Discussion 14

Dynamic Programming

Overview

- Basics of Dynamic Programming

- Dynamic Programming Examples
  - Knapsack problem
    - Two versions
  - Fibonacci Numbers
    - Stairs Climbing

Bottom-Up vs. Top Down

There are two versions of dynamic programming.

- Bottom-up.
- Top-down (or memoization).

Bottom-up:

- Iterative, solves problems in sequence, from smaller to bigger.

Top-down:

- Recursive, start from the larger problem, solve smaller problems as needed.
- For any problem that we solve, **store the solution**, so we never have to compute the same solution twice.
- This approach is also called **memoization**.

Top-Down Dynamic Programming (Memoization)

- Maintain an array/table where solutions to problems can be saved.
- To solve a problem P:
  - See if the solution has already been stored in the array.
  - If yes, return the solution.
  - Else:
    - Issue recursive calls to solve whatever smaller problems we need to solve.
    - Using those solutions obtain the solution to problem P.
    - Store the solution in the solutions array.
    - Return the solution.

Bottom-Up Dynamic Programming

- Requirements for using dynamic programming:
  - The answer to our problem, P, can be easily obtained from answers to smaller problems.
  - We can order problems in a sequence (P_0, P_1, P_2, ..., P_K) of reasonable size, so that:
    - P_k is our original problem P.
    - The initial problems, P_0 and possibly P_1, P_2, ..., P_R up to some R, are easy to solve (they are **base cases**).
    - For i > R, each P_i can be easily solved using solutions to P_0, ..., P_{i-1}.
- If these requirements are met, we solve problem P as follows:
  - Create the sequence of problems P_0, P_1, P_2, ..., P_K, such that P_k = P.
  - For i = 0 to K, solve P_K.
  - Return solution for P_K.

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.

Knapsack Problem

- We have n items with weights and values:

Item:					
Weight:	6	2	4	3	11
Value:	20	8	14	13	35

- And we have a knapsack:

Capacity: 10

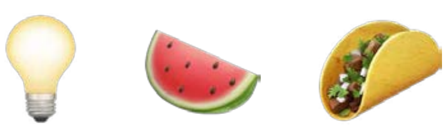
Capacity:10

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?

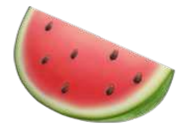
Total weight: 10
Total value: 42

- 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

Item:					
Weight:	w_1	w_2	w_3	$\dots$	w_n
Value:	v_1	v_2	v_3		v_n

Capacity: W

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

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

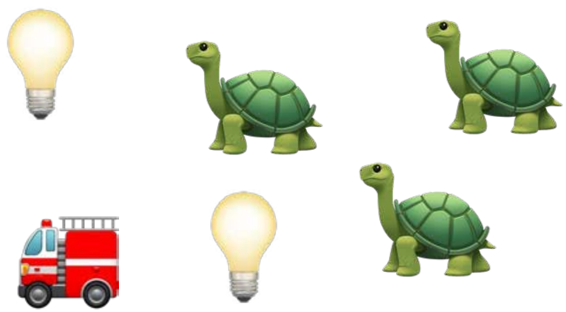
- First solve the problem for small knapsacks
- Then larger knapsacks
- Then larger knapsacks

Optimal substructure

item i

- Suppose this is an optimal solution for capacity x:

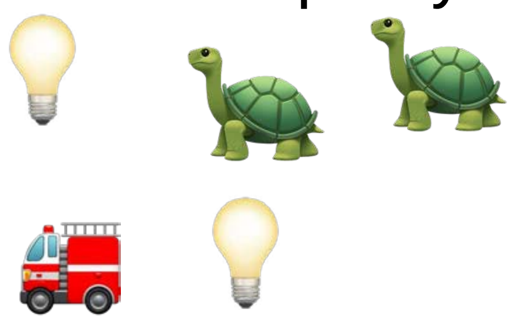
Say that the optimal solution contains at least one copy of item i.



Weight w_i
Value v_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 $V - v_i$

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 relationship

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

K[x] =max$_i$ {  +  }

The maximum is over all i so that $w_i \leq x$.

Optimal way to fill the smaller knapsack

The value of item i.

$$K[x] = \max_i \{ K[x - w_i] + v_i \}$$

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

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.

