ساختمان داده و الگوریتم ها (CE203)

جلسه بیست و یکم:
مسئله کوله پشتی

سجاد شیرعلی شهرضا

پاییز 1400

دوشنبه، 22 آذر 1400

Based on Stanford CS161 Slides from Summer 2020 by Karey Shi

اطلاع رسانی

- بخش مرتبط کتاب برای این جلسه: 15 و 16.2
- یادآوری مهلت ارسال تمرین سوم: 8 صبح روز چهارشنبه 24 آذر 1400
- امتحانک سوم: دوشنبه هفته آینده، 29 آذر 1400، در وقت کلاس
- قرار دادن نظرسنجی چهارم : مهلت ارسال: 8 صبح روز سه شنبه 30 آذر 1400
- نمرات تمرین دوم و امتحان میان ترم اعلام شده است
  - در صورت داشتن سوال در مورد نمره میان ترم، از طریق ایمیل تا قبل از ساعت 6 امروز با من مکاتبه کنید.
  - نمرات میانترم باید امروز در سامانه ثبت نهایی شوند

دستورالعمل استفاده از برنامه نویسی پویا

گام های طراحی یک الگوریتم برنامه نویسی پویا

DYNAMIC PROGRAMMING

Elements of dynamic programming:

DYNAMIC PROGRAMMING

Elements of dynamic programming:

Optimal substructure: the optimal solution of a problem can be expressed in terms of optimal solutions to smaller sub-problems.

DYNAMIC PROGRAMMING

Elements of dynamic programming:

Optimal substructure: the optimal solution of a problem can be expressed in terms of optimal solutions to smaller sub-problems.

e.g. $\mathbf{d^{(k)}[b] = \min\{\, d^{(k-1)}[b],\ \min_a\{d^{(k-1)}[a] + w(a, b)\}\,\}}$

DYNAMIC PROGRAMMING

Elements of dynamic programming:

Optimal substructure: the optimal solution of a problem can be expressed in terms of optimal solutions to smaller sub-problems.

e.g. $\mathbf{d^{(k)}[b] = \min\{\, d^{(k-1)}[b], \min_a\{d^{(k-1)}[a] + w(a, b)\}\,\}}$

Overlapping sub-problems: the subproblems overlap a lot!

This means we can save time by solving a sub-problem once & cache the answer.

(this is sometimes called “memoization”)

DYNAMIC PROGRAMMING

Elements of dynamic programming:

Optimal substructure: the optimal solution of a problem can be expressed in terms of optimal solutions to smaller sub-problems.

e.g. $\mathbf{d^{(k)}[b] = \min\{ d^{(k-1)}[b], \min_a\{d^{(k-1)}[a] + w(a, b)\} \}}$

Overlapping sub-problems: the subproblems overlap a lot!

This means we can save time by solving a sub-problem once & cache the answer.

(this is sometimes called “memoization”)

e.g. **Lots of different entries in the row $d^{(k)}$ may ask for $d^{(k-1)}[v]$**

RECIPE FOR APPLYING DP

RECIPE FOR APPLYING DP

1. **Identify optimal substructure.** What are your overlapping subproblems?

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.*

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.

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.

سوال؟

مسئله کوله پشتی

چگونگی قرار دادن اشیا با حداکثر ارزش در یک کوله پشتی

THE KNAPSACK PROBLEM

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

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**

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**

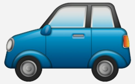
KNAPSACK PROBLEM: TWO VERSIONS

Capacity: **10**

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

KNAPSACK PROBLEM: TWO VERSIONS

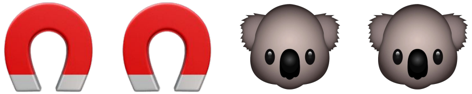
Capacity: **10**

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

KNAPSACK PROBLEM: TWO VERSIONS

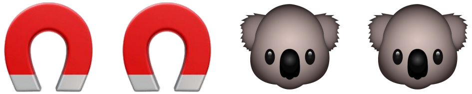
Capacity: **10**

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

سوال؟

مسئله کوله پشتی بدون حد

وقتی که از هر شی هر چقدر بخواهیم، داریم

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:	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 1: OPTIMAL SUBSTRUCTURE

SUBPROBLEMS:

Unbounded Knapsack with a smaller knapsack

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

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

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 medium 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**

We'll first solve the problem for small knapsacks

Then medium backpacks

And finally large 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:

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**

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?

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**

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 $\mathbf{w_i}$, the best value I could achieve is the value of item **i**, $\mathbf{v_i}$, *plus the optimal value for a smaller knapsack* that has capacity $\mathbf{x - w_i}$ (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**

Why does ... n for K[x]?

Basica... ilities:

My kna... r now?

Well, if I p... , *plus the*

optimal valu... ut item **i** in).

