BLG 336E Analysis of Algorithms II

Lecture 9:

Dynamic Programming II

DNA Sequencing, Knapsack Problem

Last time



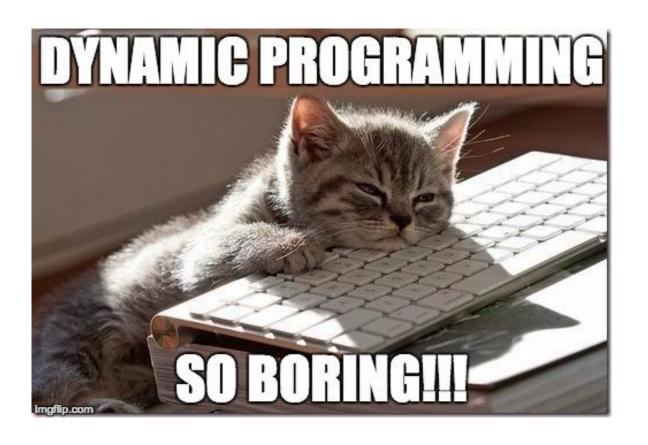
- Dynamic programming is an algorithm design paradigm.
- Basic idea:
 - Identify optimal sub-structure
 - Optimum to the big problem is built out of optima of small sub-problems
 - Take advantage of overlapping sub-problems
 - Only solve each sub-problem once, then use it again and again
 - Keep track of the solutions to sub-problems in a table as you build to the final solution.

Today

- Examples of dynamic programming:
 - 1. Longest common subsequence
 - 2. Knapsack problem
 - Two versions!
 - 3. Independent sets in trees
 - If we have time...
 - (If not the slides will be there as a reference)

The goal of this lecture

For you to get really bored of dynamic programming



Longest Common Subsequence

How similar are these two species?



DNA:

AGCCCTAAGGGCTACCTAGCTT



DNA:
GACAGCCTACAAGCGTTAGCTTG

Longest Common Subsequence

How similar are these two species?



AGCCCTAAGGGCTACCTAGCTT



GACAGCCTACAAGCGTTAGCTTG

Pretty similar, their DNA has a long common subsequence:

AGCCTAAGCTTAGCTT

DNA:

Longest Common Subsequence

- Subsequence:
 - BDFH is a subsequence of ABCDEFGH
- If X and Y are sequences, a **common subsequence** is a sequence which is a subsequence of both.
 - BDFH is a common subsequence of ABCDEFGH and of ABDFGHI
- A longest common subsequence...
 - ...is a common subsequence that is longest.
 - The longest common subsequence of ABCDEFGH and ABDFGHI is ABDFGH.

We sometimes want to find these

Applications in bioinformatics





- The unix command diff
- Merging in version control
 - svn, git, etc...

```
[DN0a22a660:~ mary$ cat file1
[DN0a22a660:~ mary$ cat file2
[DN0a22a660:~ mary$ diff file1 file2
3d2
5d3
DN0a22a660:~ mary$ ■
```

Recipe for applying Dynamic Programming

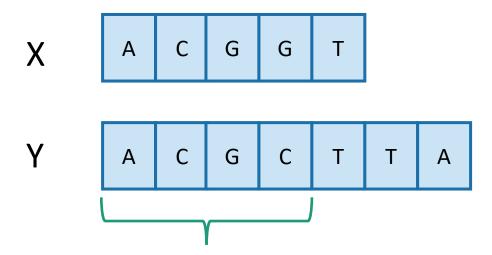
• Step 1: Identify optimal substructure.



- Step 2: Find a recursive formulation for the length of the longest common subsequence.
- Step 3: Use dynamic programming to find the length of the longest common subsequence.
- Step 4: If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual LCS.
- Step 5: If needed, code this up like a reasonable person.

Step 1: Optimal substructure

Prefixes:



Notation: denote this prefix **ACGC** by Y₄

- Our sub-problems will be finding LCS's of prefixes to X and Y.
- Let C[i,j] = length_of_LCS(X_i, Y_i)

Optimal substructure ctd.

- Subproblem:
 - finding LCS's of prefixes of X and Y.
- Why is this a good choice?
 - There's some relationship between LCS's of prefixes and LCS's of the whole things.
 - These subproblems overlap a lot.

To see this formally, on to...

Recipe for applying Dynamic Programming

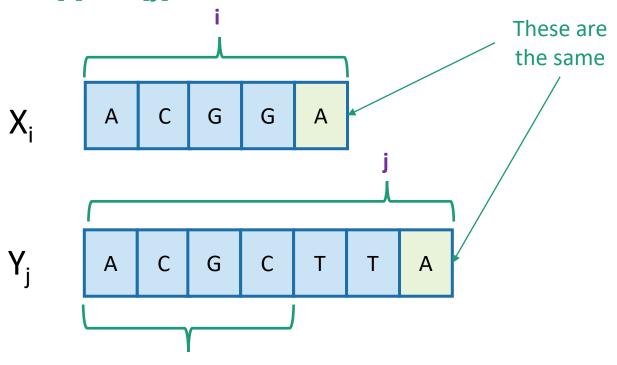
• Step 1: Identify optimal substructure.

- Step 2: Find a recursive formulation for the length of the longest common subsequence.
- Step 3: Use dynamic programming to find the length of the longest common subsequence.
- Step 4: If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual LCS.
- Step 5: If needed, code this up like a reasonable person.

Two cases

Case 1: X[i] = Y[j]

- Our sub-problems will be finding LCS's of prefixes to X and Y.
- Let C[i,j] = length_of_LCS(X_i, Y_i)



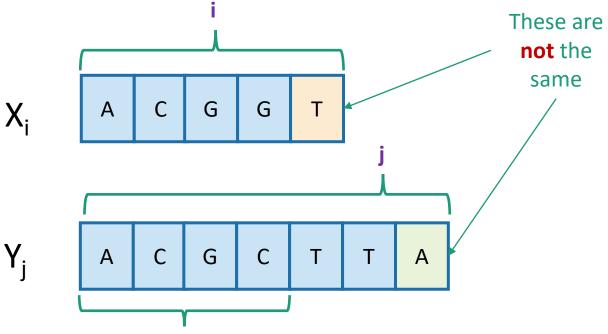
Notation: denote this prefix **ACGC** by Y₄

- Then C[i,j] = 1 + C[i-1,j-1].
 - because $LCS(X_i,Y_j) = LCS(X_{i-1},Y_{j-1})$ followed by

Two cases

Case 2: X[i] != Y[j]

- Our sub-problems will be finding LCS's of prefixes to X and Y.
- Let C[i,j] = length_of_LCS(X_i, Y_i)



Notation: denote this prefix **ACGC** by Y₄

- Then C[i,j] = max{ C[i-1,j], C[i,j-1] }.
 - either $LCS(X_i,Y_j) = LCS(X_{i-1},Y_j)$ and \top is not involved,
 - or $LCS(X_i,Y_i) = LCS(X_i,Y_{i-1})$ and A is not involved,

Recursive formulation of the optimal solution

•
$$C[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ C[i-1,j-1]+1 & \text{if } X[i] = Y[j] \text{ and } i,j > 0 \\ \max\{C[i,j-1],C[i-1,j]\} & \text{if } X[i] \neq Y[j] \text{ and } i,j > 0 \end{cases}$$

X_i A C G G A

Y_i A C G C T T A

Case 1

Case 2

Recipe for applying Dynamic Programming

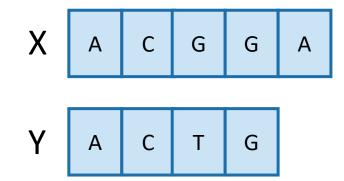
- Step 1: Identify optimal substructure.
- Step 2: Find a recursive formulation for the length of the longest common subsequence.
- Step 3: Use dynamic programming to find the length of the longest common subsequence.
- Step 4: If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual LCS.
- Step 5: If needed, code this up like a reasonable person.

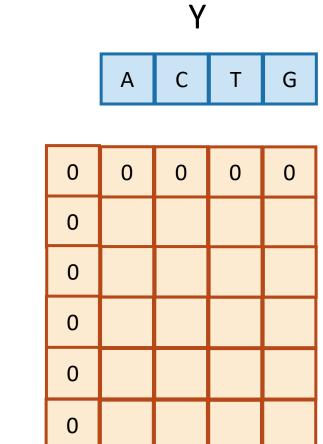
LCS DP omg bbq

- **LCS**(X, Y):
 - C[i,0] = C[0,j] = 0 for all i = 1,...,m, j=1,...n.
 - **For** i = 1,...,m and j = 1,...,n:
 - **If** X[i] = Y[j]:
 - C[i,j] = C[i-1,j-1] + 1
 - Else:
 - C[i,j] = max{ C[i,j-1], C[i-1,j] }

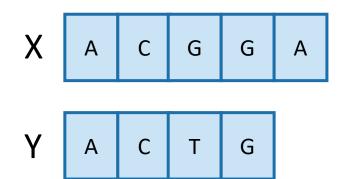
Running time: O(nm)

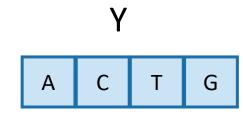
$$C[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ C[i-1,j-1] + 1 & \text{if } X[i] = Y[j] \text{ and } i,j > 0 \\ \max\{C[i,j-1],C[i-1,j]\} & \text{if } X[i] \neq Y[j] \text{ and } i,j > 0 \end{cases}$$





$$C[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ C[i-1,j-1]+1 & \text{if } X[i] = Y[j] \text{ and } i,j > 0 \\ \max\{C[i,j-1],C[i-1,j]\} & \text{if } X[i] \neq Y[j] \text{ and } i,j > 0 \end{cases}$$





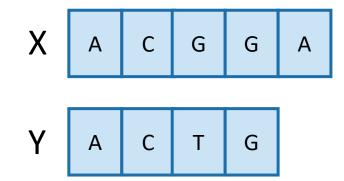
0	0	0	0	0
0	1	1	1	1
0	1	2	2	2
0	1	2	2	3
0	1	2	2	3
0	1	2	2	3

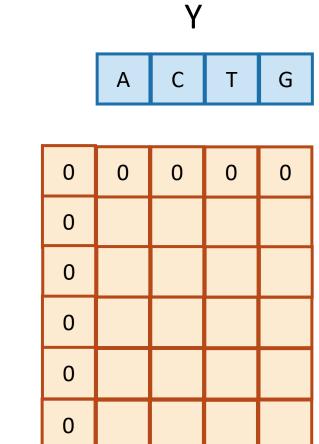
So the LCM of X and Y has length 3.

$$C[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ C[i-1,j-1] + 1 & \text{if } X[i] = Y[j] \text{ and } i,j > 0 \\ \max\{C[i,j-1],C[i-1,j]\} & \text{if } X[i] \neq Y[j] \text{ and } i,j > 0 \end{cases}$$

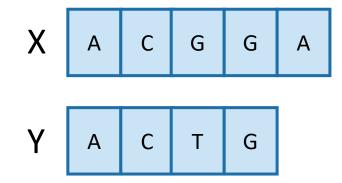
Recipe for applying Dynamic Programming

- Step 1: Identify optimal substructure.
- Step 2: Find a recursive formulation for the length of the longest common subsequence.
- Step 3: Use dynamic programming to find the length of the longest common subsequence.
- Step 4: If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual LCS.
- Step 5: If needed, code this up like a reasonable person.





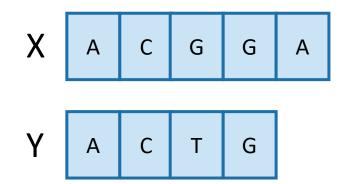
$$C[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ C[i-1,j-1]+1 & \text{if } X[i] = Y[j] \text{ and } i,j > 0 \\ \max\{C[i,j-1],C[i-1,j]\} & \text{if } X[i] \neq Y[j] \text{ and } i,j > 0 \end{cases}$$

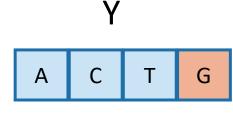


Υ						
А	С	Т	G			

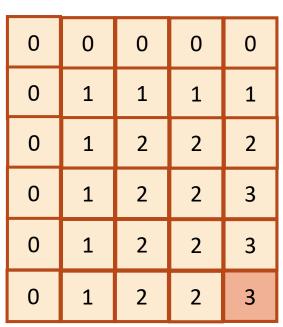
0	0	0	0	0
0	1	1	1	1
0	1	2	2	2
0	1	2	2	3
0	1	2	2	3
0	1	2	2	3

$$C[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ C[i-1,j-1]+1 & \text{if } X[i] = Y[j] \text{ and } i,j > 0 \\ \max\{C[i,j-1],C[i-1,j]\} & \text{if } X[i] \neq Y[j] \text{ and } i,j > 0 \end{cases}$$

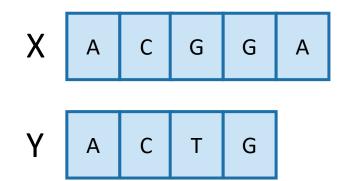


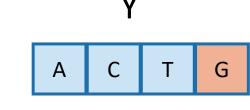


• Once we've filled this in, we can work backwards.

