size of B plus one? OK, and r star is equal to the size of A plus the size of C plus one.

So, let's take a look at what happens when these two algorithms do their thing. So, when the access to x occurs, we move x to the front of the list. OK, it goes right up to the front. So, how many inversions are created and destroyed? So, how many are created by this? That's probably a, how many inversions are created? How many inversions are created? So, we move x to the front. So what we are concerned about is that anything that was in one of these sets that came, where it's going to change in order versus down here. So, if I look in B, well, let's take a look at A. OK, so A, those are the things that are in the same order in both. So, everything that's in A, when I move x to the front, each thing in A is going to count for one more inversion.

Does everybody see that? So, I create a cardinality of A inversions. And, we are going to destroy, well, everything in B came before x in this list, and after x in this. But after we move x, they're in the right order. So, I'm going to destroy B inversions, cardinality of B inversions. OK, so that's what happens we operate with move to front. We destroy. We create A inversions and destroy B inversions, OK, by doing this movement.

OK, now, let's take a look at what opt does. So, each transpose, we don't know what opt does. He might move x this way or that way. We don't know. But each transpose, I opt, well, we're going to be interested in how many inversions it creates, and we already argued that it's going to create, at most, one inversion per transpose. So, he can go and create more inversions, OK? So, let me write it over here. Thus --

-- the change in potential is going to be, at most, twice, A minus B plus t_i. OK, so t_i, remember, was the number of transposes that opt does on the i'th step for the i'th operation. OK, so we're going to create the change in potential is, at most, twice this function. So, we are now going to look to see how we use this fact, and these two facts, this fact and this fact, OK, to show that opt can't be much better than MTF. OK, good.

The way we are going to do that is look at the amortized cost of the I'th operation. OK, what's MTF's amortized cost? OK, and then we'll make the argument, which is the one you always make that the amortized cost upper bound the true costs, OK? But the amortized cost is going to be easier to calculate. OK, so amortized cost is just C hat, actually, let me make sure I have lots of room here on the right, c_i hat, which is equal to the true cost plus the change in potential. OK, that's just the definition of amortized cost when given potential functions, OK?

So, what is the cost of operation i, OK, in this context here? OK, we accessed x there. What's the cost of operation i? Two times the rank of x, which is 2r. OK, so 2r, that part of it. OK, well, we have an upper bound on the change in potential. That's this. OK, so that's two times the cardinality of A minus cardinality of B plus t_i. OK, everybody with me? Yeah? OK, I see lots of nods. That's good. OK, that's equal to 2r plus two of size of A minus, OK, I want to plug in for B, and it turns out very nicely. I have an equation involving A, B, and r. So, I get rid of the variable size of B by just plugging that in.

OK, and so what do I plug in here? What's B equal to? Yeah, r minus size of A minus one. I wrote it the other way. OK, and then plus t_i. OK, and this is since r is A plus B plus one. OK, everybody with me still? I'm just doing algebra. We've got to make sure we do the algebra right. OK, so that's equal to, let's just multiply all this out now and get 2r plus, I have 2A here minus A. So, that's 4A. And then, two times minus r is minus 2r. Two times minus one is minus two. Oh, but it's minus-minus two, so it's plus two. OK, and then I have 2t_i. So, that's just algebra. OK, so that's not bad. We've just got rid of another variable.

What variable did we get rid of? r. It didn't matter what the rank was as long as I knew what the number of inversions was here. OK, so that's now equal to 4A plus two plus 2t_i. And, that's less than or equal to, I claim, four times r star plus t_i using our other fact. Since r star is equal to the size of A plus the size of C, plus one, then that's greater than or equal to the size of A plus one.

OK, if I look at this, I'm basically looking at A. The fact that A, what did I do here? If r star is greater than or equal to A plus one, right, so therefore, A plus one, good. Yeah, so this is basically less than or equal to 4A plus four, which is four times A plus one. I probably should have put in another algebra step here, OK, because if I can't verify it like this, then I get nervous. This is basically, at most, 4A plus four. That's four times A plus one, and A plus one is less than or equal to r star. And then, 2t_i is, at most, 4TI.