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]

$K[x] = \max_i \{$  $+$  $\}$

$= \max_i \{ K[x - w_i] + v_i \}$

Running time: O(nW)

Why does this work?

Because our recursive relationship makes sense.

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.

Let’s write a bottom-up DP algorithm

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

$K[x] = \max_i \{$  +  $\}$

$= \max_i \{ K[x - w_i] + v_i \}$

Example

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

	0	1	2	3	4
K	0				
ITEMS					

Item:			
Weight:	1	2	3
Value:	1	4	6

Capacity: 4

Example

	0	1	2	3	4
K	0	1			
ITEMS		🐢			

ITEMS[1] = ITEMS[0] + 🐢

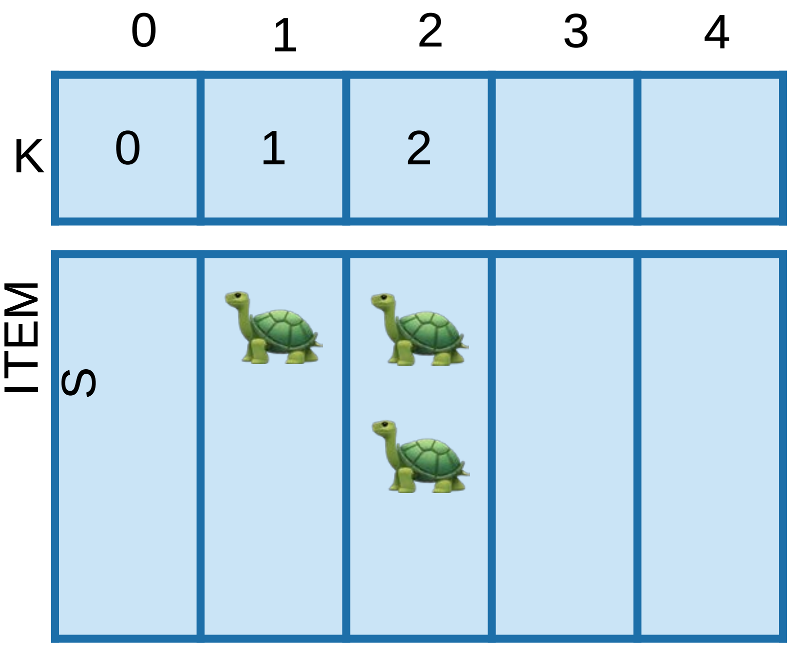
- UnboundedKnapsack(W, n, weights, values):
- K[0] = 0
- ITEMS[0] = ∅
- **for** x = 1, …, W:
  - K[x] = 0
  - **for** i = 1, …, n:
    - **If** $w_i \leq x$:
      - $K[x] = \max\{K[x], K[x - w_i] + v_i\}$
      - If K[x] was updated:
        - ITEMS[x] = ITEMS[x – w_i] ∪ { item i }
- **return** ITEMS[W]

Item:	🐢	💡	🍉
Weight:	1	2	3
Value:	1	4	6

Capacity: 4

Example

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



Item: 

