CHARLES LEISERSON: It is my great pleasure to welcome Jon Bentley, now retired from Bell Labs. Jon was my PhD thesis supervisor at Carnegie Mellon. I actually had two supervisors. The other one was HT Kung, who is now at Harvard. I guess people flee Carnegie Mellon like the plague or something.

So Jon is, as you know because you've studied some of his work, is a pioneer in software performance engineering. And he's going to talk today about a particularly neat piece of algorithmic engineering sets that centers around the so-called traveling salesperson problem, which is an NP-hard problem. NP-complete problem in fact. And so, without further ado, Jon, why don't you tell us what you've got to say?

JON BENTLEY: As Charles mentioned, I want to talk with you-- I want to tell you a story about a cool problem. This is a problem that I first heard when I was a young nerd-- not much older than this little pile of nerds in front of me-- in high school, the traveling salesperson problem. Who here has heard of the TSP at some point?

I heard about this in high school, one of the things you read about it in the math books. And a few years later, I had a chance to work on it. In 1980, I was doing some consulting, and I said, well, what you need to do is solve a TSP. Then I went home and realized that all of the stuff that I learned about it was sort of relevant but it didn't solve the problem, so I started working on it then.

Our colleague, Christos Papadimitriou, who's been at Berkeley for a long time after being at a lot of other places, once told me the TSP is not a problem. It is an addiction. So I've been hooked for coming up on 40 years now. And I want to tell you one story about a really cool program I wrote. Because this is one of the-- I've been paid to be a computer programmer for coming up on 50 years, since I've been doing it for 48 years now. This is probably the most fun, the coolest program I've written over a couple day period.

I want to tell you a story. Start off with what recursive generation is. Then the TSP, what it is.

Then I'll start with one program, and we'll make it faster and faster and faster. Again, I spend my whole life squeezing performance. This is the biggest squeeze ever. And then some principles behind that.

We'll start, though, with how do you enumerate all the elements in a set? If I want to count-- enumerate the guys between 1 and a hundred, I just count. That's no big deal. But how can I, for instance, enumerate all subsets of the set from the integers from 1 to 5? How many subsets are there of integers from 1 to 5?

AUDIENCE: 2 to the 5.

JON BENTLEY: Pardon me?

AUDIENCE: 2 to the 5.

JON BENTLEY: 2 to the 5, 32. But how do you say which ones they are? How do you go through and count them? Well, you have to decide how you represent it. You guys know all about set representations. We'll stick with bit vectors for the time being.

An iterative solution is you just count-- 0, 1, 2, 3, 4, 5, up to 31. That's pretty easy. But what does it mean to count? What does it mean to go from one integer to the next? How do you go from a given integer to the next one? What's the rule for that?

It's pretty easy, actually. You just scan over all the 0's, turning the-- you start at the right-hand side, the least significant digit, scan over all the 0's, turn it to 1. Oh, I lied to you. You scan over all the 1's, turning them to 0 until you get to the first 0. And then you turn that to a 1. So this one goes to 10. This one goes to 11. This one goes-- that one becomes 0, that one becomes 0. Then it becomes 00100.

So a pretty easy algorithm. You could do it that way. Just scan over all the 1's, turn them to 0's, take that first one and flip it around. But that doesn't generalize nicely. We're going to see a method that generalizes very nicely. This is a recursive solution to enumerate all 2 to the n subsets of a set of size n.

And the answer is this all sets of size m is just put a 0 at this end, enumerate all sets of size m minus 1. How many of these will there be? 2 to the m minus 1. How many of those 2 to the m minus 1? What do they add up to? 2 to the m. But all of these have the 0 at that end, and the one at that end. Everyone see that recursive sketch and how that works?

Here's the example. A period with 0's at this end and you fill it out. You have the 1 at that and you fill that out. If you do that, you notice that in fact we're just counting backwards-- 000, 001, 010, 3, 4, 5, 6, 7. That's the algorithm. And the cool thing is the code is really simple. I could probably write a program like that in most languages and get it correct.

So if m equals 0 in generate all subsets of size m, this doesn't occur at 1. You have a pointer going down the array. Otherwise, set the rightmost bit to 0, generate all subsets recursively, set it to 1, do it again recursively. That's a starting program. If you understand this, everything else is going to be pretty straightforward. If you don't, please speak up.

One thing that-- you people have suffered the tragedy of 14 or 15 or 16 years of educational system that has sort of beaten the creativity out of you and you're afraid to speak up. So even if something-- even if I'm up here spouting total bullshit, you'll ignore that fact and just sort of politely stare at me and nod. But this is important. I want you to understand this. If you don't understand this, speak now or forever hold it. Anyone have any questions? Please, please.

AUDIENCE: What does mean, [INAUDIBLE]?

JON BENTLEY: I'm sorry. Why did we set p to the--

AUDIENCE: [INAUDIBLE].

JON BENTLEY: So here, first I go out to the extreme rightmost and I set it to 0. Then I recursively fill those in. Then I change it from a 0 to a 1 there, and I fill all those in. So this is a program that will go through, and as it enumerates a subset, it will call the visit procedure a total of 2 to the m times, then it comes down to the bottom of the recursion. Thank you, great question. Any other questions about how this works? OK, we'll come back to this.

The traveling salesperson problem. I apologize. I will really try to say the traveling salesperson problem, but I will slip because I was raised with this being the traveling salesman problem. No connotations, no intentionality there, just senility galloping along. It's a cool problem. Abraham Lincoln faced this very problem in the years 1847 to 1853 when he-- everyone here has heard of circuit courts?

Why do they call them circuit courts? Because the court used to go out and ride a circuit to go to a whole bunch of cities. Now people in the cities come to the court. But back in the day, in

1847 to 1853, Lincoln and all of his homies would hop on their horses-- a judge, defense lawyers, prosecutors-- and go around and ride the circuit here.

And so this is the actual route that they rode where they wanted to do this effectively. It would be really stupid to start here in Springfield and go over there, then to come back here, then to go over there back and forth. What they did was try to find a circuit that minimized the total amount of distance traveled. That is the traveling salesperson problem.

We're given a set of n things. It might be a general graph. These happen to be in the plane. But you really-- helicopter service was really bad in those days, so they didn't fly there from point to point. Whether they stayed on roads, what really matters here is the graph embedded in here. I'm going to speak at this. Everything I say will be true for a graph. It will be true for geometry. I'll be sloppy about that. We'll see interfaces, how you handle both, but just cut me some slack for that.

I have actually face this myself when I worked on a system where we had a big mechanical plotter and we wanted to draw these beautiful maps where the maps would fill in dots. They happened to be precincts. Some of them were red, some of them were blue. And you wanted to draw all the red dots first and go around here. And, in fact, the plotter would draw this red dot, then that red dot, then this one, then that one. The plotter took an hour to draw the map.

I was consulted on this. Aha, you have a traveling salesperson problem. I went down. I reduced the length to about 1/10 of the original length. If it took an hour before, how long would it take now? Well, it took about half an hour. And the reason is that the plotter took about half of its time moving around about, 30 minutes moving around, and about 30 minutes just drawing the symbols. I didn't reduce the time drawing the symbols at all, but I reduced the time moving things around from about 30 minutes to about 3 minutes. That was still a big difference.