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] + [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

STEP 1: OPTIMAL SUBSTRUCTURE

SUBPROBLEMS:

Unbounded Knapsack with a smaller knapsack

Wh   x]?

Alternative interpretation:

Suppose K[x] is an optimal solution for a knapsack with capacity x, and suppose it contains one or more of the i^{th} item.

The remaining items in the knapsack must have a total weight of at most $x-w_i$, and the remaining items must also form an optimal solution!

(If not, then we could have replaced those items with a more valuable set of items and increase K[x])

opti   **i** in).

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:

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

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

STEP 2: RECURSIVE FORMULATION

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

Our recursive formulation:

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

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
    K[0] = 0
    for x = 1,...,W:
        K[x] = 0
        for i = 1,...,n:
            if w_i ≤ x:
                K[x] = max{ K[x], K[x-w_i] + v_i }
    return 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
    K[0] = 0
    for x = 1,...,W:
        K[x] = 0
        for i = 1,...,n:
            if w_i ≤ x:
                K[x] = max{ K[x], K[x-w_i] + v_i }
    return K[W]
```

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

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
    K[0] = 0
    for x = 1,...,W:
        K[x] = 0
        for i = 1,...,n:
            if w_i ≤ x:
                K[x] = max{ K[x], K[x-w_i] + v_i }
    return K[W]
```

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

Iterate over each knapsack size from smallest to largest

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

سوال؟

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
    K[0] = 0
    for x = 1,...,W:
        K[x] = 0
        for i = 1,...,n:
            if w_i ≤ x:
                K[x] = max{ K[x], K[x-w_i] + v_i }
    return K[W]
```

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

Iterate over each knapsack size from smallest to largest

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

Runtime: O(nW)

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

HOW GOOD IS O(nW)?

We'd like our runtime to scale "nicely" with our input size, which is dependent on the capacity of our knapsack, W.

Our input size is actually O(nlogW), because it takes logW bits to write down W, and it takes nlogW bits to write down all n weights (we assume the values of each item are not the dominating factor here).

HOW GOOD IS O(nW)?

We'd like our runtime to scale "nicely" with our input size, which is dependent on the capacity of our knapsack, W.

Our input size is actually O(nlogW), because it takes logW bits to write down W, and it takes nlogW bits to write down all n weights (we assume the values of each item are not the dominating factor here).

Thus, O(nW) is not actually polynomial in the input size. We call these algorithms "pseudo-polynomial".

Finding a polynomial time algorithm for Knapsack is an open problem! This problem is NP-hard.

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 4: FIND ACTUAL ITEMS

```
UNBOUNDED_KNAPSACK(W, n, weights, values):
    Initialize a size W+1 array, K
    K[0] = 0
    for x = 1,...,W:
        K[x] = 0
        for i = 1,...,n:
            if w_i ≤ x:
                K[x] = max{ K[x], K[x-w_i] + v_i }
    return K[W]
```

Suppose we want the actual set of items to use.

How can we augment our pseudocode to track which items we should include?

STEP 4: FIND ACTUAL ITEMS

```
UNBOUNDED_KNAPSACK(W, n, weights, values):
   Initialize a size W+1 array, K
   K[0] = 0
   for x = 1,...,W:
      K[x] = 0
      for i = 1,...,n:
         if w_i ≤ x:
            K[x] = max{ K[x], K[x-w_i] + v_i }
   return K[W]
```

Suppose we want the actual set of items to use.

How can we augment our pseudocode to track which items we should include?

For each knapsack size, 0 through W, we'll store the exact set of items that would provide the best value for that knapsack capacity.

STEP 4: FIND ACTUAL ITEMS

```
UNBOUNDED_KNAPSACK_ITEMS(W, n, weights, values):
    Initialize size W+1 arrays, K and ITEMS
    K[0] = 0, ITEMS[0] = { }
    for x = 1,...,W:
        K[x] = 0, ITEMS[x] = { }
        for i = 1,...,n:
            if w_i ≤ 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]
```

EXAMPLE

	0	1	2	3	4
K	0				
ITEMS					

```
UNBOUNDED_KNAPSACK_ITEMS(W, n, weights, values):
    Initialize size W+1 arrays, K and ITEMS
    K[0] = 0, ITEMS[0] = { }
    for x = 1,...,W:
        K[x] = 0, ITEMS[x] = { }
        for i = 1,...,n:
            if w_i ≤ 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]
```

Capacity: **4**

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

EXAMPLE

	0	1	2	3	4
K	0	0			
ITEMS					