Weight:	1	2	3
Value:	1	4	6

ITEMS[2] = ITEMS[1] + 

Capacity: 4

Example

	0	1	2	3	4
K	0	1	4		
ITEMS		🐢	💡		

ITEMS[2] = ITEMS[0] + 💡

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

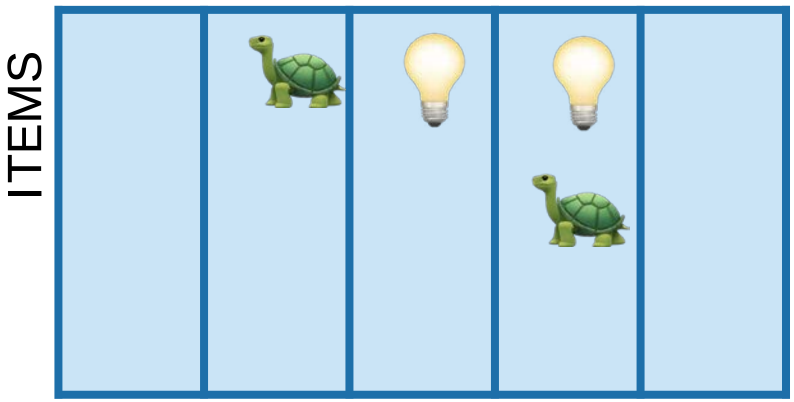
Item:	🐢	💡	🍉
Weight:	1	2	3
Value:	1	4	6

Capacity: 4

Example

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

	0	1	2	3	4
K	0	1	4	5	



Item:			
Weight:	1	2	3
Value:	1	4	6

ITEMS[3] = ITEMS[2] + 

Capacity: 4

Example

	0	1	2	3	4
K	0	1	4	6	
ITEMS		🐢	💡	🍉	

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

Item:	🐢	💡	🍉
Weight:	1	2	3
Value:	1	4	6

ITEMS[3] = ITEMS[0] + 🍉

Capacity: 4

Example

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

	0	1	2	3	4
K	0	1	4	6	7
ITEMS		🐢	💡	🍉	🍉 🐢

Item:	🐢	💡	🍉
Weight:	1	2	3
Value:	1	4	6

ITEMS[4] = ITEMS[3] + 🐢

Capacity: 4

Example

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

	0	1	2	3	4
K	0	1	4	6	8
ITEMS					 

Item:			
Weight:	1	2	3
Value:	1	4	6

ITEMS[4] = ITEMS[2] + 

Capacity: 4

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

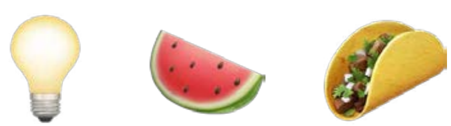
Item: 

Weight:	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

Optimal substructure: try 1

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

- First solve the problem for small knapsacks

- 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

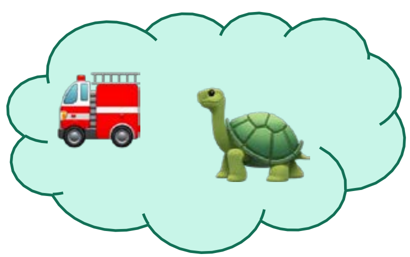
- Sub-problems:
  - 0/1 Knapsack with fewer items.



First solve the problem with few items

We'll still increase the size of the knapsacks.

Then more items

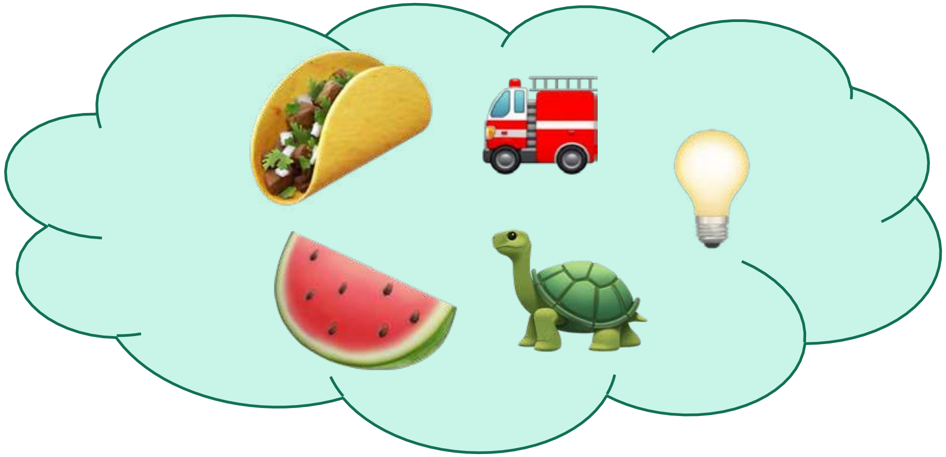


(We'll keep a two-dimensional table).

Then yet more items

Our sub-problems:

- Indexed by x and j



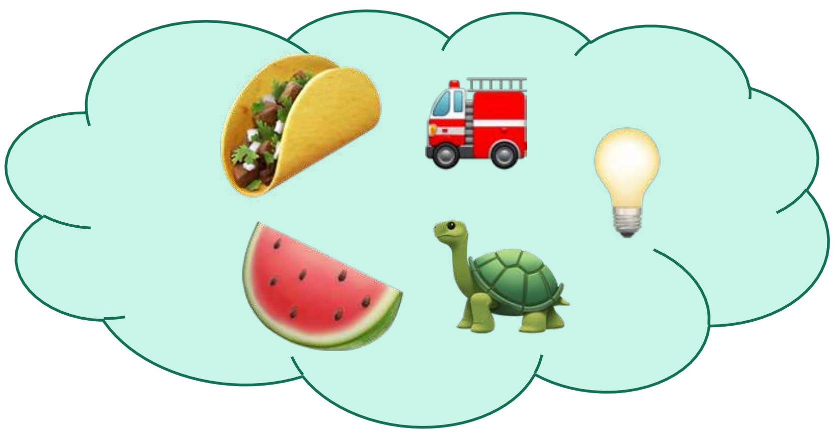
First j items

Capacity x

Two cases

item j

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



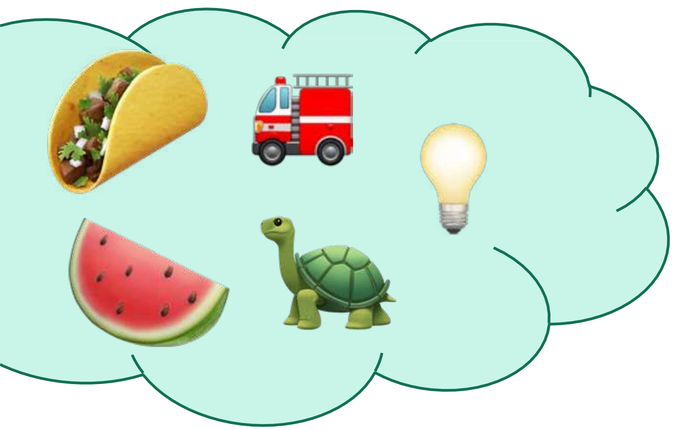
First j items

Capacity x

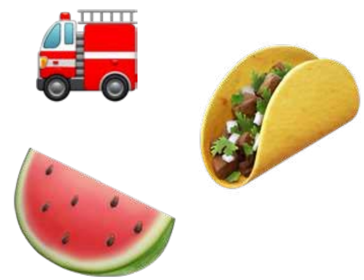
Two cases

item j

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

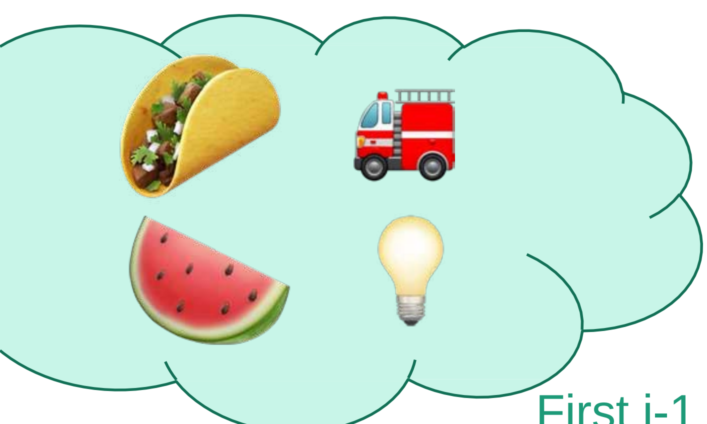


First j items

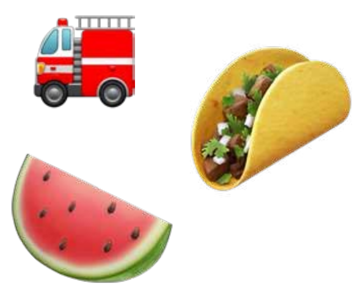


Capacity x
Value V
Use only the first j items

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



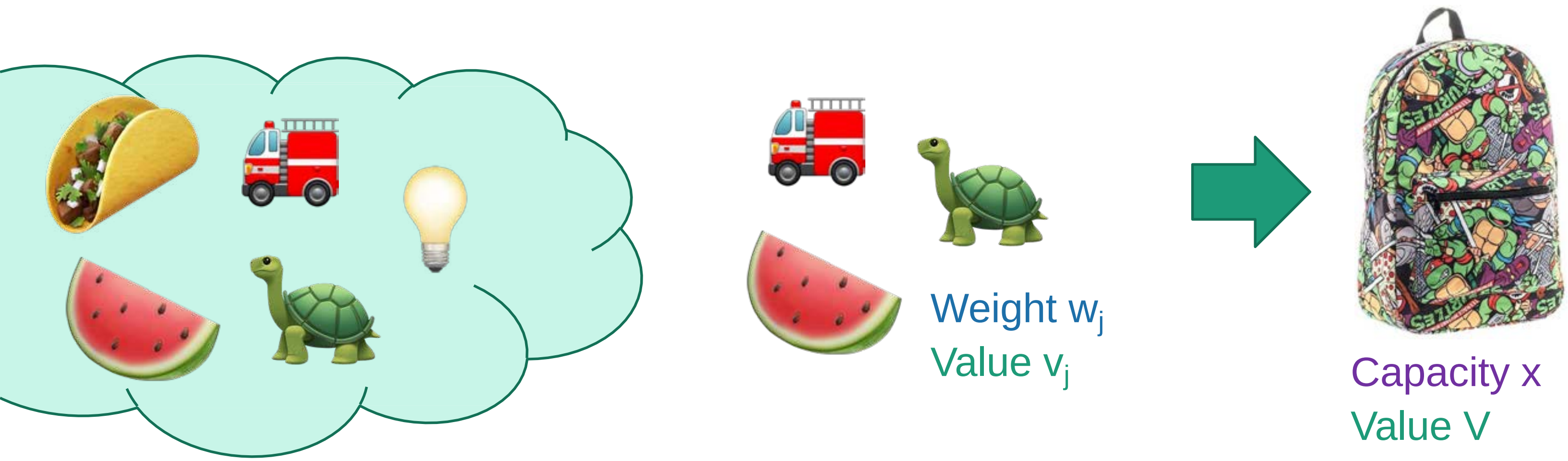
First j-1 items



Capacity x
Value V
Use only the first 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 smaller knapsack:


Recursive relationship

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

$$K[x,j] = \max\{ \underbrace{K[x, j-1]}_{\text{Case 1}} , \underbrace{K[x - w_j, j-1] + v_j}_{\text{Case 2}} \}$$

- (And K[x,0] = 0 and K[0,j] = 0).

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:
      - K[x,j] = K[x, j-1] Case 1
      - **if** $w_j \leq x$:
        - K[x,j] = max{ K[x,j], K[x – w_j, j-1] + v_j } Case 2
  - **return** K[W,n]

Running time O(nW)

Example

- 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** $w_j \leq x$:
        - K[x,j] = max{ K[x,j], K[x – w_j, j-1] + v_j }
  - **return** K[W,n]

	x=0	x=1	x=2	x=3
j=0	0	0	0	0
j=1	0			
j=2	0			
j=3	0			

current entry

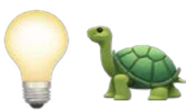
