Dynamic Programming - 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 and *W* are integer values)

Problem: How to pack the knapsack to achieve maximum total value of packed items?

Items are indivisible; you either take an item or not, The problem is called a *"0-1"* problem, because each item must be entirely accepted or rejected .

Problem in other words, is to find :

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

Where b_i is the benefit/profit provided by each item i and w_i is the weight of each item i .

Let's first solve this problem with a straightforward algorithm - brute force method:

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

Dynamic programming approach :

We can do better with an algorithm based on dynamic programming, We need to carefully identify the sub-problems

Defining a sub-problem:

Let's try this:

If items are labeled 1..n, then a sub-problem would be to find an optimal solution for S_k = {items labeled 1, 2, .. k}

But, in this case, the final solution final solution (S_n) cannot be described in terms of sub-problems (S_k).

To illustrate this, consider the following example:

Item	**Weight**	**Value**
I0	3	10
I1	8	4
I2	9	9
I3	8	11

The maximum weight the knapsack can hold is 20.

The best set of items from { I0, I1, I2} is {I0, I1, I2}
but the best set of items from {I0, I1, I2, I3} is {I0, I2, I3} .

In this example, note that this optimal solution, {I0, I2, I3} , does NOT build upon the previous optimal solution, {I0, I1, I2} .
(Instead it builds upon the solution, {I0, I2}, which is really the optimal subset of {I0, I1, I2} with weight 12 or less.)

So our definition of a sub-problem is flawed and we need another one!

Let's add another parameter: w, which will represent the maximum weight for each subset of items

The sub-problem then will be to compute V[k,w], i.e., to find an optimal solution for
S_k = {I0, I1, I2..., Ik} in a knapsack of size 'w'

Assuming knowing V[i, j],
where i=0,1, 2, ... k-1, j=0,1,2, ...w,

how to derive V[k,w]?

Recursive Formula for sub-problems:

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

It 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-wk plus the item k

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, since if it was, the total weight would be greater than w, which is unacceptable.

Second case: w_k ≤ w. Then the item k can be in the solution, and we choose the case with greater value.

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
        }
}
```

The running time of the algorithm is : O(n*W), Remember that the brute-force algorithm takes $O(2^n)$

Example of 0/1 Knapsack using dynamic programming:
n = 4 (number of elements)
W = 5 (max weight)

Elements	weight	benefit
I_1	2	3
I_2	3	4
I_3	4	5
I_4	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
for i := 1 to n
        V[i,0] = 0
```

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				
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]** // $w_i > w$

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 + 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:

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

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

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

Likewise, continue calculating until V[4,5]

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

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

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

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.
- Algorithm to choose items that can be included in the final solution:

```
i=n, k=W;
while I,k>0
{
    if V[i,k] ≠ V[i-1,k] then
    i = i-1; k = k-wi; //mark the ith item as in the knapsack
   else
    i = i-1 ;  // Assume the ith  item is not in the knapsack
}
```

Finding the Items (1)

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
```

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

$i=3$
$k=5$
$b_i=5$
$w_i=4$
$V[i,k] = 7$
$V[i-1,k] = 7$

```
i=n, k=W
while i,k > 0
    if V[i,k] ≠ 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=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] ≠ V[i−1,k] then
        mark the ith 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=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] ≠ V[i−1,k] then
        mark the ith 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] ≠ V[i–1,k] then
        mark the nth item as in the knapsack
        i = i–1, k = k-w_i
    else
        i = i–1
```

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 (memorization)
- Running time of dynamic programming algorithm vs. naïve algorithm:

  Knapsack problem: O(W*n) vs. O(2n)