Dynamic Programming II

Marcelo B. Barata Ribeiro
William Sena

Summary

1 Divide & Conquer recap

2 Four steps of dynamic programming

3 Longest Common Subsequence

4 Knapsack

- Unbounded Knapsack
- 0/1 Knapsack
- Item retrieval in Knapsack
- Knapsack complexity

5 Maximum Independent Set

Divide & Conquer recap

Divide & Conquer recap

- In divide-and-conquer, a problem is expressed in terms of sub-problems that are substantially smaller, say half the size.
  - Example of Counting Sort: “Any split of constant proportionality yields a recursion tree of depth nlog(n).” (CORMEN, T. et al, p. 176)
- In contrast, in a typical dynamic programming formulation, a problem is reduced to sub-problems that are only slightly smaller. For instance, L(j) relies on L(j-1). Thus the full recursion tree generally has polynomial depth and an exponential number of nodes.
  - However, it turns out that most of these nodes are repeats, that there are not too many *distinct* sub-problems among them.

4 Steps of Dynamic Programming

Dynamic Programming

- The main idea:
  - Compute the solutions to the sub-problems once and store the solutions in a table, so that they can be reused (repeatedly) later.

- We trade space for time!

4 steps of dynamic programming

- When developing a dynamic-programming algorithm, we follow a sequence of four steps:
  - 1. Characterize the structure of an optimal solution (Structure).
  - 2. Recursively define the value of an optimal solution (Principle of Optimality).
  - 3. Compute the value of an optimal solution, typically in a bottom-up fashion.
  - 4. Construct an optimal solution from computed information.

4 steps of dynamic programming

Step 1: Structure

- Decompose the problem into smaller problems, and find a relation between the structure of the optimal solution of the original problem and the solutions of the smaller problems.

4 steps of dynamic programming

Step 2: Principle of Optimality

- Express the solution of the original problem in terms of optimal solutions for smaller problems.

4 steps of dynamic programming

Step 3: Bottom-up computation

- Compute the value of an optimal solution (preferably) in a bottom-up fashion by using a table structure.

4 steps of dynamic programming

Step 4: Construction of optimal solution

- Construct an optimal solution from computed information.
- Note that if we need only the value of an optimal solution, and not the solution itself, then we can omit step 4.

Memoization

- In dynamic programming, we write out a recursive formula that expresses large problems in terms of smaller ones and then use it to fill out a table of solution values in a bottom-up manner.
- The formula also suggests a recursive algorithm, but we saw earlier that naive recursion can be terribly inefficient, because it solves the same sub-problems over and over again.
- On the problem which involve dynamic programming, such an algorithm would use a hash table to store the values that had already been computed. At each recursive call, the algorithm would first check if the answer was already in the table and then would proceed to its calculation only if it wasn't.

Longest Common Subsequence

Longest Common Subsequence

- When a spell checker encounters a possible misspelling, it looks in its dictionary for other words that are close by. What is the appropriate notion of closeness in this case?
- That’s where we apply a *Longest Common Subsequence* algorithm, a type of *Edit Distance*.

Longest Common Subsequence

1. Characterize the structure of an optimal solution. 🤔
2. Recursively define the value of an optimal solution. 🤔
3. Compute the value of an optimal solution, typically in a bottom-up fashion. 🤔
4. Construct an optimal solution from computed information (remember that this is optional depending on your objective). 🤔

Longest Common Subsequence

1\.

lcs("ABC", "ADE") = lcs("AB", "AD") + (if "C" == "E" then 1 else 0)

lcs("ABC", "ADE") = 1 + 0 = 1

Longest Common Subsequence

2. Let L [i] [j] be the optimal value for position i of the first string and position j of the second one

$$L[i][j] = \begin{cases} L[i-1][j-1] & if X[i-1] == Y[j-1] \\ max(L[i-1][j], L[i][j-1]) & otherwise \end{cases}$$

Longest Common Subsequence

- 3. $Let\ L\ a\ matrix\ (n+1) \times (m+1),\ where,\ n\ =\ length(X),\ m\ =\ length(Y),$

$$\forall\ i \in n,\ \forall\ j \in m\ , then\ L[I][j]\ =\ lcm(X[:i], Y[:j])$$

$$L = \begin{bmatrix} 0 & \cdots & 0 & 0 \\ \vdots & \ddots & \vdots & \vdots \\ 0 & \cdots & L[i-1][j-1] & L[i-1][j] \\ 0 & \cdots & L[i][j-1] & L[i][j] \end{bmatrix}$$

$$L[i][j] = \begin{cases} L[i-1][j-1] & if X[i-1] == Y[j-1] \\ max(L[i-1][j], L[i][j-1]) & otherwise \end{cases}$$

Longest Common Subsequence

- 4. Making the way back, it is possible to discover the subsequence common

 - Starting with i = n and j = m

$$\begin{cases} came\ from\ \leftarrow & if\ L[i][j] == L[i][j-1] \\ came\ from\ \uparrow & if\ L[i][j] == L[i-1][j] \\ X[i] == Y[j] & otherwise \end{cases}$$

Longest Common Subsequence

```
lcs X Y :=
    m, n = length X, length Y
    L = [[0, n + 1 times], m + 1 times]

    for i = 1 to m
        for j = 1 to n
            L[i][j] = if X[i - 1] == Y[j - 1]
                         then L[i - 1][j - 1] + 1
                         else  max(L[i - 1][j] , L[i][j - 1])
```

Runtime: $O(|X||Y|)$

Longest Common Subsequence

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution, typically in a bottom-up fashion.
4. Construct an optimal solution from computed information (remember that this is optional depending on your objective).

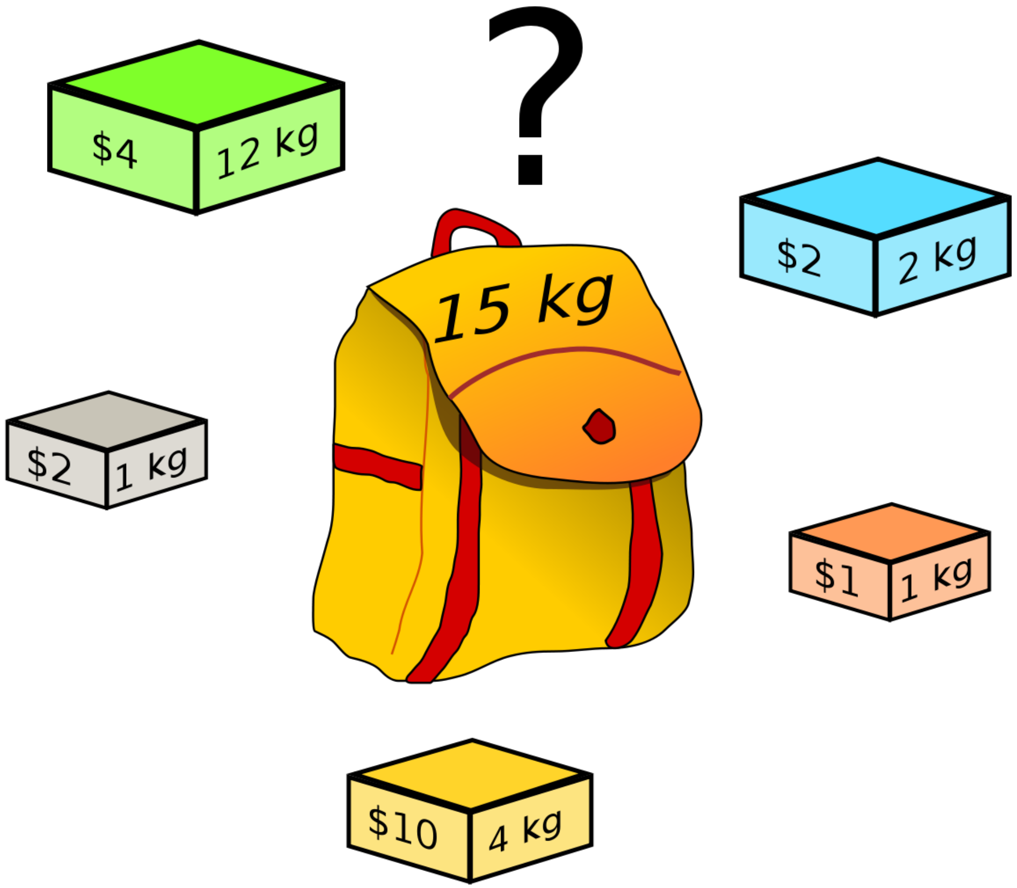
Longest Common Subsequence

Let’s see a demonstration!

