Knapsack Problem using Dynamic Programming

.

Knapsack Problem using Dynamic Programming

- **Problem :** Given a set of items, each having different weight and value or profit associated with it. Find the set of items such that the total weight is less than or equal to a capacity of the knapsack and the total value earned is as large as possible.
- The knapsack problem is useful in solving resource allocation problem. Let X = < x_1, x_2, x_3, ... , x_n> be the set of n items. Sets W = <w_1, w_2, w_3, . . . , w_n> and V = < v_1, v_2, v_3, . . . , v_n> are weight and value associated with each item in X. Knapsack capacity is M unit.
- The knapsack problem is to find the set of items which maximizes the profit such that collective weight of selected items does not cross the knapsack capacity.
- Select items from X and fill the knapsack such that it would maximize the profit. Knapsack problem has two variations. 0/1 knapsack, that does not allow breaking of items. Either add an entire item or reject it. It is also known as a binary knapsack. Fractional knapsack allows breaking of items. Profit will be earned proportionally.

Mathematical formulation

We can select any item only ones. We can put items x_i in knapsack if knapsack can accommodate it. If the item is added to a knapsack, the associated profit is accumulated.
Knapsack problem can be formulated as follow :
Maximize

$$sum_{ni=1} v_i x_i$$

subjected to

$$sum_{ni=1} w_i x_i leM$$

$$x_i in (0,1)$$

for binary knapsack

$$x_i in [0,1]$$

for fractional knapsack
We will discuss two approaches for solving knapsack using dynamic programming.

First Approach for Knapsack Problem using Dynamic Programming

If the weight of the item is larger than the remaining knapsack capacity, we skip the item, and the solution of the previous step remains as it is. Otherwise, we should add the item to the solution set and the problem size will be reduced by the weight of that item. Corresponding profit will be added for the selected item.

Dynamic programming divides the problem into small sub-problems. Let V is an array of the solution of sub-problems. V[i, j] represents the solution for problem size j with first i items. The mathematical notion of the knapsack problem is given as :

$$V[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ V[i-1, j] & \text{if } j < wi \\ \max \{V[i-1, j], vi + V[i-1, j-wi]\} & \text{if } j \geq w \end{cases}$$

V [1 …. n, 0 … M] : Size of the table

V (n, M) = Solution

n = Number of items

Algorithm

Algorithm for binary knapsack using dynamic programming is described below :

```
Algorithm DP_BINARY_KNAPSACK (V, W, M)
// Description: Solve binary knapsack problem using
dynamic programming
// Input: Set of items X, set of weight W, profit
of items V and knapsack capacity M
// Output: Array V, which holds the solution of
problem

for i ← 1 to n do
    V[i, 0] ← 0
end

for i ← 1 to M do
    V[0, i] ← 0
end

for V[0, i] ← 0 do
    for j ← 0 to M do
        if w[i]  ≤ j then
```

```
            V[i, j] ← max{V[i-1, j],  v[i]  + V[i -
1, j -  w[i]]}
        else
            V[i, j] ← V[i - 1, j]    // w[i]>j
        end
    end
end
```

The above algorithm will just tell us the maximum value we can earn with dynamic programming. It does not speak anything about which items should be selected. We can find the items that give optimum result using the following algorithm

```
Algorithm TRACE_KNAPSACK(w,  v,  M)
// w is array of weight of n items
// v is array of value of n items
// M is the knapsack capacity

SW ← { }
SP  ← { }
i ← n
j ← M

while ( j> 0 )  do
   if (V[i, j] == V[i - 1, j]) then
       i ← i - 1
   else
       V[i,  j] ← V[i, j] - vi
       j ← j - w[i]
       SW ← SW +w[i]
```

```
        SP ← SP +v[i]
    end
end
```

Complexity analysis

With n items, there exist 2^n subsets, the brute force approach examines all subsets to find the optimal solution. Hence, the running time of the brute force approach is $O(2^n)$. This is unacceptable for large n.
Dynamic programming finds an optimal solution by constructing a table of size n ´ M, where n is a number of items and M is the capacity of the knapsack. This table can be filled up in O(nM) time, same is the space complexity.

