JOHN DONG: All right, so I'm sure everybody's kind of curious about Project 2.2 Beta, so here's the preliminary performance results. There's still a bit more work to do on finalizing these numbers, but here's how they look. But before I show you guys that, I'd like to yell at you guys for a little bit. So this is a timeline of the submission deadline and when people submitted things. So from this zone to this zone is an hour, and I see like 50% of the commits made during that period.

So a little bit of background, the submission checking system automatically clones a repository up to the deadline and not a second after. So if you look at from this to this, some people took quite a jump back. So just as a warning, please try to submit things on time. 11:59 means 11:59, and to drive that point further, this is an example of a commit that I saw in somebody's repository, who's name I blanked out. Obviously, it's seven seconds past the deadline, so the automatic repository cloner didn't grab it. And the previous commit to that was like 10 days ago when Reid pushed out pentominoes grades. So don't do that either.

A little break down on how often people commit. About a quarter of the class only made one commit to their repository. Half of you guys did three to 10 commits, which seems to be an up. Under 21+, there was somebody who did 100 some commits, which was pretty impressive. Yeah, very smart dude. Committing often is a good idea, so that you don't run into a situation like before. And next time definitely, there is going to be next to zero tolerance for people who don't commit things on time for the deadlines.

Now, the numbers that people want. So for rotate, just as an interesting data point, this is the 512 by 512 case. It seems like not everybody remembered to carry over their optimizations for the 512 case over to the final, otherwise you would expect the numbers to be a little bit more similar and not off by a factor of eight. And for rotate overall, that's the distribution. The speed up factor is normalized to some constant that gives everybody a reasonable number. And there were a lot of groups that had

code that didn't build. Yes?

AUDIENCE: Does the speedup only include the rotate dot 64.

JOHN DONG: Yes. Performance was only tested on the rotate dot 64, which is what we said in the handout also. So there were a lot of groups that didn't build, which really surprised me, and I think it's probably because half of you guys pushed things after the deadline, and I presume those things contained important commits towards making your code actually work. But in this case, there was no header files, there was no cross-testing. All we did was run your make file on your code, and we replaced your test bed dot c, and your k timing, so I'm not quite sure why people have code that doesn't build.

For sort, this was the maximum size input that we allowed you guys to run. One group did really well. So all of these are correct. The correctness test is built in, and I replaced that with a clean copy that contains a couple additional checks, by the way. So I tried the other extreme, a relatively small array.

AUDIENCE: Is the top the same person?

JOHN DONG: I'm not sure whether or not the top is the same person. But the distribution wise seems like people didn't quite remember to optimize for the smallest case. And this is the current overall speedup factors for sort averaging. We did about 10 to 15 cases for rotate and sort. And that's the overall speedup distribution.

SAMAN AMARASINGHE: OK, so now you're done with individual project, so you already did the last project individually, and then we are moving into, again, a group project. So the first thing we have, setting up automated system for you to say who your group members are. So we will send you information, and with that, what you have to do is run a script saying who your group members are, both group members have to do it, and then we will basically clear that account in there.

That said, a lot of you didn't know, in the first project, how to work and what's the right mode of operations with the group. OK if we gave you to write 100,000 lines of code, it makes sense to say, OK, I'm going to divide the problem into half, one

person do one, the other person do the other half. The reason for doing the group is to try to get you to do pair programming, because talking to a lot of you, getting a lot of feedback, it looks like most of you spent a huge amount of time debugging. And since you're only writing a little amount of code, it makes a lot more sense to sit with your partner next to the screen, one person types, other person looks over, and then you have a much faster way of getting through the debugging process. So the next one, don't try to divide the problem by half. Just try to find some time, sit with each other.

Then the other really disturbing thing is there have been a couple of groups that were completely dysfunctional. We get emails saying, OK, my group member didn't talk to me, or they didn't do any work, or they were very condescending. And that's really sad, because from my experience with MIT students, when you guys go to company, you guys probably will be the best programmers there. There's no question about it. I have seen that. To the point that some people might even resent it, to have this best programmer.

But what I have seen is the fact a lot of you cannot work in the group, if you haven't developed that skill, you will not be the most impactful person. I have seen that again and again in my experience with doing a start-up. Our MIT students, their way of impacting is put all night do the entire project by themselves. Doable when you're doing a small change to a large project, but if you want to have a big change, you can't do that. You have to work with the group, figure out how to impact, how to communicate. This is more important learning than, say, trying to figure out how you can optimize something.

So being individual contributors and able to do amazing things is important, but the fact that you can't work with a group is going to really make the impact you can make much less. So please, please learn how to work with your group members. Some of them might not be as good as you are, and that will be probably true in real life, too, but that doesn't mean you can be condescending towards them, make them feel inferior. That doesn't cut it. You have to learn how to work with these people. So part of your learning is working with others, and that's a large part of

your learning. Don't consider that to be this external thing, even though you might think you can do a better job.

Just work with the other person, especially in pair programming, because where both of you are sticking together, it's much easier. Because there are four eyes on your project, they might see something you don't see, and see whether you can work together like that. And I don't want to hear any more stories saying, look, my partner was too dumb, or my partner didn't show up, he couldn't deal with that. Those are not great excuses. And so we wonder. Some of them, we will pay attention to it, because it might be one person's unilateral actions that lead to that. Still, please, try to figure out how we can work with your partners. And hope you have a good partner experience. Use pair programming. Use a lot of good debugging techniques, and the next project will be fine.

CHARLES LEISERSON: Great. We're going to talk more about caches. Whoo-whoo! OK. So for those who weren't here last time, we talked about the ideal cache model. As you recall, it has a two-level hierarchy and a cache size M bytes, and a cache line length of B bytes. It's fully associative and is optimal omniscient replacement strategy. However, we also learned that LRU was a good substitute, and that any of the asymptotic results that you can get with optimal, you could also get with LRU. The two performance measures that we talked about were the work, which deals with what the processor ends up doing, and the cache misses, which is the transfers between cache and main memory. You only have to count one direction, because what goes in basically goes out, so more or less, it's the same number.

