

# CS 2009

## Design and Analysis of Algorithms

*Waheed Ahmed*  
*Email: waheedahmed@nu.edu.pk*

# Week 7b: DYNAMIC PROGRAMMING

Another algorithm design paradigm!

# DYNAMIC PROGRAMMING

- ❑ Big Idea, hard, yet simple.
- ❑ Large class of exponential problems have a polynomial solution only via DP. DP == “Controlled Brute-force”
- ❑ Dynamic Programming is a technique for computing recurrence relations efficiently by storing partial results. (DP == “recursion + reuse”)
- ❑ Particularly used for optimization problems.

# Fibonacci Numbers

$$F_1 = F_2 = 1; \quad F_n = F_{n-1} + F_{n-2}$$

## Naive Algorithm

follow recursive definition

fib( $n$ ):

if  $n \leq 2$ : return  $f = 1$

else: return  $f = \text{fib}(n-1) + \text{fib}(n-2)$

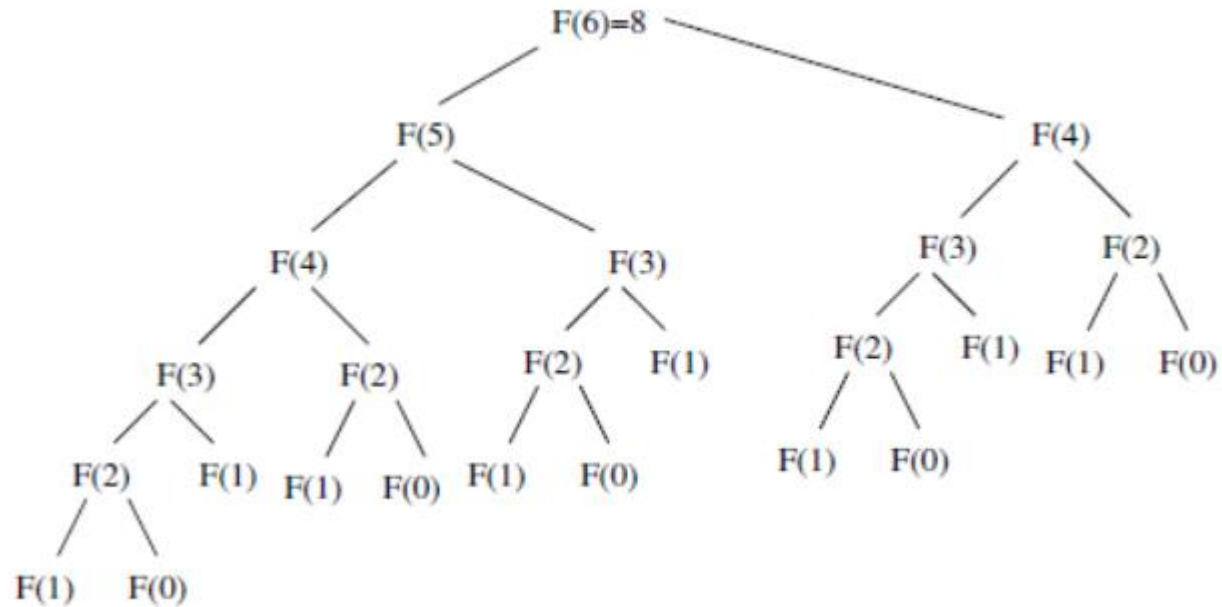
$$\begin{aligned} \implies T(n) &= T(n-1) + T(n-2) + O(1) \geq F_n \approx \varphi^n \\ &\geq 2T(n-2) + O(1) \geq 2^{n/2} \end{aligned}$$

EXPONENTIAL — BAD!

# Fibonacci Numbers

$$F_0 = 0 \text{ and } F_1 = 1$$

$$F_n = F_{n-1} + F_{n-2}$$



# Fibonacci Numbers

## Memoized DP Algorithm

Remember, remember

```
memo = { }  
fib( $n$ ):  
    if  $n$  in memo: return memo[ $n$ ]  
    else: if  $n \leq 2$  :  $f = 1$   
          else:  $f = \text{fib}(n - 1) + \text{fib}(n - 2)$   
          memo[ $n$ ] =  $f$   
    return  $f$ 
```