So, I've got this. Does everybody see where that came from? But what is r star plus t_i? What is r star plus t_i? What is it? It's c_i star. That's just c_i star. So, the amortized cost of i'th operation is, at most, four times opt's cost. OK, that's pretty remarkable. OK, so amortized cost of the i'th operation is just four times opt's cost. Now, of course, we have to now go through and analyze the total cost. But this is now the routine way that we analyze things with a potential function.

So, the costs of MTF of S is just the summation of the individual costs, OK, by definition. And that is just the sum, i equals one, to S of the amortized cost plus, minus the change in potential. OK, did I do this right? No, I put the parentheses in the wrong place. Now I've got it right. Good. I just missed a parenthesis. OK, so this is, so in the past what I did was I expressed the amortized cost as being equal to c_i plus the change in potential. I'm just throwing these two terms over to the other side and saying, what's the true cost in terms of the amortized cost? OK, so I get c hat of i plus phi sub L_i minus one minus phi of L_i, OK, by making that substitution.

OK, that's less than or equal to since this is linear. Well, I know what the sum of the amortized cost is. It's, at most, 4c_i star. So, the sum of them is, at most, to that sum, I equals one to S of 4c_i star. And then, as happens in all these things, you get a telescope with these terms. Every term is added in once and subtracted out once, except for the ones at the limit. So, I get plus phi of L_0 minus phi of L sub

cardinality of S. And now, this term is zero. And this term is greater than or equal to zero.

OK, so therefore this whole thing is less than or equal to, well, what's that? That's just four times opt's cost. And so, we're four competitive. OK, this is amazing, I think. It's not that hard, OK, but it's quite amazing that just by doing a simple heuristic, you're nearly as good as any omniscient algorithm could possibly be. OK, you're nearly as good. And, in fact, in practice, this is a great heuristic. So, if ever you have things like a hash table that you're actually seeing by chaining, OK, often it's the case that if when you access the elements, you're just bringing them up to the front of the list if it's an unsorted list that you've put them into, just bring them up to the front.

You can easily save 30 to 40% in run time for the accessing to the hash table because you will be much more likely to find the elements inside. Of course, it depends on the distribution and so forth, for empirical matters, but the point is that you are not going to be too far off from the ordering that an optimal algorithm would do, optimal off-line algorithm: I mean, amazing. OK: optimal off-line. Now, it turns out that in the reading that we assigned, so, we assigned you Sleator and Tarjan's original paper.

In that reading, they actually have a slightly different model where they count transposes that move in excess to element x towards the front of the list as free. OK, so, and this basically models, so here's the idea is if I actually have a linked list, and when I chase down, once I find x, I can actually move x up to the front with just a constant number of pointer operations to splice it out and put it up to the front.

I don't actually have to transpose all way back down. OK, so that's kind of the model that they use, which is a more realistic model. OK, I presented this argument because it's a little bit simpler. OK, and the model is a little bit simpler. But in our model, they have, when you access something, you want to bring it up to the front, or anything that you happen to go across during that time, you could bring up to the front essentially for free.

This model is the splicing in, splicing x in and out of L in constant time. Then, MTF is, it turns out, too competitive. It's within a factor of two of optimal, OK, if you use that. And that's actually a good exercise to work through. You could also go read about it in the reading to understand this better, to look to see where you would use those things. You have to have another term representing the number of, quote, "free" transposes. But it turns out that all the math works out pretty much the same. OK, let's see, another thing I promised you is, what if, to look at the case, what if they don't start with the same lists?

OK, what if the two lists are different when they start? Then, the potential function at the beginning might be as big as what? How big are the potential function start out as if the lists are different? So, suppose we're starting out, you have a list, and opt says, OK, I'm going to start out by ordering my list according to the sequence that I want to use, OK, and MTF orders it according to the sequence it must use. What list is opt going to start out with as an adversary?

Yeah, it's going to pick the reverse of what ever MTF starts out with, right, because then, if he picks the reverse, what's the number of inversions? It's how many inversions in a reverse ordered list? Yeah, n choose two, OK. Is it n choose two, or n

minus one choose two? n minus one choose two, OK, inversions that you get because it's basically a triangular number when you add them up. But in any case, it's order n^2, worst case.