```
UNBOUNDED_KNAPSACK_ITEMS(W, n, weights, values):
    Initialize size W+1 arrays, K and ITEMS
    K[0] = 0, ITEMS[0] = { }
    for x = 1,...,W:
        K[x] = 0, ITEMS[x] = { }
        for i = 1,...,n:
            if w_i ≤ 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]
```

Capacity: **4**

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

EXAMPLE

	0	1	2	3	4
K	0	0			
ITEMS					

$K[1] = \max\{\ K[1],\ K[0] + 1\ \}$

ITEMS[1] = ITEMS[0] + 

```
UNBOUNDED_KNAPSACK_ITEMS(W, n, weights, values):
    Initialize size W+1 arrays, K and ITEMS
    K[0] = 0, ITEMS[0] = { }
    for x = 1,...,W:
        K[x] = 0, ITEMS[x] = { }
        for i = 1,...,n:
            if w_i ≤ 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]
```

Capacity: **4**

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

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,...,W:
        K[x] = 0, ITEMS[x] = { }
        for i = 1,...,n:
            if w_i ≤ 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]
```

Capacity: **4**

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

EXAMPLE

	0	1	2	3	4
K	0	1	0		
ITEMS					

```
UNBOUNDED_KNAPSACK_ITEMS(W, n, weights, values):
   Initialize size W+1 arrays, K and ITEMS
   K[0] = 0, ITEMS[0] = { }
   for x = 1,...,W:
      K[x] = 0, ITEMS[x] = { }
      for i = 1,...,n:
         if w_i ≤ 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]
```

Capacity: **4**

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

EXAMPLE

	0	1	2	3	4
K	0	1	0		
ITEMS					

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

ITEMS[2] = ITEMS[1] + 

```
UNBOUNDED_KNAPSACK_ITEMS(W, n, weights, values):
    Initialize size W+1 arrays, K and ITEMS
    K[0] = 0, ITEMS[0] = { }
    for x = 1,...,W:
        K[x] = 0, ITEMS[x] = { }
        for i = 1,...,n:
            if w_i ≤ 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]
```

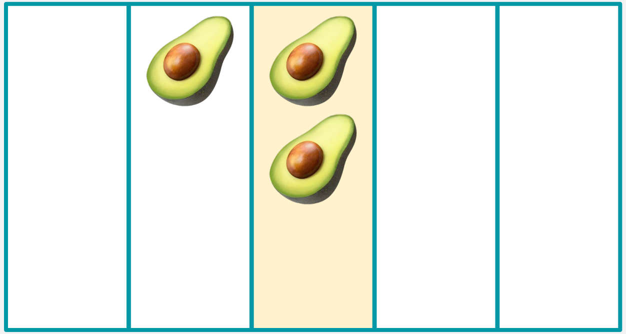
Capacity: **4**

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

EXAMPLE

	0	1	2	3	4
K	**0**	**1**	**2**		

ITEMS



```
UNBOUNDED_KNAPSACK_ITEMS(W, n, weights, values):
    Initialize size W+1 arrays, K and ITEMS
    K[0] = 0, ITEMS[0] = { }
    for x = 1,...,W:
        K[x] = 0, ITEMS[x] = { }
        for i = 1,...,n:
            if w_i ≤ 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[2] = max{ K[2], K[1] + 1 }

ITEMS[2] = ITEMS[1] + 

update!

Capacity: **4**

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

EXAMPLE

	0	1	2	3	4
K	0	1	2		
ITEMS			 		

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

ITEMS[2] = ITEMS[0] + 

```
UNBOUNDED_KNAPSACK_ITEMS(W, n, weights, values):
    Initialize size W+1 arrays, K and ITEMS
    K[0] = 0, ITEMS[0] = { }
    for x = 1,...,W:
        K[x] = 0, ITEMS[x] = { }
        for i = 1,...,n:
            if w_i ≤ 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]
```

Capacity: **4**

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

EXAMPLE

	0	1	2	3	4
K	**0**	**1**	**4**		
ITEMS					

```
UNBOUNDED_KNAPSACK_ITEMS(W, n, weights, values):
    Initialize size W+1 arrays, K and ITEMS
    K[0] = 0, ITEMS[0] = { }
    for x = 1,...,W:
        K[x] = 0, ITEMS[x] = { }
        for i = 1,...,n:
            if w_i ≤ 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[2] = max{ K[2], K[0] + 4 }

ITEMS[2] = ITEMS[0] + 

update!

