6.046J Introduction to Algorithms, Fall 2005
Transcript – Lecture 13

OK, good morning. So today we are going to, as I mentioned last week, we've started the part of the course where we are doing more things having to do with design than purely analysis. Today, we're actually going to do analysis, but it's the type of analysis that leads to really interesting design issues. And we're going to follow it up on Wednesday with an application of the methods we're going to learn today with a really interesting and practical problem. So we're talking today about amortized analysis.

And I want to motivate this topic by asking the question, how large should a hash table be? So, how large should a hash table be? Any suggestions? You have got to make a hash table, how big should I make it? Let's say it's a simple hash table, resolving collisions with chaining. How big should it be? Twice as big as you need: OK, how big would that be? So, twice the number of elements, for example, OK. As I increase the size of a hash table, what happens to the search time?

What happens to search time as I increase the size of the hash table? Yeah, but what does it, in general, do you? It decreases, right? OK, the bigger I make it, in fact, if I make it sufficiently large, then I essentially get a direct access table, and everything is worst-case, order one time. So in some sense, we'll get back to your answer in a minute. We should make it as large as possible, OK, so that the searching is cheap. The flipside of that is what? It takes a lot of space so I should make it as small as possible so as not to waste space.

So I want it big, and the happy medium, as we've discussed in our analysis, is to make it order n size for n items, OK, because making it larger than order n, the payoff in search time is not worth the extra amount of space that you are paying. OK, or at least you can view it that way. OK, however, this begs the question, which is, how do I make it, if I start out with a hash table, and I don't know how many elements are going to be hashed into it?

OK, how big should I make it? OK, so what if we don't know -- -- in advance? OK, what if we don't know n? OK, so the solution to this problem turns out it's fairly elegant? It's a strategy called dynamic tables. OK, and the idea is that whenever the table gets too many elements in it, gets too full, OK, so the idea is -- -- OK, and we say that says the table overflows, OK, we grow it and make a bigger table. So, for hashing, although there's going to be no point at which you could say that it overflows in the sense that it wouldn't be functional at least if it was done with chaining. There would be, by the way, if you were doing it with open addressing. But let's say with chaining, when it gets too big, say, as many elements as the size of the table, what we do is we grow the table.

So, the way we do that as we allocate using, in a language like C, it's called Malloc, or in a language like Java called New, a larger table. So, we create a larger table. We move the items from the old table to the new. And then, we free the old table. So, let's do an example. So, let's say I have, over here, a table of size one, and it's

empty to begin with. And I do an insert. So what I do is stick it in the table. It fits. OK, so here, I'm not going to do it with hashing. I'm just going to do it as if I just had a table that I was filling up with elements to abstract the problem. But it would work with hashing.

It would work with any kind of fixed size data structure. I insert again, oops, doesn't fit. I get an overflow. OK, so what I do is I create a new, actually, I'm going to need a little more space than that. I create a new table of size two, doubling the size. And, I copy the old value into the new. I freed this one, and now I can insert item two. So, I do it again. I get another overflow. So now, I make a table of size four. I copied these guys in, and then I insert my number three.

I do insert here. I do five. I guess I should be using ditto marks. That would be a lot smarter. Whoops, what am I doing? I overflow. And now, I make one of size eight, OK, copy these over, and now I can insert five. And I can do six, seven, etc., OK? Does everybody understand the basic idea? So, whenever I overflow, I'm going to create a table of twice the size. OK, so let's do a quick analysis of this.

So, we have a sequence of n insertion operations. OK, what is the worst-case cost of one insert operation? What's the worst case for any one of these? Yeah, it's order n, whatever the overhead is of copying; if we counted it as one, it would be basically n or n plus one because we've got to copy all those. OK, so it's order n. So therefore, if I have n of those, so the worst-case cost of n inserts is equal to n times order n, which is order n^2.