OK, so I'd like to start today by talking about some very basic algorithms that you have seen in your algorithms and data structures class, but which may look new when we start taking caches into account. So the first one here is the problem of merging two sorted arrays. So as you recall, you can basically do this in linear time. The way that the algorithm works is that it looks at the first element of the two arrays to be sorted and whichever is smaller, it puts in the output. And then it advances the pointer to the next element, and then whichever is smaller, it puts in the output. And in every step, it's doing just a constant amount of work, there are n items, so by the

time this process is done, we basically spent time proportional to the number of items in the output list. So this should be fairly familiar. So the time to emerge n elements is order n.

Now, the reason merging is useful is because you can use it in a sorting algorithm, a merge sorting algorithm. The way merge sort works is it essentially does divide and conquer on the array. It divides the array into two pieces, and it divides those each into two pieces, and those into two, until it gets down to something of unit size. And then what it does is it merges the two pairs of arrays. So for example here, the 19 and 3 got merged together to be 3 and 19. The 12 and 46 were already in order, but you still had to do work to get it there and so forth. So it puts everything in order in pairs, and then for each of those, it puts it together into fours, and for each of those, it puts it together into the final list. Now of course, the way it does this is not in the order I showed you. It actually goes down and does a walk of this tree. But conceptually, you can see that it essentially comes down to merging pairs, merging quadruples, emerging octuples, and so forth, all the way until the program is done.

So to calculate the work of Merge Sort, this is something you've seen before because it's exactly what you do in your algorithms class. You get a recurrence that says that the work, in this case is-- well, if you have only one element, it's a constant amount of work, and otherwise, I solve two problems of half the size doing order n work, which is the time to merge the two elements. So classic divide and conquer. And I'm sure you're familiar with what the solution to this recurrence is. What's the solution to this recurrence? n log n. I want to, though, step through it, just to get everybody warmed up to the way that I want to solve recurrences so that you are in a position, when we do the caching analysis-- we have a common framework for understanding how the caching analysis will work.

So we're going to solve this recurrence, and if the base case is constant, we usually omit it. It's assumed. So we start out with W of n, and what we do is we replace it by the right hand side, where we put the constant term on the top and then the two children. So here I've gotten rid of the theta, because conceptually when I'm done, I can put a big theta around the whole thing, around the whole tree, and it just makes

the math a little bit easier and a little bit clearer to see what's going on. So then I take each of those and I split those, and this time I've got n over 4. Correct? I checked for that one this time. It's funny because it was still there, actually, just a few minutes before class as I was going through. And we keep doing that until we get down to something of size one, until the recurrence bottoms out.

So when you look at a recursion tree of this nature, the first thing that you typically want to do is take a look at what the height of the tree is. In this case, we're taking a problem of size n, and we're halving it at every step. And so the number of times we have to halve the argument-- which turns out to be also equal to the work, but that's just coincidence-- is log n times. So the height is log base 2 of n. Now what we do typically is we add things up across the rows, across the levels. So on the top level, we have n. On the next level, we have n. The next level, hey, n. To add up the bottom, just to make sure, we have to count up how many leaves there are, and the number of leaves, since this is a binary tree, is just 2 to the height. So it's 2 to the log n, which is n. So then I add across all the leaves, and I get the order 1 at the base times n, which is order n.

And so now I'm in a position to add up the total work, which is basically log n levels of n is total of order n log n. So hopefully, this is all review. Hopefully this is all review. You haven't seen this before. It's really neat, isn't it? But you've missed something along the way.

So now with caching. So the first thing to observe is that merge subroutine, the number of cache misses that it has is order n over B. So as you're going through, these arrays are laid out continuously in memory. The number of misses-- you're just going through the data once-- is order n data, all going through contiguously. And so every time you bring in data, you get full spatial locality, there are n elements, it costs n over B. So is that plain? Hopefully that part's plain, because you bring in things of each axis, you get the same factor of B.

So now merge sort-- and this is, once again, the hard part is coming up with a recurrence, and then the other hard part is solving it. So there's two hard parts to

recurrences. OK, so the merge sort algorithm solves two problems of size n over 2, and then does a merge. So the second line here is pretty straightforward. I take the cache misses that I need to do, and I take a merge. I may have a few other accesses in there, but they're going to be dominated by the merge. OK, so it's still going to be theta n over B.

Now the hard part, generally, of dealing with cache analysis is how to do the base case, because the base case is more complicated than when you just do running time, you get that to run down to a base case of constant size. Here, you don't get to run down to base case of constant size. So what it says here is that we're going to run down until I have a sorting problem that fits in cache, n is going to be less than some constant times n, for some sufficiently small constant c less than 1. When in finally fits in cache, how many cache misses does it take me to sort it? Well, I only need to have the cold misses to bring that array into memory, and so that's just proportional to n over B, because for all the rest of the levels of merging, you're inside the cache. That make sense? So that's where we get this recurrence. This is always a tricky thing to figure out how to write that recurrence for a given thing.

Then, as I say, the other tricky thing is how do you solve it? But we're going to solve it essentially the same way as we did before. I'm not going to go through all the steps here, except to just elaborate. So what we're doing is we're taking, and we have n over B at the top, and then we divide it into two problems, and for each of those-- whoops, there's a c there that doesn't belong. It should just be n over 2B on both, and then n over 4B, and so forth. And we keep going down until we get to our base case.

Now in our base case, what I claim is that when I hit this base case, it's going to be the case that n is, in fact, a constant factor times n, so that n over B is almost the same as m over B. And the reason is because before I hit the base case, when I was at size twice n, and that was bigger than m. So if twice n is bigger than m-- than my constant times m-- but n is smaller than m, then it's the case that n and m are essentially the same size to within a constant factor, within a factor of two, in fact.

And so therefore, here I can say that it's order m over B.