- Running time of Brute force approach is $O(2^n)$.
- Running time using dynamic programming with memorization is O(n * M).

Example

Example: Find an optimal solution for following 0/1 Knapsack problem using dynamic programming: Number of objects n = 4, Knapsack Capacity M = 5, Weights (W_1, W_2, W_3, W_4) = (2, 3, 4, 5) and profits (P_1, P_2, P_3, P_4) = (3, 4, 5, 6).
Solution:
Solution of the knapsack problem is defined as,

$$V[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ V[i-1, j] & \text{if } j < wi \\ \max \{V[i-1, j], vi + V[i-1, j-wi]\} & \text{if } j \geq w \end{cases}$$

We have the following stats about the problem,

Item	Weight (w_i)	Value (v_i)
I_1	2	3
I_2	3	4
I_3	4	5
I_4	5	6

- Boundary conditions would be V [0, i] = V[i, 0] = 0. Initial configuration of table looks like.

	j →							
	Item Detail		**0**	**1**	**2**	**3**	**4**	**5**
i=0			0	0	0	0	0	0
i=1	w_1=2	v_1=3	0					
i=2	w_2=3	v_2=4	0					
i=3	w_3=4	v_3=5	0					
i=4	w_4=5	v_4=6	0					

Filling first column, j = 1

V [1, 1] ⇒ i = 1, j = 1, $w_i = w_1 = 2$
As, $j < w_i$, V [i, j] = V [i – 1, j]
V [1, 1] = V [0, 1] = 0

V [2, 1] ⇒ i = 2, j = 1, $w_i = w_2 = 3$
As, $j < w_i$, V [i, j] = V [i – 1, j]
V [2, 1] = V [1, 1] = 0

V [3, 1] ⇒ i = 3, j = 1, $w_i = w_3 = 4$
As, $j < w_i$, V [i, j] = V [i – 1, j]
V [3, 1] = V [2, 1] = 0

V [4, 1] ⇒ i = 4, j = 1, $w_i = w_4 = 5$
As, $j < w_i$, V[i, j] = V[i – 1, j]
V [4, 1] = V [3, 1] = 0

