LING REN: Everyone, today we're going to look at dynamic programming again. So I think I have mentioned several times, so you should all know it by heart now, the dynamic programming, its main idea is divide the problem into subproblems and reuse the results of the problems you already solved. Right? And, of course, in 6.046 we always care about the runtime. So those are the two big themes for dynamic programming.

Now, let's start with a warm-up example. It's extremely simple. Let's say we have a grid, and there's a robot from, say, coordinate 1,1 and it wants to go to coordinate m,n. So at every step, it can only either take a step up, or take a step on the right. So how many distinct paths are there for the robot to take? Is the question clear? So we have a robot at coordinate 1,1. It wants to go to coordinate m,n. And every step, it can either take a step up, or take a step to the right. How many distinct path are there that can take the robot to its destination? Any ideas how to solve that? Go ahead.

AUDIENCE: So, we define subproblems as the number of distinct paths from some point x,y to m,n. Then the number of distinct paths from some point is the number of paths if you go up if you're allowed to go up, plus the number of paths if you go right if you're allowed to go right. So if you were on the edge, [INAUDIBLE].

LING REN: Yup, yup. Does everyone got that? So, it's very simple. So, I know I have only one way to get to these points. I need to go all the way right. And only one way to get to these points. I need to go all the way up. So for all the intermediate nodes, my number of choices are-- is this board moving? Are just the number of distinct paths I can come from my left, plus the number of distinct path I can come from bottom. And then I can go in. For every node, I'll just take a sum between the two numbers on my left and on my bottom. And go from there. OK. Is that clear?

So this example is very simple, but it does illustrate the point of dynamic programming very well. You solve subproblems, and ask how many distinct path can I come here, and you reuse the results of, for example, this subproblem because you are using it to compute this number

and that number. If you don't do that, if you don't memorize and reuse the results, then your runtime will be worse. So what's the runtime of that? Speak up.

AUDIENCE: [INAUDIBLE]

LING REN: It's just m times n. Why? Because I have this many unique sub problems. One at each point, and I'm just taking the sum of two numbers at each subproblem, so it takes me constant time to merge the results from my subproblems to get my problem. So to analyze runtime, usually we ask the question how many unique problems do I have. And what's the amount of merge work I have to do at every step? That's the toy example.

Now let's look at some more complicated examples. Our first one is called make change. As its name suggests, we have a bunch of coins. s1, s2, all the way to, say, sm. So each coin has some values, like 1 cent, 5 cent, 10 cent. We're going to make change for a total of n cents, and ask what's the minimum number of coins do I need to make change of n cents.

So to guarantee that we can always make this change, we'll set s1 to be 1. Otherwise, there's a chance that the problem is unsolvable. Any ideas? Is the problem clear?

STUDENT: How do you find s1 again? Or si?

LING REN: What, these numbers? They are inputs. They are also inputs. It could be 1 cent, 5 cent, 10 cent. Or 3 cent, 7 cent. Though the smallest one is always 1. OK. I need to find a combination of them. For each of them, I have an infinite number of them. So I can find two of these, three of that, five of that, such that their sum is n. Is the problem clear? OK. Any ideas how to solve that?

So let's just use a naive or very straightforward algorithms. Go ahead.

AUDIENCE: You pick one, and then you do mc of n minus that.

LING REN: OK, great. Yeah, let's just do exhaustive search. Let's pick si. If I pick this coin, then my subproblem becomes n minus the coin value. And of course, I use the one coin. That's si. So then I think the min of this for all the i's, and that's the solution. So far so good?

OK. So what's the runtime of this algorithm? If it's not immediately obvious, then we ask how many unique subproblems are there. And how much work do I have to do to go from my subproblems to my original problem? So how many subproblem are there? So to be clear, for

this one, we have to call this recursive call again. n minus si, probably minus sj. And if you cannot compute how many subproblems are there, let's just give a bound. Any ideas? John, right?

AUDIENCE: I'm not sure there would be more than n subproblems, because the smallest amount we can subtract from the original is 1. And if we keep subtracting 1 repeatedly, we get n subproblems, and that will cover everything-- that subproblem.

LING REN: Yeah, correct. So this may not be a very tight bound, but we know we cannot have more than this number of subproblems. Actually, I don't need to even put the order there. I know we can have no more than n subproblems. They're just make change of n, n minus 1, n minus 2, all the way to make change 1. And actually, this bound is pretty tight, because we set our smallest coin is 1, so we won't make a recursive call to make change n minus 1, right? If I pick the 1 coin, the 1 cent coin first. And then from there, I will pick a 1 cent coin again. That gives me a subproblem with n minus 2. So indeed, I will encounter all the n subproblems.

