CSE- 207
Algorithms

Lecture: 12
Dynamic Programming

Fahad Ahmed
Lecturer, Dept. of CSE
E-mail: fahadahmed@uap-bd.edu

0-1 Knapsack

0-1 KNAPSACK

- **Definition:** Given items of different values and volumes, find the most valuable set of items that fit in a knapsack of fixed volume.

- The 0/1 knapsack problem means that the items are either completely or no items are filled in a knapsack.

- **Formal Definition:** There is a knapsack of capacity $c > 0$ and N items. Each item has value $v_i > 0$ and weight $w_i > 0$. Find the selection of items ($\delta_i = 1$ if selected, 0 if not) that fit, $\sum_{i=1}^{N} \delta_i w_i \leq c$, and the total value, $\sum_{i=1}^{N} \delta_i v_i$, is maximized.

Knapsack 0-1 Problem

Consider the problem having weights and profits are:

- **Weights**: {2, 3, 4, 5}
- **Profits:** {3, 4, 5, 6}
- The weight of the knapsack is 5 kg
- The number of items is 4

KNAPSACK 0-1 PROBLEM

- The above problem can be solved by using the following method:

$x_i = \{1, 0, 0, 1\}$

$= \{0, 0, 0, 1\}$

$= \{0, 1, 0, 1\}$

… … … …

1 denotes that the item is completely picked and 0 means that no item is picked.

Since there are 4 items so possible combinations will be: **$2^4 = 16$;** So. There are 16 possible combinations that can be made by using the above problem.

Once all the combinations are made, we have to select the combination that provides the maximum profit.

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.
  - In dynamic programming approach, the complicated problem is divided into sub-problems, then we find the solution of a sub-problem and the solution of the sub-problem will be used to find the solution of a complex problem.

KNAPSACK 0-1 PROBLEM

How this problem can be solved by using the Dynamic programming approach?

- 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 builds 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.

KNAPSACK 0-1 PROBLEM

- So our recursive formula for subproblems:

  B[k, w] = B[k - 1,w], if $w_k > w$

  = max { **B[k - 1,w],** B[k - 1,w - w_k] + v_k}, otherwise

- 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 Problem

Consider the problem having weights and profits are:

- **Weights**: {2, 3, 4, 5}
- **Profits:** {3, 4, 5, 6}
- The weight of the knapsack is 5 kg
- The number of items is 4

KNAPSACK 0-1 EXAMPLE

Weights: {2, 3, 4, 5}
Profits: {3, 4, 5, 6}

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

Weights: {2, 3, 4, 5}
Profits: {3, 4, 5, 6}

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					

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

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

KNAPSACK 0-1 EXAMPLE

Weights: {2, 3, 4, 5}
Profits: {3, 4, 5, 6}

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
$w\text{-}w_i = 0$

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

KNAPSACK 0-1 EXAMPLE

Weights: {2, 3, 4, 5}
Profits: {3, 4, 5, 6}

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

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

KNAPSACK 0-1 EXAMPLE

Weights: {2, 3, 4, 5}
Profits: {3, 4, 5, 6}

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					

$i = 1$
$v_i = 3$
$w_i = 2$
$w = 4$
$w\text{-}w_i = 2$

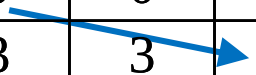
if $w_i <= w$ //item i can be in the solution
    if $v_i + B[i\text{-}1,w\text{-}w_i] > B[i\text{-}1,w]$
        $B[i,w] = v_i + B[i\text{-}1,w\text{-}w_i]$
    else
        $B[i,w] = B[i\text{-}1,w]$
else $B[i,w] = B[i\text{-}1,w]$ // $w_i > w$

