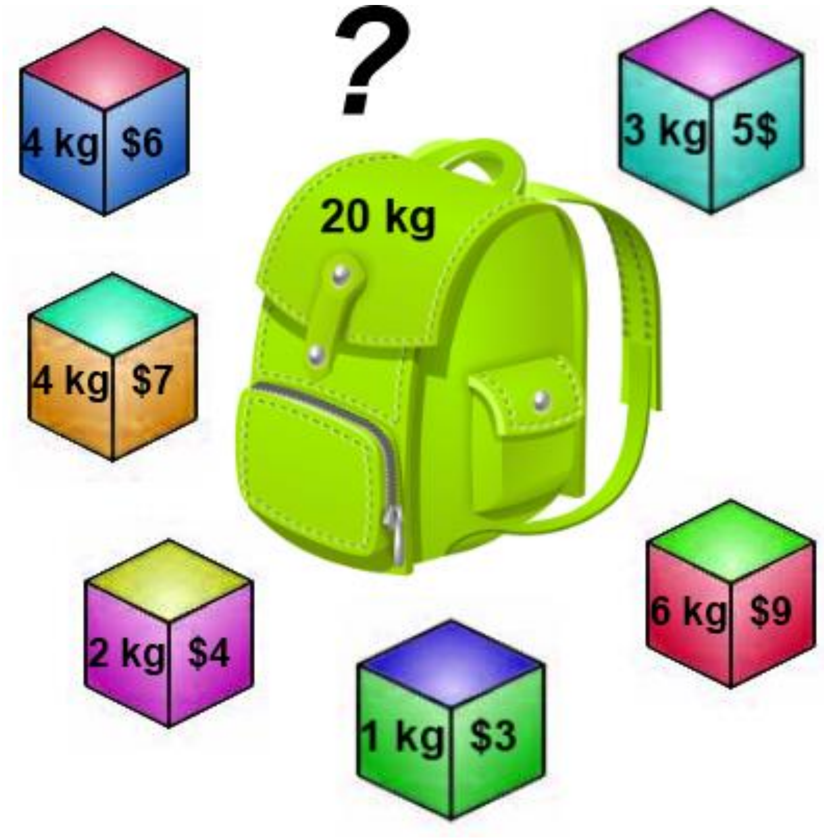
Dynamic Programming

0-1 Knapsack Problem

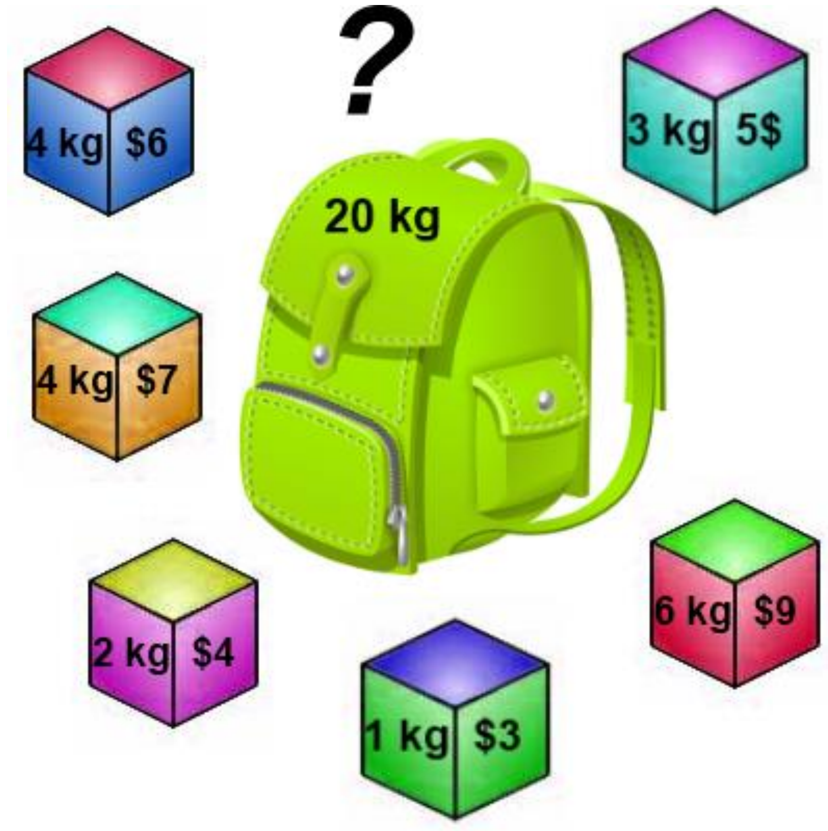
Knapsack 0-1 Problem

- The goal is to **maximize the value of a knapsack** that can hold at most W units (i.e. lbs or kg) worth of goods from a list of items I_0, I_1, … I_{n-1}.
  - Each item has 2 attributes:
    1) Value – let this be v_i for item I_i
    2) Weight – let this be w_i for item I_i



