Dynamic Programming

Dynamic Programming

High-level Idea: Break a problem into smaller subproblems (like divide-and-conquer) but dividing into **many overlapping** subproblems.

The DP technique applies to problems that obey the **principle of optimality**: the overall optimal solution can be constructed from optimal solutions to smaller subproblems

Warm-Up: Fibonacci

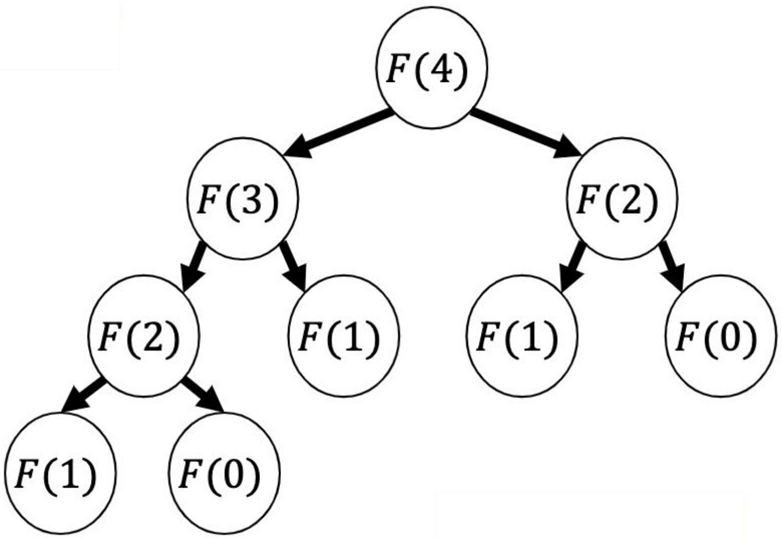
Recurrence for Fibonacci: $F(n) = F(n-1) + F(n-2)$

$F(0) = F(1) = 1$

Given a recurrence, three ways to compute its values:

1. **Top-down Recursive:** Starting at desired input, recurse down to base case(s)
2. **Top-down with Memoization:** Same as naïve, but save results as they're computed, reusing already-computed results
3. **Bottom-up Table (aka Dynamic Programming):** Start from base case(s), build up to desired result

Fibonacci Method 1: Top-down Recursive



Algorithm Fib(n):

    If n = 0 OR n = 1:

        Return 1

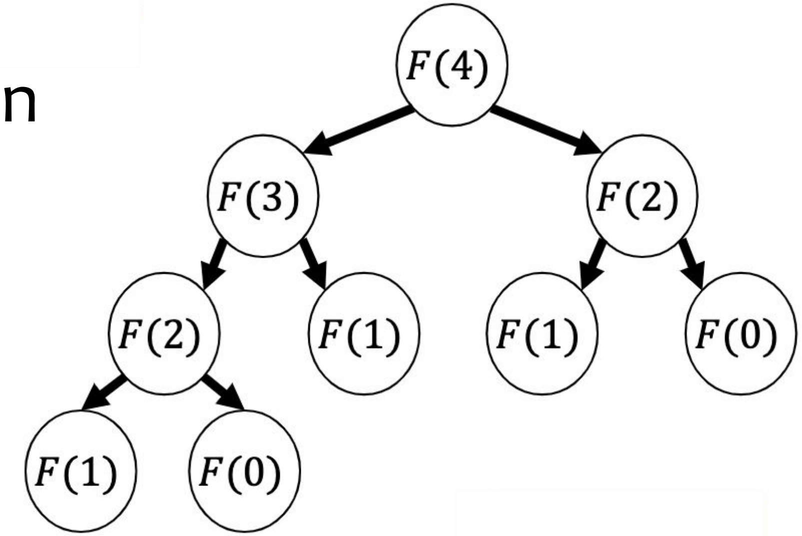
    Else:

        Return Fib(n-1) + Fib(n-2)

- **Pro:** direct translation of recurrence
- **Con:** exponential running time

Fibonacci Method 2: Top-down with Memoization

memo := array indexed from 0 to n



Algorithm Fib(n):
If n = 0 OR n = 1:
Return 1
Else if **memo**(n) is empty:
memo(n) = Fib(n-1) + Fib(n-2)
Return memo(n)