And now the question is, how many levels did I have to go down cutting things in half before I got to something of size m over B? Well, the way that I usually think about this is-- you can do it by taking the difference as I did before. The height of the whole tree is going to be log base 2 of n, and the height of this is basically log of-- what's the size of n? It's going to be log of the size of n when this occurs. Well, n at that point is something like cm. So it's basically log n minus log cm, which is basically log of n over cm. How about some questions. Yeah, question.

AUDIENCE: --the reason for why you just substituted on the left, [INAUDIBLE] over B, but on the right [INAUDIBLE].

CHARLES LEISERSON: Here?

AUDIENCE: No.

CHARLES LEISERSON: Or you mean here?

AUDIENCE: [INAUDIBLE].

CHARLES LEISERSON: On the right side.

AUDIENCE: On the right-most leaf, it's n over B. Is that all of the leaves added up because [INAUDIBLE]?

CHARLES LEISERSON: No, no, no, this is going to be all of the leaves added up here. This is the stack I have on the right hand side. So we'll get there. So the point is, the number of leaves is 2 to this-- so that's just n over cm-- times m over B. Well, n over cm times m over B, the m's cancel, and I get essentially n over b with whatever that constant is here. And so now I have n over b across every level, and then when I add those up, I have to go log of n over cm levels, which is the same as log of n over m. So I have n over B times log of n over m. Yeah, question?

AUDIENCE: Initial assumption that c is some sufficiently small number, so 1 over c would be a rather large factor.

CHARLES LEISERSON: It could potentially be a large factor, but it's a constant. In other words, it can't vary with n. So in fact, typically for something like merge sort, the constant is going to be very close to-- for most things, the constant there is typically only a few. Because the question is, how many other things do you need in order to make sure it fits in? In this case, you have n. Well, you have both the input and the output, so here, it's going to be, you have to fit both the input and the output into cache in order not to have the cache it. So it's basically going to be like a factor of 2 for merge sort. For the matrix multiplication, it was like a factor of three. So generally a fairly small number. Question?

AUDIENCE: I guess that makes sense. But I [INAUDIBLE]. So on the order of the size of the leaves, can you assume that n is more than cm, so you can substitute--

CHARLES LEISERSON: Yeah, because basically, when it hits this condition--

AUDIENCE: Right. I understand that. Then you're also saying, why isn't there more or less just one B, because there's n over cm, n is the same as cm. That's [INAUDIBLE]. At the bottom level, there should be--

CHARLES LEISERSON: Oh, did I do something wrong here? Number of leaves is--

AUDIENCE: [INAUDIBLE].

CHARLES LEISERSON: Right, right. Sorry. This is the n at the top. You always have to be careful here. So this is the n in this case, so this is some n, little n. So this is not the same n. This is the n that we had at the top. This is the notion of recurrences, like the n keeps recurring, but you know which ones-- so it can be confusing, because if you're-- so yeah. So this is not the n that's here. This is the n that started out at the top. So we're analyzing it in terms of the n. Some people write these things where they

would write this in terms of k, and then analyze it for n, and for some people, that can be helpful to do, to disambiguate the two things. I always find it wastes a variable, and you know, those variables are hard to come by. There's only a finite number of them.

OK, so are we good for this? So here, we ended up with n over b, log of n over m cache misses. So how does that compare? Let's just do a little thinking about this. Here's the recurrence, and I solved it out to this. So let's just look to see what this means. If I have a really big n, much bigger than the size of my cache, then I'm going to do a factor of B log n less misses than work, because the work is n over B log n. So if n compared to m-- let's say n was m squared or something-- then n over m would still be n, if n was as big as m squared. So if n was as big as m squared, this log of n over m would still be log n. And so I would basically have n over B log n for a factor of B log n less misses than work. If they're about the same-- did I get this right? If they're about the same size, if n is approximately m, maybe it's just a little bit bigger, then the log here disappears completely, and so I basically just have n over B misses.

AUDIENCE: [INAUDIBLE].

CHARLES LEISERSON: Yeah, but if n is like--

AUDIENCE: [INAUDIBLE].

CHARLES LEISERSON: Yeah. In fact, for this, you have to be careful as you get to the base cases. Technically, for some of this, I should be saying 1 plus log of n over m, and in some of the things I do later, I will put in the ones. But if you're looking at it asymptotically and n gets big, you don't have to worry about those cases. That just handles the case whether you're looking at n getting large or whether you're trying to handle a formula for all n, even if and n is small. Question?

AUDIENCE: [INAUDIBLE]?

CHARLES The work was n log n, yes. The work was n log n. So here we basically have n over

LEISERSON: B log n, so I'm saving a factor of B in the case where it's about the same. Did I get this right? I'm just looking at this, and now I'm trying to reverse engineer what my argument is. So we're looking at n log n versus n over B log n over m.

AUDIENCE: [INAUDIBLE PHRASE]. So that you get a factor of B less misses, because you would be getting n over B like log of n, that's the only way you're getting a factor of B less misses. So I don't understand how you're saying, for n more or less equal to m. You would want something more like, for n--

CHARLES LEISERSON: Well, if n and m are about the same size, the number of cache misses is just n over B. And the number of cache misses is n over B, and the work is n log n. So I've saved a factor of B times log n, OK? What did I say?

AUDIENCE: [INAUDIBLE].

CHARLES LEISERSON: B log m? No, I was saying that's for the case when n is much bigger than m. So let's take a look at the case-- let me just do it on the board here. Let's suppose that n is like m squared, just as an example, big number. So I'm going to look at, essentially, n over B times log of n over m. So log of n over m, so what is n over m is about m, which is about square root of n, right? So this basically ends up being approximately n over B log of square root of n, which is the same as log n, to within a constant factor. I'm going to leave out the constant factors here.

Then I want to compare that with n log n. So I get a factor of B less misses. So the first one, yes, OK. So I get a factor of B less misses, you're right. Then I get a factor of B less misses. So I think I've got these switched. So this is the case I'm doing is for n much bigger than m. So let's do the other case. I think I've got the two things switched. I'll fix it in the notes.

