0-1 KNAPSACK
USING
DYNAMIC PROGRAMMING

What is Dynamic Programming ?

- Dynamic programming is a method for solving optimization problems.

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

- It is a Bottom-Up approach.

- Coined by Richard Bellman who described dynamic programming as the way of solving problems where you need to find the best decisions one after another

Properties

- Two main properties of a problem suggest that the given problem can be solved using Dynamic Programming.
- These properties are **overlapping sub-problems and optimal substructure**.

Steps of Dynamic Programming Approach

- Dynamic Programming algorithm is designed using the following four steps −

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 the computed information.

Applications of Dynamic Programming Approach

- Matrix Chain Multiplication
- Longest Common Subsequence
- Travelling Salesman Problem
- Knapsack Problem

The 0-1 Knapsack Problem (Rucksack Problem)

- Given: A set S of n items, with each item i having
  - w_i - a positive weight
  - b_i - a positive benefit (value)

- Goal: Choose items with maximum total benefit but with weight at most W (weight of Knapsack).
  - In this case, we let T denote the set of items we take
  - **Objective**: maximize $\sum_{i \in T} b_i$
  - **Constraint**: $\sum_{i \in T} w_i \leq W$

Why not Greedy?

- Greedy approach provides the optimal solution to the fractional knapsack.
- 0-1 Knapsack cannot be solved by Greedy approach.
- It does not ensure an optimal solution to the 0-1 Knapsack

EXAMPLE:

- Capacity 200
- Items :

  X1 : v1/w1 = 12, weight : 50

  X2 : v2/w2 = 10, weight : 55

  X3 : v3/w3 = 8, weight : 10

  X4 : v4/w4 = 6, weight : 100

➢ Value of Greedy : 50 * 12 + 55 *10 + 8* 10= 1230

➢ Optimal : 8 * 100 + 55*10 + 8*10= 1430

✓ SO, Greedy Approach does not ensure an optimal solution to the 0-1 Knapsack

To solve 0-1 Knapsack, Dynamic Programming approach is required.

Recursive formula for sub-problems:

3 cases:

1. There are no items in the knapsack, or the weight of the knapsack is 0 - the benefit is 0
2. The weight of $item_i$ exceeds the weight w of the knapsack - $item_i$ cannot be included in the knapsack and the maximum benefit is V[i-1, w]
3. Otherwise, the benefit is the maximum achieved by either not including $item_i$ (i.e., V[i-1, w]),
   or by including $item_i$ (i.e., $V[i-1, w-w_i]+b_i$)

$$V[i,w] = \begin{cases} 0 & \text{for } i = 0 \text{ or } w = 0 \\ V[i-1,w] & \text{if } w_i > w \\ \max\{V[i-1,w], V[i-1,w-w_i] + b_i\} & \text{otherwise} \end{cases}$$

0-1 Knapsack Algorithm

```
for w = 0 to W  // Initialize 1st row to 0's
    V[0,w] = 0
for i = 1 to n  // Initialize 1st column to
    0's
    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]  // w_i > w
```

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 (1)

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

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

Example (2)

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

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)

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

for i = 1 to n
    V[i,0] = 0

Example (4)

Elements (weight, benefit):
(2,3), (3,4), (4,5), (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					

$i=1$

$b_i=3$

$w_i=2$

$w=1$

$w-w_i=-1$

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)

Elements (weight, benefit):
(2,3), (3,4), (4,5), (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

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)

Elements (weight, benefit):
(2,3), (3,4), (4,5), (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

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)

Elements (weight, benefit):
(2,3), (3,4), (4,5), (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					

$i=1$

$b_i=3$

$w_i=2$

$w=4$

$w-w_i=2$

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

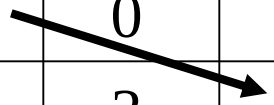
Example (8)

Elements (weight, benefit):
(2,3), (3,4), (4,5), (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					



$i=1$

$b_i=3$

$w_i=2$

$w=5$

$w-w_i=3$

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)

Elements (weight, benefit):
(2,3), (3,4), (4,5), (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					

$i=2$

$b_i=4$

$w_i=3$

$w=1$

$w-w_i=-2$

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)