So I fixed it there. When I worked at Bell Labs, we had drills that would go around, laser drills, move around on printed circuit boards to drill holes. They wanted to move it that way

I talked to people at General Motors at one point. On any day, they might have a thousand cars go through an assembly line. Some of the cars are red, some are white, some are green, some are yellow. You have to change the paint. Some of them have V6, some of them have V8. Some of them are two doors, some of them are four doors.

In what order do you want to build those cars? Well, every time you change one part of the car, you have to change the line. And what you want to do is examine, as it were, all n factorial permutations of putting the cars through and choose the one that involves the minimum amount of change. And the change from one car to another is a well-defined function. Everyone see how that weird TSP is in fact a TSP?

So all of these are cool problems. Furthermore, as a computer scientist, it's the prototypical problem. It's the E. coli of algorithmic problems. It was literally one of the first problems to be proven to be NP-hard. Held-Karp gave a polynomial time algorithm for it. There are approximation algorithms of this. Kernighan-Lin have given heuristics. It's a really famous problem. It's worth studying.

But here is what really happened to me. Here's why I'm standing in front of you today talking about this. My friend Mike Shamos, in his 1978 PhD thesis on computational geometry, talked about a number of problems. One of them was the TSP. And he shows us and he gives an example of this tour. He says, here's a set of 16 points. Here's a tour through them. Here's a traveling salesperson tour through them.

And then he says in a footnote, in fact, I'm not sure if it's a really optimal tour. I applied a heuristic several times. I'm not positive it's the shortest tour. If you wrote a thesis, it would be sort of nice to know what's going on there. Can you solve a problem that was-- this tiny little 16-element problem, 16 points in the plane. Can you really figure out what the TSP is to that? At the time, my colleague, our colleague, a really smart guy, couldn't do it. It was computationally beyond the bounds for him.

Well, in 1997 I came back to this, and I really wondered is it possible now? Computers are a whole lot faster in the 20 years. We were talking about that earlier today. 20 years, computers got faster. A lot of things got better. Have things changed enough so I can write a quick little program to solve this? I don't know. We'll see.

I did that. I talked about it. I gave a talk at Lehigh University 20 years ago. They liked it. They incorporated it into an algorithms class. The same professor gave it time and time and time again. Eventually, he retired. They asked me to come over and give this talk to them.

I can't give a talk about 20-year-old material. Computer science doesn't work that way. So I coded things. I wanted to see how things changed in two years. So this talk is about a lot of things, but especially it's about how has performance changed in 40 years. So that's one of the reasons we were-- one of things we were talking about earlier today.

I could give a bunch of titles for this talk. For you, the title I give is a sampler of performance engineering. It could be-- next week I'll give it at Lehigh. This is their final class in-- one of their final classes in algorithms and data structures. I'm going to try to tie everything they learn together.

It could be all these other things-- implementing algorithms, a lot of recursive generation, applying algorithms, really-- Charles is a fancy dancy academic. He's a professor at the Massachusetts Institute of Technology. I'm just a poor dumb computer programmer, but boy this is a fun program.

What it is not is it's not state-of-the-art TSP algorithm. People have studied the problem for well over a century. They have beautiful algorithms. I am not going to tell you about any of those algorithms for the simple reason that I don't know them. I could look them up in books, but I've never really lived the fancy state-of-the-art algorithms.

And I'm also going to just show you getting the answer. I could analyze it. I've analyzed much of these. If I had another hour or three, I could do the analysis. But I can't, so I'm just going to do the-- show you some anecdotal speeds without really the analysis.

Let's talk about some programs. A simple C program. MAXN is a maximum number, n int is going to be n, the number of cities. I'm going to have a permutation vector, where if I have the tour going from city 1 to 7 to 3 to 4, it says 1734. The distance between cities is going to be given by a distance function. There is this distance d of i, j, the distance from city i to city j.

Here's the first algorithm. What I'm going to do is generate all intact real permutations, look at them, and find the best one. It's not rocket science. The way I'm going to do this is a recursive function where I happened-- I could have done it from left to right. I am a C programmer. I always count down towards 0. So I'm going to count down that way, where all of these cities are already fixed. I'm going to permute these.

Here's the program. To search for m-- all of these have already been fixed. What I'm going to do is if m equals 1, then I check it. Otherwise, for i equals 0 up to m, for each value from 0 to minus 1, I take the ith element. I swap it. swap 3, 7 takes the third and seventh positions and swaps them. I swap that to the final thing. I call it recursively. I then swap it back to leave it in exactly the same state I found it, and I continue.

So here it's going to generate, first, all nine permuta-- put all nine digits in the last position. Then for each one of those, I'll put all eight digits in the last position, and go on down. This is really interesting, important, and subtle. If you don't follow this part, it's going to be difficult. Are there any questions at all about this? Have I lied to you yet about this? You're honest enough to tell me if I have.

AUDIENCE: You're good.

JON BENTLEY: Thank you. Anyone else?

AUDIENCE: [INAUDIBLE].

JON BENTLEY: I'm sorry. Please.

AUDIENCE: Sorry. I'm not really understanding the part that's fixed and what you're permuting, and why is that hard to fix.

JON BENTLEY: So, so far, as I recur down with-- as m moves down, all these are fixed. So I'm going to fix these things, and then I'm going to take care of all these later. So, originally, I'm going to have this array be 0-- if I have a nine-city TSP, it will be 0, 1 2, 3, 4, 5, 6, 7, 8, 9. And first I put 0 in the end and do the rest. Then I put 1 in the end, [INAUDIBLE] 9 in the end, and recur down.

But as the program is progressing, if you stop the program at any time and look at a glance at the program, you can see that, given the value of m, this parameter, the recursive function. So this is a way that I'm essentially building this tree where at the top of the tree the branching factor is 9. At each of those nine nodes, the branching factor is 8, then 7 and 6. It's going to be a big tree.

If n is 10, how big is that tree going to be? What's 10 factorial? Pardon me? When I was a nerd, we used to try to impress people of appropriate genders by going off saying things like 3628800. You can probably guess how effective that was. So 3.6 million. It's going to be a big tree. Any questions about that? Let's go.

When I check things, I just compute the sum there. I start off with the sum being the distance from 0 to p n minus first. Then I go through and add up all the pairwise things and save it. What does it mean to say it? If the sum is less than the minimum sum so far, I just copy those over, change the minsum. And to solve the whole thing, I do a search of size n.

This is a simple but powerful recursive program. You should all feel very comfortable with this. Is it correct? Does it work? Is it possible to write a program with about two dozen lines of code that works? Not the first time. But after you get rid of a few syntax errors, you check it. How do you make sure it works?

I start with n equals 3, and I put 3. Does it give me a tour? Well, it works. Think about it. For 3, 3 factorial, they're all the same tour. That part wasn't hard. 4, now that's interesting. That one works too. This program, in fact, can work.

Is it going to be a fast program? How long will it take if n equals 10? How many seconds? I'm sorry. What class have I stumbled into? Is this in fact Greek Art 303? How long will this take for n equals 10?

AUDIENCE: [INAUDIBLE].

