Lecture 5 : More on Dynamic Programming

Justin Pearson

Today's Topics

More dynamic programming

- The complexity of the coin change dynamic program again.
- Largest non-overlapping subset.
- Knapsack via Dynamic programming

Adding two numbers

Consider the very simple algorithm:

function $\textsc{Add}(x,y)$
    return $x + y$
end function

What is the complexity?

Adding two numbers

- The complexity is constant time, $O(1)$ because we assume that addition takes constant time.
- When addition is implemented in a processor it uses a binary representation of the numbers, and implements in hardware the same algorithm that you learn in school.

What is the input to the problem? It is simply two numbers, so it is constant space.

Adding two numbers

What is the complexity of:

Require: $x \geq 0$
  function $\textsc{Add}(x,y)$
    $z \leftarrow y$
    while $x > 0$ **do**
      $z = z + 1$
      $x = x - 1$
    end while
    return z
  end function

What does it do?

Adding two numbers

- The complexity is $O(x)$. It takes x steps for the while loop to finish.

Is this polynomial time or not?

Adding two numbers

- The complexity is $O(x)$. It takes x steps for the while loop to finish.

Is this polynomial time or not?

It is not polynomial time. It is pseudo-polynomial time. The input is constant space, but the time it takes depends on the value of n.

Pseudo vs polynomial time

- Think about summing the numbers in a list. This takes linear time in the input size. It does not depend on the values.
- While our stupid addition algorithm depends on the value of the input.

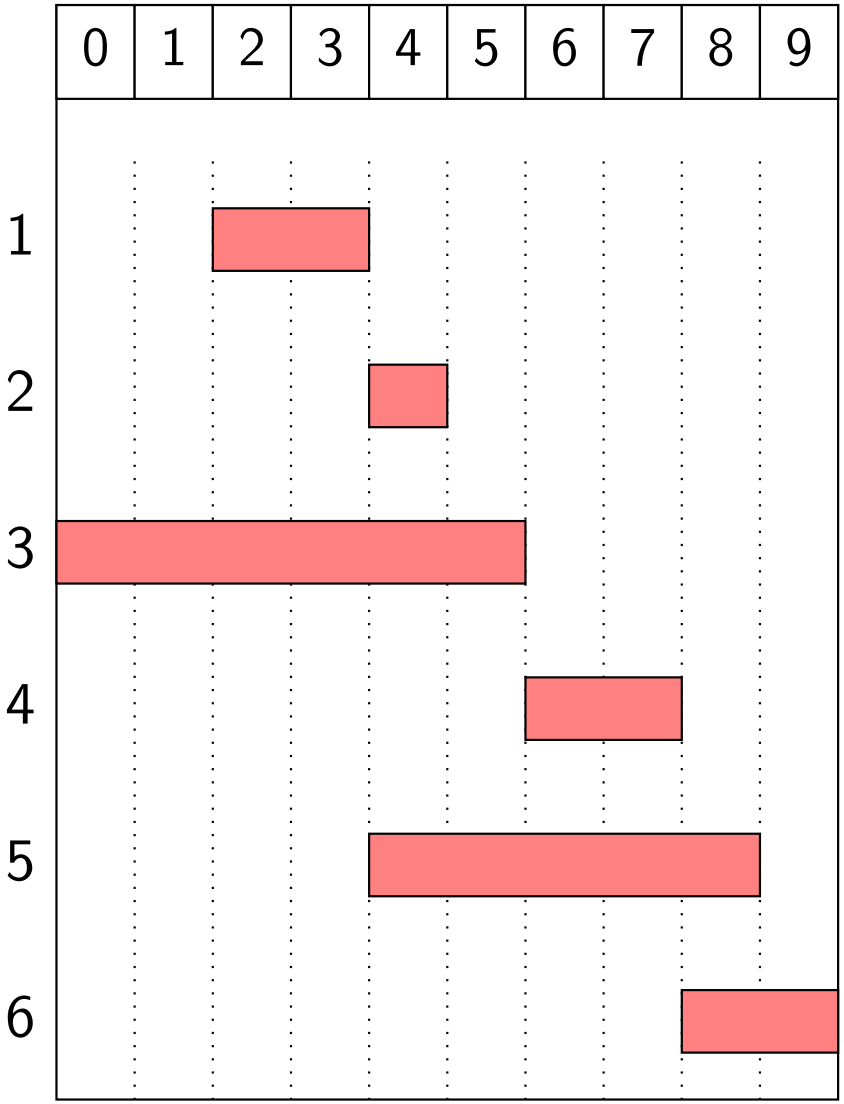
If double a number it only takes 1 more bit to represent it.