KNAPSACK 0-1 EXAMPLE

Weights: {2, 3, 4, 5}
Profits: {3, 4, 5, 6}

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					



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

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

KNAPSACK 0-1 EXAMPLE

Weights: {2, 3, 4, 5}
Profits: {3, 4, 5, 6}

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					

$i = 2$
$v_i = 4$
$w_i = 3$
$w = 1$
$w\text{-}w_i = -2$

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

KNAPSACK 0-1 EXAMPLE

Weights: {2, 3, 4, 5}
Profits: {3, 4, 5, 6}

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					

$i = 2$
$v_i = 4$
$w_i = 3$
w = 2
$w\text{-}w_i = -1$

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

KNAPSACK 0-1 EXAMPLE

Weights: {2, 3, 4, 5}
Profits: {3, 4, 5, 6}

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					

$i = 2$
$v_i = 4$
$w_i = 3$
w = 3
$w\text{-}w_i = 0$

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

KNAPSACK 0-1 EXAMPLE

Weights: {2, 3, 4, 5}
Profits: {3, 4, 5, 6}

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					

$i = 2$
$v_i = 4$
$w_i = 3$
$w = 4$
$w\text{-}w_i = 1$

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

KNAPSACK 0-1 EXAMPLE

Weights: {2, 3, 4, 5}
Profits: {3, 4, 5, 6}

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					

$i = 2$
$v_i = 4$
$w_i = 3$
$w = 5$
$w\text{-}w_i = 2$

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

KNAPSACK 0-1 EXAMPLE

Weights: {2, 3, 4, 5}
Profits: {3, 4, 5, 6}

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					

$i = 3$
$v_i = 5$
$w_i = 4$
w = 1..3
$w\text{-}w_i = -3..-1$

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

Weights: {2, 3, 4, 5}
Profits: {3, 4, 5, 6}

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					

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

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

KNAPSACK 0-1 EXAMPLE

Weights: {2, 3, 4, 5}
Profits: {3, 4, 5, 6}

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					

$i = 3$
$v_i = 5$
$w_i = 4$
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

Weights: {2, 3, 4, 5}
Profits: {3, 4, 5, 6}

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	

$i = 4$
$v_i = 6$
$w_i = 5$
w = 1..4
$w\text{-}w_i = -4..-1$

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

KNAPSACK 0-1 EXAMPLE

Weights: {2, 3, 4, 5}
Profits: {3, 4, 5, 6}

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

$i = 4$
$v_i = 6$
$w_i = 5$
$w = 5$
$w\text{-}w_i = 0$

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

KNAPSACK 0-1 EXAMPLE

Weights: {2, 3, 4, 5}
Profits: {3, 4, 5, 6}

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] ≠ 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?

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] ≠ 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

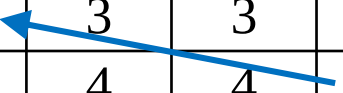
i = n , k = W
while i, k > 0
    if *B[i, k] ≠ B[i-1, k]* then
        mark the i^{th} item as in the knapsack
        i = i-1, k = k-w_i
    else
        i = i-1

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] ≠ 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] ≠ 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

How to find the items that are in the sack?

```
while (i > 0 && j > 0)
    {
        if(cost[i][j] != cost[i-1][j])
        {
            printf("%d\n",i);
            j = j-wm[i];
            i = i-1;
        }
        else
        {
            i = i-1;
        }
    }
```

KNAPSACK 0-1 PROBLEM – RUN TIME

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

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

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

O(*W*)

What is the running time of this algorithm?

O(*nW) – *of course, W can be mighty big***

What is an analogy in world of sorting?

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

KNAPSACK SIMULATION

if (w[i]>c)
T[i, c] = T[i-1,c]
else
$T[i, c] = \max(T(i-1, c), v[i] + T(i-1, c-w[i]))$

Assume,
Sack capacity, C = 10
Available Items
V = 7 2 1 6 12
W =3 1 2 4 6

i	v_i	w_i	0	1	2	3	4	5	6	7	8	9	10
0	0	0											
1	2	1											
2	1	2											
3	7	3								A			
4	6	4											
5	12	6											

Capacity

Any cell in the table represents the maximum value attained by choosing items from i items (not i^{th}) in a sack of capacity listed in the header. For example, the cell with value "A" represents that we can add items of total value "A" from 3 items and with a sack capacity=7 which is represented as T[3,7] = A

