[1] The problem of the Towers of Hanoi is one that programming texts love for showing recursion. We're ok with recursion, and that's not what I want to use it for. I want to show how you can calculate and prove the space needs of a program, and I'm just using the towers of Hanoi as an example program for that. In case you don't know the problem, I'll explain it. There are three towers, or posts, which we'll call post A, post B and post C. And there are n disks. Each disk has a hole in the center, and is sitting on a post. The picture shows the disks edge on, and shows 3 disks, but it could be any number of disks. But it's always 3 posts. The disks have different sizes, and they're numbered, with the smallest disk, disk 0, on top, and the biggest disk, disk n minus 1, on the bottom. The problem is to move all the disks from post A to post B. You can move only one disk at a time, and you can never put a bigger disk on top of a smaller disk. Post C is needed as temporary storage. [2] I'll play the game with the 3 disks shown here. First we have to [3] move the small disk to post B, [4] there, then [5] move the middle disk to post C, [6], then [7] the small disk to C [8], and now we can [9] move the big disk to its final place on [10] B. Then [11] move the little disk out of the way [12], then we can [13] put the middle disk where [14] we want it, and [15] finally move the little disk [16] where it should be. With 3 disks it's not so hard, but try it with 6 disks. You don't need to buy anything. Just use pieces of paper. They don't even need to be different sizes; just put numbers on them.

[17] We need a program that plays the game. I'll call it move pile, and it has 3 parameters. From is the post the pile starts on, to is the post the pile ends on, and using is the other post. [18] If n, the number of disks to be moved, is 0, then there's nothing to do, so that's ok. Else [19] decrease n by 1, which means forget about the bottom disk for a moment. Now let's get all the other disks off the bottom disk. We do that by calling [20] move pile, which we're in the middle of writing. It moves n disks, that's the decreased n, one at a time, never putting a bigger disk on top of a smaller one, from the post where they are, to the temporary post, making use if necessary of the remaining post. Now we've uncovered the bottom disk. So [21] call Move disk, which causes a robot arm to move a disk from from to to. Now the bottom disk is where it should be. We just have to get the rest of the disks from the using post to the to post. So [22] call move pile again. Now all the disks are where we want them, but there's one thing left to do: [23] put n back to what it was. Why bother to put n back? The reason is that [24] this call to move pile needs to know the correct value of n in order to move the right number of disks. And so [25] this call has to maintain the value of n, not destroy it. If it changes n, it has to put it back so the following calls will have the right value of n. That's why we have to [26] put it back to what it was. We would discover this if we defined move pile and move disk formally, and tried to prove the refinement. But I won't bother with that because that's not my point today. Today I want to calculate resource usage, and I'll start with [27] time. I've replaced move pile with the time it takes, [28] both the main specification, and the two recursive calls. For the measure of time I'm not using real time, and I'm not using recursive time. [29] I'm counting disk movements. So I've replaced move disk with t gets t plus 1. [30] To prove this refinement, there are 2 cases. Here's [31] the first case. We have to [32] expand the assignment and ok, and then we can use [33] context to make some substitutions in the consequent, and since 2 to the power 0 is 1, the consequent [34] simplifies to true, and so the whole thing is true. Now the [35] other case. We can use the substitution law repeatedly, so I'll [36] write the bottom line, expanding the assignment, and then [37] we should replace t by t plus (2 to the n) minus 1, so [38] here we go, and then [39] we should replace t by t plus 1, like [40] this, and then [41] replace t by t plus (2 to the n) minus 1 again, like [42] this, and now we can [43] simplify by canceling the minus 1 and plus 1 [44], and we can [45] add (2 to the n) plus (2 to the n) and get [46] 2 to the (n plus 1). Now [47] replace n by n minus 1, making [48] this, which is [49] exactly what we want. That proves the execution time is (2 to the n) minus 1, and it means don't play the game with more than about 6 or 7 disks. The original story said that in Hanoi there

are these 3 towers with 64 disks, and the monks started moving them at the beginning of time, and when they're done it's the end of time. I think they got that about right.

