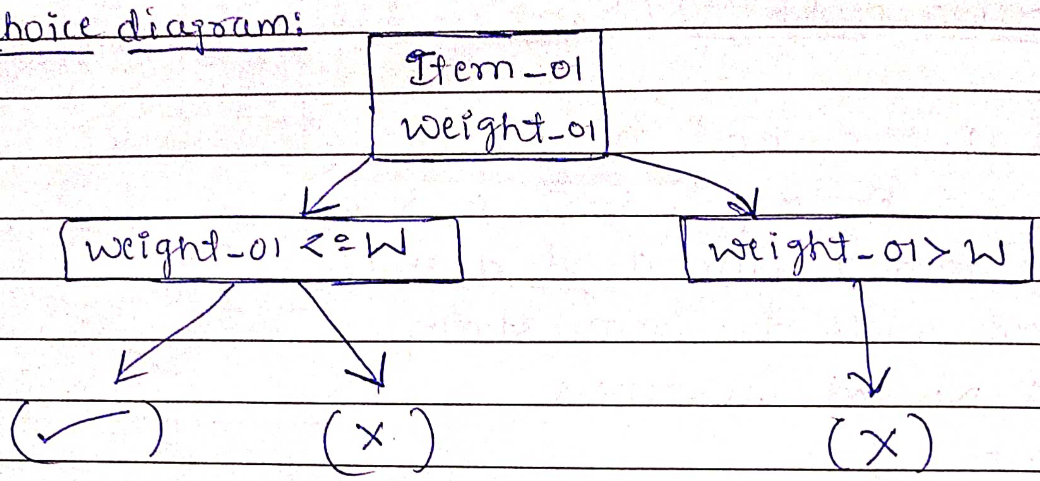
→ 0-1 KnapSack : [Recursive function]

Given weights and value of n items, put these items in a knapsack of capacity W, to get the maximum total value in the knapsack.

Given : W : capacity of knapsack.
n : Number of items.
val[0....n-1] : values associated with n items
weight[0....n-1] : weight —— " —— " ——

* The Base condition : The Best way to choose a Base condition is to check the smallest valid input, In this case, we can have 2 options i.e, if n==0 which number of items is zero, then what we can include in knapsack, nothing we can, Similarly, if the weight W==0 the capacity of knapsack is only 0 then we cannot include or add even a single item, so inboth of these cases the maxProfit will be 0

* Choice diagram:



In the above diagram, A item has a weight & value associated with it. For every item we have 2 choices,

1. If the weight of the Item <= than the weight of the Knapsack. Again, here also we have 2 choices to include it in knapsack or not.
2. If the weight of the Item > then the weight of knapsack than obviously we cannot include/add it into the Knapsack.

* Recursive Function:

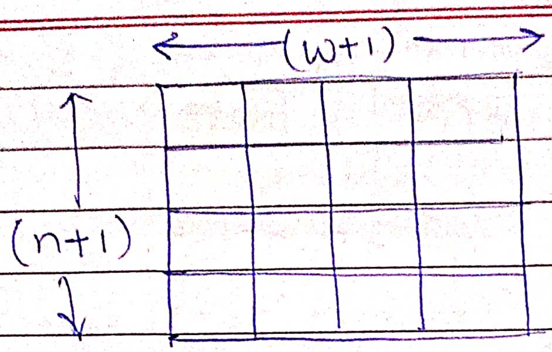
```
int knapSack(int W, int val[], int weight[], int n)
{
    if (n==0 || W==0)
        return 0;
    else if (weight[n-1] <= W)
        return max(val[n-1] + knapsack(W-weight[n-1],
                       val, weight, n-1),
                   knapsack(W, val, weight, n-1));
    else if (weight[n-1] > W)
        return knapsack(W, val, weight, n-1);
}
```

- Time complexity : $O(2^n)$ As it checks for each subset
- Auxiliary space : $O(1)$