If n and m are approximately the same, then the log is a constant, right? So this ends up being approximately n over B. And now when I take a look at the difference between the number of things, I get B log n. So I've got the two things mixed. Yeah?

AUDIENCE: As n approaches m, then the log would approach zero, but were you talking about how it technically should be--

CHARLES LEISERSON: 1 plus n, yes.

AUDIENCE: So technically, that approaches one when the log approaches zero.

CHARLES LEISERSON: Yeah.

AUDIENCE: These things are really hard for me, because they are really arbitrary. And then you're like, oh yeah, you can just put a 1 on top of there. And for example, I always miss those, because I usually try to do the math as rigorously as I can, and those ones generally do not appear, and you're like, oh, sure whatever. So how am I supposed to know that the log is actually not going to be zero, and I'm going to be like, yeah, you're not going to do any caches.

CHARLES LEISERSON: Because generally, what we're doing is we're looking at how things scale, so we're generally looking at n being big, in which case it doesn't matter. These things only matter if n's-- for example, notice here that if n goes less than m, we're in real trouble, right? Because now the log is negative. Wait, what does that mean? Well, the answer is the analysis was assuming that n was sufficiently large compared with m.

AUDIENCE: Why can't you just be like, oh, when n is for less than one, you can assume, well, n is 2n. In that case, you get log of two, which is still something or other.

CHARLES LEISERSON: Yeah, exactly. So what happens in these things is if you get right on the cusp of fitting in memory, then these analyses like, well, what exactly is the answer, is dicey. But if you assume that it doesn't fit in, what's going to happen? Or that does fit in, what is going to happen? And then the analysis right on the edge is somewhere between there. Good. So I switched these. I said this the other way around. That's funny. I went through this, and then in my notes, I had them switched, and I said, oh my gosh, I did this wrong. And I've just gone through it, and it turns out I was right in my notes.

Now, one of the things, if you look at what's going on-- let's just go back to this picture here. What's going on here is each one of the passes that we're doing to do a merge is basically taking n over B misses to do a binary merge. We're going through all the data to merge just two things, and traversing all the data. So you can imagine, what would happen if I did, say, a four-way merge? With a four-way merge, I could actually merge four things with only a little bit more than n over B misses. In fact, that's what we're going to analyze in general. So the idea is that we can improve our cache efficiency by doing multi-way merging.

So the idea here is, let's merge R, which is, let's say, less than n subarrays with a tournament. So here are R subarrays, and here's they're each, let's say, is of size n over R. And what we're going to do is merge them with a tournament, so this is a tournament where we say, who's the winner of these two, who's the winner of these two, et cetera. And then whoever wins at the top here, we take them and put them in the output, and then we repeat the tournament.

Now let's just look what happens. It takes order R work to produce the forced output. So we got R things here. To playoff this tournament, there are R nodes here. They each have to do a constant amount of comparing before I end up with a single value to put in the output. So it costs me R to get this thing warmed up. But once I find the winner, and I remove the winner whatever chain he might have come along, how quickly can I repopulate the tournament with the next guy? The next guy only has to play the tournament on the path that was there. All the other matches, we know who won. So the second guy only cost me log R to produce the next guy. And the next guy is log R, and so once we get going, to do an R-way merge only costs me log R work per element.

So the total work in merging is R, to get started, plus n log R. Well, R is less than n, so that's just n log R total to do the merging. That's the work. Now, let's take a look at what happens if I now do merge sort with R-way merges. So what I do is if I have only one element, then it's going to cost me order one time to merge it, because there's nothing to do, just put it in the output. Otherwise, I've got R problems of size n over R that I'm going to merge, and my merge takes n log R time to do the merge.

So if I look at the recursion tree, I have n log R here starting here, then I branch R ways, and then I have n over R log R to do the R-way branching at the next level, n over R squared log R at the next level, et cetera.

AUDIENCE: You said that the cost of processing is R.

CHARLES LEISERSON: --is n log R.

AUDIENCE: But is--

CHARLES LEISERSON: Upfront, there's an order R cost, but the order R cost is dominated by the n log R, so we don't have to count that separately. We just have to worry about this guy. So as I go through here, I basically end up having a tree which is only log base R of n tall, because I'm dividing things into R pieces each time, rather than into two pieces. So I only go log base R steps till I get to the base case. But I'm doing an R-way merge, so the number of leaves is still n. But now when I add across here, I get n times log R, and I go across here, I get n times log R, because I got R copies of the same thing. Now I've got R squared times n over R squared log R, and so forth. And so at every level, I have n log R. So I have n log R times the number of levels here, which is log base R of n, plus the order n work at the bottom, which we can ignore because it's going to be dominated.

And so what you notice here is, what's log base R of n? That's just log n over log R. So the log Rs cancel, and I get n log n plus n, which is just n log n. So after all that work, we still do the same amount of work, whether I do binary merging or R-way merging, the work is the same. But there's kind of a big difference when it comes to caching. So it's the same work as binary merge sort.

So let's take a look at the caching. So let's assume that my tournament fits in the cache. So I want to make sure that R is less than m over B for some constant R. So when I do constant way, when I consider the R-way merging of contiguous arrays of total size n, the entire tournament plus 1 block from each array can fit in cache. So the tournament is never going to be responsible for generating cache misses,

because I'm going to leave the tournament in memory. So if I'm the optimal algorithm, it's going to say, let's just leave the tournament in memory and bring in all the other things as we do the operation. Question?

AUDIENCE: [INAUDIBLE]

CHARLES LEISERSON: Those circles that I had, the tree.

AUDIENCE: Is that a cumulative list of the elements that you've merged in already?

CHARLES LEISERSON: I'm sorry. Is the--

AUDIENCE: Is it a cumulative list of the length arrays that you've merged already?