Filling first column, j = 2

V[1, 2] ⇒ i = 1, j = 2, $w_i = w_1 = 2$, $v_i = 3$

As, $j \geq w_i$, $V[i, j] = \max \{V[i-1, j], v_i + V[i-1, j-w_i]\}$

$= \max \{V[0, 2], 3 + V[0, 0]\}$

$V[1, 2] = \max(0, 3) = 3$

V[2, 2] ⇒ i = 2, j = 2, $w_i = w_2 = 3$, $v_i = 4$

As, $j < w_i$, $V[i, j] = V[i-1, j]$

$V[2, 2] = V[1, 2] = 3$

V[3, 2] ⇒ i = 3, j = 2, $w_i = w_3 = 4$, $v_i = 5$

As, $j < w_i$, $V[i, j] = V[i-1, j]$

$V[3, 2] = V[2, 2] = 3$

V[4, 2] ⇒ i = 4, j = 2, $w_i = w_4 = 5$, $v_i = 6$

As, $j < w_i$, $V[i, j] = V[i-1, j]$

$V[4, 2] = V[3, 2] = 3$

Filling first column, j = 3

V[1, 3] ⇒ i = 1, j = 3, $w_i = w_1 = 2$, $v_i = 3$

As, $j \geq w_i$, $V[i, j] = \max \{V[i-1, j], v_i + V[i-1, j-w_i]\}$

$= \max \{V[0, 3], 3 + V[0, 1]\}$

$V[1, 3] = \max(0, 3) = 3$

V[2, 3] ⇒ i = 2, j = 3, $w_i = w_2 = 3$, $v_i = 4$

As, $j \geq w_i$, $V[i, j] = \max \{V[i-1, j], v_i + V[i-1, j-w_i]\}$

$= \max \{V[1, 3], 4 + V[1, 0]\}$

$V[2, 3] = \max(3, 4) = 4$

V[3, 3] ⇒ i = 3, j = 3, $w_i = w_3 = 4$, $v_i = 5$

As, $j < w_i$, $V[i, j] = V[i-1, j]$

$V[3, 3] = V[2, 3] = 4$

V[4, 3] ⇒ i = 4, j = 3, $w_i = w_4 = 5$, $v_i = 6$

As, $j < w_i$, $V[i, j] = V[i - 1, j]$
$V[4, 3] = V[3, 3] = 4$

Filling first column, j = 4
V[1, 4] $\Rightarrow$ $i = 1, j = 4, w_i = w_1 = 2, v_i = 3$
As, $j \geq w_i$, $V[i, j] = \max \{V[i - 1, j], v_i + V[i - 1, j - w_i]\}$
$= \max \{V[0, 4], 3 + V[0, 2]\}$

$V[1, 4] = \max (0, 3) = 3$

V[2, 4] $\Rightarrow$ $i = 2, j = 4, w_i = w_2 = 3, v_i = 4$
As, $j \geq w_i$, $V[i, j] = \max \{V[i - 1, j], v_i + V[i - 1, j - w_i]\}$
$= \max \{V[1, 4], 4 + V[1, 1]\}$

$V[2, 4] = \max (3, 4 + 0) = 4$

V[3, 4] $\Rightarrow$ $i = 3, j = 4, w_i = w_3 = 4, v_i = 5$
As, $j \geq w_i$, $V[i, j] = \max \{V[i - 1, j], v_i + V[i - 1, j - w_i]\}$
$= \max \{V[2, 4], 5 + V[2, 0]\}$

$V[3, 4] = \max (4, 5 + 0) = 5$

V[4, 4] $\Rightarrow$ $i = 4, j = 4, w_i = w_4 = 5, v_i = 6$
As, $j < w_i$, $V[i, j] = V[i - 1, j]$
$V[4, 4] = V[3, 4] = 5$

Filling first column, j = 5
V [1, 5] $\Rightarrow$ $i = 1, j = 5, w_i = w_1 = 2, v_i = 3$
As, $j \geq w_i$, $V[i, j] = \max \{V[i - 1, j], v_i + V[i - 1, j - w_i]\}$
$= \max \{V[0, 5], 3 + V[0, 3]\}$

$V[1, 5] = \max (0, 3) = 3$

V[2, 5] $\Rightarrow$ $i = 2, j = 5, w_i = w_2 = 3, v_i = 4$
As, $j \geq w_i$, $V[i, j] = \max \{V[i - 1, j], v_i + V[i - 1, j - w_i]\}$
$= \max \{V[1, 5], 4 + V[1, 2]\}$

V[2, 5] = max (3, 4 + 3) = 7

V[3, 5] ⇒ i = 3, j = 5, $w_i = w_3 = 4$, $v_i = 5$
As, $j \geq w_i$, V [i, j] = max {V [i – 1, j], v_i + V [i – 1, j – w_i] }
= max {V [2, 5], 5 + V [2, 1]}

V[3, 5] = max (7, 5 + 0) = 7

V [4, 5] ⇒ i = 4, j = 5, $w_i = w_4 = 5$, $v_i = 6$
As, $j \geq w_i$, V [i, j] = max {V [i – 1, j], v_i + V [i – 1, j – w_i] }
= max {V [3, 5], 6 + V [3, 0]}

V[4, 5] = max (7, 6 + 0) = 7

Final table would be,

	j→							
	Item Detail		**0**	**1**	**2**	**3**	**4**	**5**
i=0			0	0	0	0	0	0
i=1	w_1=2	v_1=3	0	0	3	3	3	3
i=2	w_2=3	v_2=4	0	0	3	4	4	7
i=3	w_3=4	v_3=5	0	0	3	4	5	7
i=4	w_4=5	v_4=6	0	0	3	4	5	7