- **Pro:** way faster
- **Con:** not clear how to analyze running time

Fibonacci Method 3: Bottom-up Table (aka Dynamic Programming)

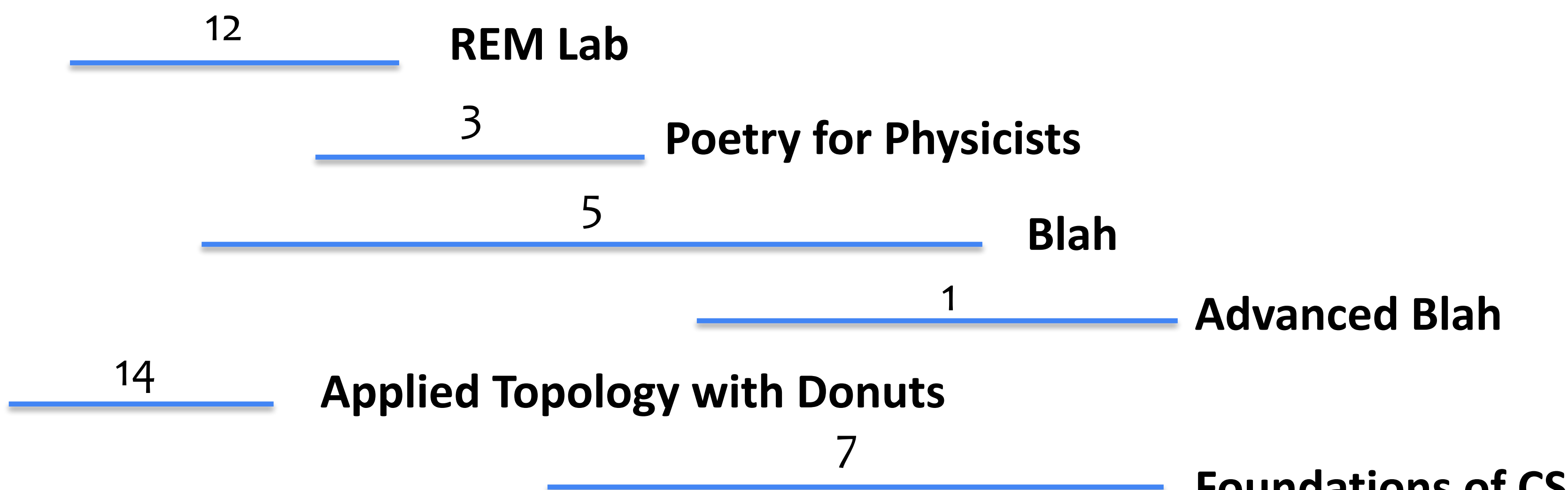
Algorithm Fib(n):

```
table := array indexed from 0 to n
table(0) = 1
table(1) = 1
for i = 2 to n:
    table(i) = table(i-1) + table(i-2)
Return table(n)
```

- **Pro:** fast, provides a roadmap for analyzing running time
- **Con:** need to translate from recurrence to table

Weighted Course Registration Problem

(aka Weighted Task Selection)



Goal: Choose a set of non-intersecting courses with largest total value.
(there may be many optimal solutions, we just seek one)

*Let the input size be n = #intervals. (Assume the weights are small enough that we can disregard their contribution to the input size.)

Weighted Task Selection Recurrence

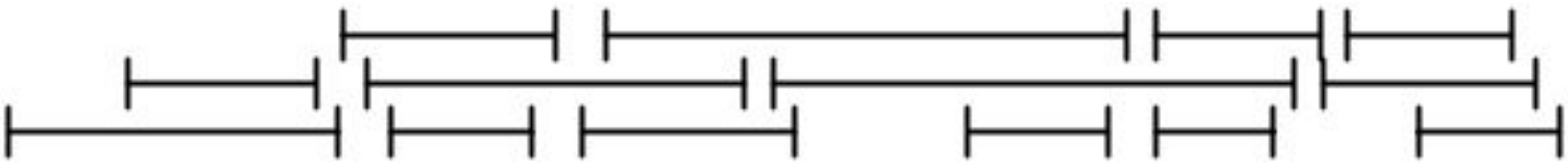
Assume the intervals $J_1, J_2, \ldots, J_n$ are given in order of **finish time**.

Let's start from J_n and work backwards.