JON BENTLEY: Pardon me?

AUDIENCE: 1 second.

JON BENTLEY: About a second. Pretty cool. For equal 20, how long will it take? A lot longer. Technically speaking, it's going to take a boatload longer.

So what I'm going to do here is-- notice that there are n factorial permutations. You do n of those at each, total of that, on this fairly fast laptop from a few years ago. But now they're all about the same. At 8 seconds, it took that. At 9 seconds, what should be the ratio-- what would you expect to be the ratio between its time at 8 and time at 9?

Well, about a factor of 9, you'd hope. Is 0.5 times 9 about 0.34? Yes, close enough. Here, going down, for 10 it's 4 seconds, 46 seconds. Yes, it's going up by a factor-- so here I've run all my examples. I ran out to 1 minute of CPU time. After that, I estimate. If this one takes 3/4 of a minute, 12 times that is 12 minutes-- 3/4 of that is 9 minutes. For 13, it's 2 hours.

How long should 14 take, ballpark? A day, ballpark. How long will 15 take if 14 takes a day?

AUDIENCE: Two weeks.

JON BENTLEY: Two weeks. How long will 16 take? Eight months. You get the idea. Are you going to go out to 20 for this one? No. Are you going to go out to 16 with this one? Can you just put this into a thesis right now? No. The problem is it's fast for really small values of n. As it becomes bigger-

- how can you make the program faster?

If you wanted to make this program faster, what would you do? What are some ideas? Give me some ideas, please. This is performance engineering. You should know this. Ideas for making it faster. Please.

AUDIENCE: You can start with arbitrary nodes. So if you take the tour, you can start anywhere, right?

JON BENTLEY: OK. So you're saying just choose one start and ignore that, ignore all the others. You don't need to take each random start. Fantastic. A factor of n. My friend in the gray T-shirt just got a factor of n. How else can you make it faster? What ideas do you have? Please.

AUDIENCE: You can start by the distance, and then reject things that were [INAUDIBLE].

JON BENTLEY: Be greedy. Follow the pig principle. If it feels good, do it. Do just local optimization. We'll get to that in a long time, but, boy, would that be a powerful technique. Other ideas, please?

AUDIENCE: Parallelize [INAUDIBLE].

JON BENTLEY: Ah. Parallelize. I would write that out, but the first I would have to do is remember how many R's and L's there are in various places. So I'll write that much. But we'll have a comment on that at the end. People tried that. Sir?

AUDIENCE: Clock the machine.

[STUDENTS LAUGH]

JON BENTLEY: Unlike you, Charles and I, at one point, attended a real engineering school at Carnegie Mellon, formerly known as CIT, Carnegie Institute of Technology. Charles, do you remember the cheer?

CHARLES LEISERSON: The cheer?

JON BENTLEY: The cheer.

CHARLES LEISERSON: I don't know how to cheer.

JON BENTLEY: 3.14159, tangent, secant, cosine, sine. Square root, cube root, log of e. Water-cooled slipstick, CIT. What's a water-cooled slipstick?

AUDIENCE: [INAUDIBLE].

JON BENTLEY: Pardon me?

AUDIENCE: [INAUDIBLE].

JON BENTLEY: It's a slide rule that you run so fast. It has to be water-cooled. So if you can just overclock the machine, just spray it with a garden hose. And as long as it makes over the finish line, you don't care if it dies when it collapses. So, sure, you can get faster machines. We'll talk about that.

How else can you make this faster? Other ideas? These are all great ideas. We'll try it. Let's see some ideas. Compiler optimizations. I just said gcc and I ran it. What should I have said instead? Instead of just gcc?

AUDIENCE: O3.

JON BENTLEY: O3. How much difference will that make? I used to know the answers to all these. [INAUDIBLE] turn on optimization, 10%. Sometimes, whoopee-freaking-do, 15%. Does turning on O3 still make it a 15% difference? We'll see. You could do that.

A faster hardware. I did this 20 years ago. I had all that number there. I'll show you some of those numbers. Modify the C code. We'll talk about all those options, but let's start with compiler optimizations.

With no options there-- how much faster will it be if I turn on optimization? This is a performance engineering class. You should know that thing. Does it matter at all? Is it going to be 15%? Is it going to not matter at all? How much will it matter to turn on optimization?

AUDIENCE: [INAUDIBLE] a lot.

JON BENTLEY: How much is a lot? I know this isn't the real engineering school of CIT, but pretend like this is kind of a semi-- one of the engineering schools. Give me a number for this.

AUDIENCE: More than 15%.

JON BENTLEY: More than 15% Do I hear more than 16%? I was surprised. If I enabled O3, it went from 4 seconds to 12-- I couldn't even time it here. It wasn't enough to time it here. 45 seconds to 1.6

seconds. I can get real times down there. I observed, ballpark here, about a factor of 25. Holy tamale.

On a Raspberry Pi, it was only a factor of 6, and on other machines it was somewhere between the two. Turning on optimization really matters. Enabling that really matters. For now on, I'm only going to show you full optimization. It's cheating not to. But just think about that, a factor of 25.

How else can I make if faster? Two machines. Back in the day, I happened to have some data laying around of running it on a Pentium Pro at 20 megahertz. Nowadays, I had this. How much faster will this machine be 20 years later? Again, pretend like you're at a real engineering school. What will it be? Please.

AUDIENCE: 20 times faster?

JON BENTLEY: 20 times faster? How did you get 20 times faster?

AUDIENCE: Well, the clock speed is 10 times faster.

JON BENTLEY: The clock speed about 10.

AUDIENCE: But I'm guessing that it has much better instructions.

JON BENTLEY: Here's what I found. On this machine, it went from a factor-- there is about a hundred-- these factors, I found, consistently were about, over the 20 years, about a factor of 150. From Moore's law, what would it be if you had 20 years if you doubled every two? That's 10 doublings. What is 2 to the 10th?

AUDIENCE: It's a thousand.

JON BENTLEY: A thousand. So Moore's law predicts a thousand. It's more than a factor of 20. I got a factor of 150 here, which is close to what Moore's law might predict, but there is some slowing down at the end. I'm not at all traumatized by this.

A speed-up of about a factor of 150, where does that come from? My guess is you get about a factor of 12 due to a faster clock speed, and another factor of 12 due to things like wider data paths. You don't have to try to its cram everything into 16-bit funnel. You have 64-bit data paths there. Deeper pipelines, more appropriate instruction sets, and compilers that exploit those instruction sets if O3 is enabled. If O3 is not enabled, sucks to be you. Questions about that? Let's go.

So we have constant factor improvements, external, modern machines, turn on optimization. But a factor of 150 and a factor 25 is a lot. We were starting off with that. That is a good start. Back in the day, if you change things from doubles to floats, it got way faster. From floats, the answer was faster yet. Does that change make much difference nowadays? No. Exactly the same runtime.

One thing that does make a difference is-- this is the definition of the geometric distance. My j is the square root of the sum of the squares of the differences. That's doing an array access, a subtraction, a multiplication, multiplication, two array accesses, subtraction, multiplication, addition, and a square root. That used to take a long time.