CHARLES LEISERSON: No, no, no. You haven't merged them. Let's just go back and make sure we understand the algorithm. The algorithm says that what we do is we compare the head of this pair and we produce a single value here, for which whatever is-- because these are already sorted to do the merge. These are already sorted. So I just have the minimum of these two here, and the minimum of these two here, and the minimum of all four of them here. So it's repeated.

When we get to the top, we have now the minimum of all of these guys, and that's the guy that's the minimum overall, we put him in the output array. And now we walk back down the path that he came from, and what we do is we say, oh, this guy had a-- let's walk down the path, let's say we get to this guy. Let's advance the pointer in here and bring out another element. And now we play off the tournament here, play off the guy here, and he advances, and he advances, whatever. And now some other path may be the minimum.

But it only took me log n work. I'm only keeping copies of the element, if you will, or the results of the comparisons along this in the tree here. And that tree, we're saying, fits in the cache, plus 1 block, the first block. Whatever cache block fits in each of these arrays, no matter how much we've gone down, one of those is fitting

in memory. So then what happens here is the entire tournament plus one block from each memory can fit in cache, and so therefore the number of cache misses that I'm going to have when I do the merge it's just essentially the time to take faults on that one cache block whenever I exceed it in each array, plus the one for the output that's similar.

And so the total number of cache misses that I'm going to have is going to be n over B, because I'm just striding straight through memory, and that tournament, I don't have to worry about, because it's sitting in cache. And there's enough sitting in cache that all the other stuff, I can just keep one block from each of them in memory and still expect to get it. In fact, you need the tall cache assumption to assume that they all fit in memory. So therefore, the R-way merge sort is then, if it's sufficiently small, once again, we have the case that it fits in memory so I only have the cold misses to get there, n over B, if n is less than cm. And otherwise, it's R copies of the number of cache misses for n over R, plus n over B. Because this is what it took us here to do the merge. We get only n over B faults when we merge, as long as the tournament fits in cache. If the tournament doesn't fit in cache, it's a more complicated analysis.

AUDIENCE: --n over B, that's cold misses. You're getting the stuff--

CHARLES LEISERSON: Yeah, basically, it's the cold misses on the data, yes, basically. Good. So now, let's do the recursion tree for this. So we basically have n over B that we're going to pay at every level, dividing by R, et cetera, down to the point where things fit in cache. And by the time it fits in cache, it's going to be m over B, because n will be approximately m, just as we had before when we were doing the binary case. As soon as the subarray completely fits in memory, I don't have to when I'm doing the sorting.

So this is now analyzing not the merging, this is analyzing the sorting now. This is the sorting, not the merging. So we get down to m over B, and I've gone now log base R, not log base 2 as we did before, but log base R n over cm. The number of leaves is n over cm, and so when I multiply this out, I get the same n over B here,

and I've got n over B at every level here. So where's the win? The win is that I have only log base R of n over cm, rather than log base 2 levels in the tree, because the amount that every level cost me was the same, asymptotically. So when I add it up, I get n over B log base R of n over m, instead of n over B log base 2 of n over m.

So how do we tune R? Well if we just look at this formula here, if I want to tune R, what should I do to R to make this be as small as possible? Make it as big as possible. But I had to assume that R was less than some constant times m so that it fits in cache. So that's, in fact, what I do, is I say R is m over B. So it fits in cache, we have at least one block for each thing that we're merging. And then when we do the analysis now, I take log base m over B here, and that's compared to the binary one, which was log base 2, which is a factor of-- because this is just log 2 over log base-- of log of m over B savings in cache misses.

Now, is that a significant number? Let's take a look. So if your l one cache is 32 kilobytes, and we have cache lines of 64 bytes, that is basically the difference in the exponents, 9x savings. For l two cache, we get about a 12x savings. For l three, we get about a 17x savings. Now of course, there are some other constants going on in here, so you can't be absolutely sure that it's exactly these numbers, but it's going to be proportional to these numbers. So that's pretty good savings to do multi-way merging. So generally when you merge, don't merge pairs. Not a very good way of doing it if you want to take good advantage of cache. May give you some ideas for how to improve some sorts that you might have looked at.

Now it turns out that there's a cache oblivious sorting algorithm, where you don't actually have-- that was a cache aware algorithm that knew the size of the cache, and we tune R to get there. There is an algorithm called funnelsort, which is based on recursively sorting and n to the 1/3 groups of n to the 2/3 items. And then you merge the sorted groups with a merging process called an n to the 1/3 funnel.

So this is more for fun, although the sorting algorithm, in my experience, from what others have told me about implementing it and so forth, is probably about 30% slower than the best hand-tuned algorithm. Whereas with matrix multiplication, the

cache oblivious algorithms are as good as any cache aware algorithm as a practical matter, here, they're off by about 20% or 30%. So interesting research topic is build one of these things and make it really efficient so that it can compete with real sorts.

So the k funnel merges k cubed items in k sorted lists, incurring this many cache funnels. Here, I did put in the one, for people who are concerned about the ones. And so then, you get this recurrence for the cache misses, because you solve n to the 1/3 problems of size n to the 2/3 recursively, plus this amount for merging. And that ends up giving you this bound, which turns out to be asymptotically optimal. And the way it works is there's basically, a k funnel is constructed recursively.

And the idea is that what we have is, we have recursive k funnels, so this is going to be a merging process, that is going to produce k cubed items by having k to the 3/2 buffers that each are taking the square root of k guys and producing k guys out. So each of these guys is going to produce k, and the square root of k of them for a total of k to the 3/2, but each of these is going to be length square root of k, so we end up with k cubed. and And they basically feed each other, and then they get merged with their own k thing, and each of these then recursively is constructed the same way.

And the basic idea is that you keep filling the buffers, I think I say this here, so that all these buffers end up being in contiguous storage. And the idea is, rather than going and just getting one element out as you do in a typical tournament, as long as you're going to go merge, let's merge a lot of stuff and put it into our buffer so we don't have to go back here again. So you sort of batch your merging in local regions, and that ends up using the cache efficiently in the local regions. Enough of sorting. Let's go on to physics.