Coin Change complexity

- The complexity of the our coin change algorithm is linear in the value that we are trying to change. So it takes at most 13 steps to work out the change for $13SEK$.
- With dynamic programming your often have to divide into smaller parameters that depend on the **values** of input parameters, and this gives you a pseudo-polynomial time dynamic programming.

Pseudo-polynomial is good when your values do not get to large.

Maximum non-overlapping intervals

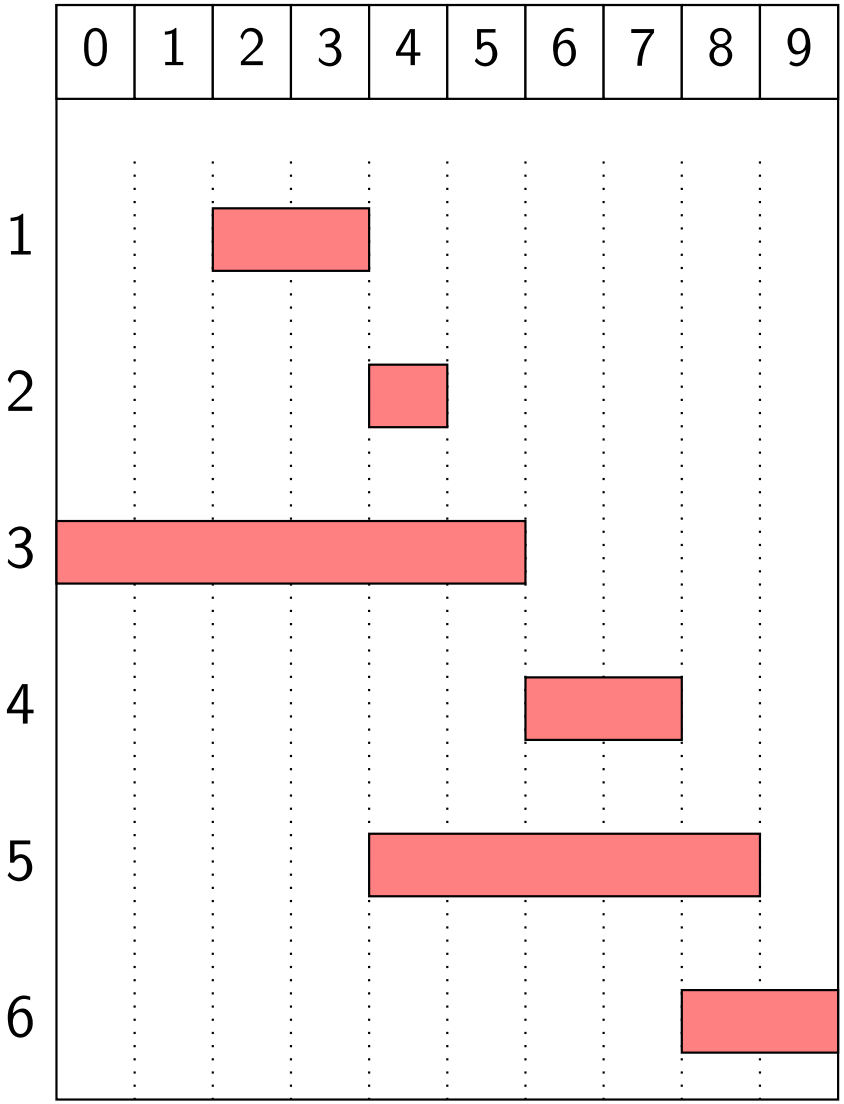


The goal is to find a largest subset of the tasks that do not overlap.