Any questions? Does that make sense? Raise hands. Yeah, not all of them can be worst-case, good. And in fact, this is totally wrong analysis. Just because one can be worst-case order n doesn't mean n are necessarily order n. OK, so this is totally wrong analysis. n inserts, in fact, take order n time in the worst case. OK, it doesn't take order n^2. So, the analysis is correct up to the point where we set the worst-case of one insert was order n. Therefore, that's the wrong step. OK, whenever you see bugs in proofs, you want to know, which step is the one that failed so you can make sure that you don't have a confusion there?

So, let's do the proper analysis, OK? So let's let c_i be the cost of the i'th insert. OK, so that's equal to i, if i minus one is an exact power of two. And it's one otherwise. OK, so as I was going through here, it was only when I inserted something where the previous one had been the exact power of two, because that was my table size. That's when I got the overflow and had to do all that copying. And otherwise, the cost, for example, for inserting six, was just one. I just inserted it. Does everybody see that? So, let's actually make a little table here so we can see this a little bit more clearly. OK, so here's i, and in the size of the table at step i, and the cost at step i.

OK, so let's see, the size of i, let's see, at step one it was one. At step two, it was two. And at step three, that's when, to get three in the table, we had to double the size here. So, this is four, and four, it fit. And then five, it had to bump up to eight. And then, six, it was eight. Seven, it was eight. Eight, it was eight. And nine, it bumps up to 16, 16, etc. So that's the size. And let's take a look at what the cost was. So, the cost here was one, OK, to insert one. The cost here was, I had to copy one, and then insert one.

So, the cost was two. Here, I had to copy two and insert one. So, the cost was three. Here, I had to just insert one. So, the cost was one. Here, I had to copy four and

insert one. So, the cost was five. Excuse me? I think it is. Yeah, see, it's i cost. The cost for five is i, OK, is five if this is a power of two. OK, one, one, and now we paid nine, and then one again. So that's the cost we are paying. It's a little bit easier to see what the costs are if I break them down. OK, so let's just redraw this as two values because there is always the cost for inserting the one thing that I want to insert.

And now, the residual amount that I have to pay is I have to pay one here. I've got to pay two additional, four additional, eight additional. That makes the pattern a little bit easier to see. OK, this is the cost of copying versus the cost of just doing the actual insert, OK? Now, if you're taking notes, leave some space here because I'm going to come back to this table later. OK, so leave a little bit of space because I'm going to come back and add some more things at there at a later time.

OK, so, I can then just add up the cost of n inserts. That's just the sum, I equals one to n of c_i, which is equal to, well, by this analysis it is essentially n because that's what this thing adds up to plus I just have to add the powers of two up to but not exceeding whatever my n was. So, if I do my algebra properly there, that's up to the floor of log n minus one, OK, of two to the J. So, I'm just adding up all the powers of two up to that aren't going to exceed my n.

And, this is what type of series? That's geometric. That's geometric, so it is bounded by its largest term. Its largest term is two to the ceiling; it's dominated by its largest term, two to the ceiling of log n minus one, which is, at most, n. OK, and then all the other terms at up to, at most, n. So this is actually less than or equal to 3n, which is order n as we want it to show. OK, that's algebra.

OK, so, thus, the average cost per insert is theta of n over n, which is theta one. So, the average cost of an insert is order one, which is what we would like it to be especially if we're building hash tables. Even though sometimes you have to pay a big price with amortized over the previous insertions that we've done, so that the overall cost of n operations is order n. And that's the notion of amortized analysis, OK, that if I look at a sequence of operations, I can spread the cost out over a whole bunch of operations, so that the average cost is order n.

So, if we sort of summarize that, OK, OK, with basically an amortized analysis, we analyze a sequence of operations to show that the average cost per operation is small, even though one operation, or several, even, may be expensive. OK, there's no probability. Even though we're doing it with averages, there's no probability going on. OK, what you do probability, and you are looking at means, there's averages. OK, here's average, but there's no probability going on. It's average performance in the worst case because n operations take me a constant amount of time per operation in the worst case. n operations take me order n time.