There are two options:

- OPT has J_n **(use it!)**

- OPT doesn't have J_n **(lose it!)**



Weighted Task Selection Recurrence

Assume the intervals $\mathbf{J_1, J_2, \ldots, J_n}$ are given in order of **finish time**.

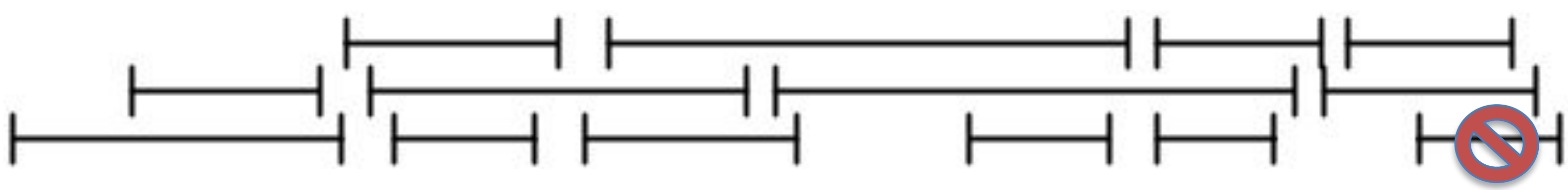
Let's start from $\mathbf{J_n}$ and work backwards.

There are two options:

- OPT has $\mathbf{J_n}$ **(use it!)**

- OPT doesn't have $\mathbf{J_n}$ **(lose it!)** ⬅️ OTV($\mathbf{J_1, \ldots, J_n}$) = OTV($\mathbf{J_1, \ldots, J_{n-1}}$)

"Optimal Task Value" i.e. the value of the optimal solution



Weighted Task Selection Recurrence

Assume the intervals $J_1, J_2, \dots, J_n$ are given in order of **finish time**.

Let's start from J_n and work backwards.

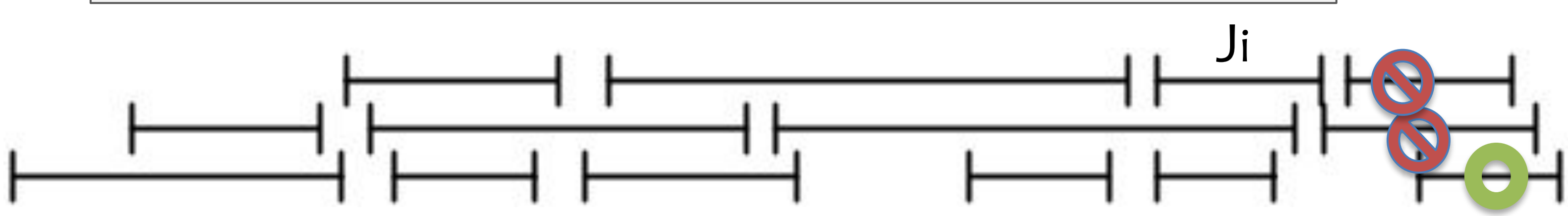
There are two options:

- OPT has J_n **(use it!)** ⬅️ $OTV(J_1, \dots, J_n) = val(J_n) + OTV(J_1, \dots, J_i)$

  J_i is the last interval that doesn't overlap with J_n

- OPT doesn't have J_n **(lose it!)** ⬅️ $OTV(J_1, \dots, J_n) = OTV(J_1, \dots, J_{n-1})$

  "Optimal Task Value" i.e. the value of the optimal solution



The Final Recurrence

You could write code to implement this recurrence as an algorithm…

Algorithm OTV($J_1, \ldots, J_n$):

    if n = 0:

        Return 0

    else:

        i = index of last interval (before J_n) that doesn't overlap with J_n

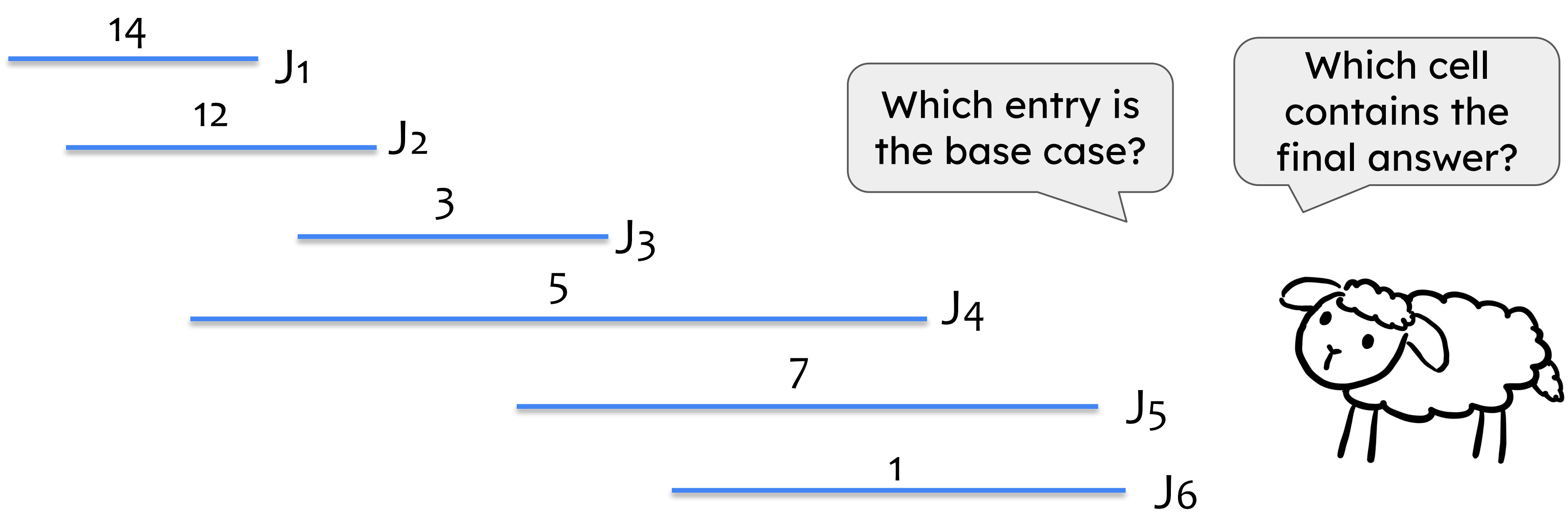
        Return max{val(J_n) + OTV($J_1, \ldots, J_i$), OTV($J_1, \ldots, J_{n-1}$)}

…but it would run in exponential time

Dynamic Programming in Action!

For reference: $OTV(J_1, \ldots, J_n) = \max\{val(J_n) + OTV(J_1, \ldots, J_i), OTV(J_1, \ldots, J_{n-1})\}$

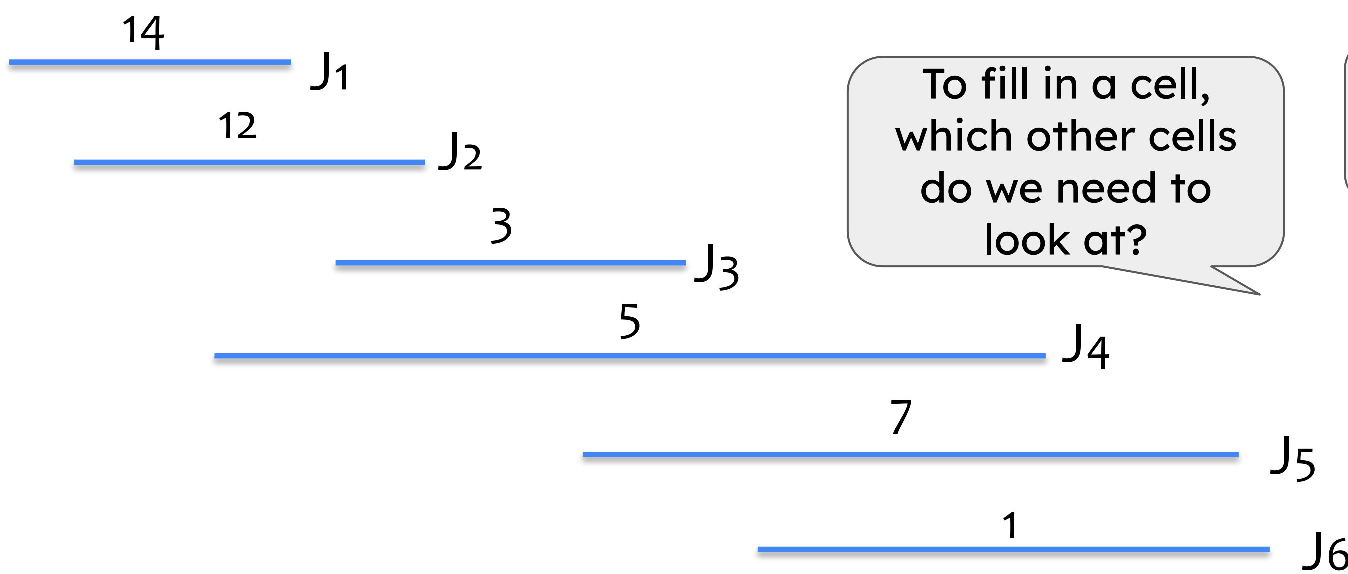
$OTV(\emptyset)$	$OTV(J_1)$	$OTV(J_1,J_2)$	$OTV(J_1,\ldots,J_3)$	$OTV(J_1,\ldots,J_4)$	$OTV(J_1,\ldots,J_5)$	$OTV(J_1,\ldots,J_6)$