relevant previous entry

Item:			
Weight:	1	2	3
Value:	1	4	6

Capacity: 3

Example

- 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** $w_j \leq x$:
        - K[x,j] = max{ K[x,j], K[x – w_j, j-1] + v_j }
  - **return** K[W,n]

		x=0	x=1	x=2	x=3
	j=0	0	0	0	0
	j=1	0	0		
	j=2	0			
	j=3	0			

current entry

relevant previous entry

Item:				
Weight:	1	2	3	
Value:	1	4	6	Capacity: 3

Example

- 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** $w_j \leq x$:
        - K[x,j] = max{ K[x,j], K[x – w_j, j-1] + v_j }
  - **return** K[W,n]

		x=1	x=2	x=3
j=0	0	0	0	0
j=1	0	1		
j=2	0			
j=3	0			

current entry

relevant previous entry

Item:			
Weight:	1	2	3
Value:	1	4	6

Capacity: 3

Example

- 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** $w_j \leq x$:
        - K[x,j] = max{ K[x,j], K[x – w_j, j-1] + v_j }
  - **return** K[W,n]

		x=1	x=2	x=3
j=0	0	0	0	0
j=1	0	1		
j=2	0	1		
j=3	0			

current entry

relevant previous entry

Item:			
Weight:	1	2	3
Value:	1	4	6

Capacity: 3

Example

- 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** $w_j \leq x$:
        - K[x,j] = max{ K[x,j], K[x – w_j, j-1] + v_j }
  - **return** K[W,n]

		x=1	x=2	x=3
j=0	0	0	0	0
j=1	0	1		
j=2	0	1		
j=3	0	1		

current entry

relevant previous entry

Item:			
Weight:	1	2	3
Value:	1	4	6

Capacity: 3

Example

- 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** $w_j \leq x$:
        - K[x,j] = max{ K[x,j], K[x – w_j, j-1] + v_j }
  - **return** K[W,n]