# Fibonacci Numbers

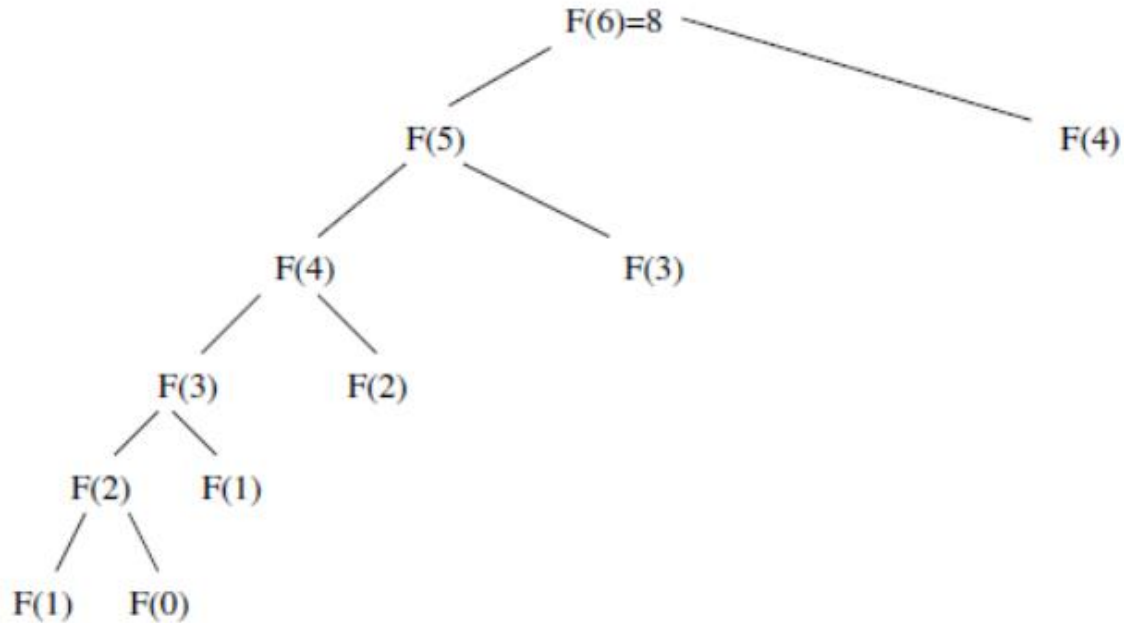


Figure 8.2: The Fibonacci computation tree when caching values

# Fibonacci Numbers

- `fib(k)` only recurses first time called,  $\forall k$
- only  $n$  nonmemoized calls:
- memoized calls free ( $\Theta(1)$  time)
- $\Theta(1)$  time per call (ignoring recursion)

❑ POLYNOMIAL — GOOD!



# Bottom-UP DP

## Bottom-up DP Algorithm

```
fib = {}  
for  $k$  in  $[1, 2, \dots, n]$ :  
    if  $k \leq 2$ :  $f = 1$   
    else:  $f = \text{fib}[k - 1] + \text{fib}[k - 2]$   
     $\text{fib}[k] = f$   
return  $\text{fib}[n]$ 
```

$\Theta(1)$

$\Theta(n)$

- exactly the same computation as memoized DP (recursion “unrolled”)

# DYNAMIC PROGRAMMING

## Two approaches for DP

(2 different ways to think about and/or implement DP algorithms)

**Bottom-up:** iterates through problems by size and solves the small problems first (kind of like taking care of base cases first & building up).

**Top-down:** instead uses recursive calls to solve smaller problems, while using memoization/caching to keep track of small problems that you've already computed answers for (simply fetch the answer instead of re-solving that problem and waste computational effort)

# RECIPE FOR APPLYING DP

1. **Identify optimal substructure.** What are your overlapping subproblems?
2. **Define a recursive formulation.** Recursively define your optimal solution in terms of sub-solutions. *Always write down this formulation.*
3. **Use dynamic programming.** Turn the recursive formulation into a DP algorithm.
4. **If needed, track additional information.** You may need to solve a related problem, e.g. step 3 finds you an optimal *value/cost*, but you need to recover the actual optimal *solution/path/subset/substring/etc.* Go back and modify your algorithm in step 3 to make this happen.






# KNAPSACK

What's the most valuable way to fill a knapsack with items?

# THE KNAPSACK PROBLEM

**What's the most valuable way to cram items into my knapsack?**

We have  $n$  items with weights and values.

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

We also have a knapsack, and it can only carry so much weight:



Capacity: **10**

# Different algorithm approaches

1. **Brute-force.** Try all possibilities until a satisfactory solution is found
2. **Divide and Conquer.** Divide problem into smaller independent sub-problems and then recursively solves these sub-problems to build solution.
3. **Greedy Algorithm.** It is used to find the best solution by taking best sub-solution at every step.
4. **Dynamic programming algorithm .** Break problem into smaller overlapping sub-problems. Sub-problems that are repeated are solved only once and result is stored (which is called memoization) and this stored result is used next time rather than recomputing solution for another same sub-problem.

# KNAPSACK PROBLEM: TWO VERSIONS



Capacity: **10**

Item:  
Weight:  
Value:



**6**

**20**



**2**

**8**



**4**

**14**



**3**

**13**



**11**

**35**

## UNBOUNDED KNAPSACK

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



Total weight:  $2 + 2 + 3 + 3 = 10$

Total value:  $8 + 8 + 13 + 13 = 42$

## 0/1 KNAPSACK

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



Total weight:  $2 + 4 + 3 = 9$

Total value:  $8 + 14 + 13 = 35$

# THE **UNBOUNDED** KNAPSACK PROBLEM

When we have infinite copies of all items



# SOME NOTATION

## UNBOUNDED KNAPSACK

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



Capacity:  **$W$**

Item:  
Weight:  
Value:



$w_1$   
 $v_1$



$w_2$   
 $v_2$



$w_3$   
 $v_3$

...



$w_n$   
 $v_n$

# RECIPE FOR APPLYING DP

1. **Identify optimal substructure.** What are your overlapping subproblems?
2. **Define a recursive formulation.** Recursively define your optimal solution in terms of sub-solutions. *Always write down this formulation.*
3. **Use dynamic programming.** Turn the recursive formulation into a DP algorithm.
4. **If needed, track additional information.** You may need to solve a related problem, e.g. step 3 finds you an optimal *value/cost*, but you need to recover the actual optimal *solution/path/subset/substring/etc.* Go back and modify your algorithm in step 3 to make this happen.