OK, each operation takes order one time, OK, but it's amortized over the n operations, OK? Yeah, question? Yes. Yes, yes, you can mix, but you don't have to. Yeah, but the point is that the basic amortized analysis is actually saying something very strong. It's giving you worst-case bounds, but over a sequence as opposed to looking at each individual element of the sequence. Now, there are three types of amortized arguments that appear in the literature.

Maybe there are more. At one point, there were two. And then, a third one was developed. So, maybe there's a fourth. The first one is an aggregate, what's called

an aggregate analysis. And this is what we just saw, OK, where basically you just analyze, what do the n operations take? OK, and then we're going to see two more today. One is called an accounting argument, and the other is a potential argument. These two are more precise because they allocate specific amortized costs to each operation.

So, one of the things about the aggregate analysis is that you can't really say what the amortized cost of a single operation is easily. You can in this case. You can say it's order one, OK? But, in the accounting and potential arguments, it gives you a much more precise way of characterizing what an amortized cost of a particular operation is. So, let's pitch in and look at the accounting method as our first method. So, these we're going to go through the exact same example. In some sense, this example, the easiest argument to make is the aggregate analysis. OK, so we're going get into arguments that, in some sense, see more complicated.

But it turns out that these methods are more powerful in many circumstances. OK, and so I want to do it in a simple situation where you have some sort of appreciation of the fact that you can look at any particular problem and approach it from different ways. OK, so the accounting method is putting yourself in the position of a financial accountant. So what you do is we are going to charge the i'th operation a fictitious amortized cost.

We'll call it c hat sub i, where we are going to use the abstraction that $1 pays for one unit of work. There's Time manipulating the data structure or whatever. OK, so the idea is you charge the cost. You say, this operation will cost you $5 or whatever. OK, and that phi is consumed to perform the operation, but there may be some unused part. So, if there's any unused amount, it's going to be stored in the bank for use by later operations.

So the idea is that if the phi that is being paid, the c_i hat phi, isn't sufficient to pay for performing the operation, then you take money out of the bank to pay for it. OK, and so you don't get arrested, what's the property that you've got to have? You've got to have the bank balance. What about the bank balance? What mathematical fact has to hold the bank balance? Yeah, it better be greater than or equal to zero, right? Most people are familiar with that.

So, the bank balance must not go negative. In other words, the amortized costs minus the costs of the operations up to that point have to always be enough to pay for all the operations that you're doing. Otherwise, you're borrowing on the future. In amortized analysis, we don't borrow on the future, at least not on the simple ones that we are doing here. OK, so that means we must have that the sum, I equals one to n of c_i, the true costs, therefore, if the balance is not going to ever go negative, must be bounded above by the amortized costs for all n.

OK, for the bank balance not to go negative, if I add up the true costs, it's got to be the case that I can always pay for them. This is what I'm charging. This is what it actually costs me. So, it better be the case that whatever I've actually had to pay to operate on that data structure, that's what this is, better be covered by the amount that I've been charging people for the use of that data structure up to that point.

And that's got to be true for all n. But notice that this now gives me a way of charging a particular operation a certain amount. So, the total amortized costs provide an upper bound on the total true costs. Total amortized costs are an upper

bound on the true costs. Any question about this? That we'll do the example of the dynamic table using this methodology. OK, so, back to the dynamic table.

OK, so what we're going to do in this case is we're going to charge an amortized cost of $3 for the i'th insert for all i. And the idea is that $1 is going to pay for an immediate insert, and $2 is going to be stored for doubling the table. And, it needs to be expanded. When the table doubles, of the stored dollars, we'll use one dollar to move a recent item, I'll call it, and one dollar we'll move an old item.

So, let's do the example. So, imagine that I'm in this situation where I have a table of size eight, and I just doubled my table. So I have four items of the table. What I'm going to do is have no dollars in my table. So, along comes an insertion of item number five. I charge $3 for it. $1 lets me put the item in the table, and I have $2 left over. So let me store those $2 in the slot corresponding to where that item is.