So, what does that do to our analysis here? It says that the cost of MTF of S is going to be, well, this is no longer zero. This is now n^2. OK, so we get that costs of MTF of S is, at most, four times opt's thing plus order n^2, OK? And, if we look at the definition, did we erase it already? OK, this is still for competitive, OK, since n^2 is constant as the size of S goes to infinity. This is, once again, sort of your notion of, what does it mean to be a constant? OK, so as the size of the list gets bigger, all we're doing is accessing whatever that number, n, is of elements. That number doesn't grow with the problem size, OK, even if it starts out as some variable number, n.

OK, it doesn't grow with the problem size. We still end up being competitive. This is just the k that was in that definition of competitiveness. OK, any questions? Yeah? Well, so you could change the cost model a little bit. Yeah. And that's a good one to work out. But if you say the cost of transposing, so, the cost of transposing is probably moving two pointers, approximately. No, one, three pointers. So, suppose that the cost of, wow, that's a good exercise, OK? Suppose the cost was three times to do a transpose, was three times the cost of doing an access, of following a pointer.

OK, how would that change the number here? OK, good exercise, great exercise. OK, hmm, good final question. OK, yes, it will affect the constant here just as when we do the free transpose, when we move things towards the front, that we consider those as free, OK. Those operations end up reducing the constant as well. OK, but the point is that this constant is independent of the constant having to do with the number of elements in the list. So that's a different constant. So, this is a constant. OK, and so as with a lot of these things, there's two things.

One is, there's the theory. So, theory here backs up practice. OK, those practitioners knew what they were doing, OK, without knowing what they were doing. OK, so that's really good. OK, and we have a deeper understanding that's led to, as I say, many algorithms for things like, the important ones are like paging. So, what's the comment page replacement policy that people study, people have at most operating systems? Who's done 6.033 or something?

Yeah, it's Least Recently Used, LRU. People have heard of that, OK. So, you can analyze LRU competitive, and show that LRU is actually competitive with optimal page replacement under certain assumptions. OK, and there are also other things. Like, people do random replacement algorithms, and there are a whole bunch of other kinds of things that can be analyzed with the competitive analysis framework. OK, so it's very cool stuff. And, we are going to see more in recitation on Friday, see a couple of other really good problems that are maybe a little bit easier than this one, OK, definitely easier than this one.

OK, they give you hopefully some more intuition about competitive analysis. I also want to warn you about next week's problem set. So, next week's problem set has a programming assignment on it. OK, and the programming assignment is mandatory, meaning, well, all the problem sets are mandatory as you know, but if you decide not to do a problem there's a little bit of a penalty and then the penalties scale dramatically as you stop doing problem sets. But this one is mandatory-mandatory.

OK, you don't pass the class. You'll get an incomplete if you do not do this programming assignment. Now, I know that some people are less practiced with programming. And so, what I encourage you to do over the weekend is spent a few minutes and work on your programming skills if you're not up to snuff in programming. It's not going to be a long assignment, but if you don't know how to read a file and write out a file, and be able to write a dozen lines of code, OK, if you are weak on that, this weekend would be a good idea to practice reading in a text file.

It's going to be a text file. Read it in a text file, decent manipulations, write out a text file, OK? So, I don't want people to get caught with this being mandatory and that not have time to finish it because they are busy trying to learn how to program in short order. I know some people take this course without quite getting all the programming prerequisites. Here's where you need it. Question? No language limitations. Pick your language. The answer will be written in, I think, Java, and Eric has graciously volunteered to use Python for his solution to this problem. We'll see whether he lives up to that promise. You did already? OK, and George wrote the Java solution. And so, C is fine. Matlab is fine. OK, what else is fine? Anything is fine.

Scheme is fine. Scheme is fine. Scheme is great. OK, so any such things will be just fine. So, we don't care what language you program in, but you will have to do programming to solve this problem. OK, so thanks very much. See you next week.