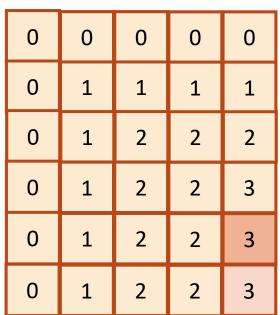


$$C[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ C[i-1,j-1]+1 & \text{if } X[i] = Y[j] \text{ and } i,j > 0 \\ \max\{C[i,j-1],C[i-1,j]\} & \text{if } X[i] \neq Y[j] \text{ and } i,j > 0 \end{cases}$$



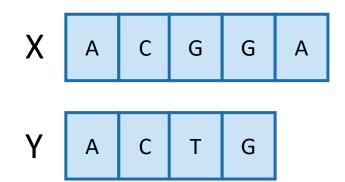


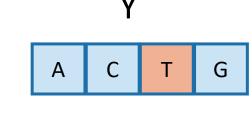
 Once we've filled this in, we can work backwards.



That 3 must have come from the 3 above it.

$$C[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ C[i-1,j-1] + 1 & \text{if } X[i] = Y[j] \text{ and } i,j > 0 \\ \max\{C[i,j-1],C[i-1,j]\} & \text{if } X[i] \neq Y[j] \text{ and } i,j > 0 \end{cases}$$

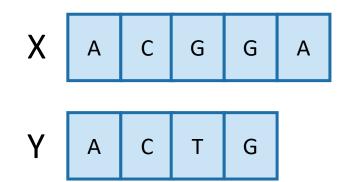


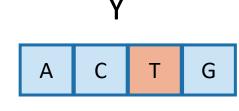


- Once we've filled this in, we can work backwards.
- A diagonal jump means that we found an element of the LCS!

This 3 came from that 2 – we found a match!

$$C[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0\\ C[i-1,j-1]+1 & \text{if } X[i] = Y[j] \text{ and } i,j > 0\\ \max\{C[i,j-1],C[i-1,j]\} & \text{if } X[i] \neq Y[j] \text{ and } i,j > 0 \end{cases}$$

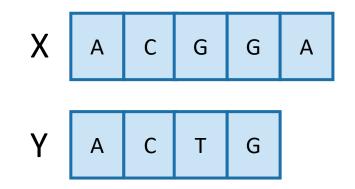


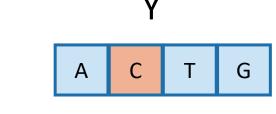


- Once we've filled this in, we can work backwards.
- A diagonal jump means that we found an element of the LCS!

That 2 may as well have come from this other 2.

$$C[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ C[i-1,j-1]+1 & \text{if } X[i] = Y[j] \text{ and } i,j > 0 \\ \max\{C[i,j-1],C[i-1,j]\} & \text{if } X[i] \neq Y[j] \text{ and } i,j > 0 \end{cases}$$

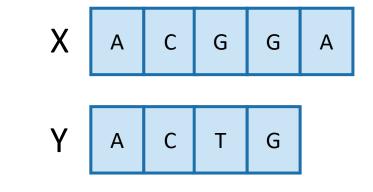


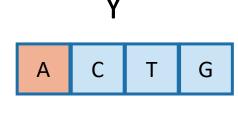


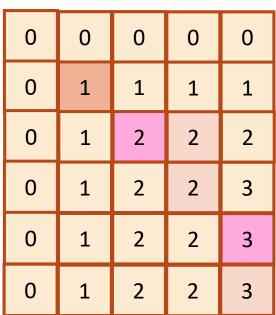
- Once we've filled this in, we can work backwards.
- A diagonal jump means that we found an element of the LCS!

G

$$C[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ C[i-1,j-1]+1 & \text{if } X[i] = Y[j] \text{ and } i,j > 0 \\ \max\{C[i,j-1],C[i-1,j]\} & \text{if } X[i] \neq Y[j] \text{ and } i,j > 0 \end{cases}$$





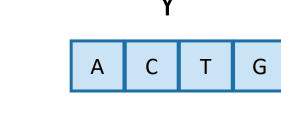


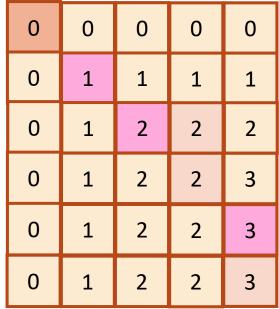
- Once we've filled this in, we can work backwards.
- A diagonal jump means that we found an element of the LCS!

CG

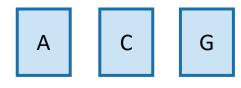
$$C[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ C[i-1,j-1]+1 & \text{if } X[i] = Y[j] \text{ and } i,j > 0 \\ \max\{C[i,j-1],C[i-1,j]\} & \text{if } X[i] \neq Y[j] \text{ and } i,j > 0 \end{cases}$$







- Once we've filled this in, we can work backwards.
- A diagonal jump means that we found an element of the LCS!



This is the LCS!

G

G

Α

$$C[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ C[i-1,j-1] + 1 & \text{if } X[i] = Y[j] \text{ and } i,j > 0 \\ \max\{C[i,j-1],C[i-1,j]\} & \text{if } X[i] \neq Y[j] \text{ and } i,j > 0 \end{cases}$$

This gives an algorithm to recover the actual LCS not just its length

- It runs in time O(n + m)
 - We walk up and left in an n-by-m array
 - We can only do that for n + m steps.
- So actually recovering the LCS from the table is much faster than building the table was.
- We can find LCS(X,Y) in time O(mn).

Recipe for applying Dynamic Programming

- Step 1: Identify optimal substructure.
- Step 2: Find a recursive formulation for the length of the longest common subsequence.
- Step 3: Use dynamic programming to find the length of the longest common subsequence.
- Step 4: If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual LCS.
- Step 5: If needed, code this up like a reasonable person.

This pseudocode actually isn't so bad

- If we are only interested in the length of the LCS:
 - Since we go across the table one-row-at-a-time, we can only keep two rows if we want.
 - If we want to recover the LCS, we need to keep the whole table.
- Can we do better than O(mn) time?
 - A bit better.
 - By a log factor or so.
 - But doing much better (polynomially better) is an open problem!
 - If you can do it let me know :D

What have we learned?

- We can find LCS(X,Y) in time O(nm)
 - if |Y|=n, |X|=m
- We went through the steps of coming up with a dynamic programming algorithm.
 - We kept a 2-dimensional table, breaking down the problem by decrementing the length of X and Y.

Example 2: Knapsack Problem

We have n items with weights and values:

 Item:
 <th

- And we have a knapsack:
 - it can only carry so much weight:



Capacity: 10



Capacity: 10











Item:

Weight:

6

2

4

3

11

Value:

20

8

14

13

35

Unbounded Knapsack:

- Suppose I have infinite copies of all of the items.
- What's the most valuable way to fill the knapsack?









Total weight: 10 Total value: 42

• 0/1 Knapsack:

- Suppose I have only one copy of each item.
- What's the most valuable way to fill the knapsack?







Total weight: 9
Total value: 35

Some notation





• Step 1: Identify optimal substructure.

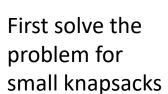


- Step 2: Find a recursive formulation for the value of the optimal solution.
- Step 3: Use dynamic programming to find the value of the optimal solution.
- Step 4: If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual solution.
- Step 5: If needed, code this up like a reasonable person.

Optimal substructure

- Sub-problems:
 - Unbounded Knapsack with a smaller knapsack.







Then larger knapsacks



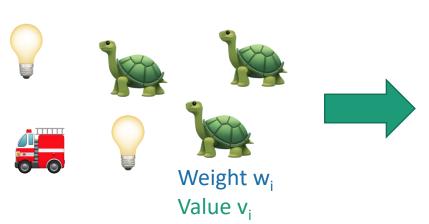
Then larger knapsacks

Optimal substructure



Suppose this is an optimal solution for capacity x:

Say that the optimal solution contains at least one copy of item i.





Capacity x Value V

Then this optimal for capacity x - w_i:





If I could do better than the second solution, then adding a turtle to that improvement would improve the first solution.

Capacity x – w_i Value V - v_i

- Step 1: Identify optimal substructure.
- Step 2: Find a recursive formulation for the value of the optimal solution.
- Step 3: Use dynamic programming to find the value of the optimal solution.
- Step 4: If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual solution.
- Step 5: If needed, code this up like a reasonable person.

Recursive relationship

• Let K[x] be the optimal value for capacity x.

$$K[x] = \max_i \left\{ \begin{array}{c} + \\ \\ \end{array} \right\}$$
 The maximum is over all i so that $w_i \leq x$. Optimal way to fill the smaller knapsack

$$K[x] = \max_{i} \{ K[x - w_{i}] + v_{i} \}$$

- (And K[x] = 0 if the maximum is empty).
 - That is, there are no i so that $w_i \leq x$

- Step 1: Identify optimal substructure.
- Step 2: Find a recursive formulation for the value of the optimal solution.
- Step 3: Use dynamic programming to find the value of the optimal solution.
- Step 4: If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual solution.
- Step 5: If needed, code this up like a reasonable person.

Let's write a bottom-up DP algorithm

- UnboundedKnapsack(W, n, weights, values):
 - K[0] = 0
 - for x = 1, ..., W:
 - K[x] = 0
 - **for** i = 1, ..., n:
 - if $w_i \leq x$:
 - $K[x] = \max\{K[x], K[x w_i] + v_i\}$
 - return K[W]

Running time: O(nW)

Why does this work?
Because our recursive relationship makes sense.

Can we do better?

- We only need log(W) bits to write down the input W and to write down all the weights.
- Maybe we could have an algorithm that runs in time O(nlog(W)) instead of O(nW)?
- Or even O(n¹⁰⁰⁰⁰⁰⁰ log¹⁰⁰⁰⁰⁰⁰(W))?

- Open problem!
 - (But probably the answer is no...otherwise P = NP)

- Step 1: Identify optimal substructure.
- Step 2: Find a recursive formulation for the value of the optimal solution.
- Step 3: Use dynamic programming to find the value of the optimal solution.
- Step 4: If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual solution.
- Step 5: If needed, code this up like a reasonable person.