Now, item six comes in. Once again, $1, charge $3, $1 is paid for the insert, $2 left over, I'm going to play $2. Let me put it down there, and so forth. The next one comes in, $2, $2 leftover, and now the ninth item comes in. So, I double the size of my table. OK, and now I copy all of these guys and to all of these here. And what happens? Look at that: I've got $8, and I've got eight items that have to be copied. Perfect. OK, so one of these dollars pays for one of the ones that was inserted in the last round, and one of them pays for an old one.

OK, and so, I copy them in, and now, none of those guys have any money. And the ninth guy comes in: he has $2 left over. And then, we keep going on. OK, so you see that by that argument, if I charge everybody $3, OK, I can always handle all of the table doubling, the charges for the table doubling because the inductive invariant that I've maintained is that after it doubles, there's nothing in the bank account.

And now, I put in $2. Well then, I can pay, and I'm now left in the same situation. OK, and it's the case that the bank balance never goes negative. So that's a really important invariant to verify. And so, therefore, the sum of the true costs, or the amortized costs, upper bound the sum of the true costs. And, since the sum of the amortized cost, here, is, if I go i equals one to n, OK, this is 3n.

So, the point is, now I bounded the sum of the true costs by 3n. OK, so let's go back to this table here, and look to see what happens, OK, if I put in c_i hat, and the bank balance. OK, so in fact, so the first thing I do is insert; I charge $3, right, and I do an insert. How much do I have left? I'm going to have $2. It turns out I'm actually going to charge $2 and have only $1 left. OK, so I'm actually going to under charge the first guy. I'm going to show you that it works if I charge everybody $3. Except the first guy: I charge $2. I can actually save a little bit on number one. OK, that for this guy I'm going to charge $3.

OK, what's the size of my bank balance when I'm done? Well, I have to copy one guy. He's all paid for, so I have $2 left. OK, people with me? OK, the next guy: I charge $3. Actually, I'm going to charge all these guys $3. OK, so here now I basically get to, I've got a table of size four. So, I basically have, I have to copy, oh, when I insert the third guy, I've got to copy two guys. That'll use up that, so I'll have only $2 left in the table after I've inserted him. OK, now I insert the fourth guy, OK, and that's a good one because now I've built up a balance here of $4 because I didn't have to copy anybody.

OK, now I insert the fifth guy. I've got to copy four items. So that expends that balance. I have, then, two left. OK, and then here basically I add two to it. And then at this point, I use it all up and go back to two, etc. OK, so you see one of the things I want you to notice is I could have charged three here. And then I would've had an extra dollar lying around throughout here. It wouldn't have mattered. It still would be upper bounded by 3n. OK, so the idea is that different schemes for charging amortized costs can work.

They don't all have to be the same. It's not like when you do in amortized analysis that there is one scheme that will work. I could have charged $4 to everybody. And it would have worked. But it turns out, I couldn't have charged two dollars for everybody. If I charged $2 for everybody, my balance would go negative, OK? My balance would go negative, but I can charge three dollars, and that will work.

OK, four, five, six, I could charge that. The bound that I would get would be simply a looser bound. Instead of it being less than or equal to 3n, it would be less than or equal to 4n or 5n, or what have you. But if I tried to do 2n, it wouldn't have worked because I wouldn't have enough money left to copy everything. What would happen is I would have only $1 in this, and then when it came time to table double, I would need to copy eight guys. And I'd only have built up a bank account of $4, sorry, if I charged $2 and had $1 left over.

OK, so to actually make these things work out, you have to play a little bit, OK, see what works, see what doesn't work. OK, no algorithmic formulas for algorithm design. OK, good. In the book, you can read about table contraction. What happens when you start deleting elements? Now you want to make the table be smaller. Now, you have to be very careful because unless you put, who remembers from physics, hysteresis? Vaguely? A couple people? OK, you have to worry about hysteresis. OK, if you're not careful, if whenever it gets to be less than a power of two, you go in the half, you can find that you're thrashing. So, you need to make it so that there is some memory in the system so that you only collapse after you've done a sufficient number of deletions, OK, and so forth.