		x=1	x=2	x=3
j=0	0	0	0	0
j=1	0	1	0	
j=2	0	1		
j=3	0	1		

current entry

relevant previous entry

Item:				
Weight:	1	2	3	
Value:	1	4	6	

Capacity: 3

Example

- 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** $w_j \leq x$:
        - K[x,j] = max{ K[x,j], K[x – w_j, j-1] + v_j }
  - **return** K[W,n]

		x=1	x=2	x=3
j=0	0	0	0	0
j=1	0	1	1	
j=2	0	1		
j=3	0	1		

current entry

relevant previous entry

Item:			
Weight:	1	2	3
Value:	1	4	6

Capacity: 3

Example

- 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** $w_j \leq x$:
        - K[x,j] = max{ K[x,j], K[x – w_j, j-1] + v_j }
  - **return** K[W,n]

	x=0	x=1	x=2	x=3
j=0	0	0	0	0
j=1	0	1	1	
j=2	0	1	1	
j=3	0	1		

current entry

relevant previous entry

Item:			
Weight:	1	2	3
Value:	1	4	6

Capacity: 3

Example

- 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** $w_j \leq x$:
        - K[x,j] = max{ K[x,j], K[$x - w_j$, j-1] + v_j }
  - **return** K[W,n]

	x=0	x=1	x=2	x=3
j=0	0	0	0	0
j=1	0	1	1	
j=2	0	1	4	
j=3	0	1		

current entry

relevant previous entry

Item:			
Weight:	1	2	3
Value:	1	4	6

Capacity: 3

Example

- 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** $w_j \leq x$:
        - K[x,j] = max{ K[x,j], K[x – w_j, j-1] + v_j }
  - **return** K[W,n]

	x=0	x=1	x=2	x=3
j=0	0	0	0	0
j=1	0	1	1	
j=2	0	1	4	
j=3	0	1	4	

current entry

relevant previous entry

Item:			
Weight:	1	2	3
Value:	1	4	6

Capacity: 3

Example

- 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** $w_j \leq x$:
        - K[x,j] = max{ K[x,j], K[x – w_j, j-1] + v_j }
  - **return** K[W,n]

	x=0	x=1	x=2	x=3
j=0	0	0	0	0
j=1	0	1	1	0
j=2	0	1	4	
j=3	0	1	4	

current entry

relevant previous entry

Item:			
Weight:	1	2	3
Value:	1	4	6

Capacity: 3

Example

- 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** $w_j \leq x$:
        - K[x,j] = max{ K[x,j], K[x – w_j, j-1] + v_j }
  - **return** K[W,n]

	x=0	x=1	x=2	x=3
j=0	0	0	0	0
j=1	0	1	1	1
j=2	0	1	4	
j=3	0	1	4	

current entry

relevant previous entry

Item:			
Weight:	1	2	3
Value:	1	4	6

Capacity: 3

Example

- 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** $w_j \leq x$:
        - K[x,j] = max{ K[x,j], K[x – w_j, j-1] + v_j }
  - **return** K[W,n]

		x=0	x=1	x=2	x=3
	j=0	0	0	0	0
🐢	j=1	0	1 🐢	1 🐢	1 🐢
💡 🐢	j=2	0	1 🐢	4 💡	1 🐢
🍉 💡 🐢	j=3	0	1 🐢	4 💡	

current entry

relevant previous entry

Item:	🐢	💡	🍉
Weight:	1	2	3
Value:	1	4	6

Capacity: 3

Example

- 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** $w_j \leq x$:
        - K[x,j] = max{ K[x,j], K[x – w_j, j-1] + v_j }
  - **return** K[W,n]

	x=0	x=1	x=2	x=3
j=0	0	0	0	0
j=1	0	1	1	1
j=2	0	1	4	5
j=3	0	1	4	0

current entry

relevant previous entry

Item:			
Weight:	1	2	3
Value:	1	4	6

Capacity: 3

Example

- 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** $w_j \leq x$:
        - K[x,j] = max{ K[x,j], K[x – w_j, j-1] + v_j }
  - **return** K[W,n]

		x=0	x=1	x=2	x=3
	j=0	0	0	0	0
🐢	j=1	0	1 🐢	1 🐢	1 🐢
💡 🐢	j=2	0	1 🐢	4 💡	5 🐢 💡
🍉 💡 🐢	j=3	0	1 🐢	4 💡	5 🐢 💡