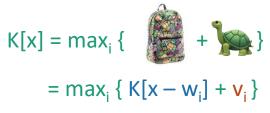
Let's write a bottom-up DP algorithm

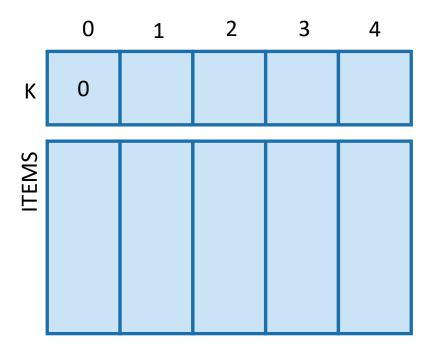
- UnboundedKnapsack(W, n, weights, values):
 - K[0] = 0
 - for x = 1, ..., W:
 - K[x] = 0
 - **for** i = 1, ..., n:
 - if $w_i \leq x$:
 - $K[x] = \max\{K[x], K[x w_i] + v_i\}$
 - return K[W]

```
K[x] = \max_{i} \{ \left[ \left[ x - w_{i} \right] + \frac{v_{i}}{v_{i}} \right] \}
```

Let's write a bottom-up DP algorithm

- UnboundedKnapsack(W, n, weights, values):
 - K[0] = 0
 - ITEMS[0] = Ø
 - for x = 1, ..., W:
 - K[x] = 0
 - **for** i = 1, ..., n:
 - if $w_i \leq x$:
 - $K[x] = \max\{K[x], K[x w_i] + v_i\}$
 - If K[x] was updated:
 - ITEMS[x] = ITEMS[x w_i] U { item i }
 - return ITEMS[W]



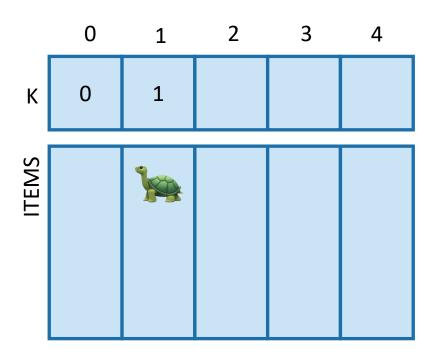


- UnboundedKnapsack(W, n, weights, values):
 - K[0] = 0
 - ITEMS $[0] = \emptyset$
 - for x = 1, ..., W:
 - K[x] = 0
 - **for** i = 1, ..., n:
 - if $w_i \leq x$:
 - $K[x] = \max\{K[x], K[x w_i] + v_i\}$
 - If K[x] was updated:
 - ITEMS[x] = ITEMS[x w_i] U { item
 - return ITEMS[W]





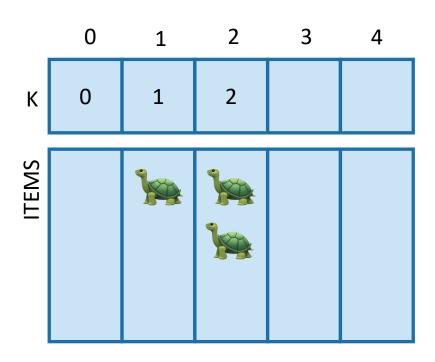
Capacity: 4



- UnboundedKnapsack(W, n, weights, values):
 - K[0] = 0
 - ITEMS $[0] = \emptyset$
 - for x = 1, ..., W:
 - K[x] = 0
 - **for** i = 1, ..., n:
 - if $w_i \leq x$:
 - $K[x] = \max\{K[x], K[x w_i] + v_i\}$
 - If K[x] was updated:
 - ITEMS[x] = ITEMS[x w_i] U { item
 - return ITEMS[W]





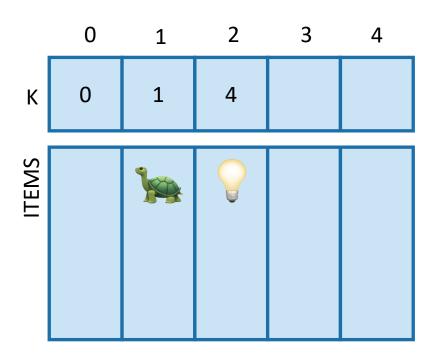


- UnboundedKnapsack(W, n, weights, values):
 - K[0] = 0
 - ITEMS $[0] = \emptyset$
 - for x = 1, ..., W:
 - K[x] = 0
 - **for** i = 1, ..., n:
 - if $w_i \leq x$:
 - $K[x] = \max\{K[x], K[x w_i] + v_i\}$
 - If K[x] was updated:
 - ITEMS[x] = ITEMS[x w_i] U { item
 - return ITEMS[W]





Capacity: 4

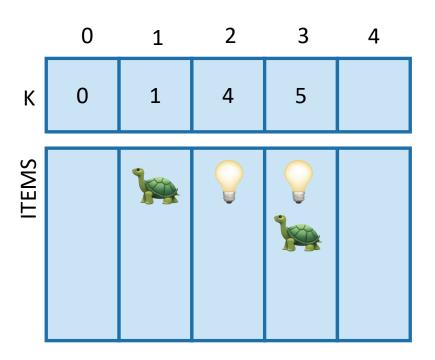


$$ITEMS[2] = ITEMS[0] +$$

- UnboundedKnapsack(W, n, weights, values):
 - K[0] = 0
 - ITEMS $[0] = \emptyset$
 - for x = 1, ..., W:
 - K[x] = 0
 - **for** i = 1, ..., n:
 - if $w_i \leq x$:
 - $K[x] = \max\{K[x], K[x w_i] + v_i\}$
 - If K[x] was updated:
 - ITEMS[x] = ITEMS[x w_i] U { item
 - return ITEMS[W]





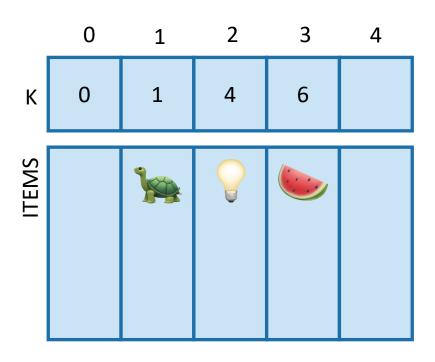


- UnboundedKnapsack(W, n, weights, values):
 - K[0] = 0
 - ITEMS $[0] = \emptyset$
 - for x = 1, ..., W:
 - K[x] = 0
 - **for** i = 1, ..., n:
 - if $w_i \leq x$:
 - $K[x] = \max\{K[x], K[x w_i] + v_i\}$
 - If K[x] was updated:
 - ITEMS[x] = ITEMS[x w_i] U { item
 - return ITEMS[W]





Capacity: 4

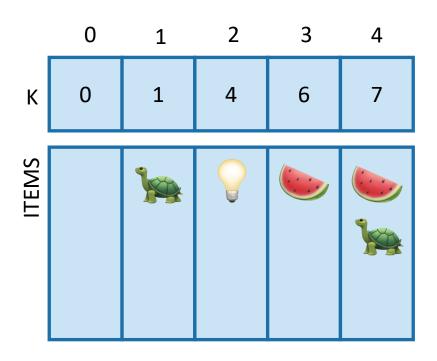


- UnboundedKnapsack(W, n, weights, values):
 - K[0] = 0
 - ITEMS $[0] = \emptyset$
 - for x = 1, ..., W:
 - K[x] = 0
 - **for** i = 1, ..., n:
 - if $w_i \leq x$:
 - $K[x] = \max\{K[x], K[x w_i] + v_i\}$
 - If K[x] was updated:
 - ITEMS[x] = ITEMS[x w_i] U { item
 - return ITEMS[W]





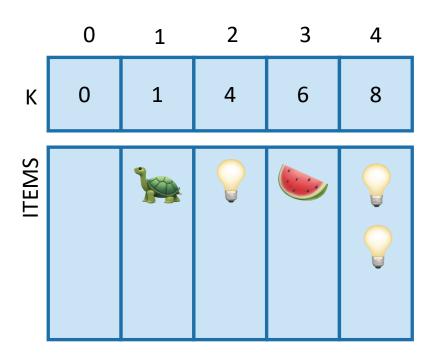
Capacity: 4



- UnboundedKnapsack(W, n, weights, values):
 - K[0] = 0
 - ITEMS $[0] = \emptyset$
 - for x = 1, ..., W:
 - K[x] = 0
 - **for** i = 1, ..., n:
 - if $w_i \leq x$:
 - $K[x] = \max\{K[x], K[x w_i] + v_i\}$
 - If K[x] was updated:
 - ITEMS[x] = ITEMS[x w_i] U { item
 - return ITEMS[W]







$$ITEMS[4] = ITEMS[2] +$$

- UnboundedKnapsack(W, n, weights, values):
 - K[0] = 0
 - ITEMS $[0] = \emptyset$
 - for x = 1, ..., W:
 - K[x] = 0
 - **for** i = 1, ..., n:
 - if $w_i \leq x$:
 - $K[x] = \max\{K[x], K[x w_i] + v_i\}$
 - If K[x] was updated:
 - ITEMS[x] = ITEMS[x w_i] U { item
 - return ITEMS[W]





- Step 1: Identify optimal substructure.
- Step 2: Find a recursive formulation for the value of the optimal solution.
- Step 3: Use dynamic programming to find the value of the optimal solution.
- Step 4: If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual solution.
- Step 5: If needed, code this up like a reasonable person.

(Pass)

What have we learned?

- We can solve unbounded knapsack in time O(nW).
 - If there are n items and our knapsack has capacity W.

- We again went through the steps to create DP solution:
 - We kept a one-dimensional table, creating smaller problems by making the knapsack smaller.



Capacity: 10













Item: Weight:

Value:

20

14

13

35

Unbounded Knapsack:

- Suppose I have infinite copies of all of the items.
- What's the most valuable way to fill the knapsack?









Total weight: 10 Total value: 42



• 0/1 Knapsack:

- Suppose I have only one copy of each item.
- What's the most valuable way to fill the knapsack?







Total weight: 9 Total value: 35

• Step 1: Identify optimal substructure.



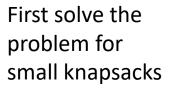
- Step 2: Find a recursive formulation for the value of the optimal solution.
- Step 3: Use dynamic programming to find the value of the optimal solution.
- Step 4: If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual solution.
- Step 5: If needed, code this up like a reasonable person.

Optimal substructure: try 1

- Sub-problems:
 - Unbounded Knapsack with a smaller knapsack.









Then larger knapsacks



Then larger knapsacks

This won't quite work...

- We are only allowed one copy of each item.
- The sub-problem needs to "know" what items we've used and what we haven't.





Optimal substructure: try 2

• Sub-problems:

• 0/1 Knapsack with fewer items.

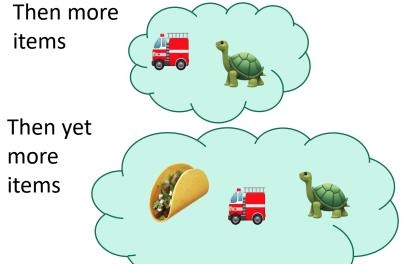
First solve the problem with few items







We'll still increase the size of the knapsacks.



(We'll keep a two-dimensional table).

Our sub-problems:

Indexed by x and j



First j items

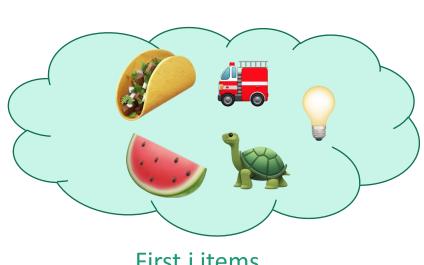


Capacity x

Two cases



- Case 1: Optimal solution for j items does not use item j.
- Case 2: Optimal solution for j items does use item j.



First j items

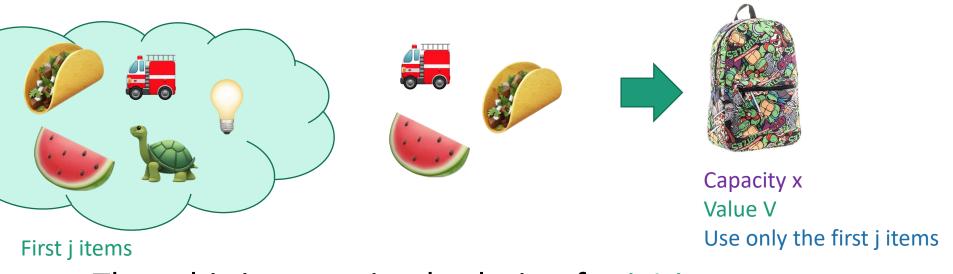


Capacity x

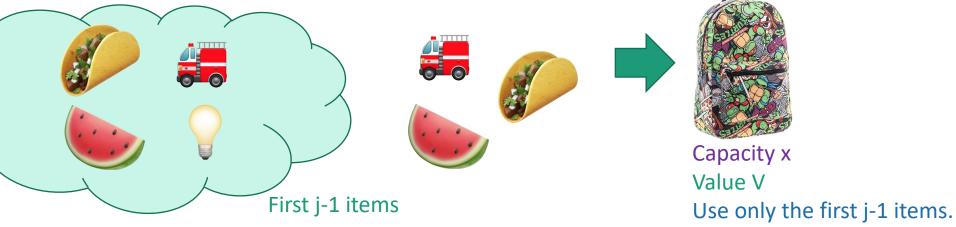
Two cases



Case 1: Optimal solution for j items does not use item j.