KNAPSACK SIMULATION

Assume,
Sack capacity, C = 10
Available Items
V = 7 2 1 6 12
W =3 1 2 4 6

if (w[i]>c)
T[i, c] = T[i-1,c]
else
T[i, c] = $\max(\ T(i-1,c),\ v[i]+T(i-1,c-w[i]))$

i	v_i	w_i	0	1	2	3	4	5	6	7	8	9	10
0	0	0											
1	2	1											
2	1	2											
3	7	3											
4	6	4											
5	12	6											

← Capacity

KNAPSACK SIMULATION

Assume,
Sack capacity, C = 10
Available Items
V = 7 2 1 6 12
W =3 1 2 4 6

if (w[i]>c)
T[i, c] = T[i-1,c]
else
T[i, c] $= \max(\ T(i-1,c),\ v[i] + T(i-1, c-w[i]))$

i	v_i	w_i	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	1											
2	1	2											
3	7	3											
4	6	4											
5	12	6											

Capacity

If i=0, no items are available, to put to the sack, the maximum value we can attain is 0.

KNAPSACK SIMULATION

Assume,
Sack capacity, C = 10
Available Items
V = 7 2 1 6 12
W =3 1 2 4 6

if (w[i]>c)
T[i, c] = T[i-1,c]
else
T[i, c] $= \max(\ T(i-1,c),\ v[i]+T(i-1,c-w[i]))$

i	v_i	w_i	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	1	0										
2	1	2	0										
3	7	3	0										
4	6	4	0										
5	12	6	0										

← Capacity

If bag capacity is 0, we can't add anything into the sack. So, attained value is 0.

KNAPSACK SIMULATION

Assume,
Sack capacity, C = 10
Available Items
V = 7 2 1 6 12
W =3 1 2 4 6

if (w[i]>c)
T[i, c] = T[i-1,c]
else
T[i, c] = max($T(i-1,c)$, $v[i] + T(i-1, c-w[i])$)

i	v_i	w_i	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	1	0										
2	1	2	0										
3	7	3	0										
4	6	4	0										
5	12	6	0										

← Capacity

T[1,1] = Max(T[1-1,1], v_1 + T[1-1,1-1])
= Max(T[0,1], 2 + T[0,0])
= Max(0, 2+0) = 2

KNAPSACK SIMULATION

Assume,
Sack capacity, C = 10
Available Items
V = 7 2 1 6 12
W =3 1 2 4 6

if (w[i]>c)
T[i, c] = T[i-1,c]
else
$T[i, c] = \max(\ T(i-1, c),\ v[i] + T(i-1, c-w[i]))$

i	v_i	w_i	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	1	0										
2	1	2	0										
3	7	3	0										
4	6	4	0										
5	12	6	0										

← Capacity

T[1,1] = Max(T[1-1,1], v_1 + T[1-1,1-1])
= Max(T[0,1], 2 + T[0,0])
= Max(0, 2+0) = 2

KNAPSACK SIMULATION

Assume,
Sack capacity, C = 10
Available Items
V = 7 2 1 6 12
W =3 1 2 4 6

if (w[i]>c)
T[i, c] = T[i-1,c]
else
T[i, c] = max($T(i-1, c)$, $v[i] + T(i-1, c-w[i])$)

i	v_i	w_i	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	1	0	2									
2	1	2	0										
3	7	3	0										
4	6	4	0										
5	12	6	0										

← Capacity

T[1,1] = Max(T[1-1,1], v_1 + T[1-1,1-1])
= Max(T[0,1], 2 + T[0,0])
= Max(0, 2+0) = 2

KNAPSACK SIMULATION

Assume,
Sack capacity, C = 10
Available Items
V = 7 2 1 6 12
W =3 1 2 4 6

if (w[i]>c)
T[i, c] = T[i-1,c]
else
T[i, c] = max($T(i-1, c)$, $v[i] + T(i-1, c-w[i])$)

i	v_i	w_i	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	1	0	2	2								
2	1	2	0										
3	7	3	0										
4	6	4	0										
5	12	6	0										

