[1] Searching today. Two standard programs, linear search and binary search. And we'll start with linear search. Find the first occurrence of item x in list L. I'll consider x and L to be constants, so there's no x prime and no L prime. I need a variable to report the position of item x, so I'll use h. h is for "here it is". Since we want the first occurrence, we search from the beginning of the list towards the end. [2] Here's a picture of list L with the first occurrence of x shown, and h prime is also shown. Maybe the picture helps to write the specification. There isn't just one right way to write it. [3] Here's the way I wrote it. The [4] left conjunct says that x does not occur in list L from the beginning to h prime. Then there's a disjunction, and the [5] left disjunct says x *does* occur in list L at position h prime. The left conjunct and the left disjunct together say that h prime is the position of the first occurrence of x. The left conjunct together with the [6] right disjunct say that x doesn't occur anywhere in the list. If that's the case, we'll report it by assigning h the length of the list. That's not the whole specification because it doesn't talk about the timing. We can add that later. Now we have to [7] refine this specification, and we start by [8] assigning h the value 0. In the [9] remaining problem, we ignore that h is 0 because we want to describe the problem at any iteration. We just keep that [10] h is less than or equal to the length of the list. And the problem looks very much like the original problem, but we just have to search from [11] h onwards, not from 0 onwards. The proof of this refinement uses the substitution law, replacing h with 0. The antecedent is – 0 less than or equal to the length of the list, which is true for any list, and the right side becomes the same as the left side.

[12] Now we have to refine the new problem. The two disjuncts suggest two tests we could make. But we can't test L.h equal x unless we know that h is less than the length of the list, and we don't know that. So we [13] have to make the other test. If h equals the length, then doing [14] nothing makes h prime equal to h, so [15] h prime equals the length, and also there is [16] no x in the list from h to h prime, so both conjuncts are satisfied. [17] Now for the else-part, we [18] were given that h is less than or equal to the length, and we've just learned [19] that h is not equal to the length, so we know [20] that h is less than the length, and we still want to solve the same problem. This new specification needs to be [21] refined. Now that we have h less than the length we can test [22] if L.h equals x. And if it does, [23] then doing nothing makes h prime equal to h, so [24] L.h prime equals x, and also there is [25] no x in the list from h to h prime, so both conjuncts are satisfied. [26] Now for the else-part, if we move along in the search by [27] adding 1 to h, then we no longer know that [28] h is less than the length. All we know is [29] it's less than or equal to the length. And we've already solved this problem, [30] so there are no unsolved problems, and we're done. Except for the timing.

For the timing, we need to make exactly the same refinements, [31] except that the specifications talk about time, and there's a t gets [32] t plus 1 just before the recursive call. [33] We need the same refinements so the law of refinement by parts applies, and says that we can conjoin the specifications. [34] The main specification now says the final time is less than or equal to the initial time plus the length of the list. In other words, the time is bounded above by the length. The time specification [35] after assigning 0 to h talks about the time remaining in the computation. Well, that's the whole time, because we haven't done anything much yet, but we ignore that h is 0, and imagine we're starting any iteration. So the time remaining is the length minus h; that's the number of items still to go. It has to be less than or equal, not just equal, because we may not have to go to the end of the list. Replacing h by 0 here makes the right side equal to the left side, and proves the refinement. To prove the [36] middle refinement, there are two cases to consider. When [37] h equals the length, then [38] the length minus h is 0, and so [39] ok says t prime equals t, which implies [40] t prime is less than or equal to t. The [41] other case just strengthens the antecedent. In the last refinement, we have [42] h is less than the length, and from ok we have t prime equals t, so together we have [43] what we want. I should mention again that in this lecture I'm just

pointing to things and talking about them, but in homework you have to write out proofs properly. [44] I'll do the last case properly. [45] Use the Substitution law to replace t with t plus 1 [46]. Then again to replace h with h plus 1 [47]. h plus 1 less than or equal to the length is the same as h less than the length, and the plus 1 and minus 1 cancel out [48], and that's the left side.

Let me go back [49] to the refinements that talk about the result. You give it to your boss, or to the customer, and say here it is, I'm done. And the boss says: oh, I forgot to tell you, it's always a [50] nonempty list. You could say fine, it works for nonempty lists. I don't have to make any changes. Or you could notice that after h is assigned 0, we know [51] that h is less than the length of the list. If you make that change, you don't have to change anything else because [52] that problem is already solved. And you've just saved an unnecessary test. That's all I want to say about linear search.