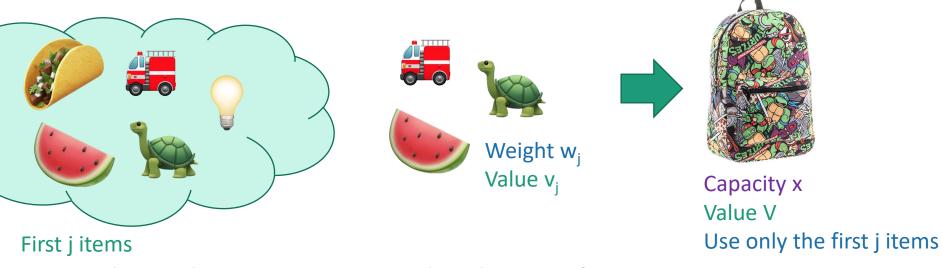
• Then this is an optimal solution for j-1 items:



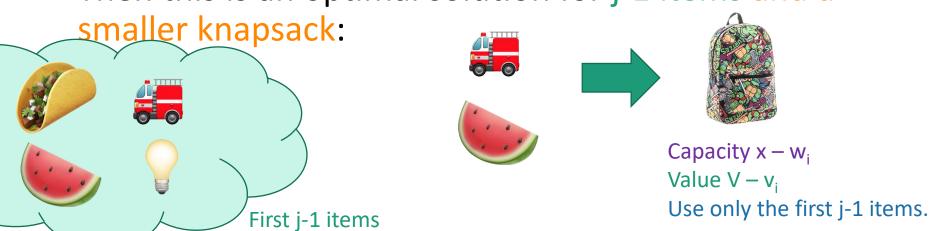
Two cases



• Case 2: Optimal solution for j items uses item j.



Then this is an optimal solution for j-1 items and a



- Step 1: Identify optimal substructure.
- Step 2: Find a recursive formulation for the value of the optimal solution.
- Step 3: Use dynamic programming to find the value of the optimal solution.
- Step 4: If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual solution.
- Step 5: If needed, code this up like a reasonable person.

Recursive relationship

- Let K[x,j] be the optimal value for:
 - capacity x,
 - with j items.

$$K[x,j] = max\{ K[x, j-1], K[x - w_{j,} j-1] + v_{j} \}$$
Case 1

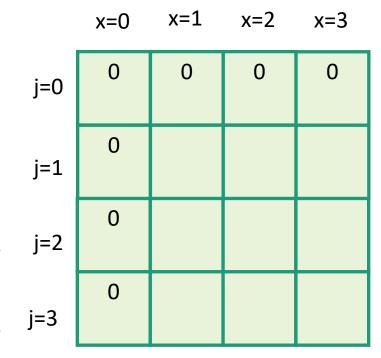
Case 2

• (And K[x,0] = 0 and K[0,j] = 0).

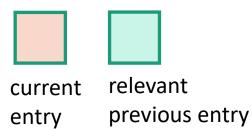
- Step 1: Identify optimal substructure.
- Step 2: Find a recursive formulation for the value of the optimal solution.
- Step 3: Use dynamic programming to find the value of the optimal solution.
- Step 4: If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual solution.
- Step 5: If needed, code this up like a reasonable person.

Bottom-up DP algorithm

```
Zero-One-Knapsack(W, n, w, v):
   • K[x,0] = 0 for all x = 0,...,W
   • K[0,i] = 0 for all i = 0,...,n
   • for x = 1,...,W:
       • for j = 1,...,n:
                               Case 1
           • K[x,i] = K[x, i-1]
           • if w_i \leq x:
                                                Case 2
               • K[x,j] = max\{ K[x,j], K[x-w_i, j-1] + v_i \}
   return K[W,n]
```



- Zero-One-Knapsack(W, n, w, v):
 - K[x,0] = 0 for all x = 0,...,W
 - K[0,i] = 0 for all i = 0,...,n
 - for x = 1,...,W:
 - **for** j = 1,...,n:
 - K[x,j] = K[x, j-1]
 - if $w_i \le x$:
 - $K[x,j] = max\{K[x,j],$ $K[x - w_i, j-1] + v_i$
 - return K[W,n]











4

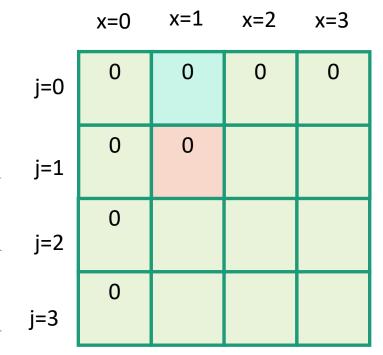




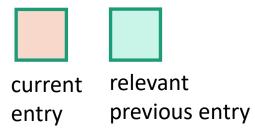
6



Capacity: 3



- Zero-One-Knapsack(W, n, w, v):
 - K[x,0] = 0 for all x = 0,...,W
 - K[0,i] = 0 for all i = 0,...,n
 - **for** x = 1,...,W:
 - **for** j = 1,...,n:
 - K[x,j] = K[x, j-1]
 - if $w_i \le x$:
 - K[x,j] = max{ K[x,j],
 K[x w_i, j-1] + v_i }
 - return K[W,n]



















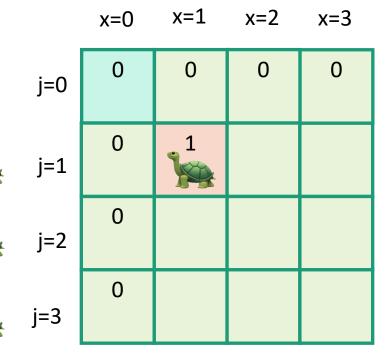
e: 1

4

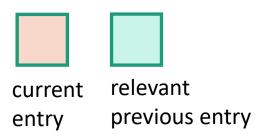
6

3

Capacity: 3



- Zero-One-Knapsack(W, n, w, v):
 - K[x,0] = 0 for all x = 0,...,W
 - K[0,i] = 0 for all i = 0,...,n
 - for x = 1,...,W:
 - **for** j = 1,...,n:
 - K[x,j] = K[x, j-1]
 - if $w_i \le x$:
 - $K[x,j] = max\{K[x,j],$ $K[x - w_i, j-1] + v_i$
 - return K[W,n]









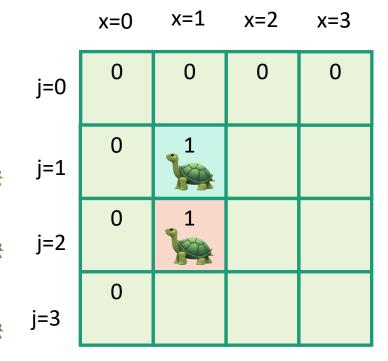


4

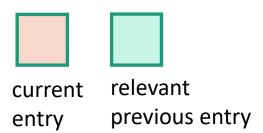




6



- Zero-One-Knapsack(W, n, w, v):
 - K[x,0] = 0 for all x = 0,...,W
 - K[0,i] = 0 for all i = 0,...,n
 - for x = 1,...,W:
 - **for** j = 1,...,n:
 - K[x,j] = K[x, j-1]
 - if $w_i \le x$:
 - $K[x,j] = max\{K[x,j],$ $K[x - w_i, j-1] + v_i$
 - return K[W,n]













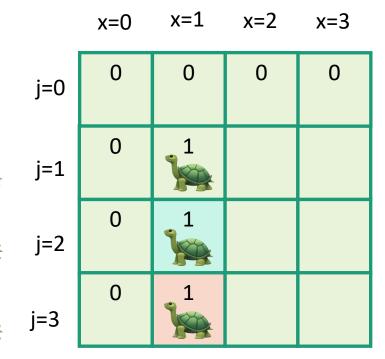
4



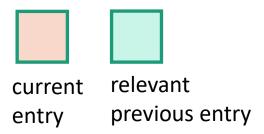
6







- Zero-One-Knapsack(W, n, w, v):
 - K[x,0] = 0 for all x = 0,...,W
 - K[0,i] = 0 for all i = 0,...,n
 - for x = 1,...,W:
 - **for** j = 1,...,n:
 - K[x,j] = K[x, j-1]
 - if $w_i \le x$:
 - $K[x,j] = max\{ K[x,j],$ $K[x - w_i, j-1] + v_i$
 - return K[W,n]









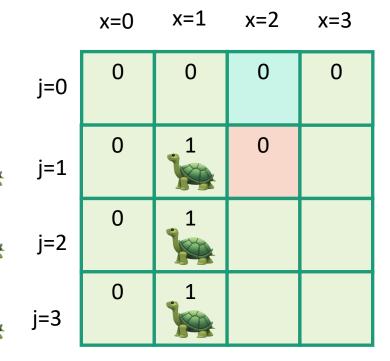




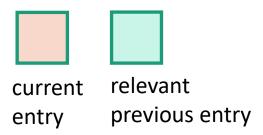
6



4



- Zero-One-Knapsack(W, n, w, v):
 - K[x,0] = 0 for all x = 0,...,W
 - K[0,i] = 0 for all i = 0,...,n
 - for x = 1,...,W:
 - **for** j = 1,...,n:
 - K[x,j] = K[x, j-1]
 - if $w_i \le x$:
 - $K[x,j] = max\{K[x,j],$ $K[x - w_i, j-1] + v_i$
 - return K[W,n]













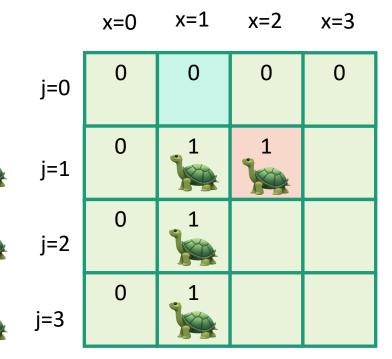
4



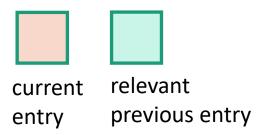
6







- Zero-One-Knapsack(W, n, w, v):
 - K[x,0] = 0 for all x = 0,...,W
 - K[0,i] = 0 for all i = 0,...,n
 - for x = 1,...,W:
 - **for** j = 1,...,n:
 - K[x,j] = K[x, j-1]
 - if $w_i \le x$:
 - $K[x,j] = max\{ K[x,j],$ $K[x - w_i, j-1] + v_i$
 - return K[W,n]



Item:

Weight: Value:





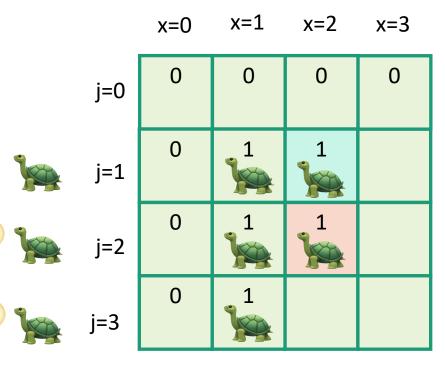
4



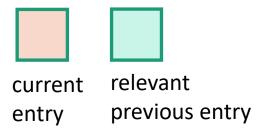


6





- Zero-One-Knapsack(W, n, w, v):
 - K[x,0] = 0 for all x = 0,...,W
 - K[0,i] = 0 for all i = 0,...,n
 - for x = 1,...,W:
 - **for** j = 1,...,n:
 - K[x,j] = K[x, j-1]
 - if $w_i \le x$:
 - $K[x,j] = max\{K[x,j],$ $K[x - w_i, j-1] + v_i$
 - return K[W,n]













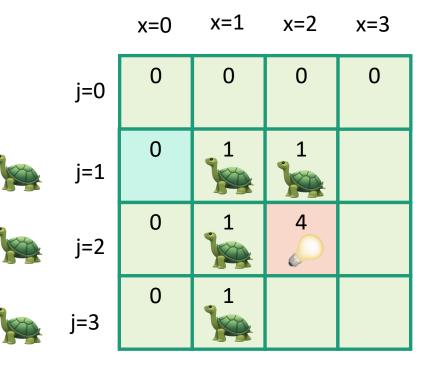




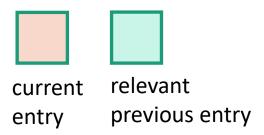
4

6

Capacity: 3



- Zero-One-Knapsack(W, n, w, v):
 - K[x,0] = 0 for all x = 0,...,W
 - K[0,i] = 0 for all i = 0,...,n
 - for x = 1,...,W:
 - **for** j = 1,...,n:
 - K[x,j] = K[x, j-1]
 - if $w_i \le x$:
 - $K[x,j] = max\{K[x,j],$ $K[x - w_i, j-1] + v_i$
 - return K[W,n]