← Capacity

Going w_1 cell back

T[1,2] = Max(T[1-1,1], v_1 + T[1-1,2-1])
= Max(T[0,1], 2 + T[0,1])
= Max(0, 2+0) = 2

KNAPSACK SIMULATION

Assume,
Sack capacity, C = 10
Available Items
V = 7 2 1 6 12
W =3 1 2 4 6

if (w[i]>c)
T[i, c] = T[i-1,c]
else
T[i, c] $= \max(T(i-1,c), v[i] + T(i-1, c-w[i]))$

i	v_i	w_i	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	1	0	2	2								
2	1	2	0										
3	7	3	0										
4	6	4	0										
5	12	6	0										

← Capacity

T[1,2] = Max(T[1-1,1], v_1 + T[1-1,2-1])
= Max(T[0,1], 2 + T[0,1])
= Max(0, 2+0) = 2

KNAPSACK SIMULATION

Assume,
Sack capacity, C = 10
Available Items
V = 7 2 1 6 12
W =3 1 2 4 6

if (w[i]>c)
T[i, c] = T[i-1,c]
else
T[i, c] $= \max(T(i-1,c), v[i] + T(i-1, c-w[i]))$

i	v_i	w_i	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	1	0	2	2	2	2	2	2	2	2	2	2
2	1	2	0										
3	7	3	0										
4	6	4	0										
5	12	6	0										

← Capacity

For any c >= 1,
T[1,c] = Max(T[1-1,1], v_1 + T[1-1,c-1])
= Max(T[0,1], 2 + T[0,c-1])
= Max(0, 2+0) = 2

KNAPSACK SIMULATION

Assume,
Sack capacity, C = 10
Available Items
V = 7 2 1 6 12
W =3 1 2 4 6

if (w[i]>c)
T[i, c] = T[i-1,c]
else
$T[i, c] = \max(T(i-1, c), v[i] + T(i-1, c-w[i]))$

i	v_i	w_i	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	1	0	2	2	2	2	2	2	2	2	2	2
2	1	2	0										
3	7	3	0										
4	6	4	0										
5	12	6	0										

← Capacity

T(2,1)= T[2-1,1] as w_2 > c
= T[1,1] = 2

KNAPSACK SIMULATION

Assume,
Sack capacity, C = 10
Available Items
V = 7 2 1 6 12
W =3 1 2 4 6

if (w[i]>c)
T[i, c] = T[i-1,c]
else
T[i, c] $= \max(T(i-1,c), v[i] + T(i-1, c-w[i]))$

i	v_i	w_i	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	1	0	2	2	2	2	2	2	2	2	2	2
2	1	2	0	2									
3	7	3	0										
4	6	4	0										
5	12	6	0										

← Capacity

T(2,1)= T[2-1,1] as $w_2 > c$
= T[1,1] = 2

KNAPSACK SIMULATION

Assume,
Sack capacity, C = 10
Available Items
V = 7 2 1 6 12
W =3 1 2 4 6

if (w[i]>c)
T[i, c] = T[i-1,c]
else
$T[i, c] = \max(T(i-1, c), v[i] + T(i-1, c-w[i]))$

i	v_i	w_i	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	1	0	2	2	2	2	2	2	2	2	2	2
2	1	2	0	2									
3	7	3	0										
4	6	4	0										
5	12	6	0										

← Capacity

Going w_2 cell back

T[2,2] = Max(T[2-1,2], v_2 + T[1-1, 2-2])
= Max(T[1,2], 1 + T[1,0])
= Max(2, 1) = 2

KNAPSACK SIMULATION

Assume,
Sack capacity, C = 10
Available Items
V = 7 2 1 6 12
W =3 1 2 4 6

if (w[i]>c)
T[i, c] = T[i-1,c]
else
T[i, c] $= \max(T(i-1,c), v[i] + T(i-1, c-w[i]))$

i	v_i	w_i	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	1	0	2	2	2	2	2	2	2	2	2	2
2	1	2	0	2	2								
3	7	3	0										
4	6	4	0										
5	12	6	0										