Find selected items for M = 5
Step 1 : Initially, i = n = 4, j = M = 5

			j →					
	Item Detail		**0**	**1**	**2**	**3**	**4**	**5**
i=0			0	0	0	0	0	0
i=1	$w_1 = 2$	$v_1 = 3$	0	0	3	3	3	3
i=2	$w_2 = 3$	$v_2 = 4$	0	0	3	4	4	7
i=3	$w_3 = 4$	$v_3 = 5$	0	0	3	4	5	7
i=4	$w_4 = 5$	$v_4 = 6$	0	0	3	4	5	7

V[i, j] = V[4, 5] = 7

V[i – 1, j] = V[3, 5] = 7

V[i, j] = V[i – 1, j], so don’t select i^{th} item and check for the previous item.
so i = i – 1 = 4 – 1 = 3

Solution Set S = { }

Step 2 : i = 3, j = 5

			j →					
	Item Detail		**0**	**1**	**2**	**3**	**4**	**5**
i=0			0	0	0	0	0	0
i=1	$w_1 = 2$	$v_1 = 3$	0	0	3	3	3	3
i=2	$w_2 = 3$	$v_2 = 4$	0	0	3	4	4	7
i=3	$w_3 = 4$	$v_3 = 5$	0	0	3	4	5	7
i=4	$w_4 = 5$	$v_4 = 6$	0	0	3	4	5	7

V[i, j] = V[3, 5] = 7

V[i – 1, j] = V[2, 5] = 7

V[i, j] = V[i – 1, j], so don’t select i^{th} item and check for the previous item.
so i = i – 1 = 3 – 1 = 2

Solution Set S = { }

Step 3 : i = 2, j = 5

			j →					
	Item Detail		**0**	**1**	**2**	**3**	**4**	**5**
i=0			0	0	0	0	0	0
i=1	$w_1 = 2$	$v_1 = 3$	0	0	3	3	3	3
i=2	$w_2 = 3$	$v_2 = 4$	0	0	3	4	4	7
i=3	$w_3 = 4$	$v_3 = 5$	0	0	3	4	5	7
i=4	$w_4 = 5$	$v_4 = 6$	0	0	3	4	5	7

V[i, j] = V[2, 5] = 7

V[i – 1, j] = V[1, 5] = 3

V[i, j] ≠ V[i – 1, j], so add item Ii = I_2 in solution set.
Reduce problem size j by w_i
j = j – w_i = j – w_2 = 5 – 3 = 2
i = i – 1 = 2 – 1 = 1

Solution Set S = {I_2}
Step 4 : i = 1, j = 2

			j →					
	Item Detail		**0**	**1**	**2**	**3**	**4**	**5**
i=0			0	0	0	0	0	0
i=1	$w_1 = 2$	$v_1 = 3$	0	0	3	3	3	3
i=2	$w_2 = 3$	$v_2 = 4$	0	0	3	4	4	7
i=3	$w_3 = 4$	$v_3 = 5$	0	0	3	4	5	7
i=4	$w_4 = 5$	$v_4 = 6$	0	0	3	4	5	7

$V[1, j] = V[1, 2] = 3$

$V[i – 1, j] = V[0, 2] = 0$

$V[i, j] \neq V[i – 1, j]$, so add item $Ii = I_1$ in solution set.
Reduce problem size j by w_i
$j = j – w_i = j – w_1 = 2 – 2 = 0$
Solution Set $S = \{I_1, I_2\}$
Problem size has reached to 0, so final solution is
$S = \{I_1, I_2\}$ Earned profit $= P_1 + P_2 = 7$

Second Approach: Set Method for Knapsack Problem using Dynamic Programming

- The first approach is suitable when knapsack capacity is small. With large knapsack, the first approach is not advisable from computation as well as memory requirement point of view.
- The secondapproach discussed below is more suitable when problem instance is large.
- In forward approach, dynamic programming solves knapsack problem as follow:

1. Set $S^0 = \{(0, 0)\}$
2. $S_1^i = \{(p, w) \mid (p – p_i) \in S^i, (w – w_i) \in S^i \}$

We can obtain S^{i+1} by invoking x_{i+1}

- If $x_i = 0$ (item x_i is excluded) then $S_1^i = S^i$
- If $x_i = 1$ (item x_i is included) then S_1^i is computed by adding (p_{i+1}, w_{i+1}) in each state of S^i.