- https://www.cs.usfca.edu/~galles/visualization/DPLCS.html
- We are going to compare “exponential” and “polynomial”.

Knapsack

Knapsack



Unbounded Knapsack

An unbounded knapsack allows repetition.

So, we want to find the items to put in an unbounded knapsack. How do we solve it? Let’s remember of the 4 steps of dynamic programming:

1. Characterize the structure of an optimal solution.

2. Recursively define the value of an optimal solution.

3. Compute the value of an optimal solution, typically in a bottom-up fashion.

4. Construct an optimal solution from computed information (remember that this is optional depending on your objective).

Unbounded Knapsack

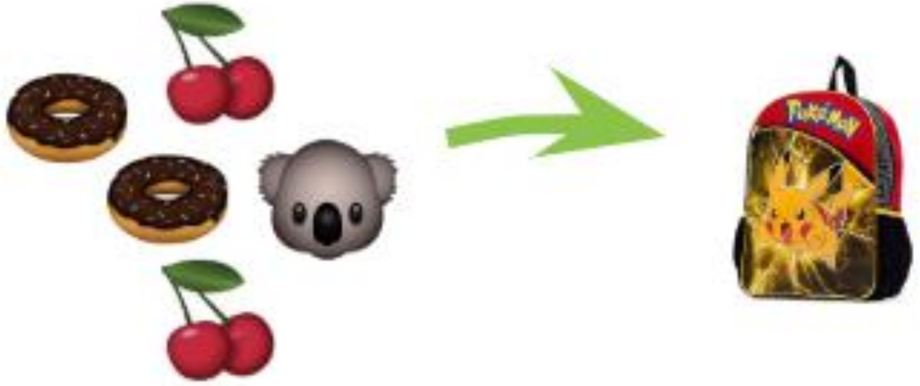
1.

- The problem statement restricts us from reducing the number of items.
- By process of elimination, we reason that we must solve the problem for smaller knapsacks.
- W(4) = W(7) – 3 = W(12)-5

Unbounded Knapsack

capacity x
value V

Then this must be an optimal solution for capacity x - w_i for item i = 



capacity x - w_i
value V - v_i

If there existed a more optimal solution, then adding a donut to that more optimal solution would improve the first solution.

Unbounded Knapsack

1. Characterize the structure of an optimal solution.

2. Recursively define the value of an optimal solution.

3. Compute the value of an optimal solution, typically in a bottom-up fashion.

4. Construct an optimal solution from computed information (remember that this is optional depending on your objective).

Unbounded Knapsack

2.

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

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

Unbounded Knapsack

1. Characterize the structure of an optimal solution.

2. Recursively define the value of an optimal solution.

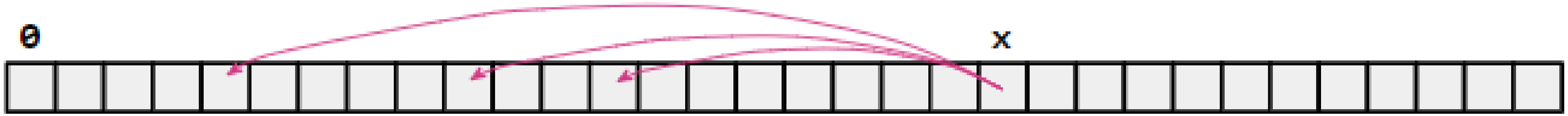
3. Compute the value of an optimal solution, typically in a bottom-up fashion.

4. Construct an optimal solution from computed information (remember that this is optional depending on your objective).

Unbounded Knapsack

(3)

- For the weight W(x) at position x, if we have items of weight 8, 11 e 16, what other weights are relevant to compare?



- Remember the recursion formula. What do we want to maximize?

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

Unbounded Knapsack

```
unbounded_knapsack W weights values :=
  n = length weights
  K = [0, W + 1 times]

  for x = 1 to W
    for i = 0 to n - 1
      if weights[i] <= x
        K[x] = max(K[x], K[x - weights[i]] + values[i])

  K[x]
```

Runtime: O(nW)

