DYNAMIC PROGRAMMING: 0/1 KNAPSACK

Instructor: Dr Tarunpreet Bhatia
Assistant Professor, CSED
TIET

Knapsack problem

There are two versions of the problem:
(1) “0-1 knapsack problem” and
(2) “Fractional knapsack problem”

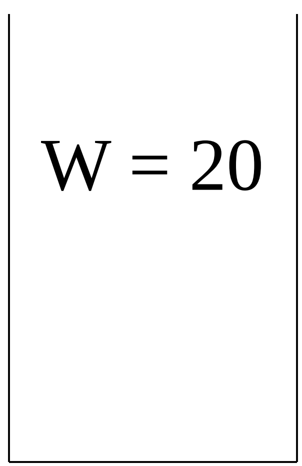
(1) Items are indivisible; you either take an item or not. Solved with *dynamic programming*
(2) Items are divisible: you can take any fraction of an item. Solved with a *greedy algorithm*.

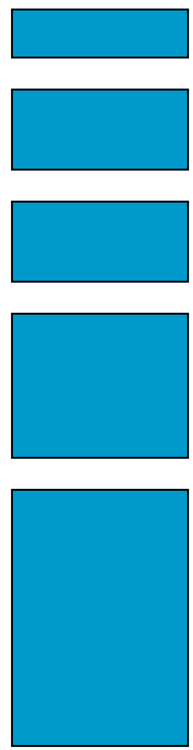
0-1 Knapsack problem

- Given a knapsack with maximum capacity W, and a set S consisting of n items
- Each item i has some weight w_i and benefit value b_i (all w_i , b_i and W are integer values)
- <u>Problem</u>: How to pack the knapsack to achieve maximum total value of packed items?

0-1 Knapsack problem: a picture

This is a knapsack
Max weight: W = 20





Items	Weight w_i	Benefit value b_i
	2	3
	3	4
	4	5
	5	8
	9	10

0-1 Knapsack problem

- Problem, in other words, is to find

$$\max \sum_{i \in T} b_i \text{ subject to } \sum_{i \in T} w_i \leq W$$

- The problem is called a *"0-1"* problem, because each item must be entirely accepted or rejected.

0-1 Knapsack problem: brute-force approach

Let's first solve this problem with a straightforward algorithm

- Since there are *n* items, there are 2^n possible combinations of items.
- We go through all combinations and find the one with the most total value and with total weight less or equal to *W*
- Running time will be $O(2^n)$

0-1 Knapsack problem: brute-force approach

- Can we do better?
- Yes, with an algorithm based on dynamic programming
- We need to carefully identify the subproblems

Let's try this:

If items are labeled *1..n*, then a subproblem would be to find an optimal solution for

S_i = *{items labeled 1, 2, .. i}*

Defining a Subproblem

If items are labeled *1..n*, then a subproblem would be to find an optimal solution for S_i = *{items labeled 1, 2, .. i}*

- This is a valid subproblem definition.
- The question is: can we describe the final solution (S_n) in terms of subproblems (S_i)?
- Unfortunately, we <u>can't</u> do that. Explanation follows….

Defining a Subproblem

$w_1 = 2$ $b_1 = 3$	$w_2 = 4$ $b_2 = 5$	$w_3 = 5$ $b_3 = 8$	$w_4 = 3$ $b_4 = 4$?	

Max weight: W = 20

For S_4:

Total weight: 14;

total benefit: 20

$w_1 = 2$ $b_1 = 3$	$w_2 = 4$ $b_2 = 5$	$w_3 = 5$ $b_3 = 8$	$w_4 = 9$ $b_4 = 10$

For S_5:

Total weight: 20

total benefit: 26

Item #	Weight w_i	Benefit b_i
1	2	3
2	3	4
3	4	5
4	5	8
5	9	10

S_4

S_5

Solution for S_4 is not part of the solution for S_5!!!

Defining a Subproblem

- As we have seen, the solution for S_4 is not part of the solution for S_5
- So our definition of a subproblem is flawed and we need another one!
- Let's add another parameter: *w*, which will represent the exact weight for each subset of items
- The subproblem then will be to compute $V[i,w]$

Defining a Subproblem

- Let's add another parameter: *w*, which will represent the maximum weight for each subset of items
- The subproblem then will be to compute *V[i,w]*, *i.e.*, to find an optimal solution for S_i = *{items labeled 1, 2, .. i} in a knapsack of size w*

Recursive Formula for subproblems

$$V[i,w] = \begin{cases} V[i-1,w] & \text{if } w_i > w \\ \max\{V[i-1,w], V[i-1,w-w_i]+b_i\} & \text{else} \end{cases}$$

It means, that the best subset of S_i that has total weight w is:

1) the best subset of S_{i-1} that has total weight $\leq w$, **or**

2) the best subset of S_{i-1} that has total weight $\leq w-w_i$ plus the item i

Recursive Formula for Subproblems

$$V[i,w]=\begin{cases} V[i-1,w] & \text{if } w_i > w \\ \max\{V[i-1,w], V[i-1,w-w_i]+b_i\} & \text{else} \end{cases}$$

The best subset of S_i that has the total weight $\leq w$, either contains item i or not.

- First case: $w_i>w$. Item i can't be part of the solution, since if it was, the total weight would be $> w$, which is unacceptable.
- Second case: $w_i \leq w$. Then the item i can be in the solution, and we choose *the case with greater value*.