Capacity: **4**

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

EXAMPLE

	0	1	2	3	4
K	0	1	4	0	
ITEMS					

```
UNBOUNDED_KNAPSACK_ITEMS(W, n, weights, values):
    Initialize size W+1 arrays, K and ITEMS
    K[0] = 0, ITEMS[0] = { }
    for x = 1,...,W:
        K[x] = 0, ITEMS[x] = { }
        for i = 1,...,n:
            if w_i ≤ 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]
```

Capacity: **4**

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

EXAMPLE

	0	1	2	3	4
K	0	1	4	0	
ITEMS					

K[3] = max{ K[3], K[2] + 1 }

ITEMS[3] = ITEMS[2] + 

```
UNBOUNDED_KNAPSACK_ITEMS(W, n, weights, values):
   Initialize size W+1 arrays, K and ITEMS
   K[0] = 0, ITEMS[0] = { }
   for x = 1,...,W:
      K[x] = 0, ITEMS[x] = { }
      for i = 1,...,n:
         if w_i ≤ 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]
```

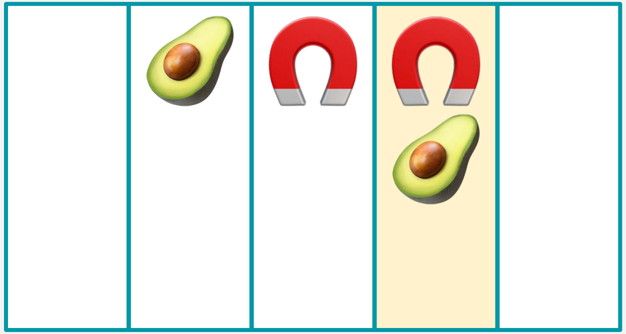
Capacity: **4**

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

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!

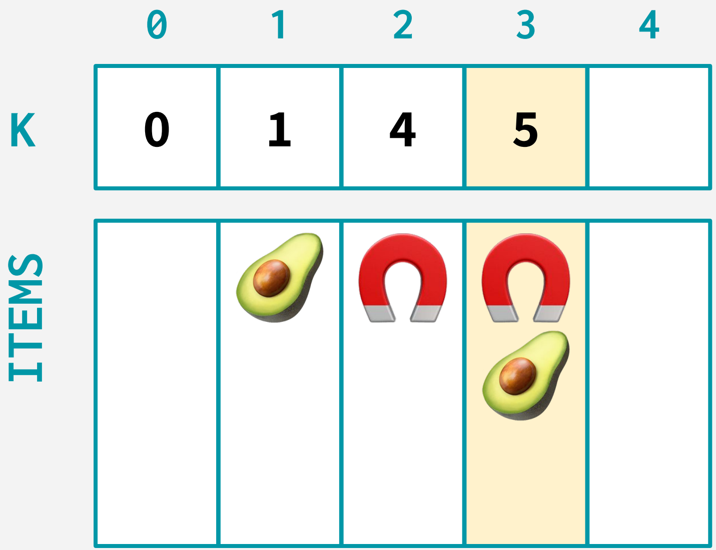
```
UNBOUNDED_KNAPSACK_ITEMS(W, n, weights, values):
    Initialize size W+1 arrays, K and ITEMS
    K[0] = 0, ITEMS[0] = { }
    for x = 1,...,W:
        K[x] = 0, ITEMS[x] = { }
        for i = 1,...,n:
            if w_i ≤ x:
                K[x] = max{ K[x], K[x-w_i] + v_i }
                if K[x] was updated:
                    ITEMS[x] = ITEMS[x-w_i] ∪ {item i}
    return ITEMS[W]
```

Capacity: **4**

Item: 

Weight:	1	2	3
Value:	1	4	6

EXAMPLE




```
UNBOUNDED_KNAPSACK_ITEMS(W, n, weights, values):
    Initialize size W+1 arrays, K and ITEMS
    K[0] = 0, ITEMS[0] = { }
    for x = 1,...,W:
        K[x] = 0, ITEMS[x] = { }
        for i = 1,...,n:
            if w_i ≤ 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]
```

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

ITEMS[3] = ITEMS[1] + 

(magnet doesn't cause update)

Capacity: **4**

Item: 

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

EXAMPLE

	0	1	2	3	4
K	0	1	4	5	
ITEMS		🥑	🧲	🧲 🥑	

```
UNBOUNDED_KNAPSACK_ITEMS(W, n, weights, values):
    Initialize size W+1 arrays, K and ITEMS
    K[0] = 0, ITEMS[0] = { }
    for x = 1,...,W:
        K[x] = 0, ITEMS[x] = { }
        for i = 1,...,n:
            if w_i ≤ 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[3] = max{ K[3], K[0] + 6 }

ITEMS[3] = ITEMS[0] + 🐨

Capacity: **4**

Item:	🥑	🧲	🐨
Weight:	1	2	3
Value:	1	4	6

EXAMPLE

	0	1	2	3	4
K	**0**	**1**	**4**	**6**	
ITEMS		🥑	🧲	🐨	

```
UNBOUNDED_KNAPSACK_ITEMS(W, n, weights, values):
    Initialize size W+1 arrays, K and ITEMS
    K[0] = 0, ITEMS[0] = { }
    for x = 1,...,W:
        K[x] = 0, ITEMS[x] = { }
        for i = 1,...,n:
            if w_i ≤ 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[3] = max{ K[3], K[0] + 6 }

ITEMS[3] = ITEMS[0] + 🐨

update!