Elements (weight, benefit):
(2,3), (3,4), (4,5), (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

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

Example (11)

Elements (weight, benefit):
(2,3), (3,4), (4,5), (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

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 + 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 (12)

Elements (weight, benefit):
(2,3), (3,4), (4,5), (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

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)

Elements (weight, benefit):
(2,3), (3,4), (4,5), (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

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)

Elements (weight, benefit):
(2,3), (3,4), (4,5), (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

b_i=5

w_i=4

w= 1..3

$w-w_i$ = -ve

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)

Elements (weight, benefit):
(2,3), (3,4), (4,5), (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

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)

Elements (weight, benefit):
(2,3), (3,4), (4,5), (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

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)

Elements (weight, benefit):
(2,3), (3,4), (4,5), (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	

$i=4$

$b_i=6$

$w_i=5$

$w= 1..4$

$w-w_i$ = -ve

```
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)

Elements (weight, benefit):
(2,3), (3,4), (4,5), (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

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
```

We’re DONE!!

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

- 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, we need to trace back through the table.

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

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 <u>not</u> in the knapsack

Finding the Items {Solution set}

Elements (weight, benefit):
(2,3), (3,4), (4,5), (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
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
        $i = i-1$

x1	x2	x3	x4

Solution set:

i. derived from sequence of **decisions**
ii. Decisions can be taken only when you kept the data ready
iii. Now the table data is ready in hand

Finding the Items (2)

Elements (weight, benefit):
(2,3), (3,4), (4,5), (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

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]* ≠ *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

x1	x2	x3	x4

Obj: I should know, Which object should be included and which object should not be included in the bag?

Finding the Items (2)

Elements (weight, benefit):
(2,3), (3,4), (4,5), (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

k= 5

b_i=6

w_i=5

V[i,k] = 7

V[i–1,k] =7

Let us start with the maximum profit @(4,5) = 7

Need to check, same 7 is in the previous row?
Yes. Means, 7 is not generated because of including 4th object. So Don't include 4th object

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
        $i = i-1$

x1	x2	x3	x4
			0

Finding the Items (3)

Elements (weight, benefit):
(2,3), (3,4), (4,5), (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=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
        $i = i-1$

x1	x2	x3	x4
		0	0

Again,
Need to check, same 7 is in the previous row?
Yes. Means, 7 is not generated because of including 3rd object. So Don't include 3rd object

Finding the Items (4)

Elements (weight, benefit):
(2,3), (3,4), (4,5), (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=2

k= 5

b_i=4

w_i=3

$V[i,k] = 7$

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

$k - w_i$=2

Again,
Need to check, same 7 is in the previous row?
No. Means, 7 is **generated** because of including 2nd object. So include 2nd object

Remaining Profit = 7-4 = **3**

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

$i = i-1$

x1	x2	x3	x4
	1	0	0

Finding the Items (5)

Elements (weight, benefit):
(2,3), (3,4), (4,5), (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=1
k= 2
b_i=3
w_i=2
V[i,k] = 3
V[i–1,k] =0
k – w_i=**0**

Again,
Need to check, same **3** is in the previous row?
No. Means, **3** is **generated** because of including 1st object. So include 1st object

Remaining Profit = **3**-3 = **0**

i=n, **k**=**W**
while i,k > 0
    if *V[i,k]* ≠ *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

x1	x2	x3	x4
1	1	0	0

Finding the Items (6)

Elements (weight, benefit):
(2,3), (3,4), (4,5), (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=0
k= 0

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
        $i = i-1$

The optimal knapsack should contain {1, 2}

x1	x2	x3	x4
1	1	0	0

Finding the Items (7)

Elements (weight, benefit):
(2,3), (3,4), (4,5), (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

The optimal knapsack should contain {1, 2}

```
i=n, k=W
while i,k > 0
    if V[i,k] ≠ 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
```

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
        < the rest of the code >
```

O(W)

Repeat *n* times

O(W)

What is the running time of this algorithm?

O(n*W)

Applications of Knapsack Problem

- Resource allocation with financial constraints
- Construction and Scoring of Heterogenous test
- Selection of capital investments

ANY QUESTIONS ???

THANK YOU