1. $S^{i+1} = MERGE\_PURGE(S^i, S_1^i)$. MERGE_PURGE does following:

For two pairs $(p_x, w_x) \in S^{i+1}$ and $(p_y, w_y) \in S^{i+1}$, if $p_x \leq p_y$ and $w_x \geq w_y$, we say that (p_x, w_x) is dominated by (p_y, w_y). And the pair (p_x, w_x) is discarded.

It also purges all the pairs (p, w) from S^{i+1} if $w > M$, i.e. it weight exceeds knapsack capacity.

2. Repeat step 1 n times

3. $f_n(M) = S^n$. This will find the solution of KNAPSACK(1, n, M).
4. for each pair $(p, w) \in S^n$

if (p, w) $\in S^{n-1}$, then set $x_n = 0$

if (p, w) $\notin S^{n-1}$, then set $x_n = 1$, update $p = p - x_n$ and $w = w - w_n$

Algorithm

Algorithm

```
DP_KNAPSACK(X, P, W, M)

// Description: Solve knapsack problem using
dynamic programming

// Input: Set of items X, profit P, weight W and
knapsack capacity M

// Output: Vector x indicating selection/rejection
of items.



S
```

0

```
= { (0, 0) }
```

for

```
i ← 0 to n - 1
```

do

```


  S
```

i

1

```
 ← {(p, w) | for each (x, y) ∈ Si, p = x + p
```

i+1

```
, w = y + w
```

i+1

```
 }

  S
```

1

i+1

```
 ← MERGE_PURGE(S
```

i

```
,S
```

1

i

```
 )
```

end

```

```

for

```
i ← n to 1
```

do

```


  Select last pair (p
```

x

```
, w
```

x

```
) ∈ S
```

n

```

```

if

```
(p
```

x

```
, w
```

x

```
) ∈ S
```

n–1

```

```

then

```
    x
```

i

```
← 0
```

else

```
    x
```

i

```
← 1
```

end

```

```

end

Examples

Example: Solve the instance of 0/1 knapsack problem using dynamic Programming : n = 4, M = 25, (P_1, P_2, P_3 P_4) = (10, 12, 14, 16), (W_1, W_2, W_3, W_4) = (9, 8, 12, 14)

Solution:

Knapsack capacity is very large, i.e. 25, so tabular approach won't be suitable. We will use the set method to solve this problem

Initially, S^0 = { (0, 0) }

Iteration 1:

Obtain S_1^0 by adding pair (p_1, w_1) = (10, 9) to each pair of S^0

$S_1^0 = S^0 + (10, 9) = \{(10, 9)\}$

Obtain S^1 by merging and purging S^0 and S_1^0

S^1 = MERGE_PURGE (S^0, S_1^0)

= { (0, 0), (10, 9) }

Iteration 2:

Obtain S_1^1 by adding pair (p_2, w_2) = (12, 8) to each pair of S^1

$S_1^1 = S^1 + (12, 8) = \{(12, 8), (22, 17)\}$

Obtain S^2 by merging and purging S^1 and S_1^1

S^2 = MERGE_PURGE(S^1, S_1^1)

= { (0, 0), (12, 8), (22, 17) }

Pair (10, 9) is discarded because pair (12, 8) dominates (10, 9)

Iteration 3:

Obtain S_1^2 by adding pair (p_3, w_3) = (14, 12) to each pair of S^2

$S_1^2 = S^2 + (14, 12)$

= { (14, 12), (26, 20), (36, 29) }

Obtain S^3 by merging and purging S^2 and S_1^2 .

S^3 = MERGE_PURGE (S^2, S_1^2)

= { (0, 0), (12, 8), (22, 17), (14, 12), (26, 20) }

Pair (36, 29) is discarded because its w > M

Iteration 4:

Obtain S_1^3 by adding pair (p_4, w_4) = (16, 14) to each pair of S^3

$S_1^3 = S^3 + (16, 14)$

= { (16, 14), (28, 22), (38, 31), (30, 26), (42, 34) }

Obtain S^4 by merging and purging S^3 and S_1^3.