- Memoization:

```
int t[n][W];
memset(t, -1, sizeof(t));
int knapsack(int W, int val[], int weight[], int n)
{
    if(n==0 || W==0)
        return 0;
    if(t[n][W] != -1)
        return t[n][W];
    else if(weight[n-1] <= W)
    {
        t[n][W] = max(val[n-1]+ knapsack(W-weight[n-1],
                      val, weight, n-1), knapsack(W, val,
                      weight, n-1));
        return t[n][W];
    }
    else if(weight[n-1] > W)
    {
        t[n][W] = knapsack(W, val, weight, n-1);
        return t[n][W];
    }
}
```

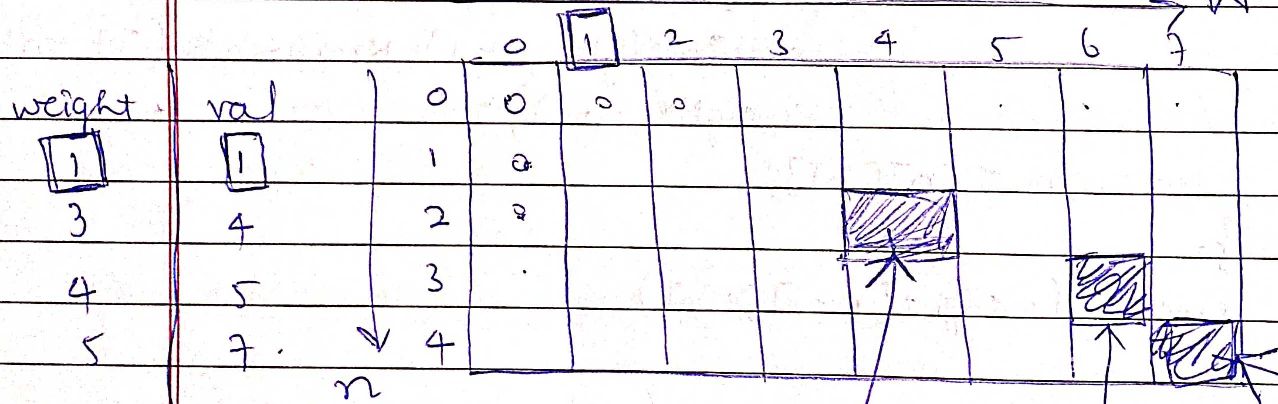
- Time complexity of above memoized function is : $O(N*M)$
- Space complexity of above function is : $O(N*M)$



- **Top Down DP:** Here, in Top down DP, the Base condition of Recursion becomes the initialization.

ex: val[] = [1, 4, 5, 7] W = 7
weight[] = [1, 3, 4, 5] n = 4.

so, we have to create a 2D matrix of size dp[n+1][W+1];



This value is a solution of subproblem i.e.,
when Knapsack capacity is 4.
W = 4
val[] = [1, 4] and
weight[] = [1, 3]

W = 6
val[] = [1, 4, 5]
weight[] = [1, 3, 4]

W = 7
val[] = [1, 4, 5, 7]
weight[] = [1, 3, 4, 5]

The Base condition of Recursion becomes the initialization of dp[][], i.e, when n==0 or W==0. Initialize the dp[0][0] matrix with 0.

(j) → W

n ↓ (i)	W=0	W=1	W=2	W=3
n=0	0	0	0	0
	0			
	0			
	0			

→ It means that we may have the capacity of Bag but no items present as $\boxed{n=0}$

↓

It means it does matter how much elements/items we can include when we have bag capacity as 0. We cannot include a single it. so, it is 0.

```
int knapsack(int W, int val[], int weight[], int n)
{  int dp[n+1][W+1];
   for(int i=0; i<n+1; i++)
      for(int j=0; j<W+1; j++)
         if(i==0 || j==0)
            dp[i][j] = 0;

   for(int i=1; i<n+1; i++)
      for(int j=1; j<W+1; j++)
      {
         if(weight[i-1] <= j)
            t[i][j] = max(val[i-1] + t[i-1][j-weight[i-1]], t[i-1][j]);
         else
            t[i][j] = t[i-1][j];
      }  return dp[n][W];
}
```