# STEP 1: OPTIMAL SUBSTRUCTURE

## SUBPROBLEMS:

Unbounded Knapsack with a smaller knapsack

$K[x]$  = optimal value you can fit in a knapsack of capacity  $x$



We'll first solve  
the problem for  
small knapsacks



Then larger  
backpacks



Then larger  
backpacks

# STEP 1: OPTIMAL SUBSTRUCTURE

## SUBPROBLEMS:

Unbounded Knapsack with a smaller knapsack





$K[x]$  = optimal value you can fit in a knapsack of capacity  $x$

Why does this make sense, and how can subproblems help me find an optimal solution for  $K[x]$ ?

Basically, I would like to take the maximum outcome over all the available possibilities:

**My knapsack has capacity  $x$ . Which item should I put in my knapsack for now?**

Well, if I put in item  $i$  with weight  $w_i$ , the best value I could achieve is the value of item  $i$ ,  $v_i$ , *plus the optimal value for a smaller knapsack* that has capacity  $x - w_i$  (i.e. the remaining space once I put item  $i$  in).

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

# STEP 1: OPTIMAL SUBSTRUCTURE

## SUBPROBLEMS:

Unbounded Knapsack with a smaller knapsack

$K[x]$  = optimal value you can fit in a knapsack of capacity  $x$

### Our high-level gameplan:

For each item  $i$  that can fit in the knapsack,  
figure out how “good” of a choice that would be:

**Value of that choice =  $[v_i] + [\text{best value with capacity } x - w_i]$**

We'll go with the most rewarding of those choices!

Weight:	$w_1$	$w_2$	$w_3$	...	$w_n$
Value:	$v_1$	$v_2$	$v_3$		$v_n$

# RECIPE FOR APPLYING DP

1. **Identify optimal substructure.** What are your overlapping subproblems?
2. **Define a recursive formulation.** Recursively define your optimal solution in terms of sub-solutions. *Always write down this formulation.*
3. **Use dynamic programming.** Turn the recursive formulation into a DP algorithm.
4. **If needed, track additional information.** You may need to solve a related problem, e.g. step 3 finds you an optimal *value/cost*, but you need to recover the actual optimal *solution/path/subset/substring/etc.* Go back and modify your algorithm in step 3 to make this happen.

# STEP 2: RECURSIVE FORMULATION

$K[x]$  = optimal value you can fit in a knapsack of capacity  $x$

Our recursive formulation:

$$K[x] = \begin{cases} 0 & \text{if there are no } i \text{ where } w_i \leq x \\ \max_i \{ K[x - w_i] + v_i \} & \text{otherwise} \end{cases}$$

The maximum is over all items  $i$  s.t.  $w_i \leq x$  (i.e. over all the items that could actually fit)

Optimal way to fill the smaller knapsack



The value of item  $i$



# RECIPE FOR APPLYING DP

1. **Identify optimal substructure.** What are your overlapping subproblems?
2. **Define a recursive formulation.** Recursively define your optimal solution in terms of sub-solutions. *Always write down this formulation.*
3. **Use dynamic programming.** Turn the recursive formulation into a DP algorithm.
4. **If needed, track additional information.** You may need to solve a related problem, e.g. step 3 finds you an optimal *value/cost*, but you need to recover the actual optimal *solution/path/subset/substring/etc.* Go back and modify your algorithm in step 3 to make this happen.



# STEP 3: WRITE A DP ALGORITHM

We'll store answers to our subproblems  $K[x]$  in a row/1-D table (this is our cache)!

Now that we've defined our recursive formulation, translating to appropriate pseudocode is straightforward: establish your base cases & define your cases!

**We'll do this in a bottom-up fashion. Why?**

Again, it's clear that we need answers to smaller knapsacks before we need answers to larger knapsacks, so we might as well just iterate from  $K[0]$  and work our way towards our final answer (which will be  $K[W]$ ).

# STEP 3: WRITE A DP ALGORITHM

$$K[x] = \begin{cases} 0 & \text{if there are no } i \text{ where } w_i \leq x \\ \max_i \{ K[x-w_i] + v_i \} & \text{otherwise} \end{cases}$$

**UNBOUNDED\_KNAPSACK**(W, n, weights, values):

Initialize a size W+1 array, K

Make sure that our  
base case is set up  
(0 capacity means  
0 value)

→ K[0] = 0

for x = 1,...,W:

← Iterate over each knapsack size  
from smallest to largest

K[x] = 0

for i = 1,...,n:

← Iterate over each possible item  
& only process those that could  
actually fit in a size x knapsack

if  $w_i \leq x$ :

K[x] = max{ K[x], K[x-w<sub>i</sub>] + v<sub>i</sub> }

return K[W]


**Runtime: O(nW)**


You do O(n) work to fill out  
each of the W entries in the  
array

# RECIPE FOR APPLYING DP

1. **Identify optimal substructure.** What are your overlapping subproblems?
2. **Define a recursive formulation.** Recursively define your optimal solution in terms of sub-solutions. *Always write down this formulation.*
3. **Use dynamic programming.** Turn the recursive formulation into a DP algorithm.
4. **If needed, track additional information.** You may need to solve a related problem, e.g. step 3 finds you an optimal *value/cost*, but you need to recover the actual optimal *solution/path/subset/substring/etc.* Go back and modify your algorithm in step 3 to make this happen.