Knapsack 0-1 Problem

- The difference between this problem and the fractional knapsack one is that you CANNOT take a fraction of an item.
  - You can either take it or not.
  - Hence the name Knapsack 0-1 problem.



Knapsack 0-1 Problem

- Brute Force
  - The naïve way to solve this problem is to cycle through all 2^n subsets of the n items and pick the subset with a legal weight that maximizes the value of the knapsack.
  - We can come up with a dynamic programming algorithm that will USUALLY do better than this brute force technique.

Knapsack 0-1 Problem

- As we did before we are going to solve the problem in terms of sub-problems.
  - So let's try to do that…

- Our first attempt might be to characterize a sub-problem as follows:
  - Let S_k be the optimal subset of elements from $\{I_0, I_1, \ldots, I_k\}$.
    - What we find is that the optimal subset from the elements $\{I_0, I_1, \ldots, I_{k+1}\}$ may not correspond to the optimal subset of elements from $\{I_0, I_1, \ldots, I_k\}$ in any regular pattern.

  - Basically, the solution to the optimization problem for S_{k+1} might NOT contain the optimal solution from problem S_k.

Knapsack 0-1 Problem

- Let’s illustrate that point with an example:

Item	Weight	Value
I_0	3	10
I_1	8	4
I_2	9	9
I_3	8	11

- **The maximum weight the knapsack can hold is 20.**

- The best set of items from $\{I_0, I_1, I_2\}$ is $\{I_0, I_1, I_2\}$
- BUT the best set of items from $\{I_0, I_1, I_2, I_3\}$ is $\{I_0, I_2, I_3\}$.
  - In this example, note that this optimal solution, $\{I_0, I_2, I_3\}$, does NOT build upon the previous optimal solution, $\{I_0, I_1, I_2\}$.
    - (Instead it build's upon the solution, $\{I_0, I_2\}$, which is really the optimal subset of $\{I_0, I_1, I_2\}$ with weight 12 or less.)

Knapsack 0-1 problem

- So now we must re-work the way we build upon previous sub-problems…
  - Let **B[k, w]** represent the maximum total value of a subset S_k with weight w.
  - Our goal is to find **B[n, W],** where n is the total number of items and W is the maximal weight the knapsack can carry.

- So our recursive formula for subproblems:

$$B[k, w] = \begin{cases} B[k-1, w], & \underline{\text{if } w_k > w} \\ \max\{B[k-1, w], B[k-1, w-w_k] + v_k\}, & \underline{\text{otherwise}} \end{cases}$$

- In English, this means that the best subset of S_k that has total weight w is:
  1) The best subset of S_{k-1} that has total weight w, or
  2) The best subset of S_{k-1} that has total weight w-w_k plus the item k

Knapsack 0-1 Problem – Recursive Formula

$$B[k,w] = \begin{cases} B[k-1,w] & \text{if } w_k > w \\ \max\{B[k-1,w], B[k-1,w-w_k]+b_k\} & \text{else} \end{cases}$$

- The best subset of S_k that has the total weight w, either contains item k or not.

- **First case:** $w_k > w$
  - Item *k* can't be part of the solution! If it was the total weight would be > w, which is unacceptable.

- **Second case:** $w_k \leq w$
  - Then the item *k* can be in the solution, and we choose the case with greater value.