Dynamic Programming in Action!

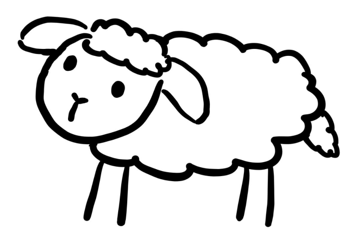
For reference: $OTV(J_1, \ldots, J_n) = \max\{val(J_n) + OTV(J_1, \ldots, J_i), OTV(J_1, \ldots, J_{n-1})\}$

$OTV(\emptyset)$	$OTV(J_1)$	$OTV(J_1,J_2)$	$OTV(J_1,\ldots,J_3)$	$OTV(J_1,\ldots,J_4)$	$OTV(J_1,\ldots,J_5)$	$OTV(J_1,\ldots,J_6)$



To fill in a cell, which other cells do we need to look at?

In which order do we fill the table?



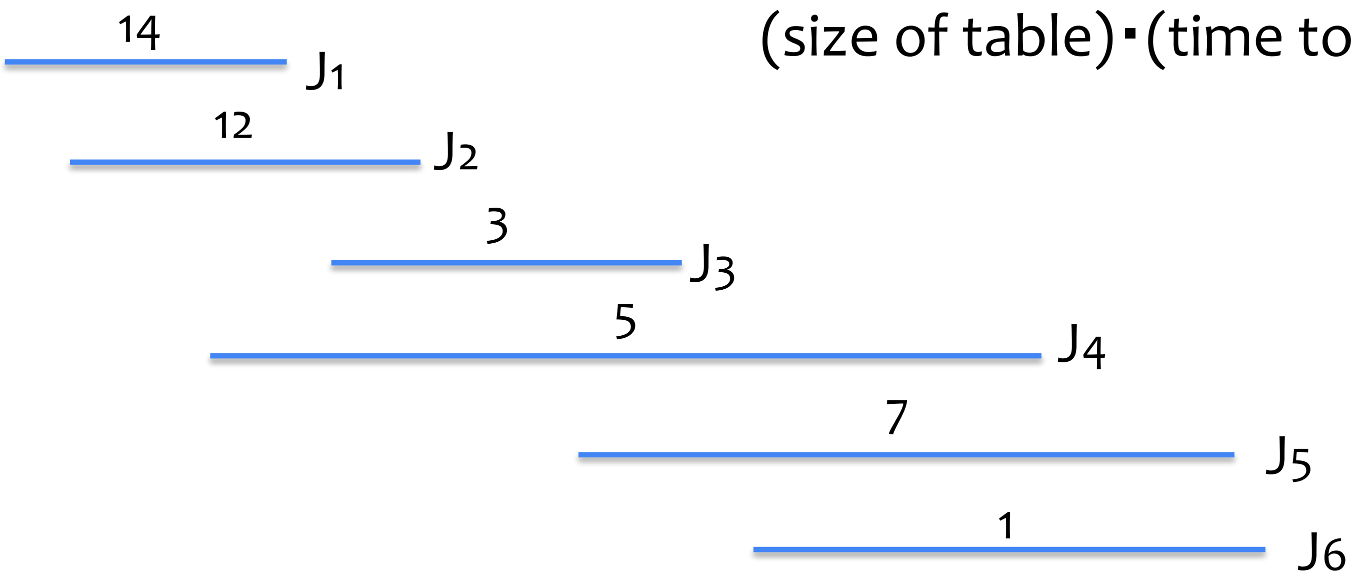
Dynamic Programming in Action!