← Capacity

T[2,2] = 2

KNAPSACK SIMULATION

Assume,
Sack capacity, C = 10
Available Items
V = 7 2 1 6 12
W =3 1 2 4 6

if (w[i]>c)
T[i, c] = T[i-1,c]
else
$T[i, c] = \max(T(i-1, c), v[i] + T(i-1, c-w[i]))$

i	v_i	w_i	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	1	0	2	2	2	2	2	2	2	2	2	2
2	1	2	0	2	2								
3	7	3	0										
4	6	4	0										
5	12	6	0										

← Capacity

Going w_2 cell back

T[2,3] = Max(T[2-1, 3], v_2 + T[2-1, 3-2])
= Max(T[1,3], 1+ T[1,1])
= Max(2, 3) = 3

KNAPSACK SIMULATION

Assume,
Sack capacity, C = 10
Available Items
V = 7 2 1 6 12
W =3 1 2 4 6

if (w[i]>c)
T[i, c] = T[i-1,c]
else
$T[i, c] = \max(T(i-1, c), v[i] + T(i-1, c-w[i]))$

i	v_i	w_i	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	1	0	2	2	2	2	2	2	2	2	2	2
2	1	2	0	2	2	3	3	3	3	3	3	3	3
3	7	3	0	2	2	7	9	9	10	10	10	10	10
4	6	4	0	2	2	7	9	9	10	13	15	15	16
5	12	6	0	2	2	7	9	9	12	14	15	19	21

← Capacity

So, simplest version is compare 1) the cell above the current cell and 2) v_i + value of w_i cell backward in previous row. Populate the current cell with whichever value is bigger.

Populate the table with this logic.

KNAPSACK SIMULATION

How to find the items that are in the bag?

i	v_i	w_i	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	1	0	2	2	2	2	2	2	2	2	2	2
2	1	2	0	2	2	3	3	3	3	3	3	3	3
3	7	3	0	2	2	7	9	9	10	10	10	10	10
4	6	4	0	2	2	7	9	9	10	13	15	15	16
5	12	6	0	2	2	7	9	9	12	14	15	19	21

← Capacity

How to find the items that are in the sack?

```
while (i > 0 && j > 0)
    {
        if(cost[i][j] != cost[i-1][j])
        {
            printf("%d\n",i);
            j = j-wm[i];
            i = i-1;
        }
        else
        {
            i = i-1;
        }
    }
```

KNAPSACK SIMULATION

How to find the items that are in the bag?

i	v_i	w_i	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	1	0	2	2	2	2	2	2	2	2	2	2
2	1	2	0	2	2	3	3	3	3	3	3	3	3
3	7	3	0	2	2	7	9	9	10	10	10	10	10
4	6	4	0	2	2	7	9	9	10	13	15	15	16
5	12	6	0	2	2	7	9	9	12	14	15	19	21

← Capacity

KNAPSACK SIMULATION

How to find the items that are in the bag?

i	v_i	w_i	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	1	0	2	2	2	2	2	2	2	2	2	2
2	1	2	0	2	2	3	3	3	3	3	3	3	3
3	7	3	0	2	2	7	9	9	10	10	10	10	10
4	6	4	0	2	2	7	9	9	10	13	15	15	16
5	12	6	0	2	2	7	9	9	12	14	15	19	21

← Capacity

Sack = {}

1. Start with 21 (Green cell) and compare with the one above it (16).
2. As 21 and 16 are not equal item# 5 is included in the sack.
3. Go 6(weight if item) units back in previous row which is the next cell to check.