Now [50] let's do space. How do we calculate space? We just add a space variable. It's a state variable, so there's s for the initial space occupied, and s prime for the final space occupied. But it's not stored in memory, and assignments to s are not executed by the computer. It's like the time variable. It's there for us to calculate the memory space that is occupied by the computation. And, like the time variable, we cannot assume that it starts at 0. Any program can be part of a larger program. And the recursive calls in our computation start in the middle of our computation. [51] The specification here just says that the space occupied is the same in the end as it was in the beginning. In the jargon, there are no space leaks. [52] The 2 recursive calls begin by pushing a return address onto a stack, and they end by popping the return address off the stack. So I've put s gets s plus 1 just before the call, and s gets s minus 1 just after the call. I suppose [53] the disk movement doesn't increase or decrease the space, and I'm not counting time right now, so it does nothing of interest. This refinement is easy to prove, but not very informative about space usage. One thing we really do want to know is [54] what is the maximum space ever used during the computation, because that tells us whether we have enough memory. We need variable m to keep track of the maximum space ever used. And [55] wherever s is being increased, we need m gets max of m and s in case it's a new maximum. We don't need anything where s is being decreased. What I really want to say about the maximum space is [56] that it's n. But since s might not be 0 to start with, I have to say it's [57] s plus n. And since m might be even bigger than that to start with, I have to say [58] it's max of m and s plus n. In case it helps, I can [59] assume m is greater than or equal to s at the start. The maximum space ever used is always at least as much as the current space. It's a bit complicated when all we really wanted to say is that the maximum is n. The proof's in the textbook, and I won't take your time with it here. You might like to try proving it before looking at the textbook. Start by simplifying the 2 long lines that are identical.

I want to finish this lecture by talking about [60] the average space occupied by a computation. This graph is supposed to be showing space usage going up and down over a period of time. To find the average space used, you take the space time product, which is the area under the curve, and divide by the time. At the [61] start time t, the space usage is s. And at the finish time t prime, the space usage s prime is equal to s. We need the space time product, so we'll use variable p. [62] The area here in this little bit, is the space time product accumulated before the start of our computation. So that's [63] p. The area in [64] this rectangle is the space time product that accumulates during our computation but due to the space that was already occupied before our computation. It's the space time product that's not our fault. And it's [65] s times the time, which is (2 to the n) minus 1. The area in [66] this part is our fault. That's the area we want to find, and divide it by the time to get the average space used by our computation. I tried several different formulas before I got it right, which is [67] this formula. I knew it had nothing to do with s and s prime. I knew it had to be less that n times 2 to the n minus 1, because that's the maximum space times the time. But I started with that, and then tried to prove it, and when the proof didn't work, I made an adjustment to the formula that looked like it might help the proof, and I did that same thing 3 or 4 times before it worked. The [68] specification says that the final value of p is whatever p started with plus the area of the rectangle that isn't our fault, plus the area above that, that is due to our computation, and variable s ends as it began. The [69] increase in the space time product occurs where time increases. This line is the move disk line, that used to be t gets t plus 1. I don't have t in this calculation, but p is increased by the current space s times the time increase, which is 1. [70] The proof is just a lot of uses of the substitution law, and it's in the book. For the [71] average space, take the part of the space time product that is due to our computation, and divide it by the time. And you can [72]

simplify that a little. It turns out to be just a bit less than n. It would be a lot easier, instead of finding the average exactly, to prove [73] that the average space is bounded above by n, but we knew that anyway.

We have several results, so by the law of refinement by parts, [74] we can put them all together. This one has all the variables in it, and the refinement is a theorem when move pile is the conjunction of all the results.

Maximum space tells us how much memory we need. Average space is a better measure of the space cost when there are parallel processes sharing the space. The space variable s should be increased at each variable declaration, and at each explicit allocation of space, and decreased when we come to the end of a scope, and at each explicit freeing of space. We didn't have any of those in this example, but we did have some return address stack activity. We look at declarations next.