- In this case it is the set of tasks $1, 2, 4$ and 6.

We will assume that the tasks are sorted in by finishing time. This adds and $O(n \log(n))$ overhead to the algorithm.

Maximum non-overlapping intervals



Definition

Let $p(i)$ to be the largest index j such that job i is non-overlapping with j, and 0 if there is no such job.

- $p(1) = 0$.
- $p(2) = 1$.
- $p(3) = 0$.
- $p(4) = 3$.
- $p(5) = 1$.
- $p(6) = 4$.

Dynamic Programming Towards the Bellman Equation

We are going to define Opt(i) that is the maximum number of non-overlapping jobs that you can schedule using only the jobs $1 \ldots i$.

Notice that are first solving the easier problem: How many jobs can I schedule? Rather than working out what the jobs are. Once we have the dynamic program we can work backwards from the solution to work out what jobs are scheduled.

We are interested in Opt(n), but we need to work out the relationship between Opt(i) and Opt($i-1$).

Why is sorting the jobs important?

We sorted the jobs by finish time. This means that when we look at Opt(i). We are only considering the jobs that finish before job i's finish time.

This means that when we are trying to work out Opt(i), we only have to consider Opt$(1), \ldots,$ Opt$(i-1)$.

Dynamic Programming Towards the Bellman Equation

Let's assume that we know what $\text{Opt}(i-1)$ is. There are two things that can happen to job i:

1. We do not use job i, so $\text{Opt}(i)$ equals $\text{Opt}(i-1)$.
2. If use job i, then the previous choice $\text{Opt}(i-1)$ might have jobs that overlap with job j. This means that we have to go back to $\text{Opt}(p(i))$ that is the maximum index for which the job does not clash with job i.

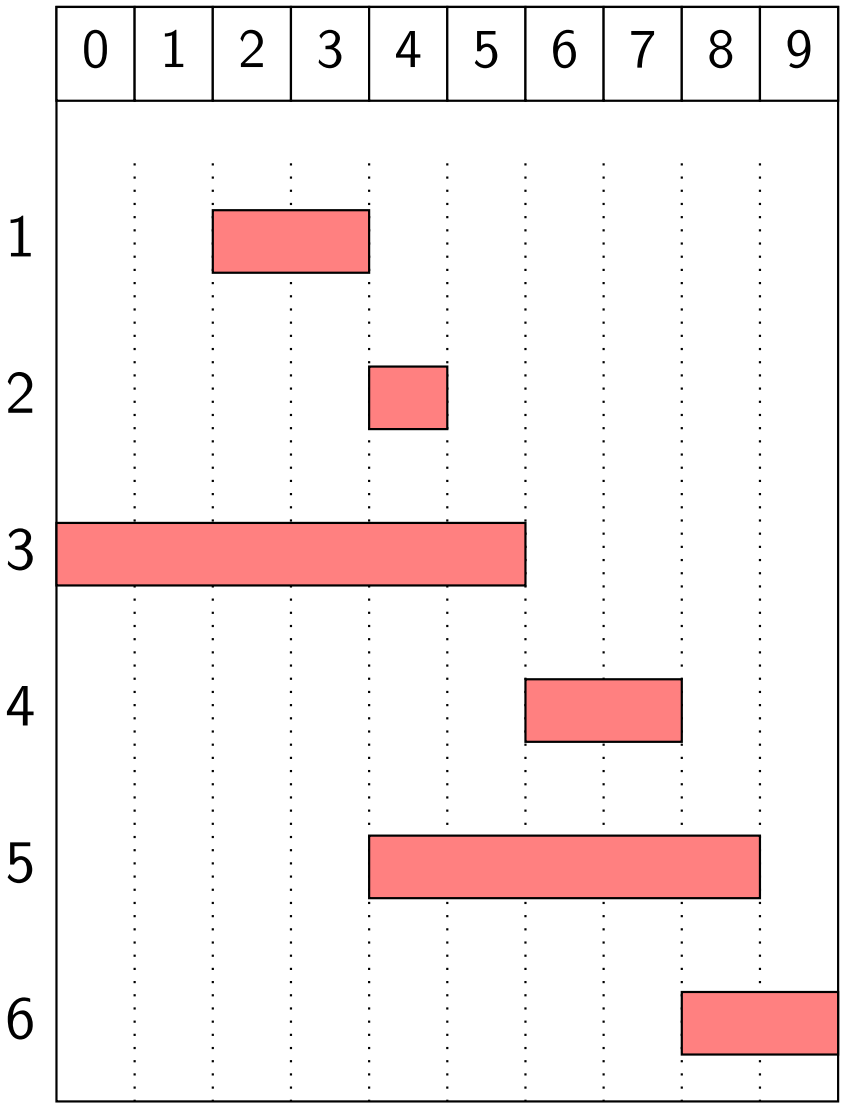
Dynamic Programming the Bellman Equation