# EXAMPLE

	0	1	2	3	4
K	0	1			
ITEMS					

$K[1] = \max\{ K[1], K[0] + 1 \}$   
**ITEMS[1] = ITEMS[0] +**   
*update!*

**UNBOUNDED\_KNAPSACK\_ITEMS**(W, n, weights, values):

Initialize size W+1 arrays, K and ITEMS

$K[0] = 0$ , **ITEMS[0] = { }**

for  $x = 1, \dots, W$ :

$K[x] = 0$ , **ITEMS[x] = { }**

for  $i = 1, \dots, n$ :

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<sub>i</sub>] ∪ {item i}**

return **ITEMS[W]**



Item:



Weight:

**1**

**2**

**3**

Value:


**1**

**4**

**6**

Capacity: **4**

# EXAMPLE

	0	1	2	3	4
K	0	1	0		
ITEMS					

$$K[2] = \max\{ K[2], K[1] + 1 \}$$

$$\text{ITEMS}[2] = \text{ITEMS}[1] + \text{avocado}$$

**UNBOUNDED\_KNAPSACK\_ITEMS**(W, n, weights, values):

Initialize size W+1 arrays, K and ITEMS

$K[0] = 0$ , **ITEMS[0] = { }**

for  $x = 1, \dots, W$ :

$K[x] = 0$ , **ITEMS[x] = { }**

for  $i = 1, \dots, n$ :

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<sub>i</sub>] ∪ {item i}**

return **ITEMS[W]**



Item:



Weight:

**1**

**2**

**3**

Value:




**1**

**4**

**6**

Capacity: **4**

# EXAMPLE

	0	1	2	3	4
K	0	1	2		
ITEMS			 		

$$K[2] = \max\{ K[2], K[0] + 4 \}$$

$$\text{ITEMS}[2] = \text{ITEMS}[0] + \text{🍷}$$

**UNBOUNDED\_KNAPSACK\_ITEMS**(W, n, weights, values):

Initialize size W+1 arrays, K and ITEMS

$K[0] = 0$ , **ITEMS[0] = { }**

for  $x = 1, \dots, W$ :

$K[x] = 0$ , **ITEMS[x] = { }**

for  $i = 1, \dots, n$ :

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<sub>i</sub>] ∪ {item i}**

return **ITEMS[W]**



Item:



Weight:

**1**

**2**

**3**

Value:



**1**


**4**

**6**

Capacity: **4**

# EXAMPLE

	0	1	2	3	4
K	0	1	4		
ITEMS					

$K[2] = \max\{ K[2], K[0] + 4 \}$   
**ITEMS[2] = ITEMS[0] + **  
*update!*

**UNBOUNDED\_KNAPSACK\_ITEMS**(W, n, weights, values):

Initialize size W+1 arrays, K and ITEMS

$K[0] = 0$ , **ITEMS[0] = { }**

for  $x = 1, \dots, W$ :

$K[x] = 0$ , **ITEMS[x] = { }**

for  $i = 1, \dots, n$ :

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<sub>i</sub>] ∪ {item i}**

return **ITEMS[W]**



Item:



Weight:

**1**

**2**

**3**

Value:





**1**


**4**

**6**

Capacity: **4**

# EXAMPLE

	0	1	2	3	4
K	0	1	4	5	
ITEMS				 	

$K[3] = \max\{ K[3], K[2] + 1 \}$   
**ITEMS[3] = ITEMS[2] +**   
*update!*

**UNBOUNDED\_KNAPSACK\_ITEMS**(W, n, weights, values):

Initialize size W+1 arrays, K and ITEMS

$K[0] = 0$ , **ITEMS[0] = { }**

for  $x = 1, \dots, W$ :

$K[x] = 0$ , **ITEMS[x] = { }**

for  $i = 1, \dots, n$ :

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<sub>i</sub>] ∪ {item i}**

return **ITEMS[W]**



Item:



Weight:

**1**

**2**

**3**

Value:

**1**




**4**


**6**

Capacity: **4**



# EXAMPLE

	0	1	2	3	4
K	0	1	4	6	
ITEMS					

$K[3] = \max\{ K[3], K[0] + 6 \}$   
**ITEMS[3] = ITEMS[0] + **  
*update!*

**UNBOUNDED\_KNAPSACK\_ITEMS**(W, n, weights, values):

Initialize size W+1 arrays, K and ITEMS

$K[0] = 0$ , **ITEMS[0] = { }**

for  $x = 1, \dots, W$ :

$K[x] = 0$ , **ITEMS[x] = { }**

for  $i = 1, \dots, n$ :

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<sub>i</sub>] ∪ {item i}**

return **ITEMS[W]**



Item:



Weight:

**1**

**2**

**3**

Value:




**1**

**4**

**6**

Capacity: **4**

# EXAMPLE

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

$$K[4] = \max\{ K[4], K[3] + 1 \}$$

$$\text{ITEMS}[4] = \text{ITEMS}[3] + \text{avocado}$$

**UNBOUNDED\_KNAPSACK\_ITEMS**(W, n, weights, values):

Initialize size W+1 arrays, K and ITEMS

$K[0] = 0$ , **ITEMS[0] = { }**

for  $x = 1, \dots, W$ :

$K[x] = 0$ , **ITEMS[x] = { }**

for  $i = 1, \dots, n$ :

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<sub>i</sub>] ∪ {item i}**

return **ITEMS[W]**



Item:



Weight:

**1**

**2**

**3**

Value:






**1**


**4**

**6**

Capacity: **4**

# EXAMPLE

	0	1	2	3	4
K	0	1	4	6	7
ITEMS					 

$K[4] = \max\{ K[4], K[3] + 1 \}$   
**ITEMS[4] = ITEMS[3] +**   
*update!*

**UNBOUNDED\_KNAPSACK\_ITEMS**(W, n, weights, values):

Initialize size W+1 arrays, K and ITEMS

$K[0] = 0$ , **ITEMS[0] = { }**

for  $x = 1, \dots, W$ :

$K[x] = 0$ , **ITEMS[x] = { }**

for  $i = 1, \dots, n$ :

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<sub>i</sub>] ∪ {item i}**

return **ITEMS[W]**



Item:



Weight:

**1**

**2**

**3**

Value:






**1**


**4**

**6**

Capacity: **4**

# EXAMPLE

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

$K[4] = \max\{ K[4], K[2] + 4 \}$   
 $ITEMS[4] = ITEMS[2] +$ 

  
*update!*

**UNBOUNDED\_KNAPSACK\_ITEMS**(W, n, weights, values):

Initialize size W+1 arrays, K and ITEMS

$K[0] = 0$ ,  $ITEMS[0] = \{ \}$

for  $x = 1, \dots, W$ :

$K[x] = 0$ ,  $ITEMS[x] = \{ \}$

for  $i = 1, \dots, n$ :

if  $w_i \leq x$ :

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

**if  $K[x]$  was updated:**

$ITEMS[x] = ITEMS[x-w_i] \cup \{ \text{item } i \}$

return  $ITEMS[W]$



Item:



Weight:

1

2

3

Value:






1


4

6

Capacity: 4

# EXAMPLE

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

$K[4] = \max\{ K[4], K[1] + 6 \}$   
**ITEMS[4] = ITEMS[1] + **  
*(koala doesn't cause update)*

**UNBOUNDED\_KNAPSACK\_ITEMS**(W, n, weights, values):

Initialize size W+1 arrays, K and ITEMS

$K[0] = 0$ , **ITEMS[0] = { }**

for  $x = 1, \dots, W$ :

$K[x] = 0$ , **ITEMS[x] = { }**

for  $i = 1, \dots, n$ :

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<sub>i</sub>] ∪ {item i}**

return **ITEMS[W]**



Item:



Weight:

**1**

**2**

**3**

Value:






**1**

**4**

**6**

Capacity: **4**

# EXAMPLE

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

**We're done!**

**UNBOUNDED\_KNAPSACK\_ITEMS**(W, n, weights, values):

Initialize size W+1 arrays, K and ITEMS

$K[0] = 0$ , **ITEMS[0] = { }**

for  $x = 1, \dots, W$ :

$K[x] = 0$ , **ITEMS[x] = { }**

for  $i = 1, \dots, n$ :

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<sub>i</sub>] ∪ {item i}**

return **ITEMS[W]**



Item:



Weight:

**1**

**2**

**3**

Value:

**1**

**4**

**6**

Capacity: **4**

# 0/1 KNAPSACK

When we have only one copy of each item

# RECIPE FOR APPLYING DP

1. **Identify optimal substructure.** What are your overlapping subproblems?
2. **Define a recursive formulation.** Recursively define your optimal solution in terms of sub-solutions. *Always write down this formulation.*
3. **Use dynamic programming.** Turn the recursive formulation into a DP algorithm.
4. **If needed, track additional information.** You may need to solve a related problem, e.g. step 3 finds you an optimal *value/cost*, but you need to recover the actual optimal *solution/path/subset/substring/etc.* Go back and modify your algorithm in step 3 to make this happen.



# STEP 1: OPTIMAL SUBSTRUCTURE

## SUBPROBLEM:

0/1 Knapsack with a smaller knapsack & *fewer items*

First solve the problem  
for a few items



Solve using  
small knapsacks



Then larger  
knapsacks



Then even larger  
knapsacks



Then more items



Then even  
more items



**This calls for a two-  
dimensional table!**

# RECIPE FOR APPLYING DP

1. **Identify optimal substructure.** What are your overlapping subproblems?
2. **Define a recursive formulation.** Recursively define your optimal solution in terms of sub-solutions. *Always write down this formulation.*
3. **Use dynamic programming.** Turn the recursive formulation into a DP algorithm.
4. **If needed, track additional information.** You may need to solve a related problem, e.g. step 3 finds you an optimal *value/cost*, but you need to recover the actual optimal *solution/path/subset/substring/etc.* Go back and modify your algorithm in step 3 to make this happen.