KNAPSACK SIMULATION

How to find the items that are in the bag?

i	v_i	w_i	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	1	0	2	2	2	2	2	2	2	2	2	2
2	1	2	0	2	2	3	3	3	3	3	3	3	3
3	7	3	0	2	2	7	9	9	10	10	10	10	10
4	6	4	0	2	2	7	9	9	10	13	15	15	16
5	12	6	0	2	2	7	9	9	12	14	15	19	21

← Capacity

included

Sack = {5}

KNAPSACK SIMULATION

How to find the items that are in the bag?

i	v_i	w_i	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	1	0	2	2	2	2	2	2	2	2	2	2
2	1	2	0	2	2	3	3	3	3	3	3	3	3
3	7	3	0	2	2	7	9	9	10	10	10	10	10
4	6	4	0	2	2	7	9	9	10	13	15	15	16
5	12	6	0	2	2	7	9	9	12	14	15	19	21

← Capacity

Not Included

Included

Weight Left: 10-6=4

Sack = {5}

1. Compare T[4,4] 9 (Green cell) with the one above it (9).
2. As both cell has same value item# 4 is not included in the sack.

KNAPSACK SIMULATION

How to find the items that are in the bag?

i	v_i	w_i	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	1	0	2	2	2	2	2	2	2	2	2	2
2	1	2	0	2	2	3	3	3	3	3	3	3	3
3	7	3	0	2	2	7	9	9	10	10	10	10	10
4	6	4	0	2	2	7	9	9	10	13	15	15	16
5	12	6	0	2	2	7	9	9	12	14	15	19	21

← Capacity

Included

Not Included

Included

Sack = {5, 3}

1. Compare T[3,4] 9 (Green cell) with the one above it (3).
2. As the cells have different values item# 3 is included in the sack.
3. Go 3 units back in previous row which is the next cell to check.

KNAPSACK SIMULATION

How to find the items that are in the bag?

i	v_i	w_i	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	1	0	2	2	2	2	2	2	2	2	2	2
2	1	2	0	2	2	3	3	3	3	3	3	3	3
3	7	3	0	2	2	7	9	9	10	10	10	10	10
4	6	4	0	2	2	7	9	9	10	13	15	15	16
5	12	6	0	2	2	7	9	9	12	14	15	19	21

← Capacity

Not Included

Included

Not Included

Included

Weight Left: 4-3=1

Sack = {5, 3}

1. Compare T[2,1] 2 (Green cell) with the one above it (2).
2. As both cells have same values item# 2 is not included in the sack.

KNAPSACK SIMULATION

How to find the items that are in the bag?

i	v_i	w_i	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	1	0	2	2	2	2	2	2	2	2	2	2
2	1	2	0	2	2	3	3	3	3	3	3	3	3
3	7	3	0	2	2	7	9	9	10	10	10	10	10
4	6	4	0	2	2	7	9	9	10	13	15	15	16
5	12	6	0	2	2	7	9	9	12	14	15	19	21

← Capacity

Included

Not Included

Included

Not Included

Included

Weight Left: 1-1=0

Sack = {5, 3, 1}

1. Compare T[1,1] 2 (Green cell) with the one above it (0).
2. As the cells have different values item# 1 is included in the sack.

KNAPSACK SIMULATION

How to find the items that are in the bag?

i	v_i	w_i	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	1	0	2	2	2	2	2	2	2	2	2	2
2	1	2	0	2	2	3	3	3	3	3	3	3	3
3	7	3	0	2	2	7	9	9	10	10	10	10	10
4	6	4	0	2	2	7	9	9	10	13	15	15	16
5	12	6	0	2	2	7	9	9	12	14	15	19	21

← Capacity

Included

Not Included

Included

Not Included

Included

Sack = {5, 3, 1}

As we have reached the 0^{th} row, we are done with item selection. So, the sack contains **1, 3 and 5** item with value = 2+7+12 = 21

KNAPSACK SIMULATION

```
for (int i = 0; i < n + 1; i++)
    {
        for (int j = 0; j < maxWeight + 1; j++)
        {
            if (i == 0 || j == 0)
                dp[i][j] = 0;

            else if (weights[i - 1] > j)
                dp[i][j] = dp[i - 1][j];

            else
                dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - weights[i - 1]] + profits[i - 1]);
        }
    }
    return dp[n][maxWeight];
```