And the book has analysis of the more general case. OK, so any questions about the accounting method? Accounting method is really very cute, OK, very cute. And, it's the one most students prefer to do, OK? They usually hate the next one until they learn it. Once they learn it, they say, ooh, that's cool. OK, but to start out with, it takes a little bit more intestinal fortitude, OK? But it's amazing. Good potential arguments are really sweet. And we are going to see one next time, so you'll definitely want to review and make sure you understand it before Wednesday's lecture because Wednesday's lecture, we're going to assume we understand potential method. OK, so let's do, enough advertisement.

I think the potential method is one of the beautiful results in algorithmic analysis, OK, just beautiful result, beautiful set of techniques. OK, and it's also, just in terms of, I mean, what do you aspire: to be a bookkeeper or to be a physicist? OK, so, the idea is we don't want to be bankers. We want to be physicists. And so, this bank account, we are going to say about the potential energy of the dynamics that that we are analyzing. OK, because, why? It delivers up work just like a spring does, for example, OK, when you study potential energy, or putting something up high and having gravity pull it down. We convert dynamic to potential, and that's exactly what we're going to be doing here, and it's similar mathematics except that in our case it turns out to be discrete mathematics rather than continuous math for most of it.

So here's the framework for the potential method. So, we start with some data structure, D_0, and operation i transforms D_(i - 1) into D_i. So, we view the operation on the data structure as a mapping, mapping one data structure to another data structure, the one from before to the one after. OK, already it's nicely mathematical. OK, and of course, the costs of operation i remains at c_i. And, now what we are going to do is define the potential function, phi, which maps the set of data structures into the reals.

So, associated with every data structure, now, is a potential, OK, a real-valued potential, often integer potential such that phi of D_0 is equal to zero. So, the initial potential is zero, and phi of D_i is greater than or equal to zero for all i. So, potential can never be nonnegative, just like the bank account because the potential is essentially representing the bank account, if you will, in the accounting method. OK, so we always want the potential to be nonnegative.

Now, actually, there are times where you use potential functions where you violate both of these properties. There's some really interesting potential arguments which don't violate like these, but for the simple ones we're going to be doing in this class, we'll just assume that these tend to be true. OK, but we will actually see some times where phi of D_0 isn't zero, it doesn't matter. OK, but generally this is what we are going to assume in the type of potential function argument that we are going to be doing.

OK, so I just want to let you know that there are bigger, there is a bigger space of potential function arguments than the one that I'm showing you here. OK, so then, under this circumstance, we define the amortized cost c_i hat with respect to phi as, and this is one of these formulas that if you can't remember it, definitely put it down on your crib sheet for the final. OK, so c_i hat is equal to c_i plus phi of D_i minus phi of D_i minus one.

OK, so this is the change in potential difference. And, let's call it delta phi i for shorthand. OK, and let's see what it means to have -- -- in the different circumstances. So, if delta of phi i is greater than zero, OK, so if this is greater than zero, then what's the relationship between c_i hat and c_i? This is greater than zero. Yeah, c_i hat is then greater than c_i, OK? Then, c_i hat is greater than c_i. And what does that mean?

That means when I do operation I, I charged more than it cost me to do the operation. So, the extra amount that I charged beyond what I actually used is being put into the bank, OK, is being stored as potential energy. So, op I stores work in the data structure for later. Similarly, if delta phi i is less than zero, then c_i hat is less than c_i. And so, the data structure delivers up work --

-- to help pay for op I, OK, for operation I. So, if it's less than zero, that means that my change in potential, that means my bank account went down as a result, and so therefore, what happens was the data structure provided work to be done in order, because the true cost was bigger than the amortized cost. So, if you think about it, the difference between looking at it from the potential function point of view versus the accounting point of view, the accounting point of view, you sort of say, here is what my amortized cost will be. Now let me analyze my bank account, make sure it never went negative. In some sense, in the potential function argument, you're saying, here's what my bank account is all the time.