OK, so having realized that, how much work do I have to do to go from here to there?

AUDIENCE: [INAUDIBLE]

LING REN: Correct. Because I'm taking the min of how many terms? m terms. So that's our runtime. Any questions so far? If not, let me take a digression. So, make change, this problem. If you think about it, it's very similar to knapsack. Has anyone not heard of this problem? Knapsack means you have a bunch of items. You want to pack these into a bag, and the bag has a certain size. So each item has a certain value, and you want to pack the items that have the largest combined value into your bag. So, why are they similar?

So in some sense, n is our size. We want to pick a bunch of coins to make the size n. And each coin here actually has a negative value, because we want to pick the min of it. If you do that, then this problem is exactly knapsack. And knapsack is NP-complete. That means we don't know a polynomial solution to it yet. However, we just found one. Our input is, m stuff and n. Our solution is polynomial to m, and polynomial to n. If this is true, then I have found the polynomial solution to one NP problem. So P equals NP. SO we should all be getting Turing award for that. So clearly something's wrong. But there's no problem with this solution. This covers all the cases. And our analysis is definitely correct.

So does anyone get what I'm asking? So what's the contradiction here? I will probably discuss

this later, in later lectures when we get to complexity or reduction. But to give a short answer, the problem is that when we say the input is n, its size is not n. So I only need log n this to represent this input. Make sense? Therefore, for log n length input, my runtime is n. That means my runtime is exponential. It's not polynomial. OK. Now that's the end of the digression.

Now let's look at another example. This one is called rectangular blocks. So in this problem, we have a bunch of blocks. Say 1, 2, all the way to n. And each of them has a length, width, and height. So it's a three-dimensional block. So I want to put blocks, stack them on top of each other to get the maximum height. But in order for j to be put on top of i, I require the length of j to be smaller then the length of i, and the width of j is also smaller with width of i.

So visually I just meant this is a block. I can put another block on there. They are smaller in width and length. But I cannot put this guy on top of it because one of its dimension is larger than the underlying block. And to make things simple, that's not allowed, rotating. So OK, I can rotate. It still doesn't fit. But you see the complication. So you allow rotate, then there's more possibility. Length and width are so one of them is north-south, the other is east-west, and you cannot change that. OK. Is the problem clear? You want to stack one on top of each other to get the maximum height.

Any ideas? Again, let's start from simple algorithm. Say, let's just try everything out. OK, go ahead.

AUDIENCE: If you try everything else, you have n factorial.

LING REN: Pardon?

AUDIENCE: It would be O of n factorial?

LING REN: You're going too fast. Let's write the algorithm first. So I want to solve my rectangle block problem, say from 1 to n. What are my subproblems?

AUDIENCE: Choose one block.

LING REN: OK. Let's choose one block.

AUDIENCE: And then you run RB of everything except that block.

LING REN: So I get its height, and then I have a subproblem. What is the subproblem? And then I'll take a max. So the difficulty here is this subproblem. So Andrew, right? So Andrew said it's just

everything except i. Is that the case? Go ahead.

AUDIENCE: It's everything except i, and anything with wider or longer than i.

LING REN: Do you get that? Not only do we have to exclude i, we also have to exclude everything longer or wider than i. So that's actually a messy problem. So let me define this subproblem to be a compatible set of w i. And let me define that to be the set of blocks where the length is smaller than the required length, and their which is also smaller than the required width. So this should remind you of the weighted interval scheduling problem, where we define a compatible set once we have chosen some block. Question?

AUDIENCE: What are we trying to do here? Are we trying to minimize h?

LING REN: Maximize h. We want to get as high as possible. I choose a block, I get its height, and then I find out the competitive remaining blocks, and I want to stack them on top of it. Everyone agrees this solution is correct? OK, then let's analyze its runtime. So how do we analyze runtime? So what's the first question I always ask?

AUDIENCE: How many subproblems?

LING REN: Yeah. I'm not sure who said that, but how many subproblems do we have?

AUDIENCE: At most n?

LING REN: At most n. Can you explain why is that the case? Or it's just a guess?

AUDIENCE: Because if n is compatible-- nothing in the compatible-- n will not be in the compatible set of anything that is in the compatible set of n.