0/1 Knapsack : Bottom Up [Tabulation]

```
int knapsack(int W, int wt[], int val[],
                int n)
{
    int DP[n+1][W+1];
    for(int i=0; i<=n; i++)
        for(int j=0; j<=W; j++)
            if(i==0 || j==0)
                DP[i][j] = 0;
                                        Initialization

    for(int i=1; i<=n; i++)
        for(int j=1; j<=W; j++)
        {
            if(wt[i-1] <= j)
                DP[i][j] = max(val[i-1] +
                        DP[i-1][j-wt[i-1]],
                        DP[i-1][j]);
            else
                DP[i][j] = DP[i-1][j];
        }

    return DP[n][W];
}
```

=> Tracing of the Above code with the following input data :

max profit: 3

W = 4 , wt =

4	5	1
1	2	3

n = 3 , val = (second row above)

DP :

n↓ \ W→ j	0	1	2	3	4
i 0	0	0	0	0	0
1	0	0	0	0	1
2	0	0	0	0	0
3	0	3	3	3	3

i = 1
j = ~~1~~, 2, ~~3~~, 4, 5

wt[i-1] => wt[0] = 4 <= j => 4 <= 1
↓
false, so,

DP[i][j] = 0
Again, 4 <= 2, false DP[1][2] = 0
4 <= 3, false DP[1][3] = 0
4 <= 4, true. max(1, 0) = 1

val[i-1] => val[1-1] => val[0] = (1 + 0, 0)
DP[i-1][j-wt[i-1]] => DP[0][0] = 0
↓ ↓
[0][4-4] = [0][0] DP[0][4] = 0
wt[i-1] => wt[0] = 4

classmate

Date
Page 3

$i = 2$
$j = \cancel{1}, \cancel{2}, \cancel{3}, \cancel{4}, 5.$

wt[i-1] => wt[1] = 5 <= 1 false, DP[2][1] = 0
5 <= 2 " DP[2][2] = 0
5 <= 3 " DP[2][3] = 0
5 <= 4 " DP[2][4] = 0

$i = 3$
$j = \cancel{1}, \cancel{2}, \cancel{3}, \cancel{4}, 5$

(i) wt[i-1] => wt[3-1] => wt[2] = 1

1 <= 1, true,

max(val[i-1] + DP[i-1][j - wt[i-1]],
DP[i-1][j]);

val[i-1] => val[3-1] => val[2] = 3
DP[i-1][j-wt[i-1]]
↓ ↓ → 3-1 = 2 => wt[2] = 1
[2] [1-1] => DP[2][0] = 0
DP[2][1] = 0.

max(0+3, 0) = 3

(ii) K = 2, true → wt[i-1] = 1 <= 3

val[i-1] = 3.
DP[i-1][j-wt[i-1]]
[2][2-1] => DP[2][1] = 0

DP[i-1][j] => DP[2][2]
0.

max(3+0, 0) = 3.

(iii) K = 3, true. wt[i-1] = 1 <= 3.

val[i-1] = 3
DP[i-1][j-wt[i-1]]
[2][3-1] => DP[2][2] = 0

DP[i-1][j] = [2][3] = 0

max(3+0, 0) = 3, i = 3

(iv) K = 4, wt[i-1] => 1 <= 3 true.

val[i-1] = 3
DP[i-1][j-wt[i-1]]
↓
[2][4 - wt[3-1]] = DP[2][4-1]
DP[2][3] => 0

DP[i-1][j] = DP[2][4]
↓
0.

max(3+0, 0) = 3

return DP[n][W] = 3