Now let me analyze what the amortized costs are. So, that's sort of the difference in approaches. One is you are sort of specifying the bank account. The other, you're specifying the amortized costs. So, we look at the, why is it that this is a reasonable way to proceed? Well, let's look at the total amortized cost of n operations. OK, that's just the sum, i equals one to n of c_i hat. That's the total amortized cost. And that's equal to, but substitution, just substitute c_i hat for this formula.

OK, so that's c_i plus phi of D_i minus phi of D_i minus one. OK, and that's equal to c_i. And now, what happens when I sum up these terms? What happens when some of these terms? What's the mathematical term we use? It telescopes. OK, every term on the left is added in once when it's I, and subtract it out when it's I minus one, except for the first and last terms. The term for n is only added in, and the term for zero is only subtracted out. OK, so that's because it telescopes. OK, so this term is what? What property do we know of this?

It's greater than or equal to zero. And this one equals zero. So, therefore, this is greater than or equal to c_i. And thus, the amortized costs are an upper bound on the true costs, which is what we want. Some of the amortized costs is an upper bound up on the sum of the true costs. OK, but here, the way that we define the amortized costs was by first defining the potential function. OK, so the potential function is sort of, as I said, the difference between the accounting and the potential method is, do you specify the bank account or do you specify the cost? OK, do you specify the potential energy at any point, or do you specify the cost at any point?

OK, but in any case, you get, this bound, also this math is nicer math. I like telescopes. OK, so the amortized costs upper bound the true costs. OK, let's do table doubling over here. So, to analyze this, we have to define our potential. OK, if anybody can guess this off the top of their head, they're better than I am. I struggled with this for probably easily a couple hours to get it right, OK, because I'm not too smart. OK, that's a potential function I'm going to use, OK, 2i minus two to the ceiling of log i.

And, we're going to assume that two to the ceiling of log of zero is equal to zero, because that's what it is in the limit. For log of zero, this becomes minus infinity, so, two to the minus infinity is zero. So, that's just going to be a mathematical convenience. Assume that. OK, so where did I get this from? I played around. I looked at that sequence that I have erased and I said, OK, because reversing, there are some problems for which defining a potential function is fairly easy. But, defining the amortized costs is hard, OK, to define the accounting. So, for this one, the accounting method is, I would say, an easier method to use. However, I'm going to show you that you still can do it with potential method if you come up with the right potential.

So, intuitively, this is basically what's left in the bank account at the I'th operation because I've put in 2i things into the bank, and I've subtracted out this many, essentially, from table doublings, OK, up to that point, OK? So, first let's observe, what is phi of D_0? Zero. So, that's good. And, the phi of D_i is greater than or equal to zero. Why is that? Why is that? So, what's the biggest that ceiling of log i could be? Ceiling of log i is either log i or log i quantity plus one, OK?

So, the biggest it is, is log i plus one. If it's log i plus one, two to the log i plus one, it's just 2i. That's the biggest it could be, right? So, two, the log of i, plus one, is

just, let's do it the other way, is i times two, OK, i for that part, two for that part. OK, so that the biggest it could be. OK, or it's just log of i. So, either this is going to be 2i minus i or 2i minus 2i. In either case, it's bigger than zero, OK?

So, those are the two properties I need for this to be a well-defined potential function. Now, that doesn't say that the amortized costs are going to be satisfied the property that things are going to be cheap, OK, that I'm going to be able to do my analysis and get the kind of bounds I want. But, it sets up to say, yes, I've satisfied the syntax of having a proper potential function. So, let's just do a quick example here just to see what this means. So, imagine that I am in the situation where I have eight, did I do that right, OK, yeah, eight slots, and say six of them are full. So then, phi by this is going to be 2i. That's two times six minus two to the 2i.