current entry

relevant previous entry

Item:	🐢	💡	🍉
Weight:	1	2	3
Value:	1	4	6

Capacity: 3

Example

- 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** $w_j \leq x$:
        - K[x,j] = max{ K[x,j], K[x – w_j, j-1] + v_j }
  - **return** K[W,n]

	x=0	x=1	x=2	x=3
j=0	0	0	0	0
j=1	0	1	1	1
j=2	0	1	4	5
j=3	0	1	4	6

current entry

relevant previous entry

Item:			
Weight:	1	2	3
Value:	1	4	6

Capacity: 3

Example

- 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** $w_j \leq x$:
        - K[x,j] = max{ K[x,j], K[x – w_j, j-1] + v_j }
  - **return** K[W,n]

	x=0	x=1	x=2	x=3
j=0	0	0	0	0
j=1	0	1	1	1
j=2	0	1	4	5
j=3	0	1	4	6

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

current entry

relevant previous entry

Item:				
Weight:	1	2	3	
Value:	1	4	6	Capacity: 3

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.

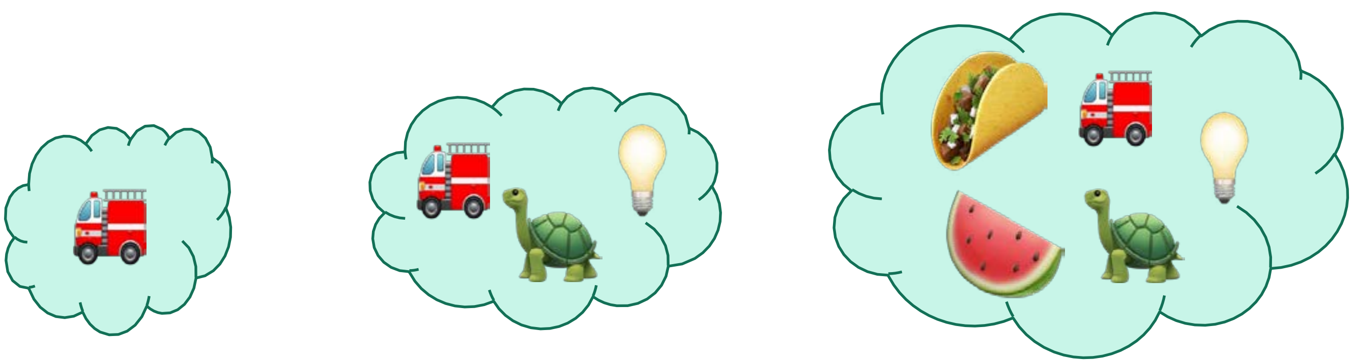
Question

- 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!

Fibonacci Numbers

- Generate Fibonacci numbers
  - 3 solutions: inefficient recursive, memoization (top-down dynamic programming (DP)), bottom-up DP.
  - Not an optimization problem but it has overlapping subproblems => DP eliminates recomputing the same problem over and over again.
    - Fibonacci(0) = 0
    - Fibonacci(1) = 1
    - If N >= 2:
    - Fibonacci(N) = Fibonacci(N-1) + Fibonacci(N-2)
  - How can we write a function that computes Fibonacci numbers?

Fibonacci Numbers

- Generate Fibonacci numbers
  - 3 solutions: inefficient recursive, memoization (top-down dynamic programming (DP)), bottom-up DP.
  - Not an optimization problem but it has overlapping subproblems => DP eliminates recomputing the same problem over and over again.
    - Fibonacci(0) = 0
    - Fibonacci(1) = 1
    - If N >= 2:
    - Fibonacci(N) = Fibonacci(N-1) + Fibonacci(N-2)
  - How can we write a function that computes Fibonacci numbers?

Fibonacci Numbers

Fibonacci(0) = 0 , Fibonacci(1) = 1

If N >= 2: Fibonacci(N) = Fibonacci(N-1) + Fibonacci(N-2)

Alternative: remember values we have already computed.