## STEP 2: RECURSIVE FORMULATION

- Let OPT be an optimal solution
- Note that presence of item  $i$  in OPT, does not forbid any other item  $j$ .
- Item  $j$  (last one) either belongs to OPT, or it doesn't.
  - If  $j \in \text{OPT}$ :
    - Optimal solution contains ' $j$ ',
    - plus optimal solution of other  $j - 1$  items,
    - But with a reduced maximum weight of  $W - w_j$
  - If  $j \notin \text{OPT}$ :
    - Optimal solution if for  $j - 1$  items,
    - with maximum allowed weight  $W$  remain unchanged.

## STEP 2: RECURSIVE FORMULATION

- $w_j > W \Rightarrow j \notin \text{OPT}$  ----- **Leave object**
  - $\text{OPT}(j, w) = \text{OPT}(j - 1, w)$
- Otherwise,  $j$  is either  $\in \text{OPT}$  or  $\notin \text{OPT}$
- If  $j \in \text{OPT}$ : ----- **Take object**
  - $\text{OPT}(j, w) = v_j + \text{OPT}(j - 1, W - w_j)$
- If  $j \notin \text{OPT}$ : ----- **Leave object**
  - $\text{OPT}(j, w) = \text{OPT}(j - 1, w)$
- $\text{OPT}(j, w) = \text{MAX}(v_j + \text{OPT}(j - 1, W - w_j), \text{OPT}(j - 1, w) )$

# STEP 2: RECURSIVE FORMULATION

$K[x, j]$  = optimal value you can fit in a knapsack of capacity  $x$  with items 1 through  $j$

Our recursive formulation:

$$K[x, j] = \begin{cases} 0 & \text{if } x \text{ or } j \text{ are } 0 \\ \max \{ K[x, j-1], K[x-w_j, j-1] + v_j \} & \text{otherwise} \end{cases}$$

Optimal way to fill the same size knapsack without using item  $j$

Optimal way to fill the smaller knapsack when we no longer have access to item  $j$

value gained by using item  $j$

# RECIPE FOR APPLYING DP

1. **Identify optimal substructure.** What are your overlapping subproblems?
2. **Define a recursive formulation.** Recursively define your optimal solution in terms of sub-solutions. *Always write down this formulation.*
3. **Use dynamic programming.** Turn the recursive formulation into a DP algorithm.
4. **If needed, track additional information.** You may need to solve a related problem, e.g. step 3 finds you an optimal *value/cost*, but you need to recover the actual optimal *solution/path/subset/substring/etc.* Go back and modify your algorithm in step 3 to make this happen.

# STEP 3: WRITE A DP ALGORITHM

$$K[x, j] = \begin{cases} 0 & \text{if } x \text{ or } j \text{ are } 0 \\ \max\{K[x, j-1], K[x-w_j, j-1] + v_j\} & \text{otherwise} \end{cases}$$

**ZERO\_ONE\_KNAPSACK**(W, n, weights, values):

Initialize a (n+1) x (W+1) table, K

K[x,0] = 0 for all x = 0,...,W

Make sure that our base case is set up (0 value for entries where we have 0 capacity or 0 items)

K[0,j] = 0 for all j = 0,...,n

Iterate over items we can consider

for j = 1,...,n:

Iterate over knapsack sizes from smallest to largest

for x = 1,...,W:

K[x,j] = K[x,j-1]

Default case: we don't use item j

if  $w_j \leq x$ :







K[x,j] = max{ K[x,j], K[x-w<sub>j</sub>, j-1] + v<sub>j</sub> }

But if item j can fit, then we'll consider using it!

return K[W,n]

**Runtime: O(nW)** You do O(1) work to fill out each of the nW entries in the table

# EXAMPLE

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

Initialize all our “base cases” first!

## ZERO\_ONE\_KNAPSACK(W, n, weights, values):

Initialize a  $(n+1) \times (W+1)$  table, K

$K[x,0] = 0$  for all  $x = 0, \dots, W$

$K[0,j] = 0$  for all  $j = 0, \dots, n$

for  $j = 1, \dots, n$ :

for  $x = 1, \dots, W$ :

$K[x,j] = K[x,j-1]$

if  $w_j \leq x$ :

$K[x,j] = \max\{ K[x,j], K[x-w_j,j-1] + v_j \}$

return  $K[W,n]$



Capacity: **3**

Item:



Weight:

**1**

Value:

**1**



**2**

**4**









**3**


**6**



# EXAMPLE

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

Default value =  $K[x, j-1]$

If  can fit, would it help?

YES!  $0 + 1$  is better!

**ZERO\_ONE\_KNAPSACK(W, n, weights, values):**

Initialize a  $(n+1) \times (W+1)$  table, K

$K[x, 0] = 0$  for all  $x = 0, \dots, W$

$K[0, j] = 0$  for all  $j = 0, \dots, n$

for  $j = 1, \dots, n$ :

for  $x = 1, \dots, W$ :

$K[x, j] = K[x, j-1]$

if  $w_j \leq x$ :

$K[x, j] = \max\{ K[x, j], K[x-w_j, j-1] + v_j \}$

return  $K[W, n]$



Item:



Weight:

1

2

3

Value:







1


4

6

Capacity: 3

# EXAMPLE

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

Default value =  $K[x, j-1]$   
 If  can fit, would it help?  
 YES!  $0 + 1$  is better!