If I replace that with a table lookup by filling out this sort of table, the distance for algorithm 2 is just the distance arrays of i sub j. That gave me a speedup factor of 2 and 1/2 or 3. Back in the day, that was a speedup factor of 25.

For you as performance engineers, you have all this intuition. Every piece of intuition you have, that I had, that was really appropriate 10 years ago is irrelevant now. You have to go back and get models to figure out how much each thing costs. But, still, it's another speedup factor of 3 just by replacing this arithmetic with a table lookup.

Algorithm 3. What we're going to do is choose the ones we need to start with. So we'll start at city 1. We'll leave 9, if we have a 9-element problem, in that position, and just search of n minus 1. It doesn't matter where you start. You're going to go back to it, so you can just choose one to start with.

Not a lot of code. Permutations are now that, distance at each. So now you've reduced n times n factorial to n factorial. Algorithm 4 is I'm computing the same sum each time. Is there a way to avoid computing the same darn sum each time? We'll carry that sum along with you.

Instead of recomputing the same thing over and over and over, start off with the sum being 0. The parameters are now m and the distance so far. s Then you just add in these remaining pieces at each point, and you solve it that way. And there it's sort of a nice piece of mathematics. I wish I had the time to analyze it.

I did a spreadsheet where I said, what's the ratio of this? And it started off as 3, 3 and 1/2, 3.6, 3.65, 3.7-- 3.718281828. What does that mean if you see a constant 3.718281828? It's 1 plus e. And once I knew what the answer was, even I, in my mathematical frailty, was able to prove that it's 1 plus e times n factorial. I'm not giving you the proof, but it's very cool. You run across these things.

So here are the four algorithms so far. On an entirely different semi-fast machine, the runtime-- here the real clock times on this machine were 10, 11, 12, 13. Real times in bold are measured times. These other times are approximate estimates. And you can see now that for size 13, you go from taking a fraction of an hour to taking a third of a minute.

We've made some programs faster. That's pretty cool. We feel good about this. This is what we do. Any questions at all? We got to go faster. How do we go faster?

To say precisely, for all these experiments, I took one data set. And if I say that runtime for size 15, I take the first 15 elements of that data set. For 16, I take the first 16 elements. 17, and so on and so forth. It's not great science. I've done the experiments where I did it on lots of random data. The trends are the same. It smooths out some of the curves, but we'll see this. The times are for initial sequence of one random set. It's pretty robust.

But the problem has factorial growth. it started factorial. It's still factorial. What does that mean? Each factor of n allows us to increase the problem size by 1 in about the same time. Faster machine and all that, we can now push into the teens. What does that mean?

You can take Abraham Lincoln's problem, and they got a tour with this length. The optimal tour looks sort of the same on this side, but it's really different over here. Charles, what figure is that? I've mentioned yesterday that if you work on the traveling salesman, every instance you see turns into a Rorschach test.

CHARLES LEISERSON: The first one is a bunny hopping, and the second one is just the head of the bunny.

JON BENTLEY: The bunny head. Everyone see that? Those are in fact the correct answers. He is a psychologically sound human being. Does anyone else want to give their Rorschach answers? A free diagnosis. Absolutely no charge. I'll completely diagnose you. but the bunny hopping and the bunny head are in fact the correct answers for here. We'll see more later.

So Abraham Lincoln, you've solved his problem now. My friend Mike Shamos could solve his problem. Did he get the optimal tour?

Well, over here he got a big part of it. But over here it's really sort of a different character. It's a fairly different character. Is it far off? Yes, about a third of a percent off. So his approach was within a third of a percent.

I've always worked-- I spent much of my career working on approximate solutions to TSPs. Those are often good enough. This algorithm, you can prove-- that he applied-- is within 50%. In the real world, it got within a third of a percent. Wow. But now we can go out and we can solve the whole problem in 16 hours.

If you were writing the thesis and you happened to do this, would it be worthwhile now to sink 16 hours of CPU time into this? You're going to go away for a weekend and leave your machine running. At the time, Charles, when we had one big computer for 60 or 70 people in that department, could we have dreamt about using 16 hours for that? On the very border. If you made it a really mellow background process, it might finish in a week or three.

All of these things change. The computers get faster. They get more available. You can devote a machine to dump 16 hours down this. But can we make it faster yet? Can we ever analyze, say, all permutations of a deck of cards? How many permutations are there of a deck of cards if you take out those jokers? What's that number?

AUDIENCE: 15 zeros?

JON BENTLEY: 1 with 15 zeros after it? It's a big number, 2 to the-- 52 factorial. I want to teach you how big 52 factorial is. People say, that problem is growing exponentially. What does that mean? It's quick is what people usually mean by it.

In mathematics, it's some constant to the n for some defined time period n. Factorial growth-- is factorial growth exponential growth? Why not? Why isn't a factorial exponential?

AUDIENCE: It's more than exponential?

JON BENTLEY: It's more than exponential. It's super exponential. We'll talk about the details here. By Sterling's approximation, you have seen in other classes that log of n factorial is n log n minus n plus O of log n for the natural log. The log base 2 of n factorial is about n log n minus 1.386n. Where have you seen this number before? n log n minus 1--

In an algorithms class, you did a lower bound on a decision tree model of sorting. There were n factorial leaves to sort. A sort algorithm must take at least as much time. So that gives you that bound. And merge sort is n log n minus n, so you're really narrow.

Where else have you seen 1.386n? That's the runtime of quick sort. All these things are coming back together here, because it's the natural log of e-- I'm sorry-- the log base 2 of e. So n factorial is not 2 to the n. It's 2 to the n log n. It's about n to the n. It's faster than any exponential function.

How big is 52 factorial? You guessed 10 to the 15th? Was that--

AUDIENCE: Yes.

JON BENTLEY: OK. If we see here, it's going to be something like 2 to the n log n. n is 52. Log of 52 is about six. So that's 2 to the 300. But there's a minus n term. Maybe 2 to the 250. It's about 2 to the 225, which is 10 to the 67th. That's a big number. How big is it? Let me put it in everyday terms.

Try this after class. Set a timer to count down 52 factorial nanoseconds, 10 to the 67th. Stand on the equator-- watch out where you are-- and take one step forward every million years. Don't rush into this. I don't want you to get all hyper about this.

Eventually, when you circle the Earth once, take a drop of water from the Pacific Ocean, and keep on going. Be careful about this. But this is an experiment. You're nerds. It's OK.

When the Pacific Ocean is empty, at that point lay a sheet of paper down, refill the ocean, and carry on. Now keep on doing that. When you're stack of paper reaches the Moon, check the timer. You're almost done. This is how big 10 to the 52nd is. The age of the universe so far is about 10 to the 26th nanoseconds. 10 to the 52nd is a long time.

Can we ever solve a problem if we look at all 10 to the 52nd options? What do we have to do instead?

AUDIENCE: Quantum computing?

JON BENTLEY: Pardon me?

AUDIENCE: Quantum computing.

JON BENTLEY: Quantum computing. OK. That's great. And I have a really cool bridge across this river out here that I'll sell you after class. Let's talk about that. Is there a nice quantum approach to this problem? Maybe. Maybe you could actually phrase this as an optimization problem where you could maybe get some mileage out of that. But we'll see.