Capacity: **4**

Item:	🥑	🧲	🐨
Weight:	**1**	**2**	**3**
Value:	**1**	**4**	**6**

EXAMPLE

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

```
UNBOUNDED_KNAPSACK_ITEMS(W, n, weights, values):
   Initialize size W+1 arrays, K and ITEMS
   K[0] = 0, ITEMS[0] = { }
   for x = 1,...,W:
      K[x] = 0, ITEMS[x] = { }
      for i = 1,...,n:
         if w_i ≤ 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]
```

Capacity: **4**

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

EXAMPLE

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

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

ITEMS[4] = ITEMS[3] + 

```
UNBOUNDED_KNAPSACK_ITEMS(W, n, weights, values):
    Initialize size W+1 arrays, K and ITEMS
    K[0] = 0, ITEMS[0] = { }
    for x = 1,...,W:
        K[x] = 0, ITEMS[x] = { }
        for i = 1,...,n:
            if w_i ≤ 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]
```

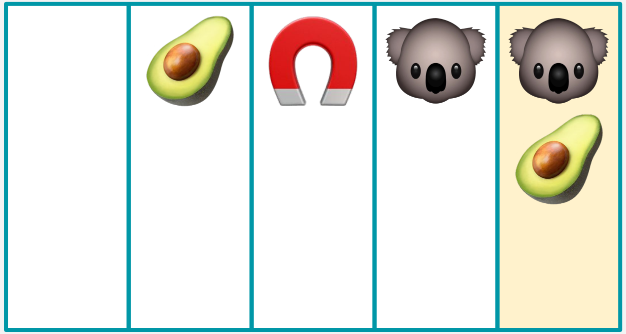
Capacity: **4**

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

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,...,W:
        K[x] = 0, ITEMS[x] = { }
        for i = 1,...,n:
            if w_i ≤ 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]
```

Capacity: **4**

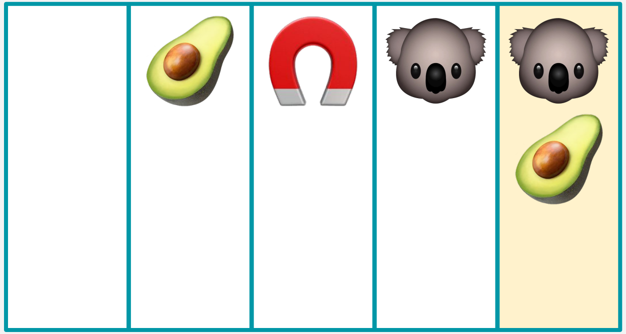
Item: 

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

EXAMPLE

	0	1	2	3	4
K	0	1	4	6	7

ITEMS



```
UNBOUNDED_KNAPSACK_ITEMS(W, n, weights, values):
    Initialize size W+1 arrays, K and ITEMS
    K[0] = 0, ITEMS[0] = { }
    for x = 1,...,W:
        K[x] = 0, ITEMS[x] = { }
        for i = 1,...,n:
            if w_i ≤ 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[4] = max{ K[4], K[2] + 4 }

ITEMS[4] = ITEMS[2] + 

Capacity: **4**

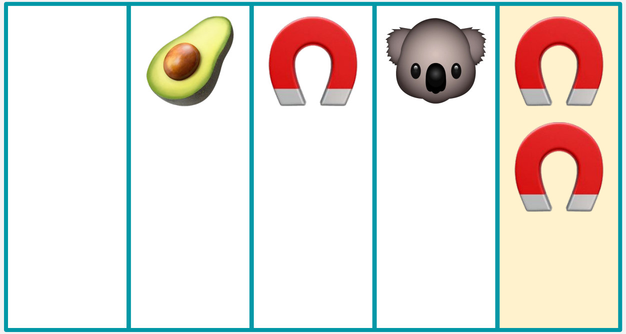
Item: 

Weight:	1	2	3
Value:	1	4	6

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!