The consideration on the previous slide gives the Bellman Equation

$$\text{Opt}(i) = \begin{cases} 0 & \text{if } i = 0 \\ \max(\text{Opt}(i-1), 1 + \text{Opt}(p(i))) & \text{if } i > 0 \end{cases}$$

Notice that $1 + \text{Opt}(p(i)))$ we use job j but we have to backtrack to job $p(j)$. $p(i) < j$ and so this should be already computed when we implement the dynamic program.

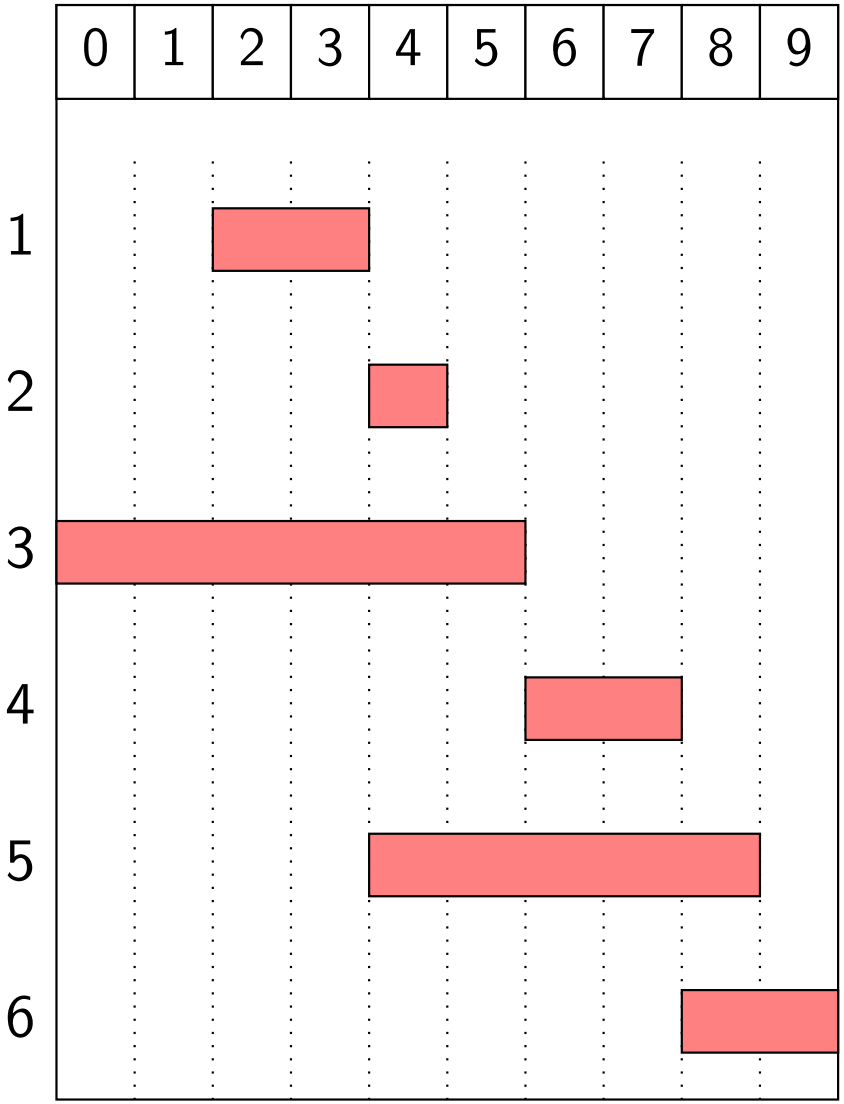
The Bellman Equation Example



$\mathrm{Opt}(i)$ is $\max(\mathrm{Opt}(i-1), 1+\mathrm{Opt}(p(i)))$ and $p(1) = p(3) = 0$, $p(2) = p(5) = 1$, $p(4) = 3$, , $p(6) = 4$ so $\mathrm{Opt}(1) = 1$.

- $\mathrm{Opt}(2) = \max(\mathrm{Opt}(1), 1 + \mathrm{Opt}(p(2))) = 2$.
- $\mathrm{Opt}(3) = \max(2, 1 + \mathrm{Opt}(0)) = 2$.
- $\mathrm{Opt}(4) = \max(2, 1 + Opt(p(4))) = 3$.
- $\mathrm{Opt}(5) = \max(2, 1 + \mathrm{Opt}(p(5)) = 3$
- $\mathrm{Opt}(6) = \max(3, 1 + \mathrm{Opt}(p(6))) = 4$

The Bellman Equation Example



Once you have computed Opt(6) equals 3, you can work back to get the schedule.

$$\text{Opt}(6) = 1 + \text{Opt}(p(6) = 4)) \ ,$$
$$\text{Opt}(4) = 1 + \text{Opt}(p(4) = 2),$$
$$\text{Opt}(2) = 1 + \text{Opt}(1).$$

So we use jobs $1, 2, 4, 6$.

The complexity of non-overlapping intervals/jobs

Assume that we have n jobs. The algorithm has two stages:

1. Sort the jobs by finish time: $O(n \log(n))$.
2. Compute Opt(n) by implementing the Dynamic programming using caching or a loop. Worst case complexity $O(n)$.

So the overall complexity is $O(n \log(n) + n) = O(n \log(n))$. This is a true polynomial time dynamic programming algorithm is only depends on the input size n.

The sorting trick allows you to replace the question, what happens before time t, to what happens before job/interval i?