For reference: $OTV(J_1, \ldots, J_n) = \max\{val(J_n) + OTV(J_1, \ldots, J_i), OTV(J_1, \ldots, J_{n-1})\}$

$OTV(\emptyset)$	$OTV(J_1)$	$OTV(J_1,J_2)$	$OTV(J_1,\ldots,J_3)$	$OTV(J_1,\ldots,J_4)$	$OTV(J_1,\ldots,J_5)$	$OTV(J_1,\ldots,J_6)$

Running time =

(size of table)·(time to fill each entry)

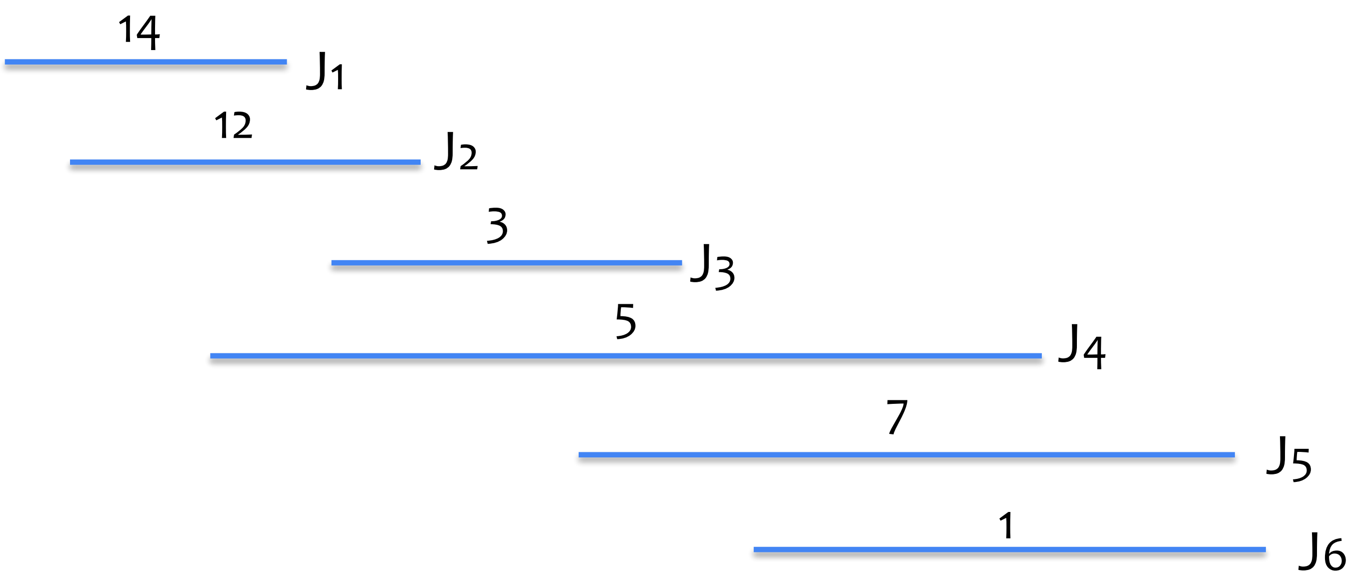


Dynamic Programming in Action!

For reference: $OTV(J_1, \ldots, J_n) = \max\{val(J_n) + OTV(J_1, \ldots, J_i), OTV(J_1, \ldots, J_{n-1})\}$

$OTV(\emptyset)$	$OTV(J_1)$	$OTV(J_1,J_2)$	$OTV(J_1,\ldots,J_3)$	$OTV(J_1,\ldots,J_4)$	$OTV(J_1,\ldots,J_5)$	$OTV(J_1,\ldots,J_6)$

Reconstructing the solution



The Final Pseudocode

```
Algorithm OTV(J1, ..., Jn):
    table := array indexed from 0 to n
    table(0) = 0
    for k = 1 to n:
        i = index of last interval (before Jk) that doesn't overlap with Jk
            // E.g. iterate through all intervals to find i
        table(k) = max{val(Jk) + table(i), table(k-1)}
    Return table(n)
```