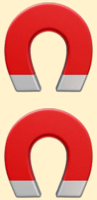
```
UNBOUNDED_KNAPSACK_ITEMS(W, n, weights, values):
   Initialize size W+1 arrays, K and ITEMS
   K[0] = 0, ITEMS[0] = { }
   for x = 1,...,W:
      K[x] = 0, ITEMS[x] = { }
      for i = 1,...,n:
         if w_i ≤ x:
            K[x] = max{ K[x], K[x-w_i] + v_i }
            if K[x] was updated:
               ITEMS[x] = ITEMS[x-w_i] ∪ {item i}
   return ITEMS[W]
```

Capacity: **4**

Item: 

Item			
Weight:	1	2	3
Value:	1	4	6

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,...,W:
        K[x] = 0, ITEMS[x] = { }
        for i = 1,...,n:
            if w_i ≤ 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]
```

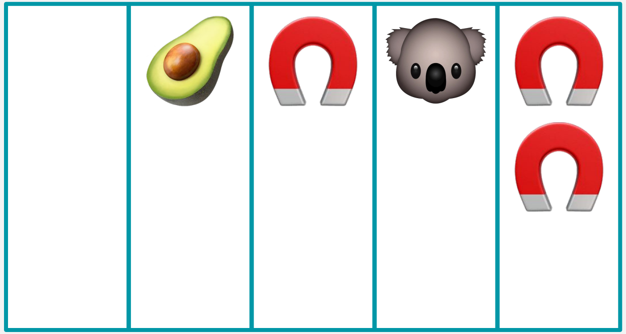
Capacity: **4**

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

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,...,W:
        K[x] = 0, ITEMS[x] = { }
        for i = 1,...,n:
            if w_i ≤ 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]
```

Capacity: **4**

Item:

Weight:	1	2	3
Value:	1	4	6

سوال؟

مسئله کوله پشتی 0/1

وقتی که از هر شی تنها یکی وجود دارد

KNAPSACK PROBLEM: TWO VERSIONS

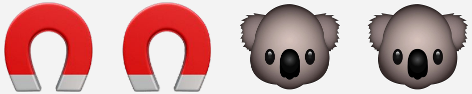
Capacity: **10**

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

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 (ATTEMPT #1):

0/1 Knapsack with a smaller knapsack

STEP 1: OPTIMAL SUBSTRUCTURE

SUBPROBLEMS (ATTEMPT #1):

0/1 Knapsack with a smaller knapsack

This doesn’t quite work. We are only allowed one copy of each item

The subproblem needs information about which items have already been used.

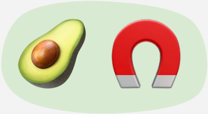
STEP 1: OPTIMAL SUBSTRUCTURE

SUBPROBLEM (ATTEMPT #2):

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

First solve the problem for a few items

Then more items



Then even more items

Solve using small knapsacks

Then medium knapsacks

And finally large knapsacks

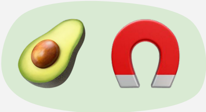
STEP 1: OPTIMAL SUBSTRUCTURE

SUBPROBLEM (ATTEMPT #2):

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

First solve the problem for a few items

Then more items



Then even more items

Solve using small knapsacks

Then medium knapsacks

And finally large knapsacks

This calls for a two-dimensional table!

STEP 1: OPTIMAL SUBSTRUCTURE

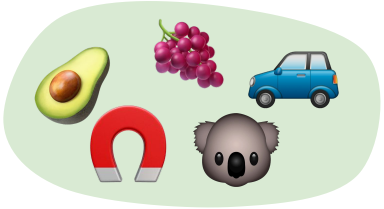
SUBPROBLEM (ATTEMPT #2):

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

Our subproblems will be indexed by x and j:

K[x, j] = optimal solution for a knapsack of size x using only the first j items

Capacity x



First j items

First sol
for

Ther

T
mor

STEP 1: OPTIMAL SUBSTRUCTURE

SUBPROBLEM (ATTEMPT #2):

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

What is the intuition behind how I'd use these subproblems?

Again, I'm making choices. But instead of deciding which of my unlimited items I'd like to add next to my knapsack, I have a binary choice of whether to include item j or not.

If I include it, then I should look to a smaller knapsack with fewer items.

If I don't include it, then I should look to the same knapsack with fewer items.

Those will be my 2 cases!

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

Our recursive formulation:

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

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

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:

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

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

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

```
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
    K[0,j] = 0 for all j = 0,...,n
    for x = 1,...,W:
        for j = 1,...,n:
            K[x,j] = K[x,j-1]
            if w_j ≤ x:
                K[x,j] = max{ K[x,j], K[x-w_j, j-1] + v_j }
    return K[W,n]
```

STEP 3: WRITE A DP ALGORITHM

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

```
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
    K[0,j] = 0 for all j = 0,...,n
    for x = 1,...,W:
        for j = 1,...,n:
            K[x,j] = K[x,j-1]
            if w_j ≤ x:
                K[x,j] = max{ K[x,j], K[x-w_j, j-1] + v_j }
    return K[W,n]
```

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

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
    K[0,j] = 0 for all j = 0,...,n
    for x = 1,...,W:
        for j = 1,...,n:
            K[x,j] = K[x,j-1]
            if w_j ≤ x:
                K[x,j] = max{ K[x,j], K[x-w_j, j-1] + v_j }
    return K[W,n]
```

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

Iterate over knapsack sizes from smallest to largest

Iterate over items we can consider

STEP 3: WRITE A DP ALGORITHM

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