So one approach is quantum computing. What's another approach? What are we going to have to do to make our program surmount this obstacle? Please.

AUDIENCE: Limit the search space?

JON BENTLEY: Pardon me?

AUDIENCE: [INAUDIBLE].

JON BENTLEY: We're going to have to limit our search space. We're going to have to prune the search space. That's the idea. Let's try it. Here's a cool problem. I was at a ceremony a few weeks ago. A friend of mine said here's this cool problem that his daughter just brought home from high school. How do you solve it?

Find all permutations of the 10 integer-- the nine integers 1 through n such that each initial substring of length m is divisible by m. So the whole darn thing is divisible by 9. Is any permutation of integers 1 through 9 divisible by 9? Well, they all sum up to numbers divisible by 9. You work that. Is it divisible-- are the first eight characters divisible by 8?

But let's start with an easy one. If you were doing it for size 3, 321 works. Is 321 divisible by 3? Is 32 divisible by 2? Is 3 divisible by 1? Thinking, then, it works. Is 132 divisible by 3? Yes. Is 13 divisible by 2? [MAKES BUZZER SOUND] That doesn't work.

So we're going to try to solve this problem. My friend Greg Conti, a really great computer security guy, gave me this problem. How do you solve it? How would you solve this problem? If this high school kid says, here's a problem I brought home from school, how do I solve it? What would you do? Ideas? I'm sorry. Please.

AUDIENCE: Yes. You could write a program where the state could be [INAUDIBLE]. Or actually just like a subset [INAUDIBLE]. Then you iterate over [INAUDIBLE].

JON BENTLEY: Great. So there are two main approaches. One is write a program. So you can either think or you can compute. Who in this room enjoys writing programs? Who enjoys thinking? Oh, that's an easy call. What's the right approach here?

Well, the right answer is you think for a while. If you solve it in the first three minutes, don't write a program. If you spend much more than five minutes on it, let's write a program and see what we learn from the program. We'll go back and forth. Never think when you should compute, never compute when you should think. How do you know which one to do? Try each. See which one gets you further faster.

If you write a program for this, What are the basic structures you have to deal with? You have to deal with nine-digit strings that are also nine-digit numbers. What's a good language for dealing with that? What would you-- if you had to write a program to do this, what language would you choose? We'll see.

How do you generate all intact real permutations of the string? Well, I hope you can see this. Here's the way that I chose to do it. I chose to have a recursive procedure search. And I'm going to have right be the part that's already fixed, left be the part that you're going to vary. I could've done it the other way, but I'll choose to do it this way.

I start with left equals that, right equals that. I end when the left is empty. So I have to recur down, just like we've been doing so far, but I'm going to do that with strings instead. And if I get to the call search of 56-- of 356 with 421978-- these are all fixed-- I'll take each one of these in turn, 3, 5, and 6, put it into here. So I'll call search of 56 with that, search of 36 with that, search of 35 with that. Everyone see how that works?

How long will the code be in your favorite language? Here's the code in my favorite language. Has anyone here ever used the AWK programming language, written by Aho, Weinberger, and Kernighan? They observed that naming a language after the initials of the authors shows a certain paucity of imagination. But it works.

So a function search of left, right, that, if left equals 0-- is null, I'll check it. Otherwise, what will I do here? The details don't matter. For i equals 1 up to the length of the left-hand side of the string, search the substring at the left starting at 1, going for i minus 1, concatenated with the substring at the left starting at i plus 1. And then take the substring in the middle, put it out in front of the right. Do that for all i values. Any questions about that? The details don't matter. It's not a big program.

If I do this, and at the end, for i equal 1 to length, if the substring of the right mod i is nonzero, then return. If it's not that, you print it out. If I run this program, how long, ballpark, will this program take for 9 factorial, ballpark? What was your answer before?

AUDIENCE: A second.

JON BENTLEY: A second. Great. Well, we'll recycle that. Reduce, reuse, recycle. We'll recycle your answers. If I call it originally with that string, it takes about 3 seconds. And it found that there was-- it searched all 9 factorial, 362880, 362,000 strings, and found only one string there that matches that. Whoops.

Are these divisible by 9? Well, they sum to a multiple of 9, sure. Is the string that ends in 72 divisible by 8? Yes, that works. 7, I'm not going to bother with. All the way down, is 38 divisible by 2? Is 381 divisible by 3? This one works. That's a pretty cool problem for a high school afternoon.

Is 3 seconds fast enough? Yes. The trade-off of thinking and programming. Write the darn program. You're done. It's cool. If you wanted to make it faster, how could you make it faster? That's what this course is all about? Always think about how you could make things faster. Please.

AUDIENCE: Well, if you just stop searching once you know one number isn't going to work.

JON BENTLEY: How early can you stop searching? That's great. So you could get constant factor speedups. Like don't check for divisibility by 1 at the end. You can change language, all that. But those are never going to matter.

The big win is going to come from pruning the search. How can you put in the search? Any winning string must have some properties of this string. What are some properties that that string has that you can check for early? Please.

AUDIENCE: The second from the left [INAUDIBLE] 2, 4, 6 or 8.

JON BENTLEY: The eighth position has to be a multiple of 2. Furthermore, if you really think about it, you can get more than that. It has to be divisible by 4. So an even number has to be in the eighth position. Anywhere else you're going to need an even number?

AUDIENCE: [INAUDIBLE].

JON BENTLEY: This position has to be even, that has to be even, that has to be even, that has to be even. In general, what's the general rule?

AUDIENCE: All the even positions [INAUDIBLE].

JON BENTLEY: Every even position has to contain an even number. There are four even numbers, there are five odd numbers. What other rule might you come up with?

AUDIENCE: The fifth position has to be 5.

JON BENTLEY: OK. Every odd position has to be an odd number. And, in particular, the fifth position has to be a 5. So those are a few rules. Even digits in even positions, odd digits in odd positions, digit 5 in position 5. Three simple rules. You can test those easily. The code is pretty straightforward.

Will that shrink the size of the search space much at all really? How big was the search space before? 9 for the first one. Now how big is the search space? For the first, if you just had the three rules-- evens going evens, odds in odds, and 5 in the middle-- how many choices do you have for the first one?

AUDIENCE: For the first, we have [INAUDIBLE].

JON BENTLEY: Four choices. For the second one, you have?

AUDIENCE: [INAUDIBLE].

JON BENTLEY: It can't be a 5. It has to be an odd number, not a 5. You have a 4. So it's 4 by 4 times 3 times 3 times 1 times 2 times 2 times 1 times-- everyone see that? We've reduce the size of the search space from a third of a million to half a thousand.

Isn't it going to be a lot of hassle to code that? I mean, is it going to take a major software development effort to code that? Well, yes, if you define that as a major software development effort.

If the parity of the string length is equal to the parity of the digit, then you can continue. If you don't have these things, you can't continue. Three lines of code allow you to do this. That's the story. Factorial grows quickly. You can never visit the entire search space. The key to speed is pruning the search. We're doing just a baby branch-and-bound, it's called.

Some fancy algorithms can be implemented in little code. That's our break. We've learned a couple of things. We're going to go back into the fray. Any questions about this diversion before we go back to the TSP? These are important lessons. We'll try to apply them now.