Draw the new recursion tree and discuss time complexity.

memoized version:

```
int Fib_mem_wrap(int i) {
  int sol[i+1];
  if (i<=1) return i;
  sol[0] = 0;  sol[1] = 1;
  for(int k=2; k<=i; k++)  sol[k]=-1;
  Fib_mem(i,sol);
  return sol[i];
}
int Fib_mem (int i, int[] sol)  {
  if (sol[i]!=-1) return sol[i];
  int res = Fib_mem(i-1, sol) + Fib_mem(i-2, sol);
  sol[i] = res;
  return res;
}
```

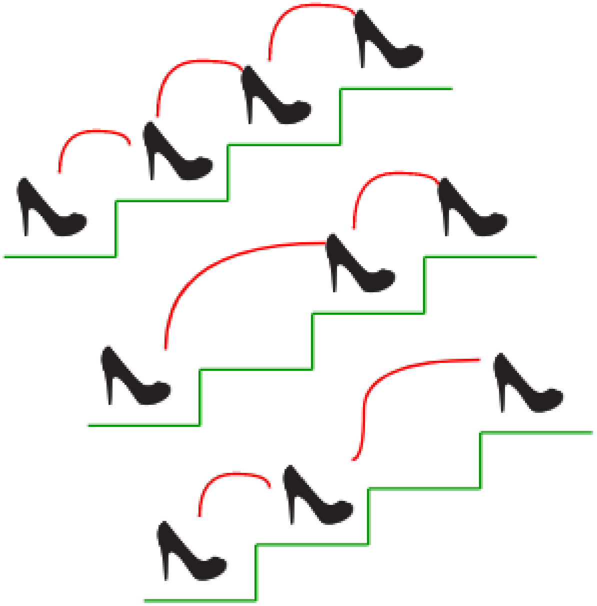
exponential version:

```
int Fib(int i)  {
  if (i < 1) return 0;
  if (i == 1) return 1;
  return Fib(i-1) + Fib(i-2);
}
```

Fibonacci and DP

- Computing the Fibonacci number is a DP problem.
- It is a counting problem (not an optimization one).
- We can make up an ‘applied’ problem for which the DP solution function is the Fibonacci function.

Stairs Climbing Problem



A lady can climb stairs one step at a time or two steps at a time (but he cannot do 3 or more steps at a time). How many different ways can they climb? E.g. to climb 4 stairs you have 5 ways: {1,1,1,1}, {2,1,1}, {1,2,1}, {1,1,2}, {2,2}

ways(n) = ways(n-1) + ways(n-2)

The above expression is actually the expression for Fibonacci numbers, but there is one thing to notice, the value of ways(n) is equal to fibonacci(n+1).

ways(1) = fib(2) = 1
ways(2) = fib(3) = 2
ways(3) = fib(4) = 3

Approaches for solving DP Problems

Greedy

- typically not optimal solution (for DP-type problems)
- Build solution
- Use a criterion for picking
- Commit to a choice and do not look back

DP

- Optimal solution
- *Write math function, **sol***, that captures the dependency of solution to current pb on solutions to smaller problems
- Can be implemented in any of the following: iterative, memoized, recursive

Brute Force

- Optimal solution
- Produce all possible combinations, [check if valid], and keep the best.
- Time: exponential
- Space: depends on implementation
- It may be hard to generate all possible combinations

Iterative (bottom-up) - BEST

- Optimal solution
- *sol* is an array (1D or 2D). Size: n+1
- Fill in *sol* from 0 to n
- Time: polynomial (or pseudo-polynomial for some problems)
- Space: polynomial (or pseudo-polynomial
- To recover the choices that gave the optimal answer, must backtrace => must keep picked array (1D or 2D).

Memoized

- Optimal solution
- Combines recursion and usage of *sol* array.
- *sol* is an array (1D or 2D)
- Fill in *sol* from 0 to n
- Time: same as iterative version (typically)
- Space: same as iterative version (typically) + space for frame stack. (Frame stack depth is typically smaller than the size of the *sol* array)

Recursive

- Optimal solution
- Time: exponential (typically) =>
- DO NOT USE
- Space: depends on implementation (code). E.g. store all combinations, or generate, evaluate on the fly and keep best seen so far.
- Easy to code given math function

Sliding window

- Improves the iterative solution
- *Saves space*
- If used, cannot recover the choices (gives the optimal value, but not the choices)

DP can solve:

- some types of **counting problems** (e.g. stair climbing)
- some type of **optimization problems** (e.g. Knapsack)
- some type of **recursively defined** pbs (e.g. Fibonacci)

QUIZ TIME