So many of you are probably studying in your linear algebra class or elsewhere, the heat equation. So, people familiar with heat diffusion? So it's a common one to do, and these were-- I have a former student, Matteo Frigo, who is a brilliant coder on anything cache oblivious. He's got the best code out there. So the 2D heat equation, what we do is let's let u(t, x, y) be the temperature at time t at point (x, y). And now

you can go through the physics and come up with an equation that looks like this, which says that basically the partial of u with respect to t is proportional to the sum of the second partials with respect to x and with respect to y.

So basically what that says is, the hotter the difference between two things, the quicker things are going to adjust, the quicker the temperature moves between them. And alpha is the thermal diffusivity, which has-- different materials have different thermal diffusivities. Say that three times fast. So if we do a simulation, we can end up with heats, say, put like this, and after it looks like this. See if we can get this running here. So now, let me see. So I can move my cursor around and make things. You can just sort of see that it simulates. You can see the simulation is actually pretty slow. Now, on my slide, I have a thing here that says-- let's see if this breaks when we do it again. There we go. So we're getting around 100 frames per minute in doing this simulation.

And so how does this simulation work? So let's take a look at that. It's kind of a neat problem. So this is what happened when I did 6.172 for a little while. It basically gave me that after a while, because it just sort of averages things, smears it out. So what's going on? Let's look at it in one dimension, because it's easier to understand than if we take on two dimensions. So assuming that we have, say, a bar which has no differential in this direction, only in this direction. So then we get to drop the partials with respect to y.

So if I take a look at that, what I can do is what's called a finite difference approximation, which you probably-- who's studied finite differences? So a few people. It's OK if you haven't. That's OK if you haven't, I'll teach it to you now. And then you're free to forget it, because that's not the part that I want you to understand, but it is interesting. So what I can do is look at the partial, for example, with respect to t, and just do an approximation the says, well, let me perturb t a little bit-- that's what it means. So I add delta t minus u of t divided by t plus delta t minus t, which gives me delta t. And I can use that as an approximation for this partial.

Then on the right hand side-- well first of all, let me get the first derivative with

respect to x. And basically here what I'll do is I'll do an approximation where I take x plus delta x over 2 minus x minus delta x over 2, and once again, the differences in the terms there ends up being delta x. And now I use that to take the next one. So basically, to take this one, I basically take the partial with respect to delta x over 2, minus the partial with x minus delta x over 2, and take the partial of that, do the approximation. And what happens is, if you look at it, when I take a partial here I'm adding delta x over 2 twice, so I end up getting just a delta x here, and then the two things on either side combined give me my original one, 2 times u(t, x), and then another one here, and now the whole thing over delta x squared.

And so what I can do is to reduce this heat equation, which is continuous, to something that we can handle in a computer, which is discrete, by saying OK, let's just do this approximation that says that this term must be equal to that term. And if you've studied the linear algebra that said that there are all kinds of conditions on convergence, and stability, and stuff like that, that are actually quite interesting from a numerical point of view, but we're not going to get into it.

But basically, I've just taken that equation here and said, OK, that's my approximation for this one. And now what do I have here? I've got u of t plus delta t, and u things u of t, and then over here, they're all with t, but now the deltas are in-- whoops, that should have been a delta x there. I don't know how that got there. That should be a delta x there. They're all in spatial over here.

So what I can do is take this, and do an iterative process to compute this. And so the idea is, let me take this and throw this term onto the right hand side, and look at u of t plus delta t as if it's t plus 1. Let me make my delta be one, essentially. Throw the delta t over here times the alpha over delta x1, and then I get basically u of tx is based on u of t of x plus 1 and of x and x minus 1. As I say, there's a typo here. That should be a delta t.

So what that says is that if I look at my one-dimensional process proceeding through time, what I'm doing is updating every point here based on the three points below it, diagonally to the right, and diagonally to the left. So this guy can be

updated because of those. These we're not going update, because they're the boundary. So these can be fixed. In a periodic stencil, they may even wrap around like a torus. So basically, I can go through and update all these with whatever that hairy equation is. And this is basically what the code is that I showed you is doing. It just keeps updating everyone based on three until I've gone through a bunch of time, and that's how the system evolves. So any questions about how I got to here?

So we're going to now look at this purely computer sciencey now. We don't have to understand any of those equations. We just have to understand the structure. The structure is that we're updating t plus 1 based on stuff on three points with some function that some physicist oracle gave us out of the blue. And so here is a pretty simple algorithm to do it. I basically have what's called the kernel, which does this updating, basically updating each one based on things. And what I'm going to do for computer science is I don't need to keep all the intermediate values.

And so what I'm going to do is do what's called an even-odd trick. Basically if I have one row, I compute the next row into another array, and then I'll reuse that first array-- it's all been used up-- and go back to the first one. So basically here, I'm just going to update t plus 1 mod 2, and just allocate two arrays of size n, and just do modding all the way up. Is that clear? And other than that, it's basically doing the same thing, and I'm doing a little bit of fancy arithmetic here by passing-- see stuff, where I'm passing the pointer to where I am in the array, so I only have to update it locally within the array. So I don't have to double indexing once I'm in the array, because I'm already indexed into the part of the array that I'm going to use, and then I am doing flipping. So this is just a little bit of cleverness. You might want to study this later.

So what's happening then is I have this double nested loop where I have a time loop on the outside, and a space loop on the inside, and I'm basically going through and using a stencil of this shape, this is called a three point stencil, because you're basically taking three points to update one point. And now if I imagine that this dimension is bigger, n here is bigger than my cache size, what's going to happen? I'm going to take a cache fault here, these are all cold misses, et cetera. But when I

get back to the beginning here, if I use LRU, nothing is going to be in memory that I happened to update over here. So I have to go and I take a cache fault on every cache line.

And so if I'm going t steps into the future from where I started, I basically have n times t updates, and I save a factor of B, because I get the spatial locality because the u of t minus 1, u of t, and u of t plus 1, are all generally on the same-- are nearby, and all within one cache line. Question?