I got great advice yesterday from people about how to do this. And I seem to have skipped-- OK, here it is. I've got it. How do we prune our search? Here we had these conditions. How can we prune the search? How can I make the program faster? What's the way I can stop doing the search?

Simplest way, don't keep doing what doesn't work. If the sum that you have so far is greater than the minimum sum, by adding more to it, are you going to make it less? What can you do? You can stop the search right there.

Is the resulting algorithm going to be faster? Maybe. It's a trade-off. I'm doing more work, which takes some time, but I might be able to prune the search space. The question is, is this benefit worth this cost? What do you think?

Well, on the same machine, algorithm 4 at size 12 took 0.6 seconds. Now it's a factor of 60 faster, a factor of 40 faster, a factor of 100 faster. Just by-- if it doesn't work, if you've already screwed up, just don't keep what doesn't work. That makes the thing a whole lot faster. Everyone see that? That's the first big win.

Can we do even better than that? Is there any way of stopping the search with more information other than, whoops, I've already gone too far? Please.

AUDIENCE: If the nodes you visited previously--

JON BENTLEY: Wait. Command voice. Speak loudly,

AUDIENCE: If the nodes you visited previously are the same, like the same subset but a different word than a search you've done before, then the answer [INAUDIBLE].

JON BENTLEY: That's a really powerful idea that Held and Karp used to reduce it from n factorial time to n squared 2 to the n time. We'll get to that. That's really powerful, but now we're looking for something not quite that sophisticated. But that's a great idea.

Can I somehow prune the search if a sum plus a lower bound on the remaining cities is greater than the minimum sum? What kind of lower bound could I get? Well, I could computed a TSP path through them. That's really powerful. That will give me a really good bound, but it's really expensive to compute.

So I could-- if this is a city I've done so far, I could compute a TSP path to the rest, which might in this case looks like this, and hook it up. That's going to be a really powerful heuristic, but it's going to be really expensive to compute. On the other hand, I could take just the distance between two random points. I'm going to choose this point and this point I happened to get the diameter of the set.

And that's a lower bound. It's going to be at least that long. And it's really cheap to compute, but is it very effective? Nyah. So the first choice is effective but too expensive. The second point is really inexpensive but not very effective.

I could also compute the nearest neighbor of each city. From this city, if I just compute its nearest neighbor among here, so it's that. This one is that. That one has its own nearest neighbor. I could compute these distances. And that's pretty inexpensive to compute, and it's a pretty good lower bound. That would work.

Who here knows what a minimum spanning tree is? Good. What I'll do here is I'll take here a minimum spanning tree. In cities, a tree is n minus 1 edges. This tree is n minus 1 edges. This is a spanning tree because it touches-- it connects all cities. And, furthermore, it's a minimum spanning tree, because, of all spanning trees, This one has the minimum total distance.

Now, the tour is going to be less-- or greater in distance than the minimum spanning tree. Why is that? If I get a tour of this, I can just knock off the longest edge. And that now becomes a minimum spanning tree. So the minimum spanning tree is a pretty good bound, a lower bound. It's cheap to compute.

Who here has ever seen an algorithm for computing minimum spanning trees? Good, good. Some of you are awake in some of their classes. What are the odds of that? I mean, what an amazing coincidence.

So what we'll do is say now that a better lower bound is to add the minimum spanning tree's remaining points. So I change this program to if sum plus the MST distance. And now I'm going to do a trick. I'm going to use word parallelism. I'm going to have the representation of the subset of the cities as a mask, a bit mask in which if the appropriate city is on, the bit is turned on. Otherwise, it's turned off.

And I just OR bits into it, and say if I compute the minimum spanning tree of this set, I can cut the search and return. And then I just compute the MST and bring this along with me, turning things off and on in the bit mask as I go down. Pretty straightforward. How much code will it cost to compute a minimum spanning tree? Ballpark? Yes.

AUDIENCE: 30 or 20 lines of code.

JON BENTLEY: About that many lines of code. This is the Prim-Dijkstra method. It takes quadratic time. For computing an MST of n points, it takes n squared time. It's quite simple. You can do it in e log log b time. But this is a simple code. It's pretty straightforward.

Will this make the program run slower or faster? What would the argument be that it might run slower? Holy moly. At every node I'm computing an MST. That takes long time and I will run slower. What's the argument to be that it might run faster? Yes, but I'm getting a much more powerful pruning. Is it worth it?

I should point out that I'm only showing the wins here to you. When I redid this myself, I went down a few wrong paths. I wish I would have documented them better now. But I might go back and see if I can find them. That would be a good thing.

But here it is. It used to take 17 seconds. Now it takes-- or 4 seconds. Now it takes 0. You like algorithms to take 0 seconds. You'd like to live in the rounding error. 4.40 to 0.2. Down here, this program is not only faster, it's a boatload faster.

And so now we can go out in this. And notice here that as you go out, the times usually get bigger, but they are bumpy, from 2.4 seconds to 0.7 seconds, to 1.8 seconds. It's because you're doing that one thing. It's just the matter of the geometry. The times that were originally really smooth now turn bumpy. I've done experiments where I do 10 different data sets, randomly drawing each one, and it's a nice smooth line. But I missed doing it here to be easy.

Before we can go out to size 17. Now we can go out to size 30. Wow. How cool is that? That's pretty powerful. Can I make this-- please.

AUDIENCE: So is it possible that the [INAUDIBLE] is chosen in such a way that this thing doesn't actually prune any bad permutations?

JON BENTLEY: That's absolutely true. And I've tried this both on random point sets. I've tried it on distance matrices. I've tried on points where they're randomly distributed around the perimeter of a circle. And so this could be a lot of time. Almost always, it's pretty effective. Again, if I had more time, I'd talk about it. But in fact we're going to go until 3:45, Charles?

CHARLES LEISERSON: 3:55.

JON BENTLEY: 3:55? When the big hand is on the 11? Oh. Sucks to be you.

[STUDENTS LAUGH]

I profiled this bad boy, and it shows that most of the time is in building minimum spanning trees. Your fear that it might take a long time, it might make it slower, has a basis. That's where all the time is going. How can I reduce the time spent in building minimum spanning trees? As I search this-- please.

AUDIENCE: Maybe don't do it every time?

JON BENTLEY: I could do some incremental minimum spanning trees because they change a lot. And so there are several responses. One is whenever you're building something over again, rather than building it from scratch, see if you can do an incremental algorithm, where you just change one bit of the minimum spanning tree. If I just add one edge into the graph, always try an incremental algorithm. That's cool. That's one sophisticated approach.

What is one-- what was another pretty idiot simpler approach? Whenever you compute something over and over again, what can you do to reduce the time spent computing it?

AUDIENCE: Store it?

JON BENTLEY: Store it. Do I ever compute the same MST over and over again? I don't know. I think maybe it's worth a try. So what I'll do is return of caching. Store rather than recompute. Cache MST distances rather than computing them.

The code looks like this. The new mask is that. If the MST distance array is less than 0, initialize everything to 0. Here I'm just going to store them all in a table of size 2 to the n. I can do direct indexing. If it's less than 0, compute it, fill in the value. If sum plus that, return. Not much code.