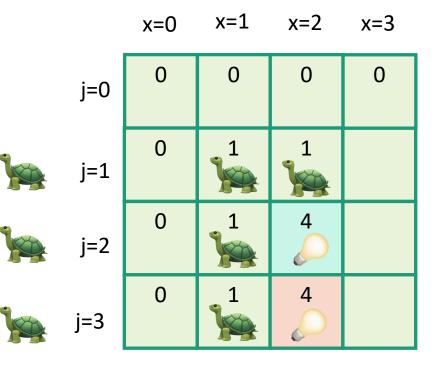




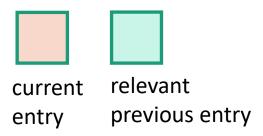
6



4



- Zero-One-Knapsack(W, n, w, v):
 - K[x,0] = 0 for all x = 0,...,W
 - K[0,i] = 0 for all i = 0,...,n
 - for x = 1,...,W:
 - **for** j = 1,...,n:
 - K[x,j] = K[x, j-1]
 - if $w_i \le x$:
 - $K[x,j] = max\{K[x,j],$ $K[x - w_i, j-1] + v_i$
 - return K[W,n]













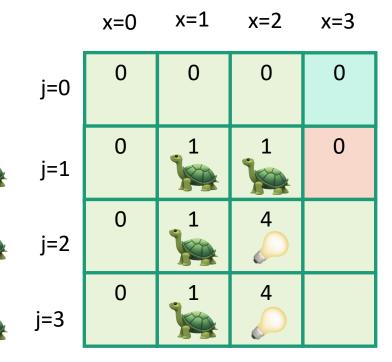
4



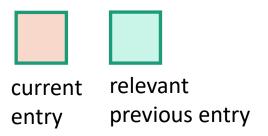


6





- Zero-One-Knapsack(W, n, w, v):
 - K[x,0] = 0 for all x = 0,...,W
 - K[0,i] = 0 for all i = 0,...,n
 - for x = 1,...,W:
 - **for** j = 1,...,n:
 - K[x,j] = K[x, j-1]
 - if $w_i \le x$:
 - K[x,j] = max{ K[x,j],
 K[x w_i, j-1] + v_i }
 - return K[W,n]













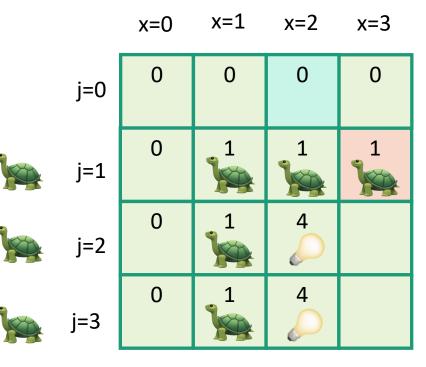
4



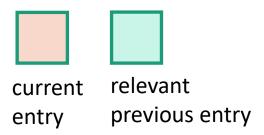


6





- Zero-One-Knapsack(W, n, w, v):
 - K[x,0] = 0 for all x = 0,...,W
 - K[0,i] = 0 for all i = 0,...,n
 - for x = 1,...,W:
 - **for** j = 1,...,n:
 - K[x,j] = K[x, j-1]
 - if $w_i \le x$:
 - K[x,j] = max{ K[x,j],
 K[x w_i, j-1] + v_i }
 - return K[W,n]

















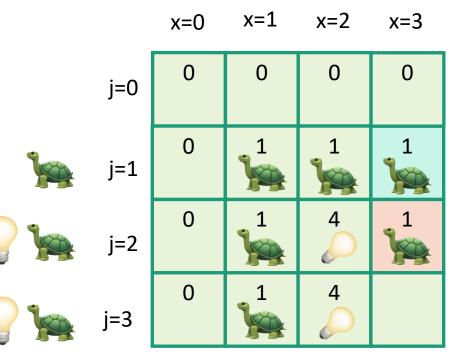
e: 1

4

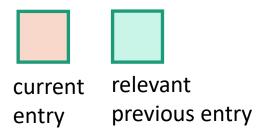
6

3

Capacity: 3



- Zero-One-Knapsack(W, n, w, v):
 - K[x,0] = 0 for all x = 0,...,W
 - K[0,i] = 0 for all i = 0,...,n
 - **for** x = 1,...,W:
 - **for** j = 1,...,n:
 - K[x,j] = K[x, j-1]
 - if $w_i \le x$:
 - $K[x,j] = max\{K[x,j],$ $K[x - w_i, j-1] + v_i$
 - return K[W,n]













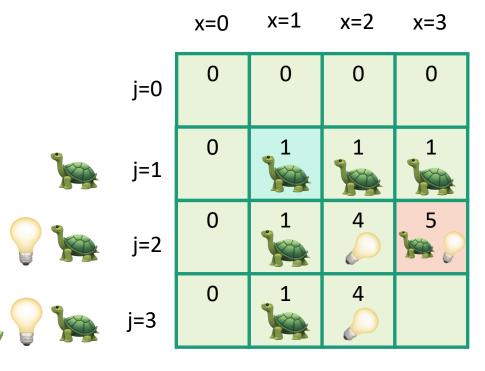




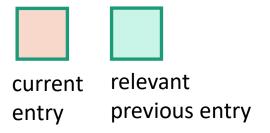
4

6

3



- Zero-One-Knapsack(W, n, w, v):
 - K[x,0] = 0 for all x = 0,...,W
 - K[0,i] = 0 for all i = 0,...,n
 - **for** x = 1,...,W:
 - **for** j = 1,...,n:
 - K[x,j] = K[x, j-1]
 - if $w_i \le x$:
 - $K[x,j] = max\{K[x,j],$ $K[x - w_i, j-1] + v_i$
 - return K[W,n]











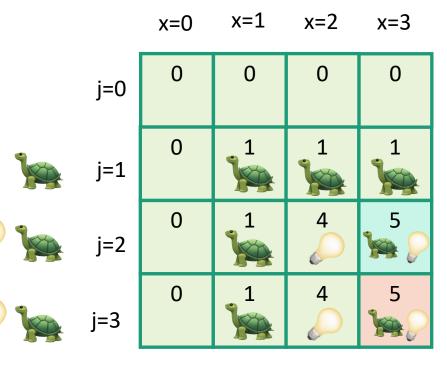
4



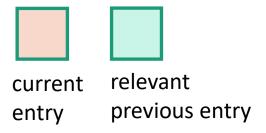


6





- Zero-One-Knapsack(W, n, w, v):
 - K[x,0] = 0 for all x = 0,...,W
 - K[0,i] = 0 for all i = 0,...,n
 - for x = 1,...,W:
 - **for** j = 1,...,n:
 - K[x,j] = K[x, j-1]
 - if $w_i \le x$:
 - K[x,j] = max{ K[x,j],
 K[x w_i, j-1] + v_i }
 - return K[W,n]













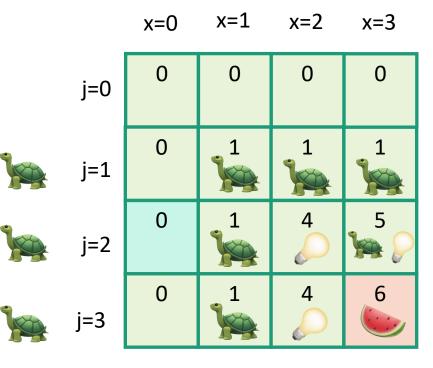
4



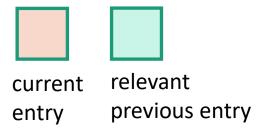


6





- Zero-One-Knapsack(W, n, w, v):
 - K[x,0] = 0 for all x = 0,...,W
 - K[0,i] = 0 for all i = 0,...,n
 - **for** x = 1,...,W:
 - **for** j = 1,...,n:
 - K[x,j] = K[x, j-1]
 - if $w_i \le x$:
 - $K[x,j] = max\{K[x,j],$ $K[x - w_i, j-1] + v_i$
 - return K[W,n]















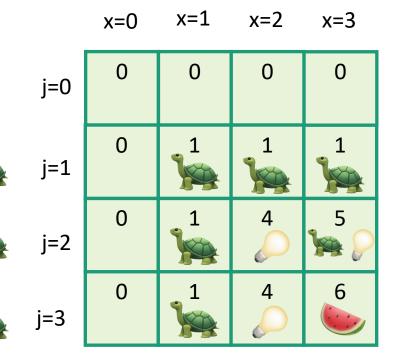
4





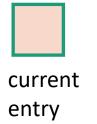






- Zero-One-Knapsack(W, n, w, v):
 - K[x,0] = 0 for all x = 0,...,W
 - K[0,i] = 0 for all i = 0,...,n
 - for x = 1,...,W:
 - **for** j = 1,...,n:
 - K[x,j] = K[x, j-1]
 - if $w_i \le x$:
 - $K[x,j] = max\{K[x,j],$ $K[x-w_i, j-1] + v_i\}$
 - return K[W,n]

So the optimal solution is to put one watermelon in your knapsack!





relevant previous entry



















6



Capacity: 3

Recipe for applying Dynamic Programming

- Step 1: Identify optimal substructure.
- Step 2: Find a recursive formulation for the value of the optimal solution.
- Step 3: Use dynamic programming to find the value of the optimal solution.
- Step 4: If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual solution.
- Step 5: If needed, code this up like a reasonable person.

What have we learned?

- We can solve 0/1 knapsack in time O(nW).
 - If there are n items and our knapsack has capacity W.

- We again went through the steps to create DP solution:
 - We kept a two-dimensional table, creating smaller problems by restricting the set of allowable items.

Question

 How did we know which substructure to use in which variant of knapsack?

Answer in retrospect:





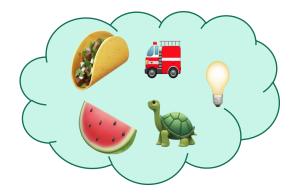


This one made sense for unbounded knapsack because it doesn't have any memory of what items have been used.

VS.





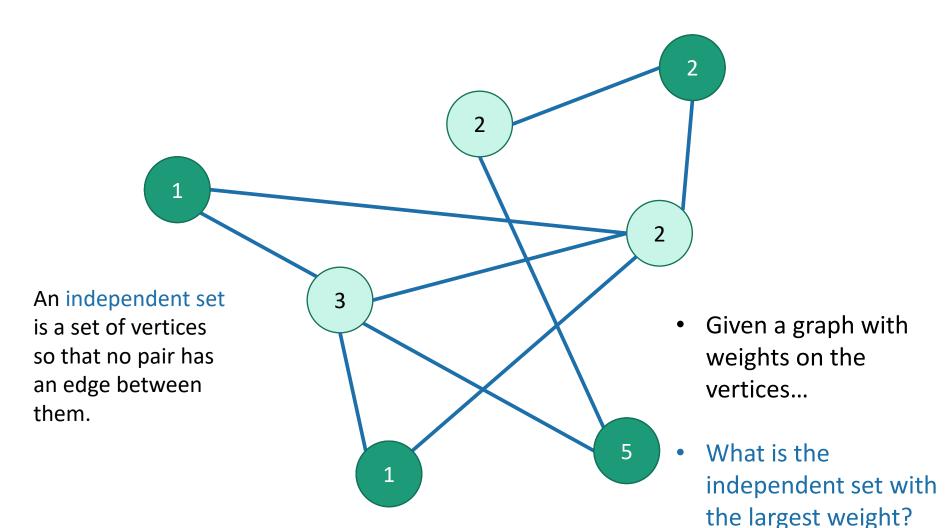


In 0/1 knapsack, we can only use each item once, so it makes sense to leave out one item at a time.

Operational Answer: try some stuff, see what works!

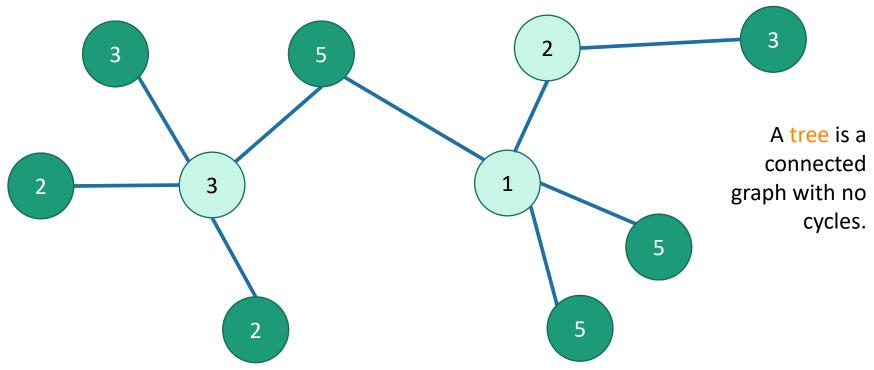
Example 3: Independent Set

if we still have time



Actually this problem is NP-complete. So we are unlikely to find an efficient algorithm

But if we also assume that the graph is a tree...