Knapsack 0-1 Algorithm

```
for w = 0 to W {   // Initialize 1st row to 0's
  B[0,w] = 0
}
for i = 1 to n {   // Initialize 1st column to 0's
  B[i,0] = 0
}
for i = 1 to n {
  for w = 0 to W {
      if $w_i$ <= w {   //item i can be in the solution
            if $v_i$ + B[i-1,w-$w_i$] > B[i-1,w]
                  B[i,w] = $v_i$ + B[i-1,w- $w_i$]
            else
                  B[i,w] = B[i-1,w]
      }
      else B[i,w] = B[i-1,w] // $w_i$ > w
  }
}
```

Knapsack 0-1 Problem

- Let’s run our algorithm on the following data:
  - n = 4 (# of elements)
  - W = 5 (max weight)
  - Elements (weight, value):

    (2,3), (3,4), (4,5), (5,6)

Knapsack 0-1 Example

i / w	**0**	**1**	**2**	**3**	**4**	**5**
0	0	0	0	0	0	0
1	0					
2	0					
3	0					
4	0					

```
// Initialize the base cases
for w = 0 to W
        B[0,w] = 0

for i = 1 to n
        B[i,0] = 0
```

Knapsack 0-1 Example

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0				
2	0					
3	0					
4	0					

$i = 1$

$v_i = 3$

$w_i = 2$

$w = 1$

$w - w_i = -1$

```
if  $w_i <= w$   //item i can be in the solution
        if $v_i + B[i-1,w-w_i] > B[i-1,w]$
            $B[i,w] = v_i + B[i-1,w- w_i]$
        else
            $B[i,w] = B[i-1,w]$
else **B[i,w] = B[i-1,w]** // $w_i > w$
```

Knapsack 0-1 Example

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3			
2	0					
3	0					
4	0					

$i = 1$

$v_i = 3$

$w_i = 2$

$w = 2$

$w - w_i = 0$

```
if  $w_i <= w$   //item i can be in the solution
        if $v_i + B[i-1,w-w_i] > B[i-1,w]$
            **$B[i,w] = v_i + B[i-1,w- w_i]$**
        else
            $B[i,w] = B[i-1,w]$
else $B[i,w] = B[i-1,w]$ // $w_i > w$
```

Knapsack 0-1 Example

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i / w	**0**	**1**	**2**	**3**	**4**	**5**
0	0	0	0	0	0	0
1	0	0	3	3		
2	0					
3	0					
4	0					

$i = 1$
$v_i = 3$
$w_i = 2$
$\mathbf{w = 3}$
$w\text{-}w_i = 1$

```
if  $w_i <= w$   //item i can be in the solution
        if $v_i + B[i-1,w-w_i] > B[i-1,w]$
            $B[i,w] = v_i + B[i-1,w- w_i]$
        else
            $B[i,w] = B[i-1,w]$
else $B[i,w] = B[i-1,w]$ // $w_i > w$
```

Knapsack 0-1 Example

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	
2	0					
3	0					
4	0					

$i = 1$

$v_i = 3$

$w_i = 2$

$\mathbf{w = 4}$

$w\text{-}w_i = 2$

```
if  $w_i <= w$   //item i can be in the solution
    if $v_i + B[i-1,w-w_i] > B[i-1,w]$
        **$B[i,w] = v_i + B[i-1,w- w_i]$**
    else
        $B[i,w] = B[i-1,w]$
else $B[i,w] = B[i-1,w]$ // $w_i > w$
```

Knapsack 0-1 Example

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

i / w	**0**	**1**	**2**	**3**	**4**	**5**
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0					
3	0					
4	0					

$i = 1$
$v_i = 3$
$w_i = 2$
$\mathbf{w = 5}$
$w\text{-}w_i = 3$

```
if  $w_i$ <= w   //item i can be in the solution
        if $v_i$ + B[i-1,w-$w_i$] > B[i-1,w]
            **B[i,w] = $v_i$ + B[i-1,w- $w_i$]**
        else
            B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w] // $w_i$ > w
```

Knapsack 0-1 Example

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0				
3	0					
4	0					

$i = 2$
$v_i = 4$
$w_i = 3$
$w = 1$
$w - w_i = -2$

```
if  $w_i <= w$   //item i can be in the solution
        if $v_i + B[i-1,w-w_i] > B[i-1,w]$
            $B[i,w] = v_i + B[i-1,w- w_i]$
        else
            $B[i,w] = B[i-1,w]$
else $B[i,w] = B[i-1,w]$ // $w_i > w$
```

Knapsack 0-1 Example

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	$\downarrow$ 3			
3	0					
4	0					