Non-Overlapping Weighted Interval Problem

Nearly the same setup as before, a number of jobs/task :

- s_i, start time of task i,
- f_i, finish time of task i,
- w_i weight of task i.

As before we assume that $f_1 \leq f_2 \leq \cdots \leq f_n$.

Now the goal is to find a set of non-overlapping tasks $\{t_1, \ldots, t_k\}$ such that the sum is

$$w_{t\_1} + \cdots + w_{t_k}$$

maximised.

Non-Overlapping Weighted Interval Problem

The Bellman equation needs a slight modification from before:

$$\text{Opt}(i) = \begin{cases} 0 & \text{if } i = 0 \\ \max(\text{Opt}(i-1), w_i + \text{Opt}(p(i))) & \text{if } i > 0 \end{cases}$$

If you don't use job i, then the optimum does not change. If you use job i, then you have weight w_i, but you have to backtrack to job $p(i)$ as before.

Important Strategy when using Dynamic Programming

Do not try to solve the assignment problem, try to solve the optimisation problem:

- How many coins do you need to give the minimum change?
- What is the maximum number of non-overlapping jobs that you can pick.
- What is the maximum weight set of non-overlapping jobs that you can pick.
- (see next problem) what is the largest set of items you can pick that maximises the value but stays withing a weight limit w.

Once you solve this problem you can then find the assignment by either going backwards through your dynamic program or using additional data structures to keep track of what choices you made. This really helps in your first assignment.

A Knapsack[1]

Knapsack is just a fancy word for a backpack. Typically the old type of backpack that was used in the army (Ränsel på Svenska).