[53] In the binary search problem, again we're looking for an item x in a list L, but not necessarily the first occurrence. Any occurrence will do. And the given list is nonempty and sorted. And the execution time must be logarithmic. This time [54] my specification will use a binary variable p, for present, to indicate whether there is such an item in the list. I could have done that last time in linear search. I'm still using variable h to indicate where it is. If x is anywhere in the list, then p prime is true, and if x isn't anywhere in the list, then p prime is false. That's the equation. And if x is present, then h prime says where. That's the implication. If x occurs several places, then this specification is nondeterministic; any of its positions will do. Once again, I'll leave the timing specification until later. In binary search, there's a segment of the list where the search continues, as in [55] this picture. The segment starts out being the whole list, and gets smaller and smaller, until there's just one item left. To save writing, [56] I define specification R to be the problem of searching in the segment from h to j. Now the [57] first refinement is easy. Set h to 0 and j to length of L, then the new problem is, given h less than j, because the list is nonempty, look for x in the segment from h to j. Now [58] we need to refine that. We should [59] test if the segment size happens to be 1, because then there's only 1 item to look at. And it's item h. [60] If L.h equals x, then set p to true and leave h where it is. If L.h is unequal to x, then set p to false, and it doesn't matter what happens to h, so we may as well leave it alone. That's what this assignment says. [61] Else what? We know h is less than j so the segment is nonempty, and we just found out it isn't size 1, so [62] we have j minus h greater than or equal to 2, and we still want to solve the problem. And [63] how do we do that? This is where we divide the segment into 2 parts. [64] Given that the segment size is at least 2, find a point, call it i, that's properly between h and j, and don't change h and j. Now the question is – which part, h to i, or i to j, should we continue to search in? So we [65] compare L of i with x. This is where it matters that the list is sorted. If L of i is smaller or equal, we continue to search in the right part, so move h up to i. If L of i is bigger than x, we look in the left half, so move j down to i. By choosing i properly between h and j, we made sure that each part, from h to i and from i to j, is nonempty. So now the remaining problem is: [66] given h is less than j, look for x in the segment from h to j. And we've already solved that problem. But there's [67] still one left.

It says we have to choose i in between h and j, but it doesn't say where. In fact, we could choose [68] h plus 1. These refinements can all be proven, and the search works. But we wouldn't be able to prove the logarithmic time bound. This would be linear search. [69] To get the best execution in the worst case, that means to make sure that the longest execution is as short as possible, we have to [70] cut the segment in half, or as nearly in half as possible.

Now here's something that is not generally known. It's usually much more important to minimize the average execution time, rather than minimizing the worst case. The worst case may never happen, or may be very rare. You really want execution to be fast for the

cases that happen all the time. For the best execution time on average, you have to cut the segment into two parts that have equal probability of containing x. That might be the same thing as cutting it in half, but maybe not. Often, different items have very different probabilities. For example, in a symbol table for a programming language, there are keywords you use all the time, like if, and there are other keywords that you never use, and didn't even know they were in the language. Suppose the most popular item is wanted half the time, and the next most popular item is wanted a quarter of the time, and the next an eighth, and so on. Then store the items in that order, and do a linear search. It's the best search there is for that probability distribution, on average.

Here's something else that isn't well known about binary search. After we cut the segment in half, would it be better or worse to see if the middle item is x? In the best case, it would be better because if the middle item is x, that would save further computing. But what about the worst case, or the average case? The answer is that in recursive time it doesn't matter. Worst case stays the same, and average case is almost identical. But for real time it makes a big difference. It's much worse to check the middle item to see if it's x, because now the loop contains 3 tests instead of 2. The binary search shown here is the best one. Notice the complete absence of plus 1s and minus 1s. Notice that we didn't start by setting variable p, or we might have called it found, to false, and then testing it each time round the loop. p gets set once, at the end, when we know its value.

We still have to do the timing. We need [71] specifications T, U, and V that talk about time, and we need to put t gets t plus 1 just before the recursive call. Let's suppose you're having trouble coming up with the right specifications. One thing you could do is to run the program on a lot of different lists, with variable t in the program, so you can [72] print out a table of times. The worst case is to search for an item that's not there, and here's what you get. You see that the time goes up just after length 1, 2, 4, 8, 16, and so on. So that tells you that the time is logarithmic, in case you didn't know it already. [73] For T, it's the ceiling, that means rounded up, of the log of the length. For U and V it's the ceiling of the log of the length of the remaining segment.

[74] I think now is a good time to tell you about 3 levels of care in programming. At the [75] lowest level, you don't bother to write specifications and refinements. You just write the code. That's the level you see most often. At the [76] middle level, you write all the specifications, but you don't bother to prove the refinements formally. That's what I have just done with binary search. Formal proofs are a lot of work, especially if you don't have an automated prover to help you. And this level is often good enough. At the [77] highest level, you write all specifications and proofs. That's what you do when you have to get it right. The formal proofs of all the refinements in binary search are in the textbook, and I leave you to read them on your own.