```
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
   K[0,j] = 0 for all j = 0,...,n
   for x = 1,...,W:
      for j = 1,...,n:
         K[x,j] = K[x,j-1]
         if w_j ≤ x:
            K[x,j] = max{ K[x,j], K[x-w_j, j-1] + v_j }
   return K[W,n]
```

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

Iterate over knapsack sizes from smallest to largest

Iterate over items we can consider

Default case: we don't use item j

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
   K[0,j] = 0 for all j = 0,...,n
   for x = 1,...,W:
      for j = 1,...,n:
         K[x,j] = K[x,j-1]
         if w_j ≤ x:
            K[x,j] = max{ K[x,j], K[x-w_j, j-1] + v_j }
   return K[W,n]
```

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

Iterate over knapsack sizes from smallest to largest

Iterate over items we can consider

Default case: we don't use item j

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

STEP 3: WRITE A DP ALGORITHM

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

```
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
   K[0,j] = 0 for all j = 0,...,n
   for x = 1,...,W:
      for j = 1,...,n:
         K[x,j] = K[x,j-1]
         if wj ≤ x:
            K[x,j] = max{ K[x,j], K[x-wj, j-1] + vj }
   return K[W,n]
```

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

Iterate over knapsack sizes from smallest to largest

Iterate over items we can consider

Default case: we don't use item j

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

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) x (W+1) table, K
    K[x,0] = 0 for all x = 0,...,W
    K[0,j] = 0 for all j = 0,...,n
    for x = 1,...,W:
        for j = 1,...,n:
            K[x,j] = K[x,j-1]
            if w_j ≤ 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	0		
🧲🥑 j=2	0			
🐨🧲🥑 j=3	0			

Default value = **K[x, j–1]**

If 🥑 can fit, would it help?

```
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
    K[0,j] = 0 for all j = 0,...,n
    for x = 1,...,W:
        for j = 1,...,n:
            K[x,j] = K[x,j-1]
            if w_j ≤ 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 🥑		
🧲🥑 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) x (W+1) table, K
    K[x,0] = 0 for all x = 0,...,W
    K[0,j] = 0 for all j = 0,...,n
    for x = 1,...,W:
        for j = 1,...,n:
            K[x,j] = K[x,j-1]
            if w_j ≤ 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 🥑		
🧲🥑 j=2	0	1 🥑		
🐨🧲🥑 j=3	0			

Default value = **K[x, j–1]**

If 🧲 can fit, would it help?

```
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
    K[0,j] = 0 for all j = 0,...,n
    for x = 1,...,W:
        for j = 1,...,n:
            K[x,j] = K[x,j-1]
            if w_j ≤ 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 🥑		
🧲🥑 j=2	0	1 🥑		
🐨🧲🥑 j=3	0			

Default value = **K[x, j–1]**

If 🧲 can fit, would it help?

It can't fit.

```
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
    K[0,j] = 0 for all j = 0,...,n
    for x = 1,...,W:
        for j = 1,...,n:
            K[x,j] = K[x,j-1]
            if w_j ≤ 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 🥑		
🧲 🥑 j=2	0	1 🥑		
🐨 🧲 🥑 j=3	0	1 🥑		

Default value = **K[x, j–1]**

If 🐨 can fit, would it help?

```
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
    K[0,j] = 0 for all j = 0,...,n
    for x = 1,...,W:
        for j = 1,...,n:
            K[x,j] = K[x,j-1]
            if w_j ≤ 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**		
j=2	**0**	**1**		
j=3	**0**	**1**		

Default value = **K[x, j–1]**

If can fit, would it help?

It can't fit.

```
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
   K[0,j] = 0 for all j = 0,...,n
   for x = 1,...,W:
      for j = 1,...,n:
         K[x,j] = K[x,j-1]
         if w_j ≤ 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 🥑	0	
🧲🥑	j=2	0	1 🥑		
🐨🧲🥑	j=3	0	1 🥑		

Default value = **K[x, j−1]**

If 🥑 can fit, would it help?

```
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
    K[0,j] = 0 for all j = 0,...,n
    for x = 1,...,W:
        for j = 1,...,n:
            K[x,j] = K[x,j-1]
            if w_j ≤ 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** 🥑	
🧲🥑 j=2	**0**	**1** 🥑		
🐨🧲🥑 j=3	**0**	**1** 🥑		

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) x (W+1) table, K
    K[x,0] = 0 for all x = 0,...,W
    K[0,j] = 0 for all j = 0,...,n
    for x = 1,...,W:
        for j = 1,...,n:
            K[x,j] = K[x,j-1]
            if w_j ≤ 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 🥑	
🧲🥑 j=2	0	1 🥑	1 🥑	
🐨🧲🥑 j=3	0	1 🥑		

Default value = **K[x, j–1]**
If 🧲 can fit, would it help?

```
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
    K[0,j] = 0 for all j = 0,...,n
    for x = 1,...,W:
        for j = 1,...,n:
            K[x,j] = K[x,j-1]
            if w_j ≤ 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 🥑	
🧲 🥑	j=2	0	1 🥑	4 🧲	
🐨 🧲 🥑	j=3	0	1 🥑		

Default value = **K[x, j–1]**

If 🧲 can fit, would it help?

YES! **0 + 4** is better!