[1] Picture taken form Wikipedia

The Knapsack problem

You have a set of items each with a value, v_i, and a weight, w_i, you want to find a subset of the items that maximises the value but stays within a weight limit w.

i	v_i	w_i
1	100 SEK	1kg
2	600 SEK	2kg
3	180 SEK	5kg
4	220 SEK	6kg
5	280 SEK	7kg

For example : the subset $\{1, 2, 5\}$ has value 980 SEK and weight 10kg.

What do we optimise?

This problem is a little more complicated there are a few obvious choices:

1. Opt(w) the optimum value of the knapsack problem with weight w.
2. Opt(i) the optimum value of the knapsack using only the first i elements.
3. Opt(i, w) the optimum value of the knapsack using only the first i elements subject to the weight limit w.

Somehow we have get the value w in there and have suitable sub-problems. If I pick item i, it has weight w_i so I solve the sub-problem with weight $w - w_i$.

This means that the third choice seems the best thing to try.

Bellman equation for Knapsack

Definition

Opt(i, w) the optimal value of the knapsack problem using items $1, \ldots, i$ subject to the weight limit w.

The goal is to hit the weight limit w.
As before two cases :

1. Opt(i, w) does not select item i, so Opt(i, w) selects the best of $\{1, 2, \ldots, i-1\}$ subject to weight limit w.
2. Opt(i, w) selects item i, with value v_i, and a new weight limit $w - w_i$ we then select the best of $\{1, 2, \ldots, i-1\}$ subject to the weight limit $w - w_i$.

We do not select an item if it too heavy.

Bellman equation for Knapsack

This gives

$$\text{Opt}(i, w) = \begin{cases} 0 & \text{if } i = 0 \\ \text{Opt}(i-1, w) & \text{if } w_i > w \\ \max(\text{Opt}(i-1, w), v_i + \text{Opt}(i-1, w-w_i)) & \text{otherwise.} \end{cases}$$

Once you have the Bellman equation you can then implement it as a top-down (with caching) or bottom up dynamic programming.

It is always (as we will do now) worth working out the bottom up program as it gives you a better idea of the complexity of the problem.

Modified Example (easier to fit on a slide)

i	v_i	w_i
1	1	1kg
2	6	2kg
3	18	3kg
4	22	5kg

Let solve it for weight limit 5kg.

Opt(i, w) for our example

We are trying to compute Opt$(4, 5)$. We do this bottom using the Bellman equation from before.
If you do not pick any items then you satisfy the weight limit, but you have value 0.

	0	1	2	3	4	5
$\{\}$	0	0	0	0	0	0

We are trying to compute Opt$(4, 5)$. We do this bottom using the Bellman equation from before.

The next row is simple if you just use item 1, then it has weight 1.

	0	1	2	3	4	5
$\{\}$	0	0	0	0	0	0
$\{1\}$	0	1	1	1	1	1

We are trying to compute Opt$(4, 5)$. We do this bottom using the Bellman equation from before.
Continuing we get :

	0	1	2	3	4	5
$\{\}$	0	0	0	0	0	0
$\{1\}$	0	1	1	1	1	1
$\{1, 2\}$	0	1				
$\{1, 2, 3\}$	0	1				
$\{1, \ldots, 4\}$	0	1				

We are trying to compute $\text{Opt}(4, 5)$. We do this bottom using the Bellman equation from before.
Interesting things start happening when we look at $\text{Opt}(2, 2)$ we get that $\text{Opt}(2, 2) = v_2 + \text{Opt}(1, 2 - w_2) = 6 + 0$.