Unbounded Knapsack

1. Characterize the structure of an optimal solution.

2. Recursively define the value of an optimal solution.

3. Compute the value of an optimal solution, typically in a bottom-up fashion.

4. Construct an optimal solution from computed information (remember that this is optional depending on your objective).

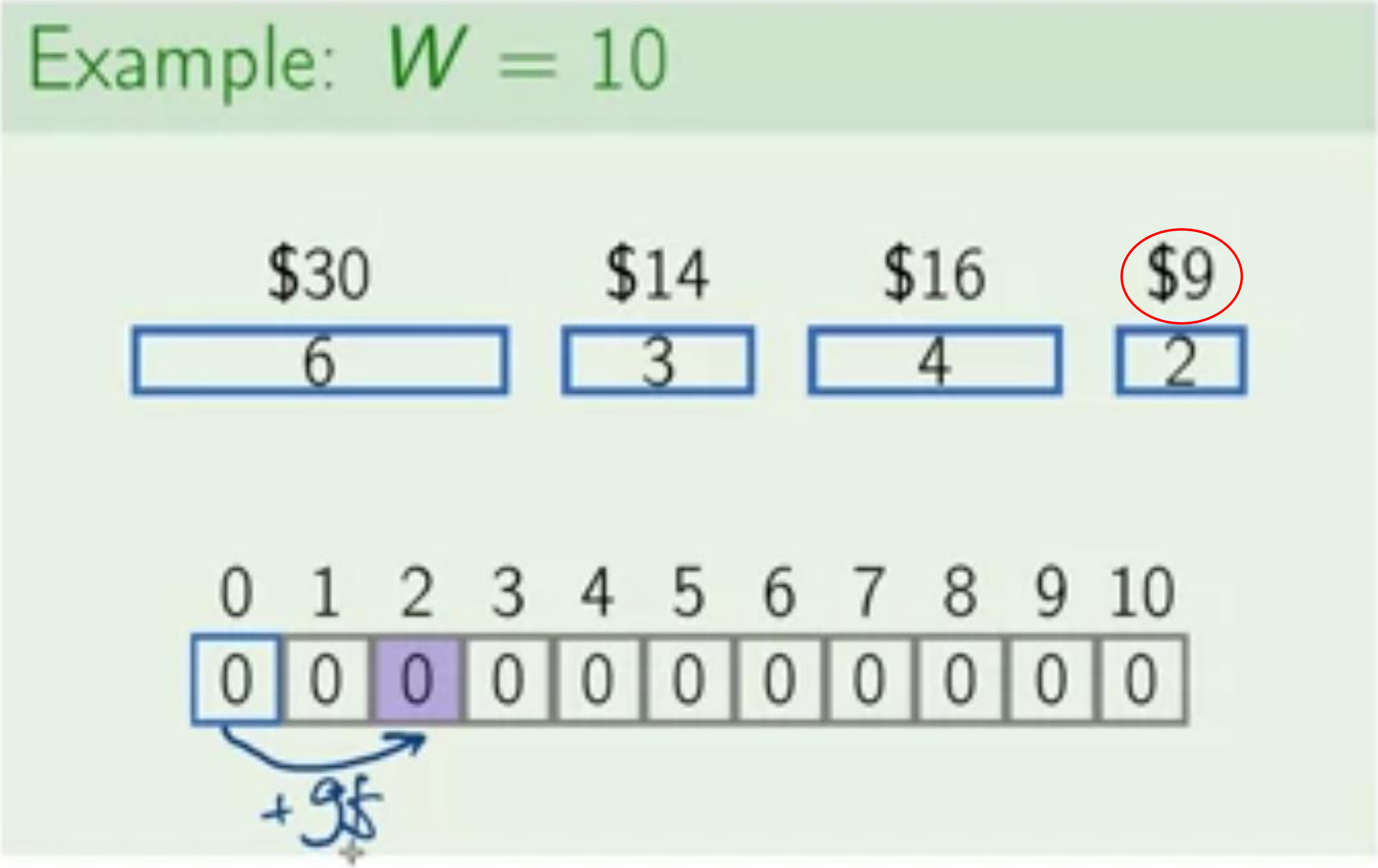
Unbounded Knapsack

Example: $W = 10$

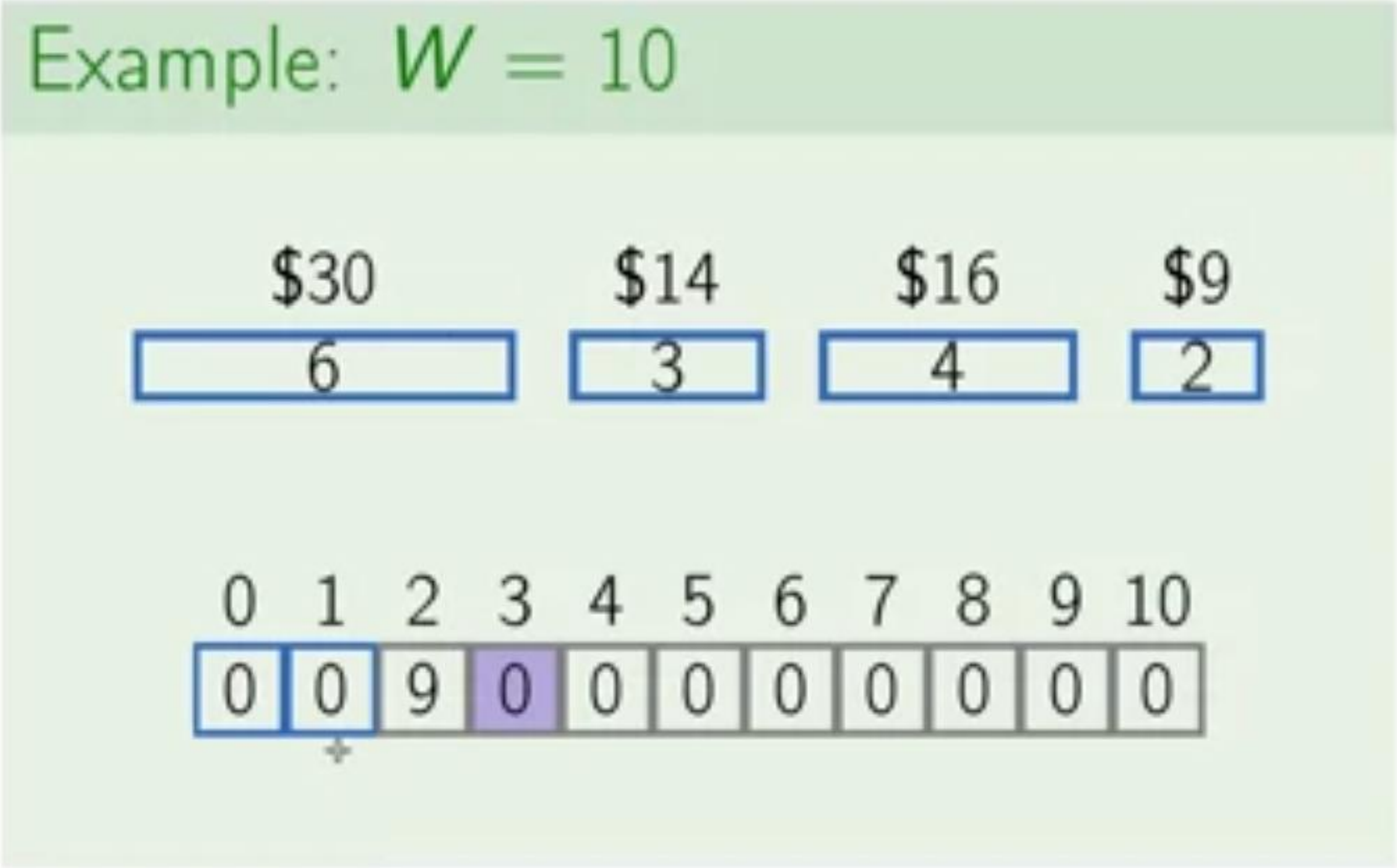
\$30	\$14	\$16	\$9
6	3	4	2

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

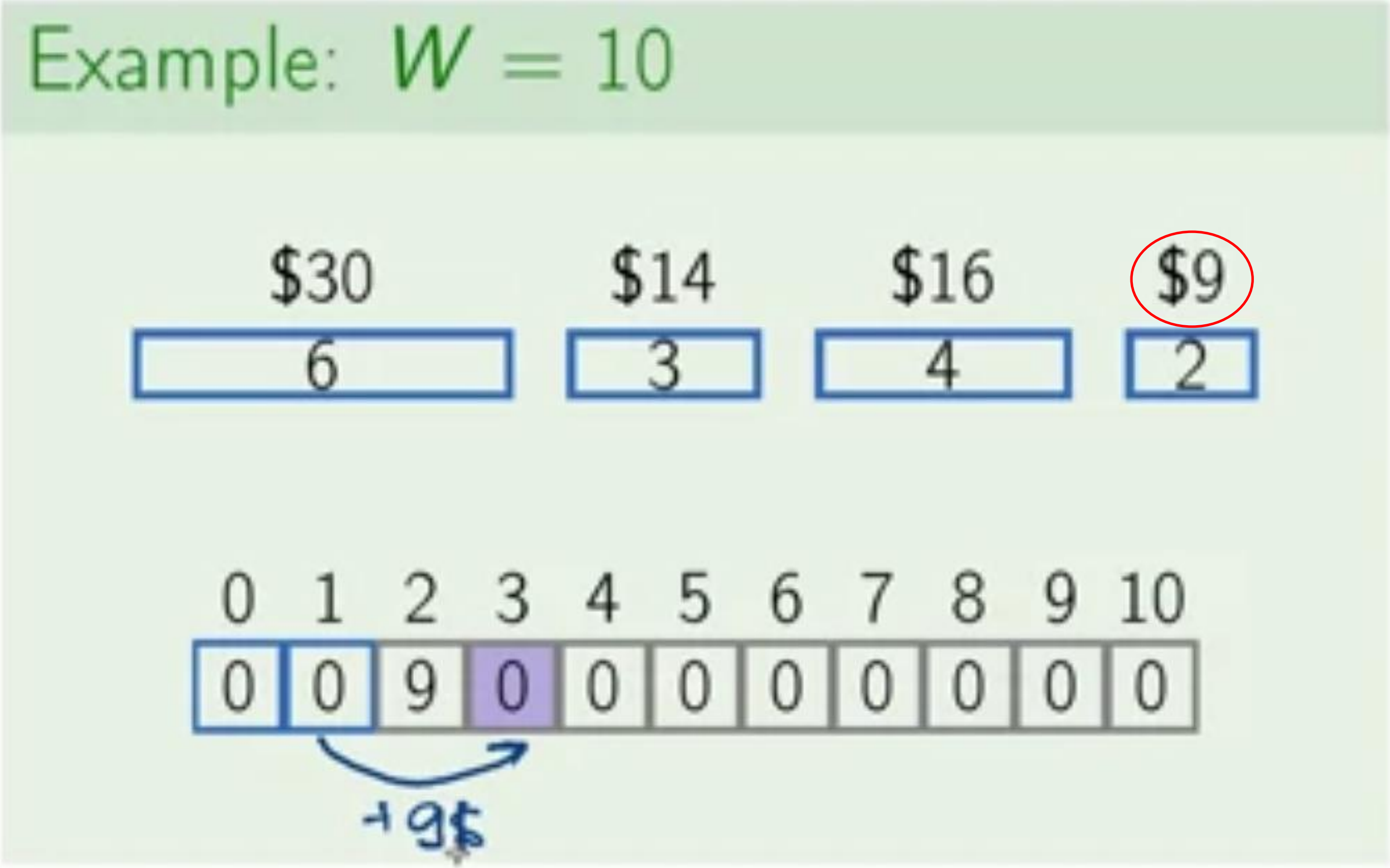
Unbounded Knapsack



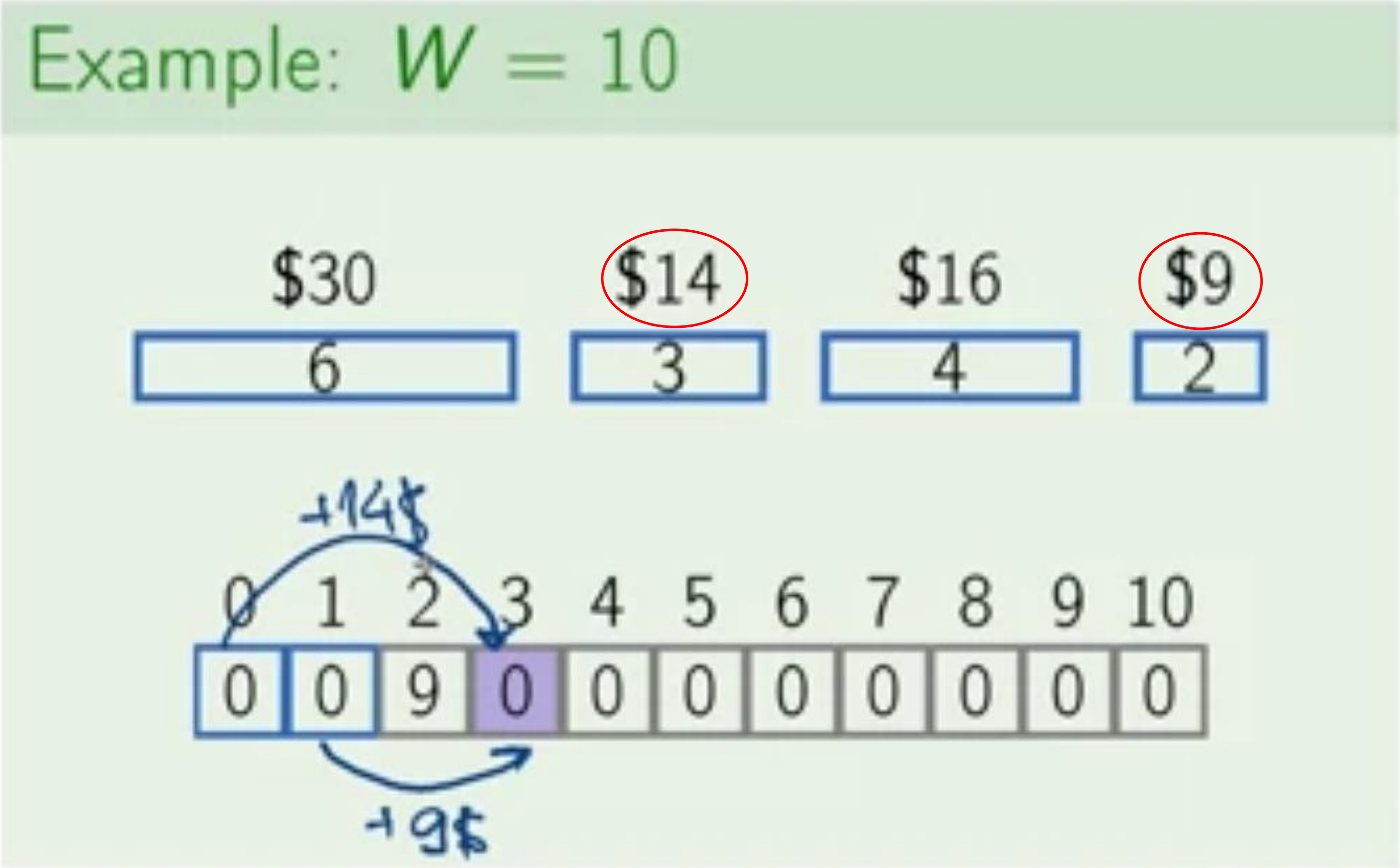
Unbounded Knapsack



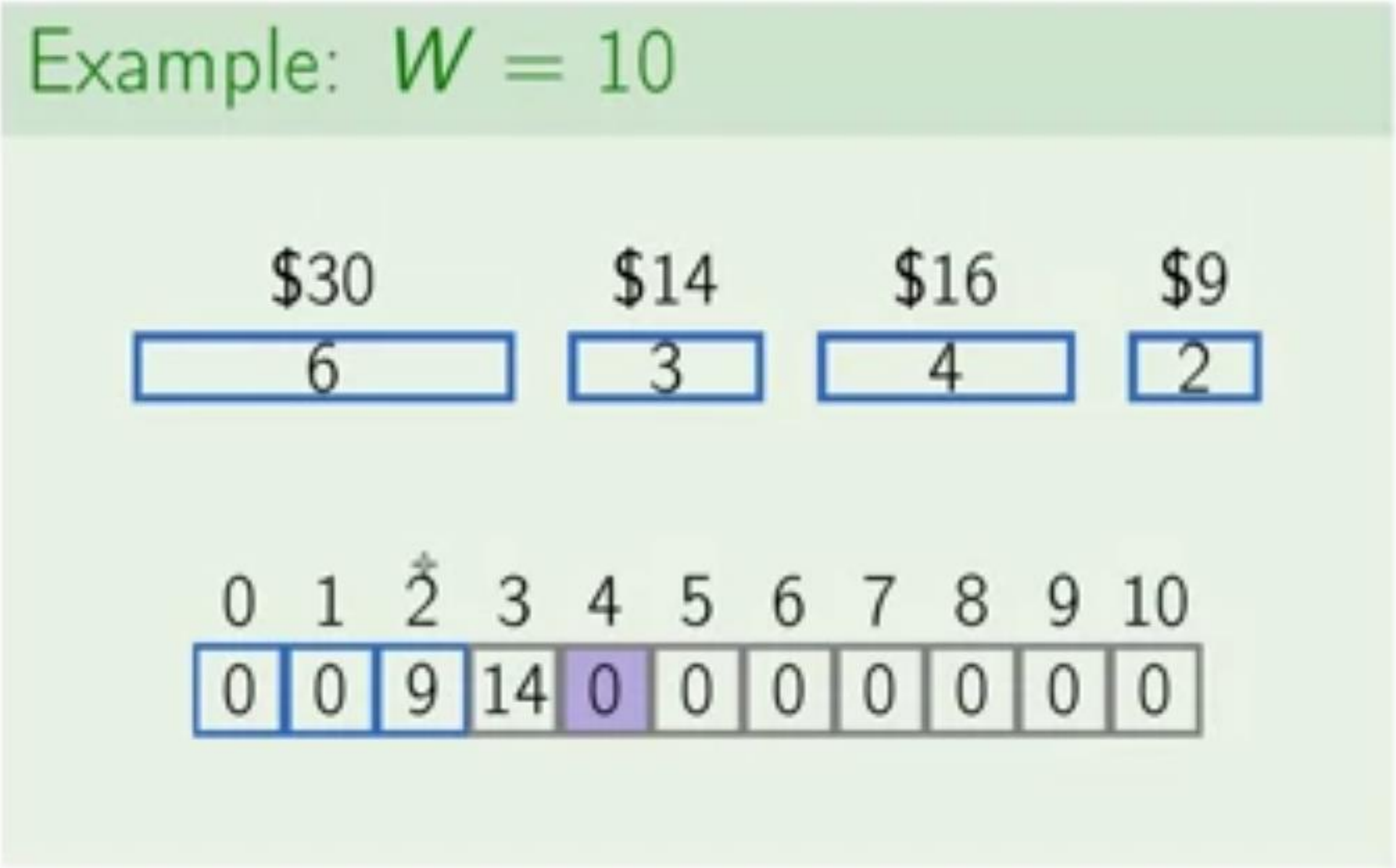
Unbounded Knapsack



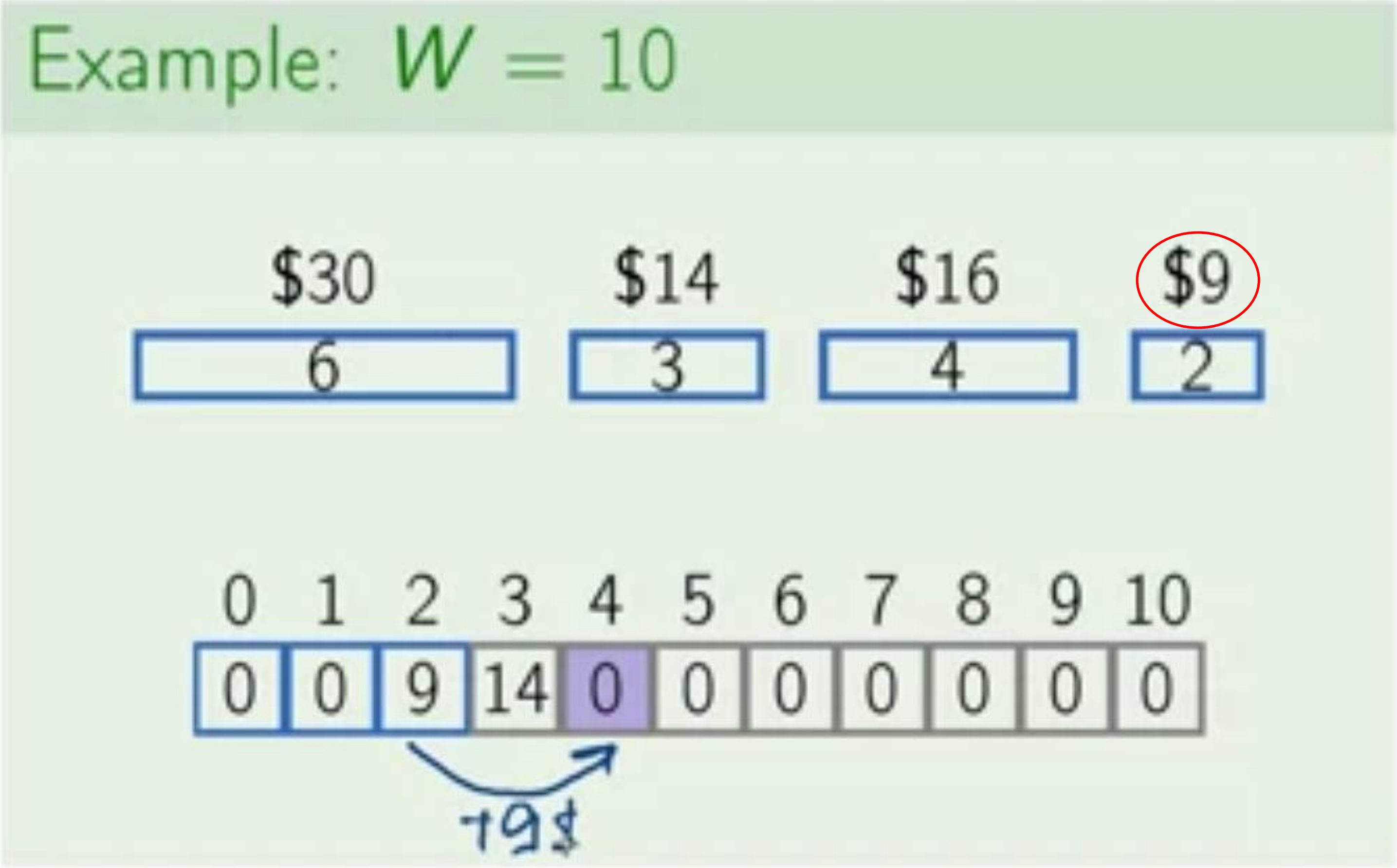
Unbounded Knapsack



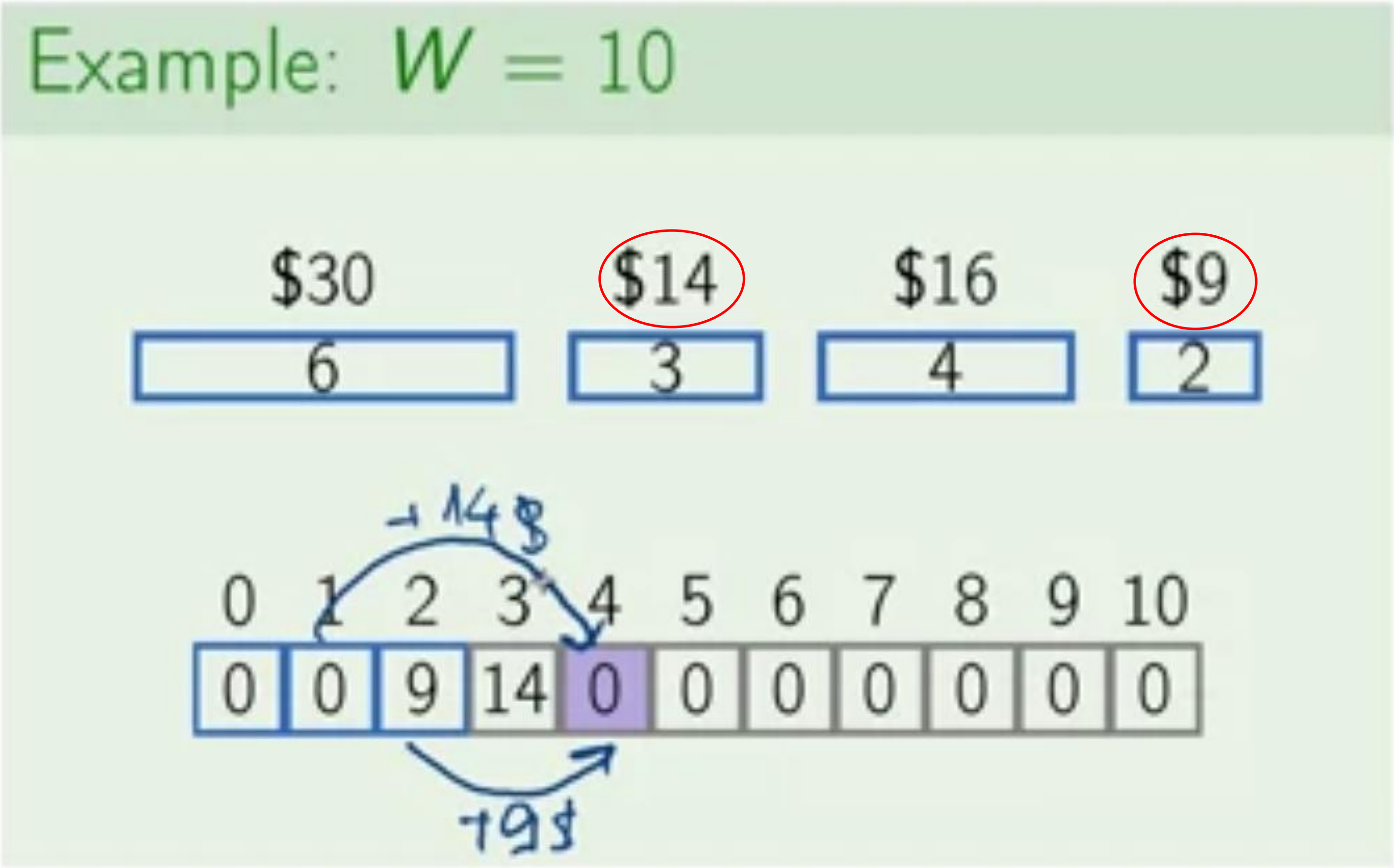
Unbounded Knapsack



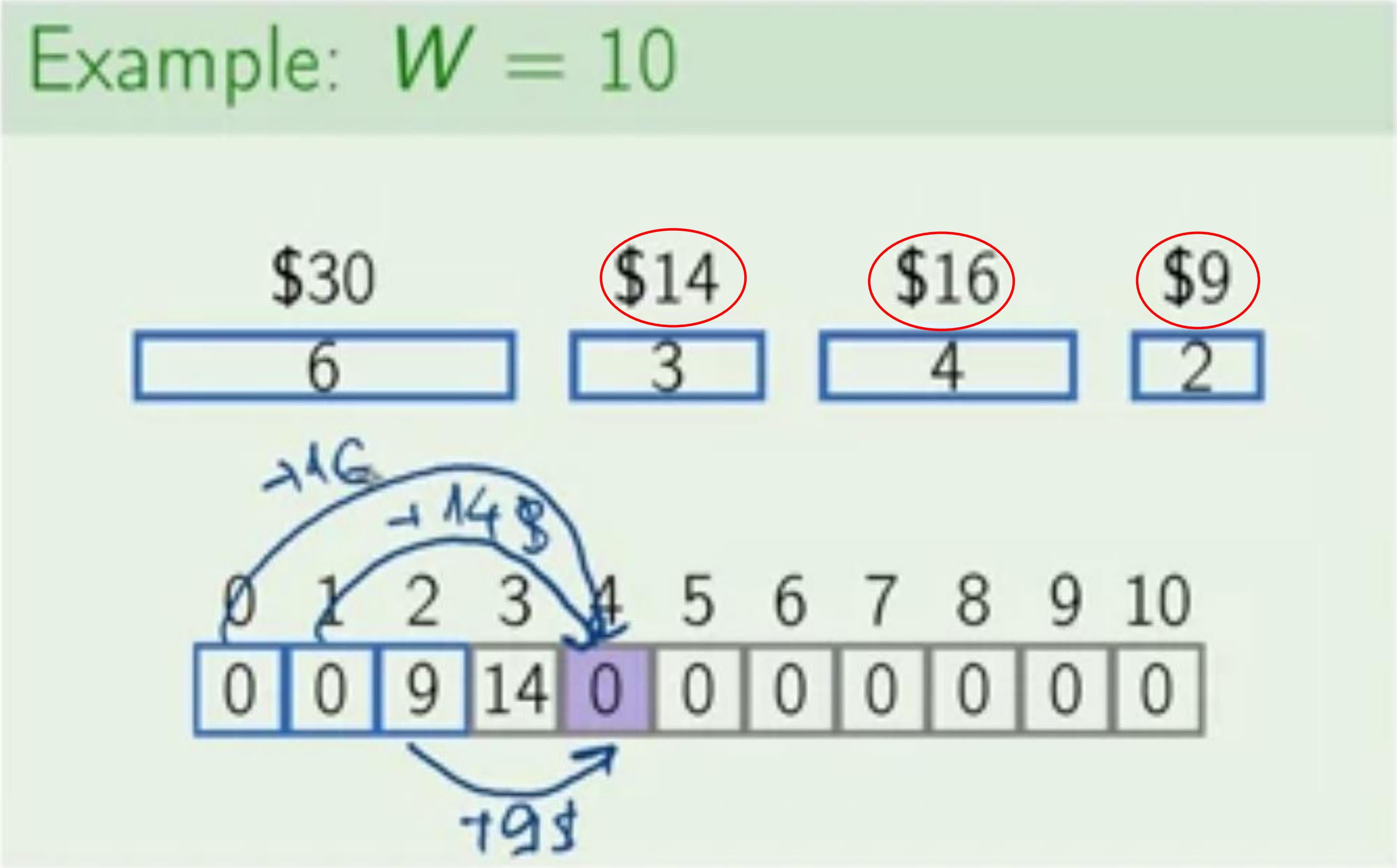
Unbounded Knapsack



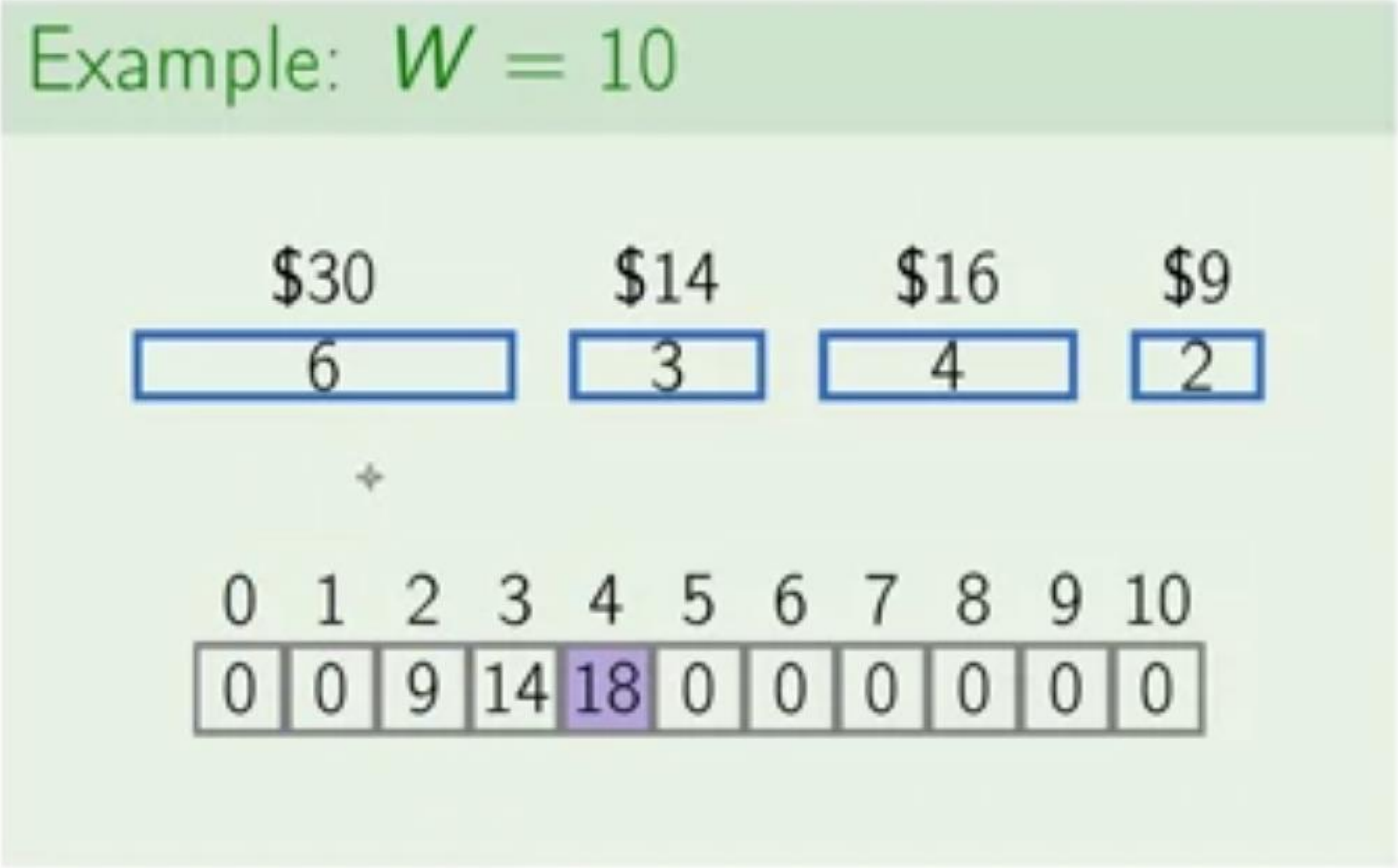
Unbounded Knapsack



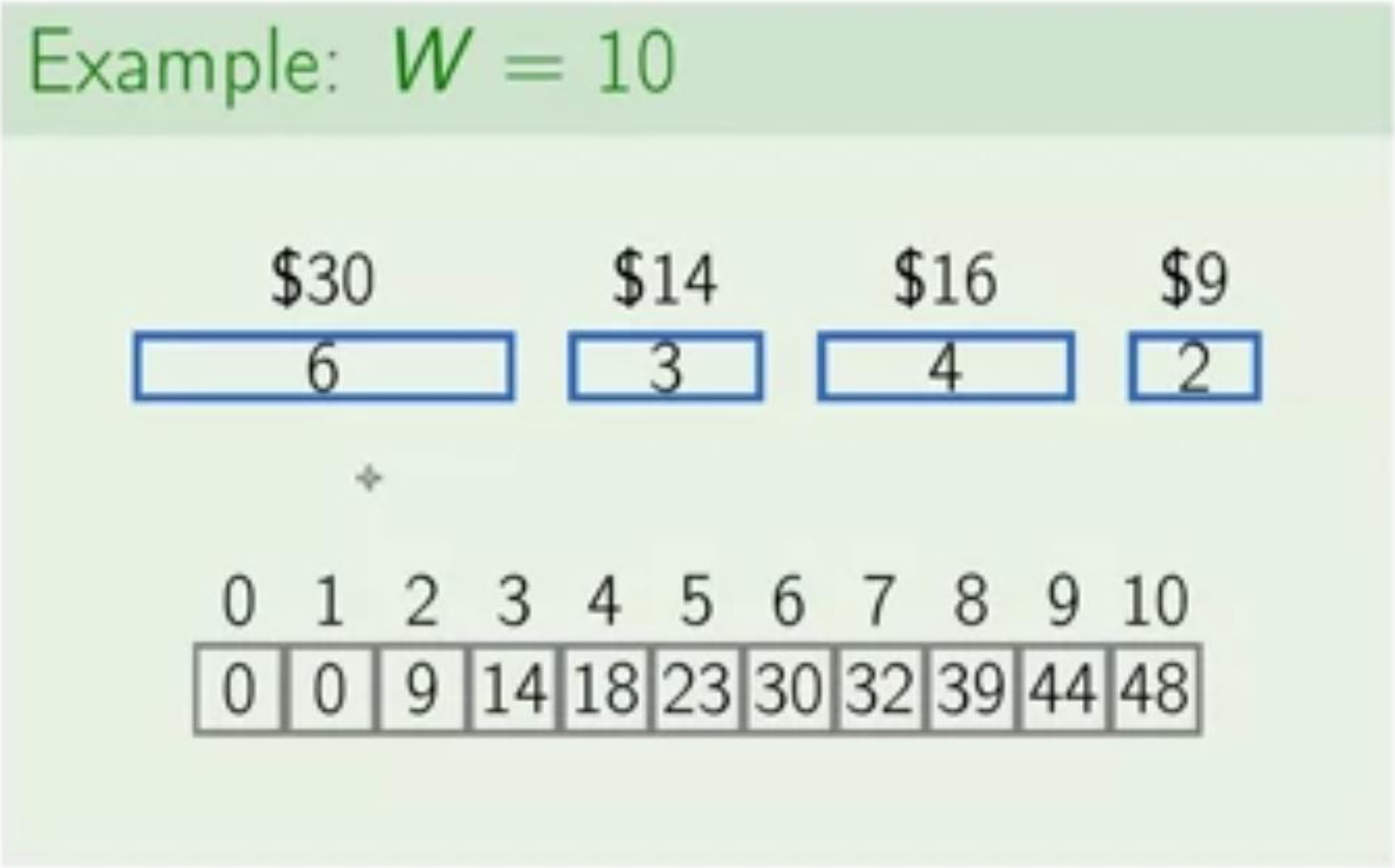
Unbounded Knapsack



Unbounded Knapsack



Unbounded Knapsack

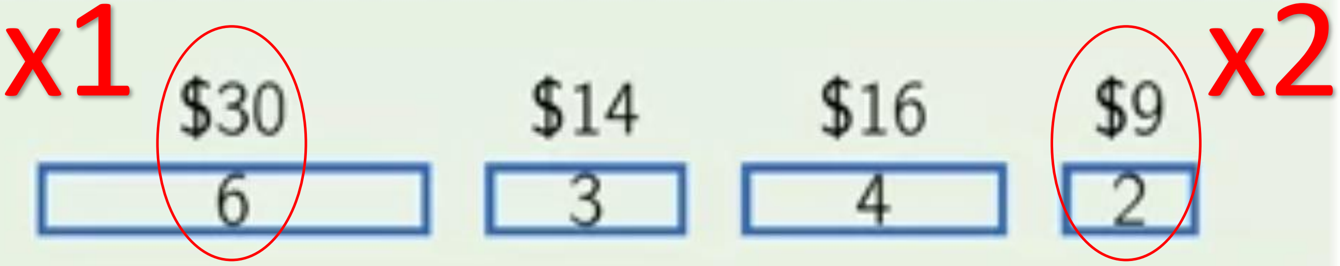

Example: $W = 10$

$30	$14	$16	$9
6	3	4	2

0	1	2	3	4	5	6	7	8	9	10
0	0	9	14	18	23	30	32	39	44	48

Unbounded Knapsack

Example: $W = 10$



0	1	2	3	4	5	6	7	8	9	10
0	0	9	14	18	23	30	32	39	44	48

0/1 Knapsack