Example

n = 4 (# of elements)
W = 8 (max weight)
Elements (weight, benefit):
(2,1), (3,2), (4,5), (5,6)

0-1 Knapsack Algorithm

```
for w = 0 to W
    V[0,w] = 0
for i = 1 to n
    V[i,0] = 0
for i = 1 to n
    for w = 0 to W
        if w_i <= w // item i can be part of the solution
            if b_i + V[i-1,w-w_i] > V[i-1,w]
                V[i,w] = b_i + V[i-1,w- w_i]
            else
                V[i,w] = V[i-1,w]
        else V[i,w] = V[i-1,w]  // w_i > w
```

Running time

```
for w = 0 to W
    V[0,w] = 0
for i = 1 to n
    V[i,0] = 0
for i = 1 to n
    for w = 0 to W
        &lt; the rest of the code &gt;
```

$O(W)$

Repeat n times

$O(W)$

What is the running time of this algorithm?

O(n*W)

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

Example

Let's run our algorithm on the following data:

n = 4 (# of elements)
W = 5 (max weight)
Elements (weight, benefit):
(2,3), (3,4), (4,5), (5,6)

Example (2)

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

for w = 0 to W

    V[0,w] = 0

Example (3)

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

    V[i,0] = 0

Example (4)

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 + V[i-1,w-w_i] > V[i-1,w]
        V[i,w] = b_i + V[i-1,w- w_i]
    else
        V[i,w] = V[i-1,w]
else V[i,w] = V[i-1,w] // w_i > w

Example (5)

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					

$i=1$
$b_i=3$
$w_i=2$
$w=2$
$w-w_i=0$

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

Example (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					

$i=1$

$b_i=3$

$w_i=2$

$w=3$

$w-w_i=1$

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

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

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

else

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

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

Example (7)

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					

$i=1$

$b_i=3$

$w_i=2$

$w=4$

$w-w_i=2$

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

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

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

else

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

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

Example (8)

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					

$i=1$

$b_i=3$

$w_i=2$

$w=5$

$w-w_i=3$

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

Example (9)

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 + V[i-1,w-w_i] > V[i-1,w]
        V[i,w] = b_i + V[i-1,w- w_i]
    else
        V[i,w] = V[i-1,w]
else V[i,w] = V[i-1,w]  // w_i > w
```

Example (10)

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 $b_i + V[i-1,w-w_i] > V[i-1,w]$
        $V[i,w] = b_i + V[i-1,w-w_i]$
    else
        $V[i,w] = V[i-1,w]$
else $V[i,w] = V[i-1,w]$ // $w_i > w$

Items:

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

3: (4,5)

4: (5,6)

Example (11)

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

Example (12)

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

Example (13)

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 + V[i-1,w-w_i] > V[i-1,w]
        V[i,w] = b_i + V[i-1,w- w_i]
    else
        V[i,w] = V[i-1,w]
else V[i,w] = V[i-1,w] // w_i > w

Example (14)

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

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

else

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

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

Example (15)

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

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

    else

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

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

Example (16)

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 + V[i-1,w-w_i] > V[i-1,w]
        V[i,w] = b_i + V[i-1,w- w_i]
    else
        V[i,w] = V[i-1,w]
else V[i,w] = V[i-1,w] // w_i > w

Example (17)

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

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

    else

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

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

Example (18)

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

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

  else

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

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

Comments

- This algorithm only finds the max possible value that can be carried in the knapsack
  - i.e., the value in V[n,W]
- To know the items that make this maximum value, an addition to this algorithm is necessary

How to find actual Knapsack Items

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

while(i>0)

    if $V[i,k] \neq V[i-1,k]$ 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

Finding the Items

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

$V[i,k] = 7$

$V[i-1,k] = 7$

i=n, k=W

while i,k > 0

if $V[i,k] \neq V[i-1,k]$ then

mark the i^{th} item as in the knapsack

$i = i-1$, $k = k-w_i$

else

$i = i-1$

Finding the Items (2)

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

$V[i,k] = 7$

$V[i–1,k] =7$

i=n, k=W

while i,k > 0

  if $V[i,k] \neq V[i–1,k]$ then

    mark the i^{th} item as in the knapsack

    $i = i–1, k = k-w_i$

  else

    $i = i–1$

Finding the Items (3)

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=5

w_i=4

$V[i,k] = 7$

$V[i-1,k] = 7$

i=n, k=W

while i,k > 0

if $V[i,k] \neq V[i-1,k]$ then

mark the i^{th} item as in the knapsack

$i = i-1$, $k = k-w_i$

else

$i = i-1$

Finding the Items (4)

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

$i=2$

$k= 5$

$b_i=4$

$w_i=3$

$V[i,k] = 7$

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

$k - w_i=2$

i=n, k=W

while i,k > 0

  if $V[i,k] \neq V[i-1,k]$ then

    mark the i^{th} item as in the knapsack

    $i = i-1,\ k = k-w_i$

  else

    $i = i-1$

Finding the Items (5)

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
$V[i,k] = 3$
$V[i-1,k] = 0$
$k - w_i = 0$

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

Finding the Items (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=0

k= 0

The optimal knapsack should contain {1, 2}

i=n, k=W

while i,k > 0

if $V[i,k] \neq V[i-1,k]$ then

mark the n^{th} item as in the knapsack

$i = i-1$, $k = k-w_i$

else

$i = i-1$

Finding the Items (7)

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 $V[i,k] \neq V[i-1,k]$ then
        mark the n^{th} item as in the knapsack
        $i = i-1,\ k = k\text{-}w_i$
    else
        $i = i-1$

The optimal knapsack should contain {1, 2}

Conclusion

- Dynamic programming is a useful technique of solving certain kind of problems
- When the solution can be recursively described in terms of partial solutions, we can store these partial solutions and re-use them as necessary.