	0	1	2	3	4	5
$\{\}$	0	0	0	0	0	0
$\{1\}$	0	1	1	1	1	1
$\{1, 2\}$	0	1	**6**	7	7	7
$\{1, 2, 3\}$	0	1				
$\{1, \ldots, 4\}$	0	1				

We are trying to compute $\text{Opt}(4, 5)$. We do this bottom using the Bellman equation from before.
$\text{Opt}(3, 5) = v_3 + \text{Opt}(2, 5 - 3)$

	0	1	2	3	4	5
$\{\}$	0	0	0	0	0	0
$\{1\}$	0	1	1	1	1	1
$\{1, 2\}$	0	1	**6**	7	7	7
$\{1, 2, 3\}$	0	1	6	18	19	**24**

We are trying to compute $\text{Opt}(4, 5)$. We do this bottom using the Bellman equation from before.
$\text{Opt}(3, 5) = v_3 + \text{Opt}(2, 5 - 3)$

	0	1	2	3	4	5
$\{\}$	0	0	0	0	0	0
$\{1\}$	**0**	1	1	1	1	1
$\{1, 2\}$	0	1	**6**	7	7	7
$\{1, 2, 3\}$	0	1	6	18	19	**24**
$\{1, \ldots, 4\}$	0	1	6	18	22	**24**

Working back from our solution we get that
$\text{Opt}(4, 5) = \text{Opt}(3, 5) = v_3 + \text{Opt}(2, 2) = v + 3 + v_2 + \text{Opt}(1, 0)$
So we are using items 3 and 2.

What is the complexity of Knapsack?

Bottom up has given a vital clue to understanding the complexity of the problem.

Given an input, what is the maximum weight input?

If W is the maximum weight value in our inputs then we have to make a table of dimensions n by W in the worst case. In the worst case the maximum weight we can carry is nW. Set every weight to W.

Constructing each entry in the table is constant time, so the complexity is

$$\Theta(nW)$$

Again this is pseudo-polynomial.

Knapsack

A lot of problems are special cases of the knapsack problem, such as the coin change problem.

But be careful, sometimes there is a simpler dynamic program that solves what you are looking for.

Longest Increasing Sub-Sequence

One last polynomial time dynamic program.
Given a sequence of integers:

$$1, 3, 2, 10, 2, 11, 5$$

Find a longest increasing sub-sequence.

$$\mathbf{1}, \mathbf{3}, 2, \mathbf{10}, 2, \mathbf{11}, 5$$

Longest Increasing Sub-Sequence

Let denote a sequence of length n by $s_1, \ldots, s_{n-1}$ Again solve the easier problem: What is the length $l_{\max}$ of a longest increasing sub-sequence:

$$l_{\max}(i) = 1 + \max\{l_{\max}(j) \mid 1 \leq j < i \text{ and } s_j < s_i\}$$

If you set the maximum of an empty set to be 0, then if no such j exists satisfying the set comprehension above you get $l_{\max}(i) = 1$.

Longest Increasing Sub-Sequence

So you go through all the smaller j, look at $l_{\max}(j)$ and pick that value. If you implement with caching or bottom up you get $O(n^2)$ complexity because for each i you look at the values of $l_{\max}(1), \ldots, l_{\max}(i-1)$.

Again solve the easier problem: What is the length? Then you can find the solution by going backwards from your solution.

Summary

- Try to solve the harder problem, how to I maximize or minimize some value or sometimes does there exist a solution. By backtracking through your dynamic program's table you can find a solution.
- Sometimes Dynamic programs are pseudo-polynomial. The complexity is polynomial (often linear) in some parameter of the input, rather than the input size. Pseudo-polynomial is still good for low values of the parameter.
- In a dynamic program you are often trying to select a subset of items. You can often use the trick of considering the first i items to get your dynamic program. Sometimes you need to sort them, sometimes you don't, but you have to think very hard about this step to see if it is correct.

Summary

- Dynamic programming is not simple recursion, sometimes you need to use n-dimensional tables (knapsack) where one of the argument is a value rather than decomposing the problem. For example, the weight.
- If you are considering the first $i-1$ items, then in the next step you have the choice:
  - Use item i
  - Don't use item i.
- With each choice, you are working out the effect on the value that you are trying to maximize.

Dynamic programming is a powerful technique for getting efficient algorithms. Working out the correct decomposition is often hard. Practice makes perfect. The internet is full of examples, and problem sets.