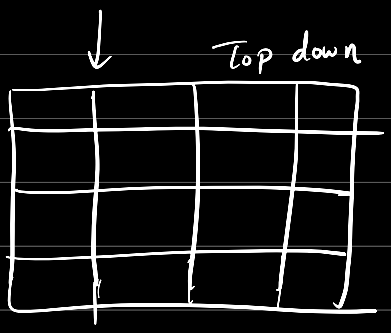
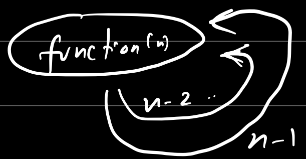
Topic :- Knapsack Poo blem using Dynamic Programming & Recursion

What is DP:- Enhanced Recursion

Top down

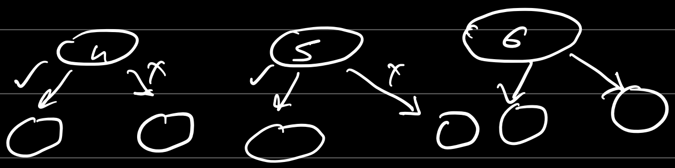


Instead of calling function everytime store it



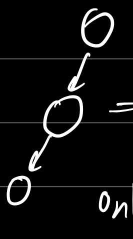
How to identify if its DP:-

① choice ⟹

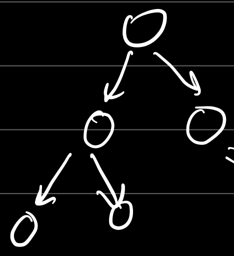


[whenever we have option to choice we can use recursion, and if we have overlapping issue use DP]

Overlapping may Occure when you have 2 choices



=> one choice no DP Only recursion



⟹ Can be solved using recursion

⟹ Can be solved using DP

② Optimal [max, min ...]

Recursion ⟹ DP solution.

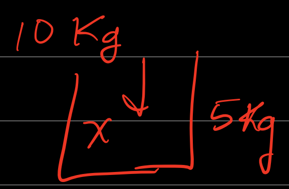
Knapsack using Recursion

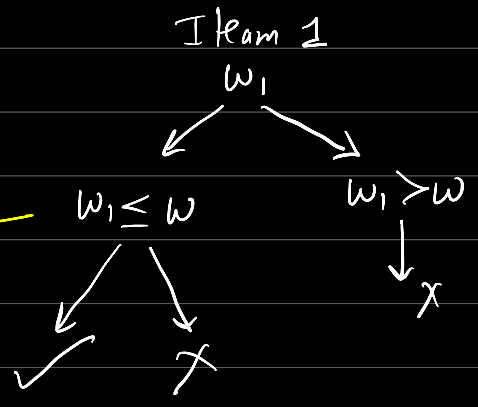
Statement:- Given a set of n iteams from 1 to n, each with a weight wi and value vi, along with the maximum weight capacity W, maximize the sum of the values of the iteams in the Knapsack so that the sum of the weights is less than or equal to the Knapsack's capacity.

I.P:- weight[] : | 1 | 3 | 4 | 5 |
value [] | 1 | 4 | 5 | 7 |
W : 7

Output:- maximum Profit

for Recursive Code ⇒ choice Diagram





if it's less we have choice

When weight of iteam is higher than weight of bag we can't include it

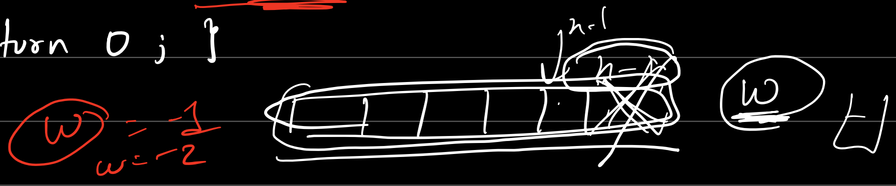
Algorithm:-

```
int Knapsack(int wt [], int val [], int W, int n)
{
```

Size of weight array

// base condition :- think about the smallest valid condition }

```
if (n==0 || w==0)
{ return 0; }
```



// logic condition ⟹ choice diagram

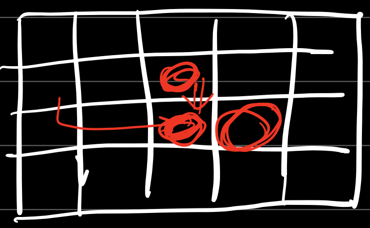
//we have 2 cases ① $w_1 \le w$ $w_1 > w$

```
if (wt[n-1] <= w) {
return max [val[n-1] + Knapsack[wt, val, w - wt[n-1], n-1],
                        Knapsack[wt, val, w, n-1)] ];
}
else {      // w1 > w
    return Knapsack(wt, val, w, n-1);
    }
}
```

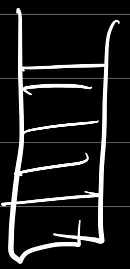
here we included the item

DP to Solve KnapSack problem.

Top-down →



Better → ?



error

Stack overflow (Recursion)

Step1 :- Initialization

Step 2 :- Recursivecall change it to Iterative call

Example:- wt = [1, 2, 3]

values = [6, 10, 12]

W = 5

wt	value	j ⟹ weights ⟶ ↓ n	0	1	2	3	4	5
0	0	0	0	0	0 ⇓	0	0	0
1	6	1	0	6	6	6	6	6
2	10	2	0	6	Max (6, 10+0)	6, 10+6=16	6, 10+6=16	6, 10+6=16
3	12	3	0	6	10	16, 12+0 = 16	16, 12+6 =18	16, 12+10 = 22 ⇓ final outpot

$t[3][4] = t[2][4-wt[2]]$
$\rightarrow [2][4-2]$
$[2][2]$

```
for (int i=1; i<n+1; i++)
{ for (int j=1; j<n+1; j++)
  {
    if (wt[i-1]<=j)
  t[i][j] = Max (val [i-1] + t [i-1] [j-wt [i-1]],
                                t [i-1][j] )
    else
      t [i][j] = t [i-1] [j] ;

  }
```