No repetition is allowed this time!!!

1. Characterize the structure of an optimal solution. 🤔
2. Recursively define the value of an optimal solution. 🤔
3. Compute the value of an optimal solution, typically in a bottom-up fashion. 🤔
4. Construct an optimal solution from computed information (remember that this is optional depending on your objective). 🤔

0/1 Knapsack

1. What if we use a two-dimensional table to remember what items were used? So instead of just increasing W from left to right, we would also iterates over items from top to down.

0/1 Knapsack

1. What if we use a two-dimensional table to remember what items were used? So instead of just increasing W from left to right, we would also iterates over items from top to down.

2.

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

0/1 Knapsack

1. What if we use a two-dimensional table to remember what items were used? So instead of just increasing W from left to right, we would also iterates over items from top to down.

2.
$$K[x,j] = \begin{cases} 0 & \text{if } x \text{ or } j \text{ are } 0 \\ \max\{K[x,j-1],\ K[x-w_j,j-1] + v_j\} & \text{otherwise} \end{cases}$$

3. Let’s build some algorithm!

0/1 Knapsack

```
zero_one_knapsack W weights values :=
  n = length weights
  K = [[0, n + 1 times], W + 1 times]

  for x = 1 to W
    for i = 1 to n
      K[x][i] = K[x][i – 1]
      if weights[i - 1] <= x
        K[x][i] = max(K[x][i], K[x - weights[i]][i - 1] + values[i - 1])

  K[x]
```

Runtime: O(nW)

0/1 Knapsack

1. What if we use a two-dimensional table to remember what items were used? So instead of just increasing W from left to right, we would also iterates over items from top to down.

2.

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

3. Let’s build some algorithm!

0/1 Knapsack

1. What if we use a two-dimensional table to remember what items were used? So instead of just increasing W from left to right, we would also iterates over items from top to down.

2.

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

3. Let’s build some algorithm!

4. Construct an optimal solution from computed information.

0/1 Knapsack

1. What if we use a two-dimensional table to remember what items were used? So instead of just increasing W from left to right, we would also iterates over items from top to down.

2.
$$K[x,j] = \begin{cases} 0 & \text{if } x \text{ or } j \text{ are } 0 \\ \max\{K[x,j-1],\ K[x-w_j,j-1] + v_j\} & \text{otherwise} \end{cases}$$

3. Let's build some algorithm!

4. Construct an optimal solution from computed information.

We will show (4) later

0/1 Knapsack

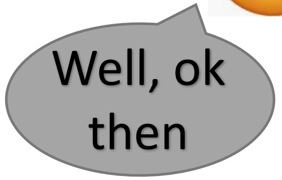
