Dynamic Programming

0/1 Knapsack Problem

* **Informal Description:**

- There are 'n' objects, which need to be stored in a knapsack of capacity 'W'.
- Each object 'i' has weight 'w_i' and some value (benefit profit, size) 'v_i'.
- We need to find subset of objects to store such that
(i) Objects have combined size of atmost W.
(ii) The total benefit of the stored object is as large as possible

- We cannot store parts of objects, it is the whole object or nothing.

* **Formal Description:-**
- Given two n-tuples of positive numbers
$\{v_1, v_2 \ldots v_n\}$ and $\{w_1, w_2 \ldots w_n\}$

and $W > 0$, goal is to determine the subset $T \subseteq \{1, 2, \ldots n\}$ (of objects to store) that

$$\text{maximizes } \sum_{i \in T} v_i ,$$

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

* **Developing a DP Algorithm for 0/1 knapsack:**

I] Characterize the structure of optimal solution:
Decompose the problem into smaller problems.

- Construct an array $V[0 \ldots n, 0 \ldots W]$.
- For $1 \le i \le n$ and $0 \le w \le W$, the entry $V[i, w] \rightarrow$ stores the maximum combined benefit value of any subset of object $\{1, 2, \ldots i\}$ of combined size at most w.

- If all the entries of this array is computed, then the entry at $V[n, W]$ will contain the maximum benefit value of objects that can fit into the knapsack.

II] Recursively define the value of an optimal solution.

→ Initial settings :- Set

$$V[0, w] = 0 \quad \text{for } 0 \le w \le W$$
$$V[i, 0] = 0 \quad \text{for } 0 \le i \le n$$
$$V[i, w] = -\infty \quad \text{for } w < 0$$

→ Recursive step :- To compute $V[i, w]$ there is only 2 possibilities,

(i) Leave object i :-
With objects $\{1, 2 \ldots i-1\}$ and storage limit w,
$V[i, w] = V[i-1, w]$.

(ii) Take object i :- (only possible if $w_i \le w$)

- After spending w_i size of knapsack, if we take object 'i', then benefit value gained is 'v_i'.

Scanned by CamScanner

∴ With objects $\{1, 2, \ldots i-1\}$ and storage limits $(w - w_i)$,

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

∴ To summarize,

$$V[i, w] = \max\left(V[i-1, w], v_i + V[i-1, w-w_i]\right)$$

for $1 \le i \le n$ and $1 \le w \le W$

III Compute the maximum benefit value (optimal solution)

- Bottom up computing is used for computing $V[i, w]$
- Bottom :- $V[0, w] = 0$ for all $0 \le w \le W$
  $V[i, 0] = 0$ for all $0 \le i \le n$
- Then table $V[i, w]$ is computed using equation defined in step II in row major form.

$V[i,w]$	$w=0$	1	2	....	W
$i=0$	0	0	0	....	0
1	0				
2	0				
3	0				
⋮	⋮				
n	0				

bottom

up

Algorithm:

```
knapsack_value(v, w, n, W)
O(W)     1.  for w = 0 to W  step +1  do
         2.      V[0, w] = 0
             for i = 1 to n
                 V[i, 0] = 0
O(n)     3.  for i = 1 to n  step +1  do
O(n×W)   4.      for w = 1 to W  step +1  do
         5.          if ((w[i] ≤ w))
                         if (v[i] + V[i-1, w-w[i]] > V[i-1, w]))
         6.                  V[i, w] = v[i] + V[i-1, w-w[i]]
         7.                  keep[i, w] = '✓'
         8.              else
         9.                  V[i, w] = V[i-1, w]
         10.                 keep[i, w] = 'X'
                     else
                         V[i, w] = V[i-1, w]    // wi > w
                         keep[i, w] = 'X'
```

Analysis:

Running time of the procedure is $O(nW)$, since rest all steps from line 5 to 10 can be computed in $O(1)$ time.

[IV] Constructing an optimal solution i.e (object in knapsack)

- To compute the actual subset, an auxillary array keep[i, w] is used in algorithm.

i.e if keep[i, w] = 1, then $i \in T$ and this can be repeated for keep[i-1, W-wi]

else if keep[i, w] = 0, then $i \notin T$ and this can be repeated for keep[i-1, W]

Algorithm

print_knapsack_subset (keep, n, W, w)

$\Theta(n)$

1. for i = n to 1 step -1 do
2. if (W > 0 and keep[i, W] == 'V')
3. print i
4. W = W - w[i]

Analysis :-

Running time of the procedure is $O(n)$

⇒ **Problem 1:** Consider the knapsack capacity W = 8 and total no. of elements 4, as shown in table.

i	v_i	w_i
1	1	2
2	2	3
3	5	4
4	6	5

⇒ **Solution :** n = 4 W = 8

Step 1 : calculation of V and keep table

v_i	w_i	V[i,w]	0	1	2	3	4	5	6	7	8
		0	0	0	0	0	0	0	0	0	0
1	2	1	~~0~~	✗ 0	✓ 1	1	✓ 1	✓ 1	✓ 1	✓ 1	✓ 1
2	3	2	0	✗ 0	✗ 1	✓ ~~2~~	✓ 2	✓ 3	✓ 3	✓ 3	✓ 3
5	4	3	0	✗ 0	✗ 1	✗ ~~2~~	✓ 5	✓ 5	✓ 6	✓ 7	✓ 7
6	5	4	0	✗ 0	✗ 1	✗ 2	✗ 4	✓ 6	✓ 6	✓ 7	✓ 8

∴ Maximum value earned = V [n, W] = 8

Step 2: Printing ~~ete~~ items in knapsack.

∴ Items in the knapsack are 2, 4.

⇒ problem 2

Q Solve the following using 0/1 knapsack

item (i)	value (vi)	weight (wi)
1	18	3
2	25	5
3	27	4
4	10	3
5	15	6

with knapsack capacity 12.

Vi	Wi	V[i,w]	0	1	2	3	4	5	6	7	8	9	10	11	12
		0	0	0	0	0	0	0	0	0	0	0	0	0	0
18	3	1	0	× 0	× 0	✓ 18	✓ 18	✓ 18	✓ 18	✓ 18	✓ 18	✓ 18	✓ 18	✓ 18	✓ 18
25	5	2	0	× 0	× 0	× 18	× 18	✓ 25	✓ 25	✓ 25	✓ 43	✓ 43	✓ 43	✓ 43	✓ 48
27	4	3	0	× 0	× 0	× 18	✓ 27	✓ 27	✓ 27	✓ 45	✓ 45	✓ 52	✓ 52	✓ 52	✓ 70
10	3	4	0	× 0	× 0	× 18	× 27	× 27	✓ 28	× 45	× 45	× 52	✓ 55	✓ 55	× 70
15	6	5	0	× 0	× 0	× 18	× 27	× 27	× 28	× 45	× 45	× 52	× 55	× 55	× 70

∴ Maximum value earned $= V[n, W] = 70$.

<u>step2</u> :- Printing items in knapsack.

Items are 3, 2, 1