LING REN: OK, that's very tricky. I didn't get that. Can you say that again?

AUDIENCE: Because for example, if you start with n, then everything that's in the compatible set of n. n won't be in the compatible set of that.

LING REN: OK. I think I got what you said. So, if we think there are only n subproblems, what are they? They have to be compatible sets l1, w1, then l2, w2. These are the n unique subproblems you are thinking about. Is there any chance that I will get a compatible set like something like l3 but w5? If I ever have this subproblem then, well, my number of subproblems are kind of exploding.

Yeah, I see many of you are saying no. Why not? Because if we have a subproblem, say, compatible set of l i and w i, and if we go from here, and choose the next block, say t, it's guaranteed that t is shorter and narrower. That means our new subproblem, or new compatible set becomes-- our new subproblem needs to be compatible with t instead of i. So, the only subproblems I can get are these ones. I cannot have one of these. The number of subproblems are n. And how much work do I have to do at each level?

AUDIENCE: n.

LING REN: n, because I'm just taking the max, and there are n potential choices inside my max. So runtime n squared. OK, we're not fully done, because there is an extra step when we're trying to do this. We have to figure out what each of these are. Because once I go into this subproblem, I need to take a max on all the blocks that's in this set. I have to know what blocks are in that set. Is that hard? So how would you do that?

AUDIENCE: You just check for all of them, and that's O of n.

LING REN: OK. So, I check all of them. That's O of n. I'm pretty sure you just meant scanning, scan the entire thing, and pick out the compatible ones. But that's for this subproblem. We have to do it for every one. Or there may be a better way. So I think the previous TA is telling me there's a better way to do that. So in order to find the entire compatible stuff, he claims he can do it in n log n, but I haven't checked that, so I'm not sure. This is a folklore legend here. Yeah, we'll double check that offline. But assuming if I don't have this, then figure out all these subproblems will also take n squared. Then my total runtime is n squared plus n squared, and still n squared. Question?

AUDIENCE: Is the n log n solution giving us sorting this by [INAUDIBLE]?

LING REN: Yeah, I think it should be something along those lines, but yeah, I haven't figured out whether you sort by length or by width. You can only sort by one of them. So after sorting, say let's sort by length. Then after sorting, I may get something like this. And if I'm asking what's the compatible set of width this guy, I still have to kick all of them out. Yeah, so it's not entirely clear to me how to do it, but I think you can potentially consider having another, say, binary search tree that's sorted by width, and you can go in and just delete everything larger than a certain width. So that's the, yeah. OK, go ahead.

AUDIENCE: Can you convert into a directed graph, where each pair of shapes that's compatible, you do an

edge. And then path find.

LING REN: OK. OK. But constructing that graph already takes O n squared, correct? Yeah, OK, let's move on. I don't have time to figure this out. So, this problem is remotely similar to interval scheduling, weighted interval scheduling, in a sense that it has some compatible set. And in the very first lecture and recitation, we have two algorithm for weighted interval scheduling, and one of them is better than the other. And this one looks like the naive algorithm. So, does anyone remember what the better algorithm is for weighted interval scheduling?

But instead of checking every one as my potential lowest one, it really doesn't make sense to do that. Because for the very small ones, I shouldn't put them as my bottom one. I should try the larger ones first as the very bottom one. Go ahead. Oh, you're not--

AUDIENCE: You could create a sorted list of length n with the width. So you know that items that are later in the list, they're not going to be in the first level of the tower.

LING REN: Yeah, correct. So, just in the same line of thought as weighted interval scheduling, let's first sort them. But then, it's a little tricky because do I sort by length or width? So I'm not sure yet, so let's just sort by length and then width. So this means if they have the same length, then I'll sort them by width. So I can create a sorted list. Let me just assume that it's in-place sort, and now I have the sorted list. So once I have that, the potential solutions I should consider is that whether or not I put my first block as the bottom one. It doesn't make sense for me to put a later one down. So my original problem becomes taking the max, and whether or not I choose block one. If I do, then I get its weight-- height, sorry. And my subproblem is the ones compatible with it.

If I do not choose it, then my sub problem is like what Andrew first said, from 2 all the way to n. So why is this correct? So I claim this covers all the cases. Either h1 is chosen as the first bottom one, or it's not. It's not chosen at all. It's impossible for h1 to be somewhere in the middle, because it has the longest, largest length. OK. So how many subproblems do I have? Go ahead. Still n. So there are all of these compatible set of l1 w1, l2 w2. But it looks like I do have some new subproblems.