## ZERO\_ONE\_KNAPSACK(W, n, weights, values):

Initialize a  $(n+1) \times (W+1)$  table, K  
 $K[x, 0] = 0$  for all  $x = 0, \dots, W$   
 $K[0, j] = 0$  for all  $j = 0, \dots, n$   
 for  $j = 1, \dots, n$ :  
   for  $x = 1, \dots, W$ :  
      $K[x, j] = K[x, j-1]$   
     if  $w_j \leq x$ :  
        $K[x, j] = \max\{ K[x, j], K[x-w_j, j-1] + v_j \}$   
 return  $K[W, n]$



Capacity: **3**

Item:



Weight:

**1**

**2**

**3**

Value:

**1**

**4**

**6**

# EXAMPLE

	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	

Default value =  $K[x, j-1]$

If 🍷 can fit, would it help?

YES!  $1 + 4$  is better!

**ZERO\_ONE\_KNAPSACK(W, n, weights, values):**

Initialize a  $(n+1) \times (W+1)$  table, K

$K[x, 0] = 0$  for all  $x = 0, \dots, W$

$K[0, j] = 0$  for all  $j = 0, \dots, n$

for  $j = 1, \dots, n$ :

for  $x = 1, \dots, W$ :

$K[x, j] = K[x, j-1]$

if  $w_j \leq x$ :

$K[x, j] = \max\{ K[x, j], K[x-w_j, j-1] + v_j \}$

return  $K[W, n]$



Item:



Weight:

1

2

3

Value:







1

4


6

Capacity: 3

# EXAMPLE

	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

Default value =  $K[x, j-1]$

If  can fit, would it help?

YES!  $0 + 6$  is better!

**ZERO\_ONE\_KNAPSACK(W, n, weights, values):**

Initialize a  $(n+1) \times (W+1)$  table, K

$K[x, 0] = 0$  for all  $x = 0, \dots, W$

$K[0, j] = 0$  for all  $j = 0, \dots, n$

for  $j = 1, \dots, n$ :

for  $x = 1, \dots, W$ :

$K[x, j] = K[x, j-1]$

if  $w_j \leq x$ :

$K[x, j] = \max\{ K[x, j], K[x-w_j, j-1] + v_j \}$

return  $K[W, n]$



Item:



Weight:

1

2

3

Value:

1

4

6

Capacity: 3

# RECIPE FOR APPLYING DP

1. **Identify optimal substructure.** What are your overlapping subproblems?
2. **Define a recursive formulation.** Recursively define your optimal solution in terms of sub-solutions. *Always write down this formulation.*
3. **Use dynamic programming.** Turn the recursive formulation into a DP algorithm.
4. **If needed, track additional information.** You may need to solve a related problem, e.g. step 3 finds you an optimal *value/cost*, but you need to recover the actual optimal *solution/path/subset/substring/etc.* Go back and modify your algorithm in step 3 to make this happen.

# RECIPE FOR APPLYING DP

1. Identify the substructure. What does your base case look like?

**Try to add code to the ZERO-ONE-KNAPSACK pseudocode to recover the actual item set that contributes to the optimal solution.**

The example diagram basically shows how to track it!

algorithm in step 3 to make this happen.

Let  $W = 10$  and

$i$	1	2	3	4
$v_i$	10	40	30	50
$w_i$	5	4	6	3

Let  $W = 10$  and

$i$	1	2	3	4
$v_i$	10	40	30	50
$w_i$	5	4	6	3

$$K[i, w] \leftarrow \text{MAX}(v_j + K[i - 1, w - w_i], K[i - 1, w])$$

[illegible]



Let  $W = 10$  and

$i$	1	2	3	4
$v_i$	10	40	30	50
$w_i$	5	4	6	3

$$K[i, w] \leftarrow \text{MAX}(v_i + K[i - 1, w - w_i], K[i - 1, w])$$

K [i, w]	W=0	W=1	W=2	W=3	W=4	W=5	W=6	W=7	W=8	W=9	W = 10
i = 0	0	0	0	0	0	0	0	0	0	0	0
i = {1}	0	0	0	0	0	10	10	10	10	10	10
i={1,2}	0	0	0	0	40	40	40	40	40	50	50
i={1,2,3}	0	0	0	0	40	40	40	40	40	50	70
i={1,2,3,4}	0	0	0	50	50	50	50	90	90	90	90

# RECIPE FOR APPLYING DP

1. **The method described does not tell which subset gives the optimal solution. (It is {2,4} in this example).**

2. **D** terms
- Step 4: **track additional information (keep table)**

3. **U** m.
4. **If needed, track additional information.** You may need to solve a related problem, e.g. step 3 finds you an optimal *value/cost*, but you need to recover the actual optimal *solution/path/subset/substring/etc.* Go back and modify your algorithm in step 3 to make this happen.