AUDIENCE: The x's, what are the x's for?

CHARLES LEISERSON: Sorry. I should have put the legend on here. The x's are a miss. So I do a miss when I update these, and then these I don't miss on, because it was brought in when I accessed that. And then I do a miss, and I'll do it-- so basically I do it, then I shift over the stencil by one, and then I won't get a miss. So I'm just looking at the misses on the reads, not misses on the writes. I should have made that clear, too. But the point is, the writes don't help you, because it's all out of memory by the time I get up here. To the second row, if this is longer than my cache size, none of that's there if I'm using LRU.

AUDIENCE: You have also a miss, like you need to get two [INAUDIBLE].

CHARLES LEISERSON: Yeah, but what I'm saying is I'm only looking at the read misses. Yes, there are write misses as well, but basically, I'm only doing the read misses. You can look at the write misses as well. It makes the picture messier. So we've basically have nt over b. However this, let me tell you, is the way that everybody codes it. and And if you have a machine where you have any bandwidth issues to memory, especially for these large problems, this is not a very good way to do it, as it turns out.

So it turns out that what you want to do is, as we've seen, divide and conquer is a really good way to do it. But in this case, when we're doing divide and conquer, we're actually not going to use rectangles, we're going to use trapezoids. And the reason is that a trapezoid has the nice property that-- notice that if I have all these points in memory, then notice that I can compute all the guys that are read on the

next level, and then I can compute all the guys that are next on the next level. And so for example, if you imagine that this part here fit within cache, I could actually keep going. I didn't have to stop here, I could keep going right up to a triangle if I wanted to, and compute all the values without having any more cache misses than those needed to bring in, essentially, one row-- two rows, actually, because I'm reusing the rows as I go up.

So what we're going to do is traverse trapezoidal regions of space-time points such that the points are between an upper limit, T1, and a low one, T0, and between an x0 and an x1, where now I have slopes here that are going to be, in general, this is plus 1 minus 1. And in fact, sometimes it will be straight, in which case we'll call it 0. It's really the inverse of the slope, but we'll still call it zero rather than infinity. So it's 1 over the slope. There's a name for that, right? Is that called the run or something? I forget, I don't remember my calculus.

So that's what we're going to do. And we're going to leave the upper and right borders undone and include, so it's going to be a sort of half open trapezoid on the left and bottom, closed on the left and bottom, and open on the top and right. So the width is basically the midpoint here, and the height is the height, because they're always going to have parallel axes here. So here's how are our recursion is going to work. If the height is 1, then we can compute all space-time points in any way we want. I can just go through them if I want, because they're all independent. None depends on anybody else. So that's going to be our base case.

If the width is greater than twice the height, however, then what we're going to do is we're going to cut the trapezoid through the middle of the slope of minus 1. And that will produce two new trapezoids, which we then will recursively compute all the elements of. So I'll start out with a trapezoid. Basically, if it ends up that it's a long and wide one, I'm going to make what's called a space cut, and cut it this way, and then I'm going to recursively do this one and then this one. And notice that I can do that because-- all these guys I can do, but then when I get to the border here, this will already have been done by the time I'm computing these guys.

So the requirement is that I've got to do things according to that map of triples that I showed you before, but I don't have to do them in the same order. I don't have to do the whole bottom row first. In this case, I can compute the whole trapezoid here, and then I can compute this trapezoid here, and then all the values that I'll need will have already been computed over here, that are on the boundary of this trapezoid. The other type of cut I'll do is what happens when a trapezoid gets too tall for me. So if the trapezoid is too tall, then what we'll do is we'll slice it through the middle, but the other way. We call that a time cut. So we won't take it all the way through time, we'll only take it partially through time. Now you can show, and I'm not going to show this in detail, but you can show that if I do this, my trapezoids are always sort of medium sized. I never get long, long skinny ones. If I start with something that's sort of got a good aspect ratio, I maintain a good aspect ratio through the entire code.

So here's the implementation. This is what Matteo Frigo wrote, and I've modified it a little bit. So basically, we pass in it the values that let us identify the trapezoid, t0, t1, x0, and then the slope on the left side, x1 in the slope on the right side, where the dx0 and the dx1s are all either 0, 1, or minus 1. And then what I do is I look at what the height is that my trapezoid is going to operate on. And if the height is 1, well, then I just run through all the elements, and I just compute the kernel-- that program that I showed you before, that kernel-- on all the elements. Nothing really to be done there, just go through and compute them individually with a four loop.

Otherwise, if I've got the situation where the trapezoid is big, then I do this comparison, which I promise you-- you can work out the math if you wish-- which I promise you tells you whether or not it's more than twice the height, as I said before, whether the width is more than twice the height. And if so, I compute the middle point, and then I partition it into two trapezoids, and I recursively call them. And otherwise, I simply cut the time in half, and then I do the bottom half and then the upper half.

So getting all those parameters exactly right takes a little bit of thinking, makes my brain hurt, but Matteo is brilliant at this kind of coding. So let's see how well this

does. So I'm not going to do a detailed analysis that I did before, but basically what's going on is at this level, if I'm doing divide and conquering, I'm only doing a constant amount of work managing this stuff. So my caches that I'm taking in the internal part of the tree are all going to be order one.

Now each leaf is going to represent a trapezoid, which is going to be approximately h times w, where h and w are approximately equal, because they're going to be shaped-- This is assuming I start out with a number of iterations to do that is at least as large as the number of points that I have to go on. If I start out with something that's really thin and flat, then it's not going to be the case. But if I start out with something that's deep enough, then I'm going to be able to make progress in an unconventional order into time by moving the time non-uniformly through the space. So each leaf represents a fairly balanced trapezoid.

Each leaf basically is going to-- if you look that the direction of the trapezoid is in time, so this represents the spatial dimension, and if I have something of size w, I can access it with only w over B misses. And when that fits in cache, where w is some constant less than m, so w is order m. So each of these things that's a leaf is only going to occur w over B misses.