1. What if we use a two-dimensional table to remember what items were used? So instead of just increasing W from left to right, we would also iterates over items from top to down. 

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

3. Let’s build some algorithm! 

4. Construct an optimal solution from computed information. 

We will show (4) later



0/1 Knapsack

- 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)

0/1 Knapsack

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1						
2						
3						
4						

for w = 0 to W

    B[0,w] = 0

0/1 Knapsack

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0					
2	0					
3	0					
4	0					

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

0/1 Knapsack

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					

$i=1$
$b_i=3$
$w_i=2$
$w=1$
$w-w_i=-1$

if w_i <= w // item i can be part of the solution
    if $b_i + B[i-1,w-w_i] > B[i-1,w]$
        $B[i,w] = b_i + B[i-1,w-w_i]$
    else
        $B[i,w] = B[i-1,w]$
else **B[i,w] = B[i-1,w]** // $w_i > w$

0/1 Knapsack

Items:

$\boxed{1: (2,3)}$
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
b_i=3
w_i=2
w=2
$w-w_i$ =0

if w_i <= w // item i can be part of the solution
  if b_i + B[i-1,w-w_i] > B[i-1,w]
    B[i,w] = b_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

0/1 Knapsack

Items:

$\boxed{1: (2,3)}$

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					

$i=1$

$b_i=3$

$w_i=2$

$w=3$

$w-w_i=1$

```
if wi <= w // item i can be part of the solution
      if bi + B[i-1,w-wi] > B[i-1,w]
         B[i,w] = bi + B[i-1,w- wi]
      else
         B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]  // wi > w
```

0/1 Knapsack

Items:

$\boxed{1: (2,3)}$
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					

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

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

0/1 Knapsack

Items:

$\boxed{1: (2,3)}$

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					

$i=1$

$b_i=3$

$w_i=2$

$w=5$

$w-w_i=3$

```
if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_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
```

0/1 Knapsack

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					

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

if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_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

0/1 Knapsack

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**			
3	0					
4	0					

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

if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_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