These do not exist before. However, there are only n of them. They're just a suffix of the entire set. So I still have O of n subproblems. And at each step, I'm doing constant amount of work. There are just two items. So we found an order n solution. Are we done? Is it really order n? OK, no.

AUDIENCE: You still have to find the c.

LING REN: Yeah. I still have to find all these c's. And first, I actually have a sort step. That sort step is n log n. Yeah, then again, well, if we do it naively, then it's again n squared, because I have to find this compatible set, each of them. But if there's an n log n solution to find these compatible sets, then my final runtime is n log n. Make sense? Any questions so far?

OK. So now we actually have a choice. So we can either go through another DP example, I do have another one. But Nancy, one of the lecturers suggested, that it seems that many people have some trouble understanding yesterday's lecture on universal hashing and perfect hashing. So we can also consider going through that. Well, of course, the third option is to just call it a day. So, let me just take a poll. How many people before we go over the hash stuff? How many people prefer another DP example? OK. Sorry guys. How many people just want to leave? It's fine. OK. Great. That's it.

OK. So, so much for DP. We do have another example. We will release it in recitation notes. For those of you who are interested, you can take a look. So, well, sure you all know that we haven't go into DP in the main lectures yet. So this is really just a warm up to prepare you to go to the more advanced DP concepts. And also, DP will be covered in quiz 1. But the difficulty will be strictly easier than the examples we covered here. OK?

Now let's review universal and perfect hashing. So it's not like I have a better way to teach it. Our advantage here is that we have fewer people, so you can ask questions you have. So let me start with the motivating example. So why do we care about hash? It's because we want to create a hash table of, say, n. It has n bins. And we will receive input, say, k0, k1, all the way to k n minus 1. n keys. And we'll create a hash function to each of them to map them to one of the bins. That the hope is that if n is theta m, or in the other way, m is theta n, then each bin should contain a constant number of keys.

So to complete the picture, all the keys are drawn from a universe that has size u. And this u is usually pretty large. Let's say it's larger than m squared. It's larger than the square of my hash table size. But let me first start with a negative result. So if my hash function is deterministic, then there always exists a series of input that all map to the same thing. We call that worst case. We don't like the worst case. Why? Because in that case, the hash is not doing anything. We still have all of the items in the same list.

Why is that lemma true? Because by a very simple pigeonhole argument, so imagine I insert all of the keys in the universe into my hash table. I would never do that in practice. It's just a thought experiment. So by a simple pigeonhole argument, if u is greater than m squared, then at least some bin will contain more than m elements. Well, if it just so happens that my inputs are these m keys, then my hash will hash all of them to the same bin. Make sense? So this is the problem we're trying to solve. We don't want this worst case. And it does say that if h is deterministic, we cannot avoid that. There always exist a worst case. So what's the solution? Then the solution is to randomize h.

However, I can't really randomize h. If h take some key, if my hash function maps a key into a certain bin, well, the next time I call this hash function, it better give the same bin. Otherwise I cannot find that item. So h needs to be deterministic. So now our only choice is to pick a random h. Make sense? Every hash function is deterministic, but we will pick a random one from a family of hash functions. So in some sense, this is cheating. Why? Because all I'm saying is I will not choose a hash function beforehand. I will wait for the user to insert inputs. If I have too many collisions, I'll choose another one. If I have too many collisions. I'll choose another one.

OK. I think I forgot to mention one thing that's important. So you may ask why do I care? Why do I care about that worst case? What's the chance of it happening in practice? It's very low, but in algorithms, we really don't like making assumptions on inputs. Why? Because if you imagine you're running, say, a website, a web server, and you code has some has table in it. So if your competitor, or someone who hates you, wants to put you out of business, and if he knows your hash function, he can create a worst case input. That will make your website infinitely slow.

So what we are saying here is I don't tell him what hash function I'll use. I'll say I choose one. If he figures out the wrong input, the worst case input, I'm going to change my hash function and use another one. Make sense?

Now the definition of universal hash function is that if I pick a random h from my universal hash function family, the probability that any key i mapped to the same bin as any key j should be less or equal than 1 over m, where m is my hash table. This is really the best you can get. If the hash function is really evenly distributing things, you should get this property.

So we have seen one universal hash function in the class. I'll just go over the other example,

which is ak plus b modulo p, and then modulo m. So p is a prime number that is greater than the universe size. We'll see why this is a universal hash function. So to do that, we just need to analyze the collision probability. So if I have two key, that k1 and k2 that map to the same bin, that means they must have this property. After taking the mod m, their difference should be a multiple of m. Because if this is true after taking the modulo m, they will map to the same bin. Make sense?