What's that? Sorry, minus two to the ceiling of log i. So, i is six, right, so log of i is log of six. The ceiling of it is three. So, that's minus 2^3, which is eight. So, that's 12 minus eight. That's four. And if you think about this in the accounting method, these would be zeros, and these would be twos, right, for the accounting method if we do the same thing, because this is halfway through, right, all zeros, and that we add two for each one that we're going in. So, I function is, in fact, telling me what the actual cost is. OK, everybody with me? OK, so that's what we mean by this particular potential function.

OK, so now let's add up the amortized cost of the i'th insert. OK, so that's the amortized cost of the i'th insert, just by definition. OK, and now that's equal to, well, what is c_i? Do we still have that written down somewhere, or have we erased that at this point? I think we erased it. Well, we can write it down again. It is i if i minus one is an exact power of two. And, it's one otherwise. That's c_i. That's this term, plus, and now phi of D_i: so, what is that?

phi of D_i is this business, 2i minus two ceiling of log i minus 2i minus one minus two ceiling of log of I minus one. OK, so that's the amortized cost. That's a nice, pretty formula, right? OK, let's hope it simplifies a little. OK, so that's equal to, well, we have the i and the one here, if, etc., that business, plus, OK, well, we have some things we can cancel here, right? So here, we have 2i minus 2i. That cancels. And then we have what's left over here is a minus two. So, that's a plus two. And now, I have minus this term plus this term.

That's a lot prettier. OK, it's still a mess. We've got to case analysis. Why is it suggestive of a case analysis? We have a case. OK, so let's do a case analysis. OK, so case one, I minus one is an exact power of two. So then, c_i hat is equal to, well, c_i is now just i. That's that case. And then we have the rest there, plus two, minus two, ceiling of log i minus two ceiling of log of i minus one.

OK, and that's equal to i plus two. Well, let's see. If I minus one is an exact power of two, what is this term? i minus one is an exact power of two. Plus, thank you, sorry about that. It's good to have students, let me say, because, boy, my math is so bad. This is actually why I end up being a pretty good theoretician is because I don't ever trust what I've written down. And so, I write it down in a way that I can verify it because otherwise I just am not smart enough to carry through five equations in a row and expect that everyone is going to be transformed appropriately.

OK, so you write it down. So I always write it down so I can verify it. And that fortunately has the side benefit that other people can understand what I've done as

well. OK, so what is this one? This is two to the log of i minus one because the ceiling, if this is an exact power of two, right, then ceiling of log of i minus one is just log of i minus one. So, this is two to the log of i minus one, which is i minus one, right? Yeah? OK. OK, if it's an exact power of two, then the log is an integer, right?

So, taking the ceiling, it doesn't matter, get rid of the ceiling. OK, this one, however, is not an exact power of two. But what is it? It's just one more than this guy. We know that i minus one is not an exact power of two, so it's going to be the next bigger one. OK, so that means this is what? So, it's going to be, how do these two compare? How much bigger is this one than this one? It's twice the size. We know what this one is.

OK, or it can reason it from first principles. This is going to be the log of i minus one plus one. OK, and so then you can reduce it to this. So, you've got to think about those floors and ceilings, right? It's like, what's happening in the round off there? OK, so now we can simplify this. OK, so what do we have here? We have, OK, so if I multiply this through, I have an i plus two minus 2i plus two plus i minus one. I know a lot of you, probably 90% of you will do this step. You will go directly from this step to the last step. And that's where 30% of you, or some number, will get it wrong.

OK, so let me encourage you to do that step. OK, it's easier to find your bugs if you take it slow. Actually, taking it slow is faster in the long run. It's hard to teach young stallions, or phillies, or whatever, OK? Just take it easy. Just patience, just do it slow, get it right. It's actually faster. OK, everybody knows the tortoise and hare story. Yeah, yeah, yeah, OK, but nobody believes it. OK, so now here we have 2i here, an i here, and an i here. And then, that leaves us with two plus two minus one, equals three. Awesome, awesome. OK, amortized cost is three when i minus one is an exact power of two.