Problem:

find a maximal independent set in a tree (with vertex weights).

Recipe for applying Dynamic Programming

• Step 1: Identify optimal substructure.



- Step 2: Find a recursive formulation for the value of the optimal solution
- Step 3: Use dynamic programming to find the value of the optimal solution
- Step 4: If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual solution.
- Step 5: If needed, code this up like a reasonable person.

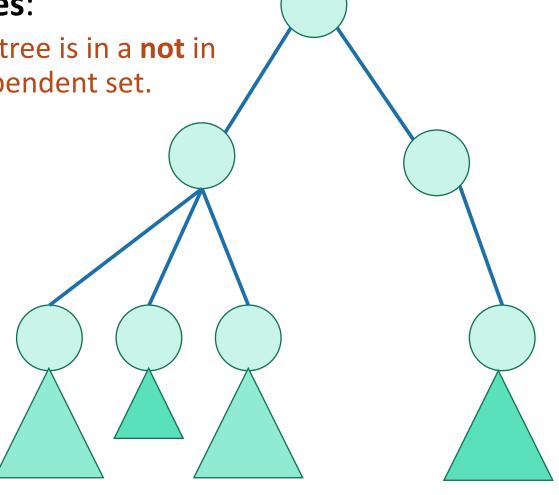
Optimal substructure

• Subtrees are a natural candidate.



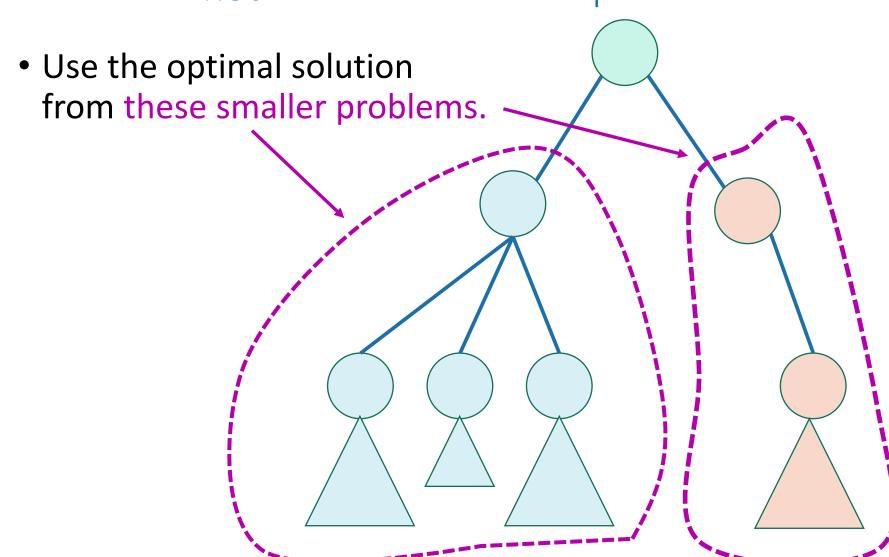
1. The root of this tree is in a **not** in a maximal independent set.

2. Or it is.

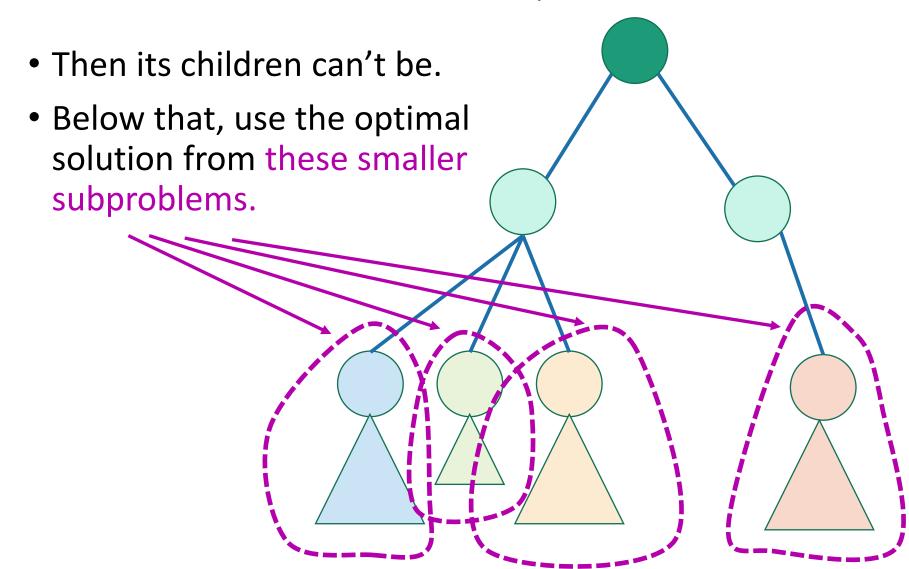


Case 1:

the root is **not** in an maximal independent set



Case 2: the root is in an maximal independent set



Recipe for applying Dynamic Programming

- Step 1: Identify optimal substructure.
- Step 2: Find a recursive formulation for the value of the optimal solution.
- Step 3: Use dynamic programming to find the value of the optimal solution
- Step 4: If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual solution.
- Step 5: If needed, code this up like a reasonable person.

Recursive formulation: try 1

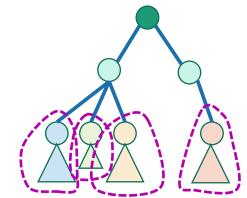
• Let A[u] be the weight of a maximal independent set in the tree rooted at u.

•
$$A[u] = \sum_{v \in u.\text{children}} \sum_{v \in u.\text{children}} A[v]$$

max $\begin{cases} \sum_{v \in u.\text{children}} A[v] \end{cases}$

weight(u) + $\sum_{v \in u.\text{grandchildren}} A[v]$

When we implement this, how do we keep track of this term?



Recursive formulation: try 2

Keep two arrays!

- Let A[u] be the weight of a maximal independent set in the tree rooted at u.
- Let $B[u] = \sum_{v \in u. \text{children}} A[v]$

•
$$A[u] = \max \begin{cases} \sum_{v \in u.\text{children}} A[v] \\ \text{weight}(u) + \sum_{v \in u.\text{children}} B[v] \end{cases}$$

Recipe for applying Dynamic Programming

- Step 1: Identify optimal substructure.
- Step 2: Find a recursive formulation for the value of the optimal solution.
- Step 3: Use dynamic programming to find the value of the optimal solution.
- Step 4: If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual solution.
- Step 5: If needed, code this up like a reasonable person.

A top-down DP algorithm

- MIS_subtree(u):
 - if u is a leaf:
 - A[u] = weight(u)
 - B[u] = 0
 - else:
 - **for** v in u.children:
 - MIS_subtree(v)
 - $A[u] = \max\{\sum_{v \in u, \text{children}} A[v], \text{ weight}(u) + \sum_{v \in u, \text{children}} B[v]\}$
 - $B[u] = \sum_{v \in u.\text{children}} A[v]$
- MIS(T):
 - MIS_subtree(T.root)
 - return A[T.root]

Initialize global arrays A, B the recursive calls.

Running time?

- We visit each vertex once, and at every vertex we do O(1) work:
 - Make a recursive call
 - look stuff up in tables
- Running time is O(|V|)

Why is this different from divide-and-conquer?

That's always worked for us with tree problems before...

- MIS_subtree(u):
 - if u is a leaf:
 - return weight(u)
 - else:
 - **for** v in u.children:
 - MIS_subtree(v)
 - return max{ $\sum_{v \in u. \text{children}} \text{MIS_subtree}(v)$,

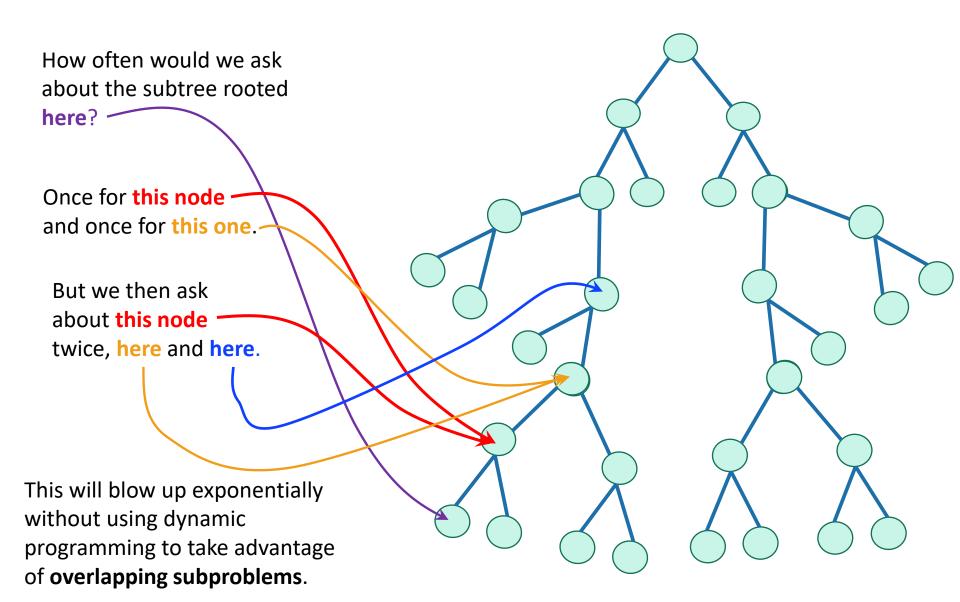
```
weight(u) + \sum_{v \in u.\text{grandchildren}} \text{MIS\_subtree}(v) }
```

- MIS(T):
 - return MIS_subtree(T.root)

This is exactly the same pseudocode, except we've ditched the table and are just calling MIS_subtree(v) instead of looking up A[v] or B[v].

Why is this different from divide-and-conquer?

That's always worked for us with tree problems before...



Recipe for applying Dynamic Programming

- Step 1: Identify optimal substructure.
- Step 2: Find a recursive formulation for the value of the optimal solution.
- Step 3: Use dynamic programming to find the value of the optimal solution.
- Step 4: If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual solution.
- Step 5: If needed, code this up like a reasonable person.

What have we learned?

 We can find maximal independent sets in trees in time O(|V|) using dynamic programming!

 For this example, it was natural to implement our DP algorithm in a top-down way.

Recap

- Today we saw examples of how to come up with dynamic programming algorithms.
 - Longest Common Subsequence
 - Knapsack two ways
 - (If time) maximal independent set in trees.
- There is a **recipe** for dynamic programming algorithms.

Recipe for applying Dynamic Programming

- Step 1: Identify optimal substructure.
- Step 2: Find a recursive formulation for the value of the optimal solution.
- Step 3: Use dynamic programming to find the value of the optimal solution.
- Step 4: If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual solution.
- Step 5: If needed, code this up like a reasonable person.

Recap



- Today we saw examples of how to come up with dynamic programming algorithms.
 - Longest Common Subsequence
 - Knapsack two ways
 - (If time) maximal independent set in trees.
- There is a **recipe** for dynamic programming algorithms.
- Sometimes coming up with the right substructure takes some creativity

JON KLEINBERG • ÉVA TARDOS

SECTION 6.4

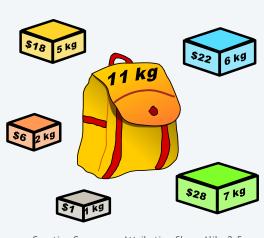
6. DYNAMIC PROGRAMMING

- weighted interval scheduling
- segmented least squares
- knapsack problem
- ► RNA secondary structure

Goal. Pack knapsack so as to maximize total value of items taken.

- There are n items: item i provides value $v_i > 0$ and weighs $w_i > 0$.
- Value of a subset of items = sum of values of individual items.
- Knapsack has weight limit of W.
- Ex. The subset $\{1, 2, 5\}$ has value \$35 (and weight 10).
- Ex. The subset { 3, 4 } has value \$40 (and weight 11).