$i = 2$

$v_i = 4$

$w_i = 3$

$w = 2$

$w\text{-}w_i = -1$

```
if  $w_i <= w$   //item i can be in the solution
        if $v_i + B[i-1,w-w_i] > B[i-1,w]$
            $B[i,w] = v_i + B[i-1,w- w_i]$
        else
            $B[i,w] = B[i-1,w]$
else $B[i,w] = B[i-1,w]$ // $w_i > w$
```

Knapsack 0-1 Example

Items:

1: (2,3)

$\boxed{2: (3,4)}$

3: (4,5)

4: (5,6)

i / w	**0**	**1**	**2**	**3**	**4**	**5**
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4		
3	0					
4	0					

$i = 2$

$v_i = 4$

$w_i = 3$

$\mathbf{w = 3}$

$w\text{-}w_i = 0$

```
if  $w_i <= w$   //item i can be in the solution
        if $v_i + B[i-1,w-w_i] > B[i-1,w]$
            $B[i,w] = v_i + B[i-1,w- w_i]$
        else
            $B[i,w] = B[i-1,w]$
else $B[i,w] = B[i-1,w]$ // $w_i > w$
```

Knapsack 0-1 Example

Items:

1: (2,3)

$\boxed{\text{2: (3,4)}}$

3: (4,5)

4: (5,6)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	
3	0					
4	0					

$i = 2$

$v_i = 4$

$w_i = 3$

$\mathbf{w = 4}$

$w\text{-}w_i = 1$

```
if  $w_i <= w$   //item i can be in the solution
        if $v_i + B[i-1,w-w_i] > B[i-1,w]$
            $\mathbf{B[i,w] = v_i + B[i-1,w- w_i]}$
        else
            $B[i,w] = B[i-1,w]$
else $B[i,w] = B[i-1,w]$ // $w_i > w$
```

Knapsack 0-1 Example

Items:

1: (2,3)

$\boxed{\text{2: (3,4)}}$

3: (4,5)

4: (5,6)

i / w	**0**	**1**	**2**	**3**	**4**	**5**
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0					
4	0					

$i = 2$

$v_i = 4$

$w_i = 3$

$\mathbf{w = 5}$

$w\text{-}w_i = 2$

```
if  $w_i <= w$   //item i can be in the solution
    if $v_i + B[i-1,w-w_i] > B[i-1,w]$
        $B[i,w] = v_i + B[i-1,w- w_i]$
    else
        $B[i,w] = B[i-1,w]$
else $B[i,w] = B[i-1,w]$ // $w_i > w$
```

Knapsack 0-1 Example

Items:

1: (2,3)

2: (3,4)

$\boxed{\text{3: (4,5)}}$

4: (5,6)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	↓ 0	↓ 3	↓ 4		
4	0					

$i = 3$

$v_i = 5$

$w_i = 4$

$\mathbf{w = 1..3}$

$w - w_i = -3..-1$

```
if  $w_i <= w$   //item i can be in the solution
        if $v_i + B[i-1,w-w_i] > B[i-1,w]$
            $B[i,w] = v_i + B[i-1,w- w_i]$
        else
            $B[i,w] = B[i-1,w]$
else **B[i,w] = B[i-1,w]** // $w_i > w$
```

Knapsack 0-1 Example

Items:

1: (2,3)

2: (3,4)

$\boxed{3: (4,5)}$

4: (5,6)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	
4	0					

$i = 3$
$v_i = 5$
$w_i = 4$
$\mathbf{w = 4}$
$w\text{-}w_i = 0$

```
if  $w_i$ <= w   //item i can be in the solution
        if $v_i$ + B[i-1,w-$w_i$] > B[i-1,w]
            **B[i,w] = $v_i$ + B[i-1,w- $w_i$]**
        else
            B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w] // $w_i$ > w
```

Knapsack 0-1 Example

Items:

1: (2,3)

2: (3,4)

$\boxed{3: (4,5)}$

4: (5,6)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	$\downarrow$ 7
4	0					

$i = 3$

$v_i = 5$

$w_i = 4$

$\mathbf{w = 5}$

$w\text{-}w_i = 1$

```
if  $w_i <= w$   //item i can be in the solution
    if $v_i + B[i-1,w-w_i] > B[i-1,w]$
        $B[i,w] = v_i + B[i-1,w- w_i]$
    else
        **B[i,w] = B[i-1,w]**
else B[i,w] = B[i-1,w] // $w_i > w$
```

Knapsack 0-1 Example

Items:

1: (2,3)