The DP Recipe

1. Write recurrence ← usually the trickiest part
2. Size of table: How many dimensions? Range of each dimension?
3. What are the base cases?
4. To fill in a cell, which other cells do I look at? In which order do I fill the table?
5. Which cell(s) contain the final answer?
6. Running time = (size of table)·(time to fill each entry)
7. To reconstruct the solution (instead of just its size) follow arrows from final answer to base case

The Final Pseudocode

Algorithm OTV($\mathbf{J_1, \ldots, J_n}$):

 table := array indexed from 0 to n ⬅ step 2 of DP recipe

 table(0) = 0 ⬅ step 3 of DP recipe

 for k = 1 to n: ⬅ step 4 of DP recipe

  i = index of last interval (before $\mathbf{J_k}$) that doesn't overlap with $\mathbf{J_k}$

  table(k) = max{val($\mathbf{J_k}$) + **table**(**i**), **table**(**k-1**)}

 Return table(n) ⬅ step 5 of DP recipe

⬆ steps 1,4 of DP recipe

Why is it called “dynamic programming”?

We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word, research.

[…]

His face would suffuse, he would turn red, and he would get violent if people used the term, research, in his presence.

[…]

The RAND Corporation [where Bellman worked] was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, **I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation.**

[…]

In the first place I was interested in planning, in decision making, in thinking. But planning, is not a good word for various reasons. I decided therefore to use the word, ‘programming.’

[…]

It’s impossible to use the word, dynamic, in a pejorative sense. Try thinking of some combination that will possibly give it a pejorative meaning. It’s impossible. Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities.

-Richard Bellman (introduced dynamic programming in 1953)

Another example of DP:

Longest Increasing Subsequence (LIS)

Input: Array A of n numbers

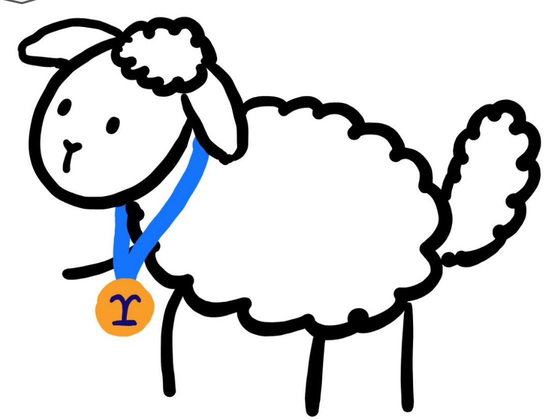
Output: Longest increasing subsequence of A

Example: What is the LIS of [1,4,3,7,2]?

Recurrence for LIS

It’s easy! Check it out!

$$\mathrm{LIS}(A[1..n]) = \begin{cases} \mathrm{LIS}(A[1..n-1]) + 1 & \text{if } A[n] > A[n-1] \quad \textbf{(use it!)} \\ \mathrm{LIS}(A[1..n-1]) & \text{otherwise} \quad \textbf{(lose it!)} \end{cases}$$



Counterexample:

Recurrence for LIS

Defn: Let END-LIS(**A[1..i]**) be the length of the LIS of **A[1..i]** ***that ends in A[i].***

When A = [1, 3, 4, 2, 6], what is END-LIS(A[1..i]) for each i?

Recurrence:

END-LIS(**A[1..i]**) = 1 + max{END-LIS(**A[1..j]**) among all j<i with **A[j]**<**A[i]**}
(if such j exists)

Base case: END-LIS(**A[1..i]**) =1 if A[i] is the smallest element so far in A

Let's follow the DP Recipe

A =	3	6	1	4	5	9	2	4
END-LIS(A[1..i])								

Pseudocode

Algorithm LIS(A[1..n]) :

    table := array indexed from 1 to n

    for i = 1 to n:

        if A[i] is the minimum so far: // determined by scanning A[1..i-1]

            table(i) = 1 // base case

        Else:

            table(i) = 1+ max{**table**(j) among all j<i with **A[j]**<**A[i]**}

                // find j by scanning **table**

    Return $\max_i$ **table**(i)