S^4 = MERGE_PURGE (S^3, S_1^3)

= { (0, 0), (12, 8), (14, 12), (16, 14), (22, 17), (26, 20), (28, 22) }

Pair (38, 31), (30, 26) ,and (42, 34) are discarded because its w > M

Find optimal solution
Here, n = 4.

Start with the last pair in S^4, i.e. (28, 22)
(28, 22) ∈ S^4 but (28, 22) ∉ S^3
So set $x_n = x_4 = 1$
Update,

$p = p - p_4 = 28 - 16 = 12$
$w = w - w_4 = 22 - 14 = 8$
$n = n - 1 = 4 - 1 = 3$

Now n = 3, pair (12, 8) ∈ S^3 and (12, 8) ∈ S^2
So set $x_n = x_3 = 0$
$n = n - 1 = 3 - 1 = 2$

Now n = 2, pair(12, 8) ∈ S^2 but (12, 8) ∉ S^1
So set $x_n = x_2 = 1$
Update,

$p = p - p_2 = 12 - 12 = 0$
$w = w - w_2 = 8 - 8 = 0$
Problem size is 0, so stop.

Optimal solution vector is $(x_1, x_2, x_3, x_3) = (0, 1, 0, 1)$ Thus, this approach selects pair (12, 8) and (16, 14) which gives profit of 28.

Example: Generate the sets S^i, $0 \leq i \leq 3$ for following knapsack instance. N = 3, $(w_1, w_2, w_3) = (2, 3, 4)$ and $(p_1, p_2, p_3) = (1, 2, 5)$ with M = 6. Find optimal solution.

Solution:
Initially, $S^0 = \{(0, 0)\}$

Iteration 1:
Obtain S_1^0 by adding pair $(p_1, w_1) = (1, 2)$ to each pair of S^0
$S_1^0 = S^0 + (1, 2) = \{(1, 2)\}$
Obtain S^1 by merging and purging S^0 and S_1^0

S^1 = MERGE_PURGE (S^0, S_1^0)
= { (0, 0), (1, 2) }

Iteration 2:
Obtain S_1^1 by adding pair (p_2, w_2) = (2, 3) to each pair of S^1
S_1^1 = S^1 + (2, 3) = {(2, 3), (3, 5)}
Obtain S^2 by merging and purging S^1 and S_1^1
S^2 = MERGE_PURGE(S^1, S_1^1)
= { (0, 0), (1, 2), (2, 3), (3, 5) }

Iteration 3:
Obtain S_1^2 by adding pair (p_3, w_3) = (5, 4) to each pair of S^2
S_1^2 = S^2 + (5, 4) = {(5, 4), (6, 6), (7, 7), (8, 9) }
Obtain S^3 by merging and purging S^2 and S_1^2 .
S^3 = MERGE_PURGE (S^2, S_1^2)
= { (0, 0), (1, 2), (2, 3), (5, 4), (6, 6) }

Pair (7, 7) and (8, 9) are discarded because their w > M

Pair (3, 5) is discarded because pair (5, 4) dominates (3, 5)

Find optimal solution:
Here, n = 3.

Start with the last pair in S^3, i.e. (6, 6)
(6, 6) $\in S^3$ but (6, 6) $\notin S^2$
So set x_n = x_3 = 1
Update,

p = p – p_3 = 6 – 5 = 1
w = w – w_3 = 6 – 4 = 2
Now n = 2, pair(1, 2) $\in S^2$ and (1, 2) $\in S^1$
So set x_n = x_2 = 0
Now n = 1, pair(1, 2) $\in S^1$ but (1, 2) $\notin S^0$
So set x_n = x_1 = 1
Optimal solution vector is (x_1, x_2, x_3) = (1, 0, 1)

Thus, this approach selects pair (1, 2) and (5, 4)