But do you really want to store-- to blast it out and to use a lazy-- I'm using lazy evaluation of this table here. Only when I need it do I fill in a value. That's not effective. Rather than storing all 2 to the n tables, what can I do instead? What's our favorite data structure for storing stuff?

Hash table. A cache via hash. So the key to happiness. You can write that down too. Store them in a hash table. If sum plus MST distance lookup-- oh, but I have to implement a hash table now.

How much code is that going to be? Ballpark? What does it cost to build a hash table? Roughly. Come on. Yes. About that many lines.

So just go down the hash table. If you find it, return it. Otherwise, make a new node, compute the distance, put it in there, fill in the values, and you're done. Is it going to be faster? Oh, we'll see. Who reads xkcd on a regular basis? The rest of you are bad, bad, bad people, and you should feel very guilty until you go to xkcd.com and start reading this on a regular basis.

I mean, like wow. This is two deep psychological insights in one lecture for no additional fee. Sir.

CHARLES LEISERSON: Were you resolving collisions by chaining them?

JON BENTLEY: Right, by chaining, yes.

CHARLES LEISERSON: Why bother? Why not just store the place value and keep a key to make sure that it's the value associated with the one that you want?

JON BENTLEY: That is a great question, and the answer is left as an exercise for the listener. We've got about 20 minutes, Charles.

CHARLES LEISERSON: Code, less code.

JON BENTLEY: It would be, yes. And it's well worth a try. All of these things are empirical questions. One thing that's really important to learn as a performance engineer is that your intuition is almost always wrong. Always try to experiment to see.

It's a great question. When I get home, I'll actually-- when I leave here, I'm going to go up to try to climb Mount Monadnock. Who here has ever climbed Mount Monadnock? Yes. I finished climbing all 115 4,000-foot peaks in the Northeastern US last year. I've never climbed Monadnock. I'm really eager to give it a try tomorrow.

xkcd. Brute force n factorial. The Held-Karp dynamic programming algorithm uses the grown-up version of dynamic programming for n squared 2 to the n, but even better. Algorithm 6 looks like that if I cache the TSPs. Does it have to be faster? No. Is it faster? Oh, by about a factor of 15 there. By about a factor of 25 there, 26 there.

You can go out now much further, 6 and 8. So we've done that. Is there any other way to make this program faster? We've pruned the search like crazy. Any other way to make it faster? Please.

AUDIENCE: [INAUDIBLE].

JON BENTLEY: I forget what happens at 39. Let's see. At 39, it went over a minute. And, like I said, this thing goes up and down. I guess it just hit some weird bumps in the search space.

That's something else. The first algorithm is completely predictable. The other algorithms, you have to get more and more into analysis. And now the times go up and down. There is a trend. And, basically, I'm taking an exponent and I'm lowering-- I turned it from super exponential to exponential, and then I'm being down on the exponent right now.

Can you make this run faster? What we're going to do is take this idea of a greedy search. I've can have smarter researching. Better than a random order, I'm going to do a better starting tour. And what I'm going to do is always at each point sort the points to look at the nearest one to the current one first. Start with a random one. Then for the next one, always look at the nearest point first, then the second nearest, the third nearest, et cetera.

So I'll go in that order. That should make the search smarter, and that should guide me rather quickly to the initial starting tour. Rather than just a random tour, I'll have a good one that will give me a better prune of the search space. Will that make a difference? We'll have to include a sort. I'll get two birds with one modification.

By a really dumb insertion sort, which takes up that many lines of code, I'll visit the nearest city first, then others in order. If I do that, here it's a factor of 2, there it's a factor of 8, a factor of 4. But it seems to work. It gives you some-- as you go out especially. I can now go out further. I lied. I didn't stop my search at 60 seconds there. But I can now go up further, and it seems to be a lot faster.

So in 1997, 20 years ago, I was really happy to get out to 30. The question now is, in 20 more years, how much bigger can I go? If I just depend on Moore's law alone, in 20 years a factor of a thousand. At 30, 30 times 31 times 32, that's a factor I can go up by Moore's law. With a

[INAUDIBLE] algorithm, it would give me two more cities at this size in 20 years.

Can I get from 30 on to anything interesting by combining Moore's law, and compiler technology, and all the algorithms. How far can I go? Well, I was going to give a talk at Lehigh. So I could go out-- in under a minute, I could go to the 45-city tour. Charles answered this yesterday, so he is completely clear.

Rorschach test. Who's willing to go out-- what do you see there?

AUDIENCE: A puppy.

JON BENTLEY: Dancing doggy. That was my answer, dancing doggy. I like that a lot. That's the obvious answer. But Charles-- and this shows a profoundly profound mind. Professor Leiserson, what is this?

CHARLES LEISERSON: This is a dog doing his business [INAUDIBLE].

JON BENTLEY: OK. So any Freudians, you feel free to go to town on that one. 45-city tour, it's pretty cool. Dancing doggy. How far can it go? I got out to 45 in under a minute.

46-- I broke my rule of this-- I went over the minute boundary. This was my Thanksgiving 2016 cycle test. I was just going hog wild. I was willing to spend the-- I had to give a-- I was doing this Wednesday night. I had to give a lecture on Monday. A hundred hours of CPU time. How far can I go?

47. Yes. Yikes, factor of 5. When do I think? When do I run? Should I go back and [? work on it. ?] 52-- wouldn't it be sweet to be able to go out to 52 factorial? Wouldn't that be cool? 48-- that's not bad. That's looking pretty good there, actually.

Oh, ouch, ouch. That's going to take a-- so that about 2 hours right there. But 50, whoo, edge of my seat. The turkey was smelling good, but 51. And can I get to 52? Will it make it? Will I have to go back to my-- whew. 3 hours and 7 minutes.

By combining all of these things, we're able to go out to something that is just obscene. 52 is obscenely huge. We're able to get out there by a combination of all of these things, of some really simple performance engineering techniques. If you're going to work on a real TSP, read the literature, study that. I hope we can come across some things that I've written about approximation algorithms.

But if you really need them, forget the approximation algorithms because they're too short. There's a huge literature. I haven't told you any of that. Everything that I've done here are things that you, as a person who has completed this class, should be able to do. All these things are well within your scope of practice, as we say. You will not be sued for malpractice.

How much code is the final thing? About that much. You build an MST. You had a hash table. Charles points out you could nuke three or four of those lines. You have the sort here. Altogether about 160 lines.

Where have we been? We started we could get out to 11. Store the distances. Out to 12. Fix the starting city. That was a big one. Accumulate distance along the way. These were all good. But then by pruning the search, we started making the big things. Add the MST, store the distances in a hash table, visit the cities in a greedy algorithm. Each one of these gave us more and more and more power as we went out there, till we're finally able to go out pretty far.

There are a lot of things you can do. Parallelism, faster machines, more code tuning, better hashing. That malloc is just begging to be removed. Better pruning, a better starting tour, better bounds. I can take the MST length plus the nearest-- that's why I do this MST-- plus the nearest neighbor to each of the ends. I can get that. Would that make a big difference? Empirical question. Easy to find out.