```
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
    K[0,j] = 0 for all j = 0,...,n
    for x = 1,...,W:
        for j = 1,...,n:
            K[x,j] = K[x,j-1]
            if w_j ≤ 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	
j=2	0	1	4	
j=3	0	1	4	

Default value = **K[x, j–1]**

If can fit, would it help?

```
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
    K[0,j] = 0 for all j = 0,...,n
    for x = 1,...,W:
        for j = 1,...,n:
            K[x,j] = K[x,j-1]
            if w_j ≤ 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 🥑	
🧲 🥑 j=2	0	1 🥑	4 🧲	
🐨 🧲 🥑 j=3	0	1 🥑	4 🧲	

Default value = **K[x, j–1]**

If 🐨 can fit, would it help?

No, it can't fit.

```
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
    K[0,j] = 0 for all j = 0,...,n
    for x = 1,...,W:
        for j = 1,...,n:
            K[x,j] = K[x,j-1]
            if w_j ≤ 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 🥑	0
🧲🥑 j=2	0	1 🥑	4 🧲	
🐨🧲🥑 j=3	0	1 🥑	4 🧲	

Default value = **K[x, j–1]**

If 🥑 can fit, would it help?

```
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
    K[0,j] = 0 for all j = 0,...,n
    for x = 1,...,W:
        for j = 1,...,n:
            K[x,j] = K[x,j-1]
            if w_j ≤ 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 🧲	
🐨🧲🥑	j=3	0	1 🥑	4 🧲	

```
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
    K[0,j] = 0 for all j = 0,...,n
    for x = 1,...,W:
        for j = 1,...,n:
            K[x,j] = K[x,j-1]
            if w_j ≤ x:
                K[x,j] = max{ K[x,j], K[x-w_j,j-1] + v_j }
    return K[W,n]
```

Default value = **K[x, j–1]**

If 🥑 can fit, would it help?

YES! **0 + 1** is better!

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 🧲	1 🥑
🐨 🧲 🥑	j=3	0	1 🥑	4 🧲	

Default value = **K[x, j–1]**

If 🧲 can fit, would it help?

```
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
    K[0,j] = 0 for all j = 0,...,n
    for x = 1,...,W:
        for j = 1,...,n:
            K[x,j] = K[x,j-1]
            if w_j ≤ 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) x (W+1) table, K
    K[x,0] = 0 for all x = 0,...,W
    K[0,j] = 0 for all j = 0,...,n
    for x = 1,...,W:
        for j = 1,...,n:
            K[x,j] = K[x,j-1]
            if w_j ≤ 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 🧲	5 🥑🧲

Default value = **K[x, j–1]**

If 🐨 can fit, would it help?

```
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
    K[0,j] = 0 for all j = 0,...,n
    for x = 1,...,W:
        for j = 1,...,n:
            K[x,j] = K[x,j-1]
            if w_j ≤ 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 🧲	6 🐨

```
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
    K[0,j] = 0 for all j = 0,...,n
    for x = 1,...,W:
        for j = 1,...,n:
            K[x,j] = K[x,j-1]
            if w_j ≤ x:
                K[x,j] = max{ K[x,j], K[x-w_j,j-1] + v_j }
    return K[W,n]
```

Default value = **K[x, j–1]**

If 🐨 can fit, would it help?

YES! **0 + 6** is better!

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 🧲	6 🐨

And we're done! So the optimal solution here is to put one 🐨 in our knapsack for a value of 6!

```
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
    K[0,j] = 0 for all j = 0,...,n
    for x = 1,...,W:
        for j = 1,...,n:
            K[x,j] = K[x,j-1]
            if w_j ≤ 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**

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

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 5 to make this happen.

KNAPSACK PROBLEM: TWO VERSIONS

Capacity: **10**

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

UNBOUNDED KNAPSACK

We have infinite copies of all the items.

What's the most valuable way to fill the knapsack?

Subproblems: answers to smaller knapsack capacities!

(our cache was a 1D array)

0/1 KNAPSACK

We have only one copy of each item.

What's the most valuable way to fill the knapsack?

Subproblems: answers to smaller knapsack capacities & smaller item sets!

(our cache was a 2D table)

سوال؟