Now, the total space number of points I have to go after is n times t. N is going to be the full dimension this way, t is the height that way. And so since each leaf has hw points, I have nt over hw leaves. And the number of internal nodes is just the leaves minus 1, so they can't contribute substantially, because there's only order one misses I'm taking here, whereas I've got something on the order of w over B misses for this. So therefore, now I can do my math. The number of cache misses I'm going to take is-- well, how many leaves do I have? nt over hw. And what does each one cost us? W over B.

And so now, when I multiply that out, well, hw is about m squared, and w over B is about m over B. And so I get nt over MB as being the total number of savings. so whereas the original algorithm only got nt over B, we've got this factor of a memory cache size in there showing us that we have far fewer cache misses. So the cache

misses end up not being an issue for this. Any questions about that?

So I want to show you a simulation of this three point stencil and comparing the two things. So this is going to be the looping version, where the red dots are where the cache misses are, and this is going to be the trapezoidal one. And basically, I have an n of 95 and a t of 87, and what I'm going to do is assume a fully associative LRU cache that fits four points on a cache line, where the cache size is 32, two to the fifth as opposed to two to the 15th, it's really little. If I get a cache hit, I'm going to call it one cycle. If I get a cache miss, I'm going to call it 10 cycles. We're going to race them. So on the left is the current world champion, the looping algorithm. And on the right is the cache oblivious trapezoid algorithm. So let's go.

So you can see that it's basically, it's made a space cut there, but it's made a time cut across the top there. It said, this is too tall, so let me cut it this way. And that guy's, meanwhile, taking all those-- you can see how many cache misses he's taking. Let's speed him up. That's one way you can do it, is make it think.

So let's see what happens if I have a cache of size eight. So here we go. I think I'm just doing the same thing. I know I can show you the cuts. Can I show you the cuts? I know. I think it's because I'm not-- OK, let's try it. There we go. Now I'm showing the cuts as they go on. Let's do that again. We'll go fast and do it again. So those are the cuts that it's making to begin with as it's doing the divide and conquer. So I think this is the same size cache. So now I think I'm doing a bigger cache. I think I did a bigger cache, but I'm not sure I gave the other guy a bigger cache. Yeah, because it doesn't matter for the guy on the left, right? As long as the cache line is the same length and as long as it's not big enough, he's going to do the same thing. He didn't get to take advantage of the fact that the cache was bigger, because it was still smaller than the array that he's striping out there. Anyway, we can play with these all day.

So if you make the cache lines bigger, then of course it'll go faster, because he'll have fewer misses. He'll get to bring it in. So let's see here. So let's now do it for real. So this is a two-dimensional problem. You can use the same thing to do what

end up being three-dimensional trapezoids. In fact, you can generalize this trapezoid method to multiple dimensions. So this is the looping one. So let's start that one out. So it's going about 104 frames a minute. I think by resizing it, the calibration is off.

But in any case, let's switch to the cash oblivious version. Anybody notice something? Slower. Why is that? I gave code exactly as I had up there. No, it's not because it's two dimensions.

AUDIENCE: [INAUDIBLE].

CHARLES LEISERSON: I'm sorry?

[INTERPOSING VOICES]

CHARLES LEISERSON: Yeah, so now it's the trapezoiding at only 86 frames. What do you suppose is going on there?

AUDIENCE: You have [INAUDIBLE].

CHARLES LEISERSON: Yeah. So this is a case where if you look at the code I wrote, I went down to a t, a delta t, of one in my recursion. I recursed all the way down. Let's see what happens if instead of going all the way down, playing the trapezoid game on little tiny trapezoids, suppose I go down only to, say, when t is 10, and then do essentially the row major ones. So I'm basically coarsening the leaves of the thing. So to do that, I do this. So now we go-- ah.

So I have to coarsen in order to overcome the procedure call overhead. It has nothing to do with the cache. It has to the fact that the way that you implement recursion, recursion and function calls have a cost to them. And if what you're going to do is do a little tiny update of those few floating point operations-- let's go back to the looping just to see. The looping is going about 107, 108, and trapezoiding at 136. So unfortunately, you need a voodoo variable, but it's a voodoo variable not to overcome the cache, but rather to deal with what's the overhead in using the

processor when you do function calls.

So let's see. How coarse can we make it? Let's try five, a coarsening of five? That's still pretty good. That's still pretty good. How about four? Still doing 131 frames a minute. How about three? Oh, we lost something there. How about two? So at a coarsening of two, I go 138, whereas the looping goes at about the same. I can't do 20. I didn't program that in. I just programmed up to 10. So if I go down to one, however, then you see it's not that efficient. But if I pick any number that's even slightly larger, that gives me just enough that the function call overhead ends up not being a substantial cost of the things.

So let me just wrap up now. So I just have a couple more things. So I'm not going to really talk about these, but there are lots of cash oblivious algorithms that have been discovered in the last 10 or 15 years for doing things like matrix transposition, which is similar to rotating a matrix. You can do that in a cache oblivious fashion. Strassen's algorithm, which does matrix multiplication using fewer than n cubed operations. The FFT can be computed in a cache oblivious fashion. And LUD composition is a popular thing to solve systems.

In addition, there are cache oblivious data structures, and here are just a few of them. There's cache oblivious B-Trees and priority queues, and doing things called ordered-file maintenance. There's a whole raft. There's probably now several hundred papers written on cache oblivious algorithms, so something you should be aware of and understand how it is that you go about designing an algorithm of this nature. Not all of them are straightforward. For example, the FFT one does divide and conquer but not by dividing it into two. It divides it into square root of n pieces of size square root of n each in order to get a good cache efficient algorithm that doesn't have any tuning parameters. But almost all of them, since they're recursive, do have this annoying thing that you have to still coarsen the base case in order to get really good performance if you're not doing a lot of work in the leaves of the recursion. So any questions?