2: (3,4)

3: (4,5)

$\boxed{4: (5,6)}$

i / w	**0**	**1**	**2**	**3**	**4**	**5**
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	↓ 0	↓ 3	↓ 4	↓ 5	

$i = 4$

$v_i = 6$

$w_i = 5$

$\mathbf{w = 1..4}$

$w - w_i = -4..-1$

```
if  $w_i <= w$   //item i can be in the solution
        if $v_i + B[i-1,w-w_i] > B[i-1,w]$
            $B[i,w] = v_i + B[i-1,w- w_i]$
        else
            $B[i,w] = B[i-1,w]$
else **B[i,w] = B[i-1,w]** // $w_i > w$
```

Items:

1: (2,3)

2: (3,4)

3: (4,5)

$\boxed{\text{4: (5,6)}}$

Knapsack 0-1 Example

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	$\downarrow$ 7

$i = 4$

$v_i = 6$

$w_i = 5$

$\mathbf{w = 5}$

$w\text{-}w_i = 0$

```
if  $w_i <= w$   //item i can be in the solution
    if $v_i + B[i-1,w-w_i] > B[i-1,w]$
        $B[i,w] = v_i + B[i-1,w- w_i]$
    else
        **B[i,w] = B[i-1,w]**
else $B[i,w] = B[i-1,w]$ // $w_i > w$
```

Knapsack 0-1 Example

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

We're DONE!!

The max possible value that can be carried in this knapsack is ***$7***

Knapsack 0-1 Algorithm

- This algorithm only finds the max possible value that can be carried in the knapsack
    - The value in B[n,W]

- To know the ***items*** that make this maximum value, we need to trace back through the table.

Knapsack 0-1 Algorithm
Finding the Items

- Let i = n and k = W

```
if $B[i, k] \neq B[i-1, k]$ then
        mark the ith item as in the knapsack
        $i = i-1$, $k = k-w_i$
else
        $i = i-1$   // Assume the ith item is not in the knapsack
                    // Could it be in the optimally packed knapsack?
```

Knapsack 0-1 Algorithm Finding the Items

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

Knapsack:

i / w	**0**	**1**	**2**	**3**	**4**	**5**
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

$i = 4$
$k = 5$
$v_i = 6$
$w_i = 5$
B[i,k] = 7
B[i-1,k] = 7

```
i = n , k = W
while  i, k > 0
        if $B[i, k] \neq B[i-1, k]$ then
                mark the i^th item as in the knapsack
                $i = i-1, k = k-w_i$
        else
                $i = i-1$
```

Knapsack 0-1 Algorithm Finding the Items

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

Knapsack:

i / w	**0**	**1**	**2**	**3**	**4**	**5**
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

$i = 3$
$k = 5$
$v_i = 5$
$w_i = 4$
B[i,k] = 7
B[i-1,k] = 7

```
i = n , k = W
while  i, k > 0
        if $B[i, k] \neq B[i-1, k]$ then
                mark the i^th item as in the knapsack
                $i = i-1, k = k-w_i$
        else
                $i = i-1$
```

Knapsack 0-1 Algorithm Finding the Items

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

Knapsack:

Item 2

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

$i = 2$

$k = 5$

$v_i = 4$

$w_i = 3$

$B[i,k] = 7$

$B[i-1,k] = 3$

$k – w_i = 2$

```
i = n , k = W
while  i, k > 0
        if $B[i, k] \neq B[i-1, k]$ then
                mark the i^th item as in the knapsack
                $i = i-1, k = k-w_i$
        else
                $i = i-1$
```

Knapsack 0-1 Algorithm
Finding the Items

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

Knapsack:

Item 2
Item 1

i / w	**0**	**1**	**2**	**3**	**4**	**5**
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

$i = 1$
$k = 2$
$v_i = 3$
$w_i = 2$
$B[i,k] = 3$
$B[i-1,k] = 0$
$k - w_i = 0$

```
i = n , k = W
while  i, k > 0
        if $B[i, k] \neq B[i-1, k]$ then
                *mark the $i^{th}$ item as in the knapsack*
                $i = i-1, k = k-w_i$
        else
                $i = i-1$
```

Knapsack 0-1 Algorithm Finding the Items

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

Knapsack:

Item 2

Item 1

i / w	**0**	**1**	**2**	**3**	**4**	**5**
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

$i = 1$

$k = 2$

$v_i = 3$

$w_i = 2$

$B[i,k] = 3$

$B[i-1,k] = 0$

$k - w_i = 0$

k = 0, so we're DONE!