0/1 Knapsack

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**		
3	0					
4	0					

$i=2$
$b_i=4$
$w_i=3$
$w=3$
$w-w_i=0$

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

0/1 Knapsack

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**	
3	0					
4	0					

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

if $w_i \le w$ // item i can be part of the solution
  if $b_i + B[i-1,w-w_i] > B[i-1,w]$
    $B[i,w] = b_i + B[i-1,w-w_i]$
  else
    $B[i,w] = B[i-1,w]$
else $B[i,w] = B[i-1,w]$ // $w_i > w$

0/1 Knapsack

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					
4	0					

$i=2$
$b_i=4$
$w_i=3$
$w=5$
$w-w_i=2$

if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_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

0/1 Knapsack

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**		
4	0					

$i=3$

$b_i=5$

$w_i=4$

$w= 1..3$

if w_i <= w // item i can be part of the solution

    if $b_i + B[i-1,w-w_i] > B[i-1,w]$

        $B[i,w] = b_i + B[i-1,w- w_i]$

    else

        $B[i,w] = B[i-1,w]$

else **B[i,w] = B[i-1,w]** // $w_i > w$

0/1 Knapsack

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**	
4	0					

$i=3$
$b_i=5$
$w_i=4$
$w= 4$
$w- w_i=0$

```
if w_i <= w // item i can be part of the solution
      if b_i + B[i-1,w-w_i] > B[i-1,w]
         B[i,w] = b_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
```

0/1 Knapsack

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					

$i=3$
$b_i=5$
$w_i=4$
$w= 5$
$w- w_i=1$

```
if w_i <= w // item i can be part of the solution
      if b_i + B[i-1,w-w_i] > B[i-1,w]
         B[i,w] = b_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
```

0/1 Knapsack

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**	

$i=4$
$b_i=6$
$w_i=5$
$w= 1..4$

if w_i <= w // item i can be part of the solution
    if $b_i + B[i-1,w-w_i] > B[i-1,w]$
        $B[i,w] = b_i + B[i-1,w- w_i]$
    else
        $B[i,w] = B[i-1,w]$
else **B[i,w] = B[i-1,w]** // $w_i > w$

0/1 Knapsack

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**

$i=4$
$b_i=6$
$w_i=5$
$w= 5$
$w- w_i=0$

if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_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

Retrieving items from Knapsack

- 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, an addition to this algorithm is necessary.

Retrieving items from Knapsack

- All of the information we need is in the table.
- $B[n,W]$ is the maximal value of items that can be placed in the Knapsack.
- Let i=n and k=W

  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

  $i = i-1$ // Assume the i^{th} item is not in the knapsack

  // Could it be in the optimally packed knapsack?

Retrieving items from Knapsack

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

i=4
k= 5
b_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
        $i = i-1,\ k = k\text{-}w_i$
    else
        $i = i-1$

Retrieving items from Knapsack

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

i=4
k= 5
b_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] ≠ 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
```

Retrieving items from Knapsack

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

i=3
k= 5
b_i=6
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
    $i = i-1$

Retrieving items from Knapsack

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

i=2

k= 5

b_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] ≠ 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
```

Retrieving items from Knapsack

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

i=1

k= 2

b_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] ≠ B[i−1,k] then
        mark the ith item as in the knapsack
        i = i−1, k = k-w_i
    else
        i = i−1
```

Retrieving items from Knapsack

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

i=0
k= 0

The optimal knapsack should contain {1, 2}

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

Retrieving items from Knapsack

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

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

The optimal knapsack should contain {1, 2}

Retrieving items from Knapsack

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution, typically in a bottom-up fashion.
4. Construct an optimal solution from computed information (remember that this is optional depending on your objective).

Knapsack complexity

Knapsack complexity

- Brute force:
  - $O(2^n)$
- Using dynamic programming:
  - $O(Wn)$
- So, is dynamic programming always good?

Knapsack complexity

- Brute force:
  - $O(2^n)$
- Using dynamic programming:
  - $O(Wn)$
- So, is dynamic programming always good?
- NO!!!

Knapsack complexity

- Brute force:
  - $O(2^n)$
- Using dynamic programming:
  - $O(Wn)$
- So, is dynamic programming always good?
- NO!!!
- The DP runtime of $O(nW)$ is better than our brute-force runtime of $O(2^n)$,
  - provided that $W < c2^n$ for some $c > 0$

Knapsack complexity and NP-completeness

- Knapsack is NP-complete and we found a strategy to turn it O(nW).
- Then P = NP?
- ALSO NO!!!
- O(Wn) is pseudo-polynomial.

Knapsack complexity and NP-completeness

- Time complexity commonly measures the time that an algorithm takes as a function of the **length in bits** of its input. From this perspective, Knapsack is NP-complete.
- W is not polynomial in the length of the input, which is what makes O(nW) *pseudo*-polynomial.
- Consider W = 1,000,000,000,000. It only takes 40 bits to represent this number, so input size = 40, but the computational runtime uses the factor 1,000,000,000,000 which is $O(2^{40})$.
- So the runtime is more accurately said to be $O(n.2^{\text{bits in W}})$, which is exponential.
- Therefore, O(nW) is exponential in the number of bits required to write out the input.
    - Adding one more bit to the end of the representation of W doubles its size and doubles the runtime.

Maximal Independent Set

Maximal Independent Set

- A subset of nodes $S \subset V$ is an *independent set* of graph $G = (V,E)$ if there are no edges between them.
- Finding the largest independent set in a graph is believed to be intractable.
- However, when the graph happens to be a *tree*, the problem can be solved in linear time, using dynamic programming.
- So here's the algorithm: Start by rooting the tree at any node r. Now, each node defines a subtree. This immediately suggests subproblems:
- I(u) = size of largest independent set of subtree hanging from u.
- Our final goal is I(root).

Maximal Independent Set

- Dynamic programming proceeds as always from smaller subproblems to larger ones, that is to say, bottom-up in the rooted tree.
- Suppose we know the largest independent sets for all subtrees below a certain node u; in other words, suppose we know I(w) for all descendants w of u. How can we compute I(u)? Let's split the computation into two cases: any independent set either includes u or it doesn't.

Maximal Independent Set

1. Characterize the structure of an optimal solution. 🤔
2. Recursively define the value of an optimal solution. 🤔
3. Compute the value of an optimal solution, typically in a bottom-up fashion. 🤔
4. Construct an optimal solution from computed information (remember that this is optional depending on your objective).

Maximal Independent Set

1.

- If the independent set includes u, then we get one point for it, but we aren't allowed to include the children of u.
- Therefore we move on to the grandchildren. This is the first case in the formula. On the other hand, if we don't include u, then we don't get a point for it, but we can move on to its children.

2.

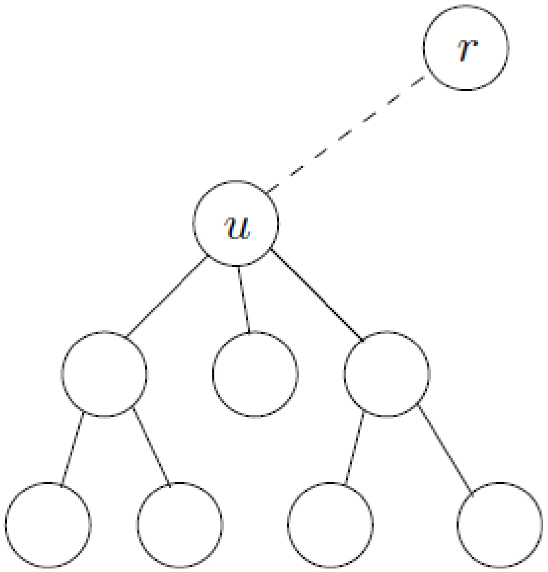
$$I(u) = \max\left\{1 + \sum_{\text{grandchildren } w \text{ of } u} I(w), \sum_{\text{children } w \text{ of } u} I(w)\right\}$$

Maximal Independent Set

```
mis node :=
  w = if is_leaf(node)
        then node.weight
        else  max(node.weight + sum([mis in_all node.grandchildren]),
                      sum([mis in_all node.children]))
  w
```

Runtime: O(|V|)

Maximal Independent Set



References

- DASGUPTA, C. H. et al. Algorithms.
- CORMEN, Thomas et al. Introduction to Algorithms.
- University of Stanford
- Coursera
- University of Nebraska–Lincoln

Thanks

Marcelo Barata Ribeiro
marcelobbribeiro@gmail.com

•

William Sena

will182neves@gmail.com