# RECIPE FOR APPLYING DP

## ZERO\_ON\_KNAPSACK(W, n, weights, values):

Initialize a  $(n+1) \times (W+1)$  table, K

$K[x,0] = 0$  for all  $x = 0, \dots, W$

$K[0,j] = 0$  for all  $j = 0, \dots, n$

for  $j = 1, \dots, n$ :

  for  $x = 1, \dots, W$ :

$K[x,j] = K[x,j-1]$

    if  $w_j \leq x$ :

$K[x,j] = \max\{ K[x,j], K[x-w_j,j-1] + v_j \}$

return  $K[W,n]$

## ZERO\_ONE\_KNAPSACK(W, n, weights, values):

Initialize a  $(n+1) \times (W+1)$  table, K

$K[x,0] = 0$  for all  $x = 0, \dots, W$

$K[0,j] = 0$  for all  $j = 0, \dots, n$

for  $j = 1, \dots, n$ :

  for  $x = 1, \dots, W$ :

$K[x,j] = K[x,j-1]$

    if  $w_j \leq x$  and  $K[x-w_j,j-1] + v_j > K[x,j]$ :

$K[x,j] = K[x-w_j,j-1] + v_j$

      keep  $[j, w] = 1$

    else

$K[x,j] = K[x,j]$

      keep  $[j, w] = 0$

K = W

for  $j = n, \dots, 1$

  if ( keep  $[j, k] == 1$  )

    output j

$k = k - w[j]$

return  $K[W,n]$

Main table

	W=0	W=1	W=2	W=3	W=4	W=5	W=6	W=7	W=8	W=9	W = 10
	0	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	10	10	10	10	10	10
$i=\{1,2\}$	0	0	0	0	40	40	40	40	40	50	50
$i=\{1,2,3\}$	0	0	0	0	40	40	40	40	40	50	70
$i=\{1,2,3,4\}$	0	0	0	50	50	50	50	90	90	90	90

Keep table

	W=0	W=1	W=2	W=3	W=4	W=5	W=6	W=7	W=8	W=9	W = 10
	0	0	0	0	0	0	0	0	0	0	0
$l = \{1\}$	0	0	0	0	0	1	1	1	1	1	1
$i=\{1,2\}$	0	0	0	0	1	1	1	1	1	1	1
$i=\{1,2,3\}$	0	0	0	0	0	0	0	0	0	0	1
$i=\{1,2,3,4\}$	0	0	0	1	1	1	1	1	1	1	1

# Example

- Let  $W = 8$
- $w_i = \{2, 3, 4, 5\}$
- $v_i = \{3, 4, 5, 7\}$
- $M[j, w] \leftarrow \text{MAX}(v_j + M[j - 1, w - w_j], M[j - 1, w])$

$V_i$	$W_i$	index	$W=0$	$W=1$	$W=2$	$W=3$	$W=4$	$W=5$	$W=6$	$W=7$	$W = 8$
0	0	0	0	0	0	0	0	0	0	0	0
3	2	1	0								
4	3	2	0								
5	4	3	0								
7	5	4	0								

# Example

- Let  $W = 8$
- $w_i = \{2, 3, 4, 5\}$
- $v_i = \{3, 4, 5, 7\}$
- $M[j, w] \leftarrow \text{MAX}(v_j + M[j - 1, w - w_j], M[j - 1, w])$

$V_i$	$W_i$	index	$W=0$	$W=1$	$W=2$	$W=3$	$W=4$	$W=5$	$W=6$	$W=7$	$W = 8$
0	0	0	0	0	0	0	0	0	0	0	0
3	2	1	0	3	3	3	3	3	3	3	3
4	3	2	0								
5	4	3	0								
7	5	4	0								

# Example

- Let  $W = 8$
- $w_i = \{2, 3, 4, 5\}$
- $v_i = \{3, 4, 5, 7\}$
- $M[j, w] \leftarrow \text{MAX}(v_j + M[j - 1, w - w_j], M[j - 1, w])$

$V_i$	$W_i$	index	W=0	W=1	W=2	W=3	W=4	W=5	W=6	W=7	W = 8
0	0	0	0	0	0	0	0	0	0	0	0
3	2	1	0	3	3	3	3	3	3	3	3
4	3	2	0	3	3	4	4				
5	4	3	0								
7	5	4	0								

# Example

- Let  $W = 8$
- $w_i = \{2, 3, 4, 5\}$
- $v_i = \{3, 4, 5, 7\}$
- $M[j, w] \leftarrow \text{MAX}(v_j + M[j - 1, w - w_j], M[j - 1, w])$

$V_i$	$W_i$	index	$W=0$	$W=1$	$W=2$	$W=3$	$W=4$	$W=5$	$W=6$	$W=7$	$W = 8$
0	0	0	0	0	0	0	0	0	0	0	0
3	2	1	0	3	3	3	3	3	3	3	3
4	3	2	0	3	3	4	4	7	7	7	7
5	4	3	0	3	3	4	5	5+3	5+3	5+4	5+4
7	5	4	0	3	3	4	5	8	8	7+3 =10	7+4 = 11



# RECIPE FOR APPLYING DP

1. **The method described does not tell which subset gives the optimal solution. (It is {2,4} in this example).**

2. **D** terms
- Step 4: **track additional information (keep table)**

3. **U** m.
4. **If needed, track additional information.** You may need to solve a related problem, e.g. step 3 finds you an optimal *value/cost*, but you need to recover the actual optimal *solution/path/subset/substring/etc.* Go back and modify your algorithm in step 3 to make this happen.