Assumption. All values and weights are integral.



Creative Commons Attribution-Share Alike 2.5 by Dake

i	v_i	w_i
1	\$1	1 kg
2	\$6	2 kg
3	\$18	5 kg
4	\$22	6 kg
5	\$28	7 kg

weights and values can be arbitrary positive integers

knapsackinstance (weight limit W= 11)



Which algorithm solves knapsack problem?

- A. Greedy-by-value: repeatedly add item with maximum v_i .
- B. Greedy-by-weight: repeatedly add item with minimum w_i .
- C. Greedy-by-ratio: repeatedly add item with maximum ratio v_i/w_i .
- D. None of the above.





i	v_i	W_i
1	\$1	1 kg
2	\$6	2 kg
3	\$18	5 kg
4	\$22	6 kg
5	\$28	7 kg

knapsadkinstance (weight limit W= 11)

Dynamic programming: quiz 3



Which subproblems?

- A. OPT(w) = optimal value of knapsack problem with weight limit w.
- **B**. OPT(i) = optimal value of knapsack problem with items 1, ..., i.
- C. OPT(i, w) = optimal value of knapsack problem with items 1, ..., i subject to weight limit w.
- D. Any of the above.

Dynamic programming: two variables

Def. OPT(i, w) = optimal value of knapsack problem with items 1, ..., i, subject to weight limit w.

Goal. OPT(n, W).

possibly because $w_i > w_i$

Case 1. OPT(i, w) does not select item i.

• OPT(i, w) selects best of $\{1, 2, ..., i-1\}$ subject to weight limit w.

Case 2. OPT(i, w) selects item i.

optimal substructure property (proof via exchange argument)

- Collect value v_i .
- New weight limit = $w w_i$.
- OPT(i, w) selects best of $\{1, 2, ..., i-1\}$ subject to new weight limit.

Bellman equation.

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

Knapsack problem: bottom-up dynamic programming

KNAPSACK
$$(n, W, w_1, ..., w_n, v_1, ..., v_n)$$

FOR $w = 0$ TO W
 $M[0, w] \leftarrow 0$.

FOR $i = 1$ TO n

FOR $w = 0$ TO W

IF $(w_i > w)$ $M[i, w] \leftarrow M[i-1, w]$.

ELSE

 $M[i, w] \leftarrow \max \{ M[i-1, w], v_i + M[i-1, w-w_i] \}$.

RETURN $M[n, W]$.

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

Knapsack problem: bottom-up dynamic programming demo

i	v_i	w_i			
1	\$1	1 kg		$\begin{cases} 0 \\ OPT(i-1, w) \\ \max \{OPT(i-1, w), v_i + OPT(i-1, w-w_i)\} \end{cases}$	
2	\$6	2 kg	$OPT(i, w) = \langle \cdot \rangle$	OPT(i-1,w)	
3	\$18	5 kg		$\max \{OPT(i-1, w), v_i + OPT(i-1, w-w_i)\}$	
4	\$22	6 kg			
5	\$28	7 kg			

weight limit w

	0	1	2	3	4	5	6	7	8	9	10	11
{ }	0	0	0	0	0	0	0	0	0	0	0	0
{1}	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0 ←		6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	- 18 ∢	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	- 40
{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	35	40

subset of items 1, ..., i

OPT(i, w) = optimal value of knapsack problem with items 1, ..., i, subject to weight limit w

Truck – 10t capacity

Optimum cargo combination:

- •Item 1: \$5 (3t)
- •Item 2: \$7 (4t)
- •Item 3: \$8 (5t)

Output function f(i,w)



Optimum output of a combination of items 1 to i with a cumulated weight of w or less.

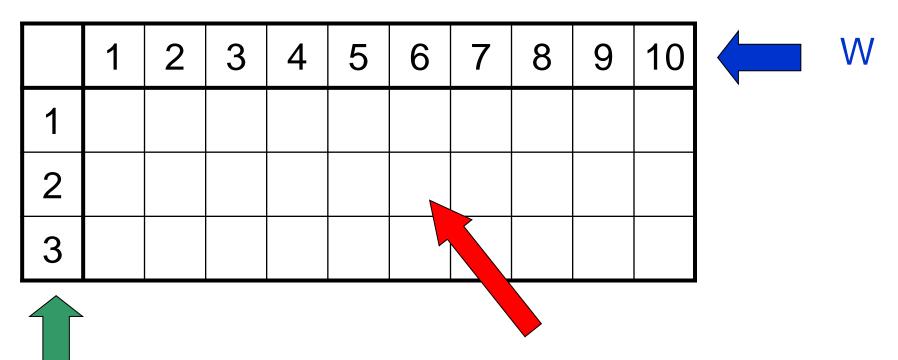
- •ltem 1: x1=\$5; w1=3t
- •Item 2: x2=\$7; w2=4t
- •Item 3: x3=\$8; w3=5t

Output function f(i,w)

ONE Item i + optimum combination of weight w-wi

NO Item i + optimum combination items 1 to i-1

Table



f(**i**,**w**)

•Item 1: x1=\$5; w1=3t

•Item 2: x2=\$7; w2=4t

•Item 3: x3=\$8; w3=5t

•Item 2: x2=\$7; w2=4t

•Item 3: x3=\$8; w3=5t

Knapsack Problem

	1	2	3	4	5	6	7	8	9	10	
1			U	sing	j on	ly i	tem	1			
2											
3											





•Item 2: x2=\$7; w2=4t

•Item 3: x3=\$8; w3=5t

Knapsack Problem

	1	2	3	4	5	6	7	8	9	10	
1											
2			Usiı	ng d	nly	ite	m 1	& 2			
3											





•Item 2: x2=\$7; w2=4t

•Item 3: x3=\$8; w3=5t

Knapsack Problem

	1	2	3	4	5	6	7	8	9	10	W
1											
2											
3			Usi	ng	iten	ns 1	, 2	& 3			

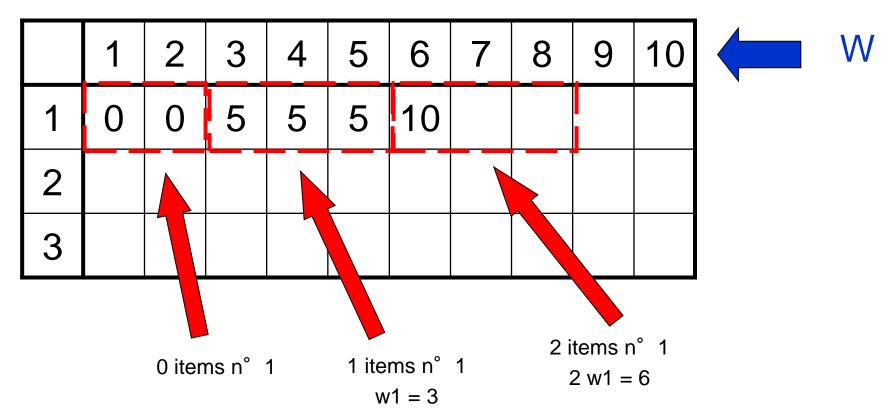


•Item 2: x2=\$7; w2=4t

•Item 3: x3=\$8; w3=5t

Knapsack Problem





•Item 2: x2=\$7; w2=4t

•Item 3: x3=\$8; w3=5t

Knapsack Problem

	1	2	3	4	5	6	7	8	9	10
1	0	0	5	5	5	10	10	10	15	15
2	0 •	0	5	7						
3										

$$w - w2 = 5 - 4 = 1$$

$$f(i,w)=Max[xi + f(i,w-wi); f(i-1,w)]$$

•Item 2: x2=\$7; w2=4t

•Item 3: x3=\$8; w3=5t

Knapsack Problem

	1	2	3	4	5	6	7	8	9	10
1	0	0	5	5	5	10	10	10	15	15
2	0	0	5	7	7					
3										

$$f(i,w)=Max[xi + f(i,w-wi); f(i-x,w)]$$

•Item 2: x2=\$7; w2=4t

•Item 3: x3=\$8; w3=5t

Knapsack Problem

	1	2	3	4	5	6	7	8	9	10
1	0	0	5	5	5	10,	10	10	15	15
2	0	0,	5	7	7					
3										

$$w - w2 = 6 - 4 = 2$$

$$+ x2 (= 7)$$

$$f(i,w)=Max[xi + f(i,w-wi); f(i-1,w)]$$

•Item 2: x2=\$7; w2=4t

•Item 3: x3=\$8; w3=5t

Knapsack Problem

	1	2	3	4	5	6	7	8	9	10
1	0	0	5	5	5	10,	10	10	15	15
2	0	0	5	7	7	10				
3										

$$+ x2 (= 7)$$

•Item 2: x2=\$7; w2=4t

•Item 3: x3=\$8; w3=5t

Knapsack Problem

COMPLETED TABLE

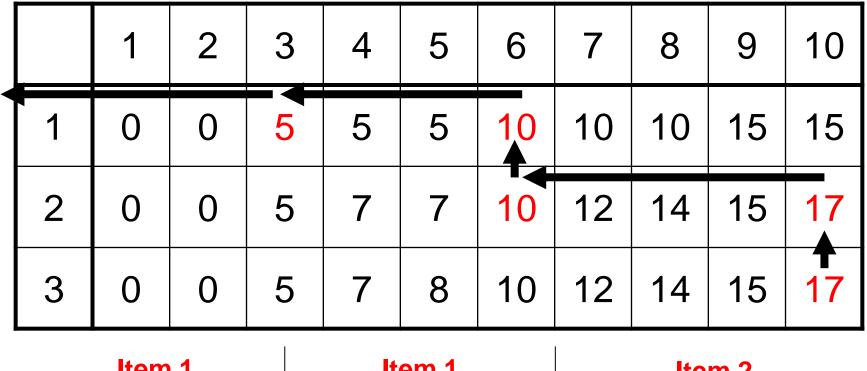
	1	2	3	4	5	6	7	8	9	10
1	0	0	5	5	5	10	10	10	15	15
2	0	0	5	7	7	10	12	14	15	17
3	0	0	5	7	8	10	12	14	15	17

•Item 1: x1=\$5; w1=3t •Item 2: x2=\$7; w2=4t

•Item 3: x3=\$8; w3=5t

Knapsack Problem

Path



Item 1 Item 1 Item 2

Optimal: $2 \times ltem 1 + 1 \times ltem 2$

Knapsack problem: running time

Theorem. The DP algorithm solves the knapsack problem with n items and maximum weight W in $\Theta(n|W)$ time and $\Theta(n|W)$ space.

Pf.

- Takes O(1) time per table entry.

 weights are integers between 1 and W
- There are $\Theta(n|W)$ table entries.
- After computing optimal values, can trace back to find solution: OPT(i, w) takes item i iff M[i, w] > M[i-1, w].

Remarks.

- Algorithm depends critically on assumption that weights are integral.
- Assumption that values are integral was not used.

Knapsack Problem: Bottom-Up

Knapsack. Fill up an n-by-W array.

```
Input: n, w_1, ..., w_N, v_1, ..., v_N
for w = 0 to W
  M[0, w] = 0
for i = 1 to n
   for w = 1 to W
      if (w_i > w)
          M[i, w] = M[i-1, w]
      else
          M[i, w] = \max \{M[i-1, w], v_i + M[i-1, w-w_i]\}
return M[n, W]
```

Knapsack Algorithm

W + 1

W = 11

		0	1	2	3	4	5	6	7	8	9	10	11
n + 1	ф	0	0	0	0	0	0	0	0	0	0	0	0
	{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
	{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
	{1,2,3}	0	1	6	7	7	18	19	24	25	25	25	25
	{1,2,3,4}	0	1	6	7	7	18	22	24	28	29	29	40
	{1,2,3,4,5}	0	1	6	7	7	18	22	28	29	34	34	40

OPT: { 4, 3 }

value = 22 + 18 = 40

Weight 6 3 18 5 4 22 6 5 28

Value

Item

Dynamic programming: quiz 4



Does there exist a poly-time algorithm for the knapsack problem?

- A Yes, because the DP algorithm takes $\Theta(n \ W)$ time.
- **B.** No, because $\Theta(n \ W)$ is not a polynomial function of the input size.
- No, because the problem is **NP**-hard.
- Unknown.