The optimal knapsack should contain:

Item 1 and Item 2

Knapsack 0-1 Problem – Run Time

```
for w = 0 to W          O(W)
  B[0,w] = 0

for i = 1 to n          O(n)
  B[i,0] = 0

for i = 1 to n          Repeat n times
  for w = 0 to W                O(W)
      < the rest of the code >
```

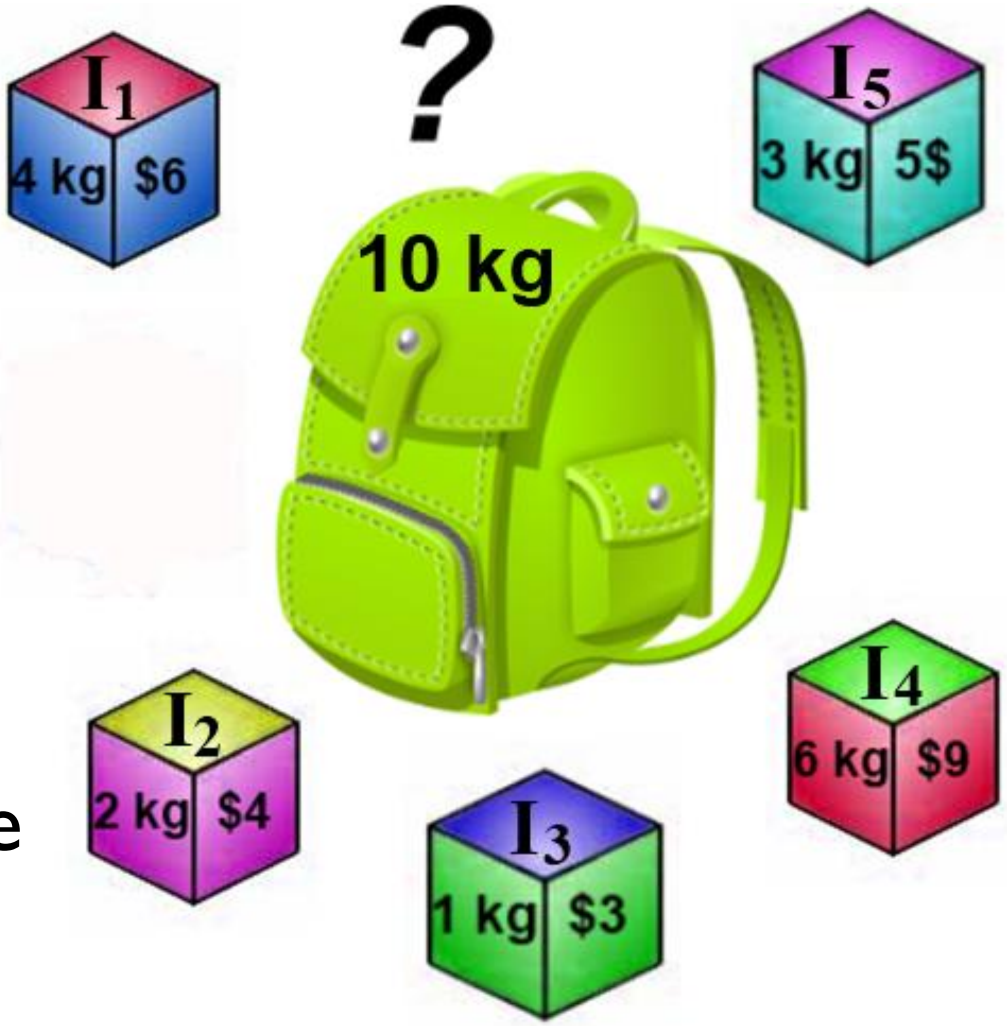
What is the running time of this algorithm?

$O(n*W)$

Remember that the brute-force algorithm takes: **$O(2^n)$**

Knapsack Problem

1) Fill out the dynamic programming table for the knapsack problem to the right.
2) Trace back through the table to find the items in the knapsack.



I_1
4 kg
$6
?
10 kg
I_5
3 kg
5$
I_2
2 kg
$4
I_3
1 kg
$3
I_4
6 kg
$9