Can I move by pruning tests earlier? Better sorting. This is really cool. Can I maybe just sort once for each city to get that sorted list, then go through that, precompute and sort, and select the things in order? Is that going to be a win in this context? The main ideas here are caching, precomputing, storing this, avoiding the work. Can I change that n squared algorithm to just a linear time selection? All of these things are really fun to look at.

I've tried to tell you about incremental software development. I started off with around 30, 40 lines of code. It grew to 160. But altogether all the versions come to about 600 lines of code. You've now seen more than you need for one life about recursive generation. It's a surprisingly powerful technique if you ever need to use it. No excuses now. You're obligated to build it immediately.

Storing precomputed results, partial sums, early cut-offs. Algorithms and data structures. These are things that sounded fancy in your algorithms class, but you just pull them out when you need them. Vectors, strings, arrays and bit vectors, minimum spanning trees, hash tables, insertion sort. It's easy. It's a dozen lines of code here. two dozen lines of code there.

I believe that Charles may had mentioned earlier that I wrote a book in 1982 about code tuning. At the time, you did these in the smaller programs. Now compilers do all that for you. But these ideas-- some of these ideas still apply. Store precomputed results. Rather than [INAUDIBLE] elimination in an expression, you now put interpoint distances in a matrix or a table of MST lengths.

Lazy evaluation. You compute the n squared distances eagerly but only the MSTs that you need. Don't bother computing them all. That's essentially what Held and Karp does. Short-circuiting monotone functions, reordering tests, word parallelism. These are the things that you as performance engineers can do quite readily.

I had a lot of tools behind the scenes. I wish I could come back and give you another hour about how I really did this with the analysis and the tools that I used. I had a driver to make the experiments easy, a whole bunch of profilers. Where is the time really going here? What should I focus on? Cost models that allowed me to estimate those, how much does an MST cost. A spreadsheet was my lab notebook for graphs of performance, all sorts of curve fitting.

But these are the main things I wanted to tell you about. The big hand is getting about nine minutes away from the 11. Professor Leiserson, is there anything else that these fine, young semi-humanoids need to know about this material?

CHARLES LEISERSON: Does anybody happen to see any analogies with the current project 4. Maybe people could chat a little bit about where they see analogies [INAUDIBLE].

JON BENTLEY: I don't know it, but one of my first exposures to MIT was when I had Donovan as a software systems book, and it was dedicated to 6.51 graduate students. I saw that I thought, that bastard. I'm sure that the six students really worked hard on it, but to say that the seventh student worked only a little much more over halfway and then to be so precise, that's just cruel. What a son of a bitch that guy had to be.

So I don't know what project 4 is, but is it Leiserchess? Oh, great. I know what that is. So what things-- have you used any of these techniques? Did you ever prune searches? Did you ever store results? What did you do in project 4? You're delegating this. That's a natural leader right there.

AUDIENCE: We talked about search pruning-- we already have--

JON BENTLEY: Speak up so all of them can hear, please.

AUDIENCE: Commander voice. So we already have --everybody in this room knows-- alpha-beta pruning. [INAUDIBLE] It's got search. I don't know how many teams are already working on search but at least my team is working on changing order representation first. So we haven't gotten into pruning search yet, but that's definitely on the horizon [INAUDIBLE].

JON BENTLEY: Is there anyone here from the state of California? I was born in California. When you hear alpha beta, apart from the search, what do you think of?

AUDIENCE: The grocery store.

JON BENTLEY: There's a grocery store there called Alpha Beta. And when Knuth wrote a paper on that topic, he went out and bought a box of Alpha Beta prunes that he had in his desk. So he was an expert in two senses on alpha beta pruning. So good. Other techniques? Please.

AUDIENCE: The hashing. There's one function [INAUDIBLE] takes a long time, and suggested maybe you could somehow keep track of the laser path with a hash table [INAUDIBLE].

JON BENTLEY: Great. Did you resolve collisions at all? Or did you just have one element there with a key? How did you address the problem that Charles mentioned of-- what kind of hashing did you use?

AUDIENCE: So we haven't used caching yet.

JON BENTLEY: Other techniques?

CHARLES LEISERSON: Yes. That's a classic example of the fastest way to compute is not to compute at all.

JON BENTLEY: In general, in life no problem is so big that it can't be run away from. These things about avoiding work and being lazy are certainly models for organizing your own life. The lazy evaluation really works in the real world. Other questions? Was that a question or a random obscene hand gesture?

AUDIENCE: [INAUDIBLE].

JON BENTLEY: Please.

AUDIENCE: [INAUDIBLE] state-of the-art [INAUDIBLE]?

JON BENTLEY: Oh. That's a great question. I worked on this problem a lot in the early 1990s with my colleague David Johnson, who literally wrote the book on NP-completeness. An MIT PhD guy. We were really happy we're in-- at the time, in a couple of hours of CPU time we could solve 100,000 city problems to within a few percent. We were able to solve a million city problems in a day of CPU time to within a few percent.

And we were ecstatic. That was really big. So we could go out that big to within a few percent. If we worked really, really hard, we can get 10,000 problems down within a half a percent. But if you want to go all the way to have not only the optimal solution but a proof that it's optimal, for a while people bragged about we finally solved that problem. This will let you see about what was done. We solved the problem of all 48 state capitals.

So for a while that was the state of the art. And then that number has crept over time. And now you can get exact solutions to some famous problems into the tens of thousands by using lots and lots of really clever searching the branching down with really clever lower bounds to guide it up. And you at one point get a tour, and you can make that tour. But then you get a proof of a lower bound along with it to do that.

CHARLES LEISERSON: Hey, old man, I want to let you know that there are actually now 50 states in the union.

JON BENTLEY: No. What time did this happen? You can tell that I am much, much, much older than Charles, and he never lets me hear the end of it. I trust that the rest of you-- this is like the third free deep psychology insight, is be kind to old people ignore the example that the kid over there sets and show some class and respect to me and my fellow geezers.

CHARLES LEISERSON: We were both born in 1953.

JON BENTLEY: But I was born in the good part of 1953. In particular, I was born before Her Majesty the Queen of England assumed the throne. Can you make the same claim?

CHARLES LEISERSON: I cannot make the same claim.

JON BENTLEY: I'm sorry. He can, but only because he's a sneaky bastard. Can you make it truthfully is the question that I should have asked. Other questions?

This class can be very important. Like I said, I spent the past almost half century as a working computer programmer. The majority of that thing I've done most is performance engineering. It's allowed me to do a number of really interesting things. I've been able to dabble in all sorts of computational systems, ranging from automated gerrymandering.

Every time you make a telephone call in this country, if it's, say, a call from inside an institution like a hospital of a university, it uses some code that I wrote, some of the performance things. If you make a long-distance call, it uses code that I wrote. If you've ever used something called Google internet search or maps, or stocks or anything else, that uses some algorithms I've done.

It's incredibly satisfying. It's been a very, very fulfilling way for me to spend a big chunk of my life. I am grateful. It's allowed me to make friends, whom I've known for almost half a century, and to our wonderful dear people. And it's been a great way for my life. I hope that performance engineering is as good to you as it has been to me. Anything else, professor?

CHARLES LEISERSON: Thank you very much, Jon.

JON BENTLEY: Thank you.

[STUDENTS APPLAUD]