COIN CHANGING



Problem. Given n coin denominations $\{c_1, c_2, ..., c_n\}$ and a target value V, find the fewest coins needed to make change for V (or report impossible).

Recall. Greedy cashier's algorithm is optimal for U.S. coin denominations, but not for arbitrary coin denominations.

Ex. $\{1, 10, 21, 34, 70, 100, 350, 1295, 1500\}$. Optimal. $140 \neq 70 + 70$.



















JON KLEINBERG • ÉVA TARDOS

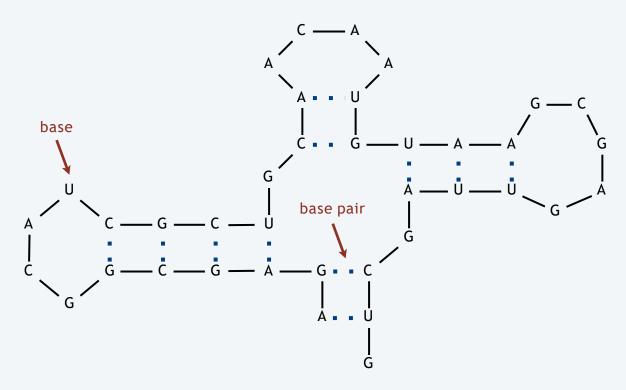
SECTION 6.5

6. DYNAMIC PROGRAMMING I

- weighted interval scheduling
- segmented least squares
- knapsack problem
- ► RNA secondary structure

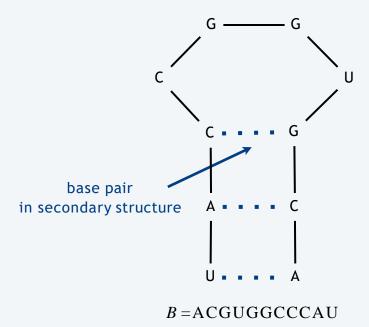
RNA. String $B = b_1b_2...b_n$ over alphabet $\{A, C, G, U\}$.

Secondary structure. RNA is single-stranded so it tends to loop back and form base pairs with itself. This structure is essential for understanding behavior of molecule.

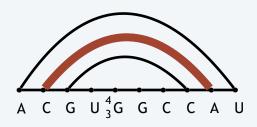


Secondary structure. A set of pairs $S = \{(b_i, b_j)\}$ that satisfy:

■ [Watson-Crick] S is a matching and each pair in S is a Watson-Crick complement: A–U, U–A, C–G, or G–C.



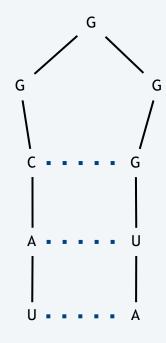
 $S = \{ (b_1, b_{10}), (b_2, b_9), (b_3, b_8) \}$



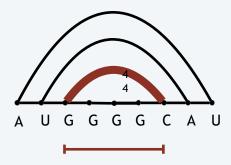
Ss not asecondary structure (C-A is not avalid Watson-Orick pair)

Secondary structure. A set of pairs $S = \{(b_i, b_i)\}$ that satisfy:

- [Watson-Crick] S is a matching and each pair in S is a Watson-Crick complement: A–U, U–A, C–G, or G–C.
- [No sharp turns] The ends of each pair are separated by at least 4 intervening bases. If $(b_i, b_j) \in S$, then i < j 4.



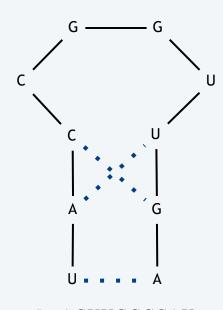
B = AUGGGGCAU $S = \{ (b_1, b_{10}), (b_2, b_9), (b_3, b_8) \}$



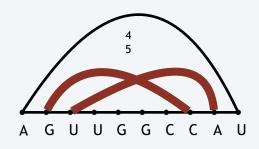
Sis not a secondary structure (≤4 intervening bases between Gand C)

Secondary structure. A set of pairs $S = \{(b_i, b_i)\}$ that satisfy:

- [Watson-Crick] S is a matching and each pair in S is a Watson-Crick complement: A–U, U–A, C–G, or G–C.
- [No sharp turns] The ends of each pair are separated by at least 4 intervening bases. If $(b_i, b_i) \in S$, then i < j 4.
- [Non-crossing] If (b_i, b_j) and (b_k, b_ℓ) are two pairs in S, then we cannot have $i < k < j < \ell$.



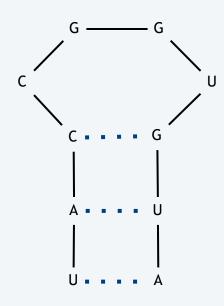
B = ACUUGGCCAU $S = \{ (b_1, b_{10}), (b_2, b_8), (b_3, b_9) \}$



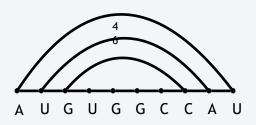
Sis not a secondary structure (G-Cand U-A cross)

Secondary structure. A set of pairs $S = \{(b_i, b_i)\}$ that satisfy:

- [Watson-Crick] S is a matching and each pair in S is a Watson-Crick complement: A–U, U–A, C–G, or G–C.
- [No sharp turns] The ends of each pair are separated by at least 4 intervening bases. If $(b_i, b_i) \in S$, then i < j 4.
- [Non-crossing] If (b_i, b_j) and (b_k, b_ℓ) are two pairs in S, then we cannot have $i < k < j < \ell$.



B = AUGUGGCCAU $S = \{ (b_1, b_{10}), (b_2, b_9), (b_3, b_8) \}$

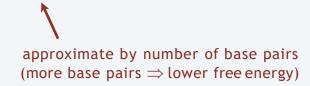


Sis a secondary structure (with 3 base pairs)

Secondary structure. A set of pairs $S = \{(b_i, b_i)\}$ that satisfy:

- [Watson-Crick] S is a matching and each pair in S is a Watson-Crick complement: A–U, U–A, C–G, or G–C.
- [No sharp turns] The ends of each pair are separated by at least 4 intervening bases. If $(b_i, b_i) \in S$, then i < j 4.
- [Non-crossing] If (b_i, b_j) and (b_k, b_ℓ) are two pairs in S, then we cannot have $i < k < j < \ell$.

Free-energy hypothesis. RNA molecule will form the secondary structure with the minimum total free energy.



Goal. Given an RNA molecule $B = b_1b_2...b_n$, find a secondary structure S that maximizes the number of base pairs.

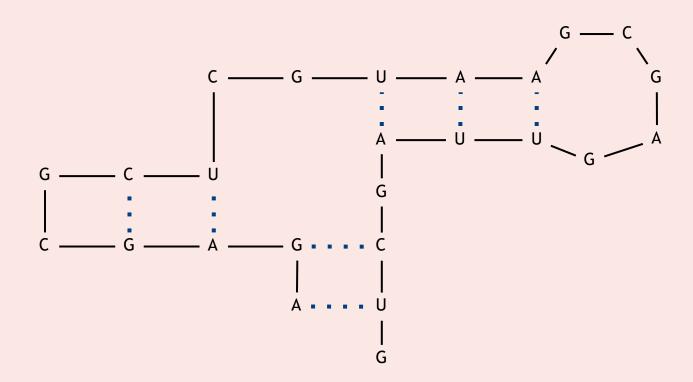


Is the following a secondary structure?

- A. Yes.
- B. No, violates Watson-Crick condition.
- C. No, violates no-sharp-turns condition.



D. No, violates no-crossing condition.



Dynamic programming: quiz 6



Which subproblems?

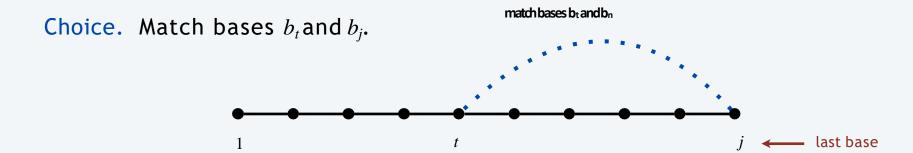
- A. $OPT(j) = \max \text{ number of base pairs in secondary structure}$ of the substring $b_1b_2...b_j$.
- B. $OPT(j) = \max \text{ number of base pairs in secondary structure}$ of the substring $b_j b_{j+1} \dots b_n$.
- C. Either A or B.
- D. Neither A nor B.



RNA secondary structure: subproblems

First attempt. $OPT(j) = \text{maximum number of base pairs in a secondary structure of the substring } b_1b_2 \dots b_j$.

Goal. OPT(n).



Difficulty. Results in two subproblems (but one of wrong form).

- Find secondary structure in $b_1b_2...b_{t-1}$. ← OPT(t-1)
- Find secondary structure in $b_{t+1}b_{t+2}\dots b_{j-1}$. need more subproblems (first base no longer b_1)

Dynamic programming over intervals

Def. OPT(i, j) = maximum number of base pairs in a secondary structure of the substring $b_i b_{i+1} \dots b_j$.

Case 1. If $i \ge j - 4$.

• OPT(i, j) = 0 by no-sharp-turns condition.

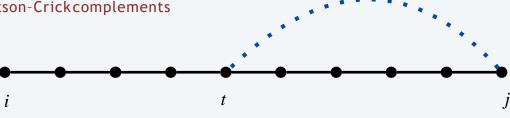
Case 2. Base b_j is not involved in a pair.

• OPT(i, j) = OPT(i, j-1).

Case 3. Base b_j pairs with b_t for some $i \le t < j - 4$.

Non-crossing condition decouples resulting two subproblems.

 $\begin{tabular}{ll} \bullet OPT(i, & j) = 1 + \max_t \{ & OPT(i, & t-1) + OPT(t+1, & j-1) \}. \\ & & \text{match bases } \mathbf{b_i} \text{ and } \mathbf{b_t} \text{ and } b_i \text{ are Watson-Crick complements} \\ \end{tabular}$





In which order to compute OPT(i, j)?

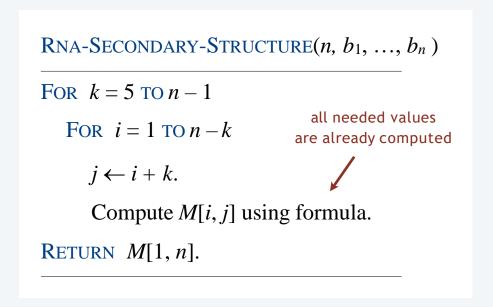
- A. Increasing i, then j.
- B. Increasing j, then i.

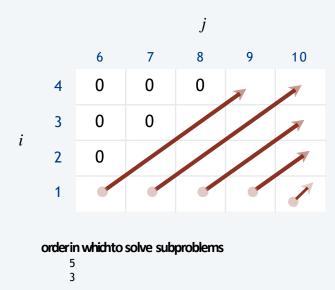


- C. Either A or B.
- D. Neither A nor B.

Bottom-up dynamic programming over intervals

- Q. In which order to solve the subproblems?
- A. Do shortest intervals first—increasing order of 🖵 🗓





Theorem. The DP algorithm solves the RNA secondary structure problem in $O(n^3)$ time and $O(n^2)$ space.

Dynamic programming summary

Outline.

typically, only a polynomial number of subproblems

- Define a collection of subproblems.
- Solution to original problem can be computed from subproblems.
 - Natural ordering of subproblems from "smallest" to "largest" that enables determining a solution to a subproblem from solutions to smaller subproblems.

Techniques.

- Binary choice: weighted interval scheduling.
- Multiway choice: segmented least squares.
- Adding a new variable: knapsack problem.
- Intervals: RNA secondary structure.

Top-down vs. bottom-up dynamic programming. Opinions differ.

NEXT LECTURE

- Min-cut
- · Karger's Algorithm

	_	
Week	Date	Topic
1	21-Feb	Introduction. Some representative problems
2	28-Feb	Stable Matching
3	7-Mar	Basics of algorithm analysis.
4	14-Mar	Graphs (Project 1 announced)
5	21-Mar	Greedy algorithms-I
6	28-Mar	Greedy algorithms-II
7	4-Apr	Divide and conquer (Project 2 announced)
8	11-Apr	Dynamic Programming I
9	18-Apr	Dynamic Programming II
10	25-Apr	Network Flow-I (Project 3 announced)
11	2-May	Midterm
12	9-May	Network Flow II
13	16-May	NP and computational intractability-l
14	23-May	NP and computational intractability-II