KNAPSACK SIMULATION RECURSIVE

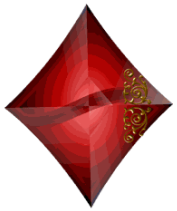
```
int knapSack(int W, int wt[], int val[], int n)
{
    if (n == 0 || W == 0)
        return 0;

    if (wt[n - 1] > W)
        return knapSack(W, wt, val, n - 1);

    else
        return max( val[n - 1]+knapSack(W - wt[n - 1], wt, val, n - 1),
                    knapSack(W, wt, val, n - 1));
}
```

Unbounded Knapsack

(Repetition of items allowed)

Wt. = 5
Value = 10

Wt. = 3
Value = 20



Wt. = 8
Value = 25

Wt. = 4
Value = 8

KNAPSACK **UNBOUNDED**

Given a knapsack weight **W** and a set of **n** items with certain value val_i and weight wt_i, we need to calculate the maximum amount that could make up this quantity exactly.

This is different from classical Knapsack problem, here we are allowed to use unlimited number of instances of an item.

KNAPSACK UNBOUNDED

Example:
items: {Apple, Orange, Melon}
weights: {1, 2, 3}
profits: {15, 20, 50}
capacity: 5

Different Profit Combinations:
5 Apples (total weight 5) => 75 profit
1 Apple + 2 Oranges (total weight 5) => 55 profit
3 Apples + 1 Orange (total weight 5) => 65 profit
2 Apples + 1 Melon (total weight 5) => 80 profit
1 Orange + 1 Melon (total weight 5) => 70 profit

Best Profit Combination : 2 Apples + 1 Melon with 80 profit.

KNAPSACK SIMULATION - UNBOUNDED

- Can include multiple instances of the same resource

i	v_i	w_i	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	1	0										
2	1	2	0										
3	7	3	0										
4	6	4	0										
5	12	6	0										

← Capacity

if (w[i]>c)
  T[i, c] = T[i-1,c]
else
  $T[i, c] = \max(T[i-1, c], v[i] + T[i, c-w[i]])$

KNAPSACK SIMULATION - UNBOUNDED

- Can include multiple instances of the same resource

i	v_i	w_i	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	1	0	2	4	6	8	10	12	14	16	18	20
2	1	2	0	2	4	6	8	10	12	14	16	18	20
3	7	3	0	2	4	7	9	11	14	16	18	21	23
4	6	4	0	2	2	7	9	11	14	16	18	21	23
5	12	6	0	2	2	7	9	11	14	16	18	21	23

← Capacity

if (w[i]>c)
  T[i, c] = T[i-1,c]
else
  $T[i, c] = \max(T[i-1, c], v[i] + T[i, c-w[i]])$

KNAPSACK SIMULATION - UNBOUNDED

How to find the items that are in the bag?

i	v_i	w_i	0	1	2	3	4	5	6	7	8	9	10	← Capacity
0	0	0	0	0	0	0	0	0	0	0	0	0	0	Not Included
1	2	1	0	2	4	6	8	10	12	14	16	18	20	**Included – 1 times**
2	1	2	0	2	4	6	8	10	12	14	16	18	20	Not Included
3	7	3	0	2	4	7	9	11	14	16	18	21	23	**Included – 3 times**
4	6	4	0	2	2	7	9	11	14	16	18	21	23	Not Included
5	12	6	0	2	2	7	9	11	14	16	18	21	23	Not Included

Sack = {3, 3, 3, 1}

As we have reached the 0^{th} row, we are done with item selection. So, the sack contains one quantity of item#**1 and 3 quantity of item#3** with value = 1*2+3*7 = 23

Reference

- Chapter 15 (15.1 and 15.3) (Cormen)
- http://www.shafaetsplanet.com/?p=3638
- https://www.javatpoint.com/0-1-knapsack-problem
- https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/