Now I can quickly write it as a times the difference of the key equals a multiple of m, mod p. Now, k1 and k2 are not equal, so they are nonzero. And in this group, based on some number theory, we have an inverse element for it. So, if this happens, we'll call it a bad a. How many bad a's do I have? One of a will make this equation holds with i equals 1. Another a make the equation holds with i equals 2. But how many such a's do I have? At most, because this equation can hold with m, 2m, 3m, all the way to p over m floored m. This is the total number of possible ways this equation can hold.

So how many bad a's do I have? I have p over m, over the total number of a's, which is p minus 1. Oh, yeah, I forgot to mention that. So a is from 1 to p minus one. OK. So I can always choose my p to be not a multiple of m. If I do that, this floor-- so, then p and p minus 1 do not cross the boundary of modulo m. Then this is true, and this is less than 1 over m. So this is a universal hash function family.

So what's the randomness here? The randomness is a. I'll pick an a to get one of my hash, and if it doesn't work, I pick another a.

AUDIENCE: What is b? What is b?

LING REN: p is a prime number I choose--

AUDIENCE: [INAUDIBLE]

LING REN: b?

AUDIENCE: Yeah.

LING REN: Oh, b. I think it's also a random number. Yeah, so, actually it's not needed, but I think there's some deep reason that they keep it in the hash function. I'm not sure why.

Now once we have that, once we have universal hash, people also want perfect hashing,

which means I want absolutely 0 collision. So how do I do that? Let me first give a method 1. I'll just use any universal hash function, but I choose my m to be n squared. I claim this is a perfect hash function with certain probability. Why? Because I want to calculate probability no collision. Yeah, 1 minus probability I do have a collision. And I can use a union bound. That's the probability that any pair has a collision. Any pair of hx equals hy. How many pairs do I have?

AUDIENCE: N choose 2.

LING REN: Yeah. n choose 2, which is this number. So if it's a universal hash function, then any collision, any two colliding, the probability is 1 over m. So I choose my m to be n squared, so this one is larger than 1/2. So what I'm saying, to get a perfect hash function, I'll just use the simplest way. I select the universal hash function with m equals n squared. I have a probability more than 1/2 to succeed. Or if I don't succeed, I'll choose another one until I succeed. So this is a randomized algorithm, and we can make it a Monte Carlo algorithm or Las Vegas algorithm.

So I can either say if I choose alpha log n times, then what's the chance that none of my choice satisfies perfect hashing? My failure probability is less than this. My each chance I have a half success rate, and I try this many times, what's the chance of all of them failing? This is 1 over n raised to alpha. Of course, I can also say, I'll keep trying until I succeed. Then I have a 100 percent success rate, but my runtime could potentially go unbounded. Make sense? OK. This sounds like a perfect solution. The only problem is that the space complexity of this method is n squared, because I choose my m hash table size to be n squared. So this is the only thing we don't want in this simple method.

Our final goal, is to have a perfect hash function that has space O of n, and also runtime some polynomial in n, and failure probability arbitrarily small. And the idea there is this two-level hashing. So, I choose h1 first to hash my keys into bins. And for each bin, say I get l1 elements here, l2 elements here, so on and so forth. I'll choose each of the bins to be a second level perfect hashing. So we can use the method one to choose this small one. If I choose m1, which is the hash table size of this guy, to be l1 squared, then I know after alpha log n trial, this one should be a perfect hashing. After another alpha log n trial, I should resolve all the conflicts in l2 to make it a perfect hashing. Make sense?

So after n log n trials, I will resolve all the conflicts in my second level hashing. Question?

AUDIENCE: It was mentioned in the lecture that this only works if there are no inserts or deletes, or

something like that?

LING REN: Let me think about that offline. I'm not sure about that. OK. So the only remaining problem is we need to figure out whether we achieve this space O of n. What is this space complexity of this algorithm? It's n plus l i squared, because each table size is the square of the elements in it. And finally, we have that Markov inequality or I think something like that, to prove this is the case with-- so my space is O of n, also with the probability of greater than 1/2. I can keep going. I'll try alpha log n times on my first level hash function, until my space is O of n. Once I get to that point, I'll try choosing universal hash functions for my smaller tables, until I succeed.

OK? That's it for hashing and DP.