OK, case two. OK, i minus one is not an exact power of two. So then we have c_i hat is equal to, now instead of i it's one plus, and then the two minus two to the ceiling of log i plus two to the ceiling of log of i minus one. OK, now what can somebody tell me about these two terms in the case where i minus one is not an exact power of two? What are they? Equal. Why is that? Yeah, the ceiling is going to do the same thing to both, going to take it up to the same integer. So these two things are equal, which means this is equal to three.

OK, so therefore, n inserts, OK, so now I say, oh, the amortized cost is three for every operation for every insert. So therefore, n inserts costs, well, the amortized cost of each is three. So n of them, the amortized cost is 3n. That's an upper bound on the worst-case true costs. So, n inserts costs order n in the worst case. OK, there is a bug in this analysis. It's a minor bug. It's the one I pointed out before the first insert has amortized cost of two, and not three.

I didn't actually deal with that one carefully enough. OK, so that's an exercise to just go and look to see where it is that that happens and how you show that, in fact, the amortized cost of the first one is two, OK, where that shows up. OK, so to summarize, actually let me summarize over here, conclusions about amortized analysis. So amortized costs provide a clean abstraction for data structure performance.

So, what I can tell somebody, so suppose I built a dynamic table, for example, OK. It's easier to say, in terms of your own performance modeling, it costs a constant

amount of time for each insert. As long as you don't care about real-time behavior, but only the aggregate behavior, that's a great abstraction for the performance Rather than saying it's got that complicated thing which sometimes cost you a lot, how do they reason about that? But you could say every operation costs me order one, that's really simple. But they have to understand, it's order one in an amortized sense, OK. So, if they do have a real-time constraint to make, amortized doesn't cut it.

OK, but for many problems, it's perfectly good. It lets me explain it rather simply. OK, we will see some other data structures that have amortized costs where different operations have different amortized costs. And the nice thing about that is I just add up, what's the cost of all my different operations, OK, where there is a different cost for each operation? Some will be log n. Some will be order one, or whatever. Add them up: that's an upper bound on the true costs. OK: tremendous simplification in abstracting and reasoning about those complicated data structures. OK, now, so this is probably, this is huge. OK, abstraction, you know, computer science, what teach you through four years of undergraduate, and another year if you go on to M.Eng., and then if you get a Ph.D., it's another 15 years or whatever it takes to get a Ph.D.

OK, all you teach about is abstraction: abstraction, abstraction, abstraction. So, this is a powerful abstraction: quite good. Now, we learned three methods. In general, any method can be used. You can convert one to the other. But each has situations where it is arguably simplest or most precise. So, any of the methods can be used. However, you must learn all of them, OK, because there are going to be some situations where you need one, where it's better to do one, better to do another. If you're reading in the literature, you want to know these different ways of doing it. And that means that even though you may get really comfortable with accounting, OK, in an exam, or whatever, I may say solve this with a potential function argument.

So, you want to be comfortable with all the methods, OK. Last point is that, in fact, different potential functions or accounting costs may yield different bounds. OK, so when you do an amortized analysis, there's nothing to say that one set of costs is better than another. So, just as an example, OK, in any data structure generally that supports delete, I can amortize all the deletes against the inserts. So, in general, what I could do is say deletes are free, OK, and charge twice as much for each insert.

OK, charging enough when I do the insert to amortize it against the delete. That you could do with an accounting method. You can also do it with a potential method, the potential, then, being the number of items that I actually have in my data structure times the cost of the delete for each of those items. OK, so the point is that I can allocate costs in different ways. Or I could have amortized costs which are equal to the true costs. So, there are different ways that I could assign amortized costs. There is no one way, OK, and choosing different ones may yield different bounds. It may not, but it may yield different bounds. OK, generally it does yield different ones. OK, next time: an amazing use of potential functions. OK, the stuff is cool, but let me tell you, Wednesday's lecture: amazing. Amazing the type of analysis we are going to be able to do.