There are two type of knapsack

1. 0-1 knapsack (that is solve by Dp {https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/})
2. fractional kanpsack(that is solved by Greedy Approach not DP {https://www.geeksforgeeks.org/fractional-knapsack-problem/}).
3. Unbounded Knapsack(Solve by Dp {https://www.geeksforgeeks.org/unbounded-knapsack-repetition-items-allowed })

------------------++++++++++++++++++++++++++++++++++--

In this we are going to discuss 0-1 knapsack.

Let see the problem statement of 0-1 knapsack

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

In other words, given two integer vectors val[0..n-1] and wt[0..n-1] which represent values and weights associated with n items respectively.

Also given an integer W which represents knapsack capacity, find out the maximum value subset of val[] such that sum of the weights of this

a subset is smaller than or equal to W. You cannot break an item, either pick the complete item or don’t pick it (0-1 property).

----------------Identification-----------------

Question aries that how can we say that this 0-1 knapsack is DP problem ??

So if you remember that there are two condition to identify

1. **choice** -- In this given question there are two choices for every Item, what are those choices , either we include this item in the knapsack or not.
2. **Optimal** -- In the output question asking for max total values.
   |
   {min , max , largest , smallest etc.}

So, you surely say that is Dp promblem.

----------O-1 knapsack recursive solution----------------

condition no 1.

for every item ke pass 2 choice hai usko kanpscak me include kare ya nhi..

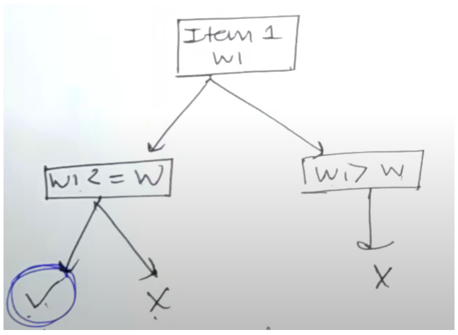
condition no 2

and we check every item's weight if the weight is greater or smaller than the capacity of the knapsack.

Let see how an you find the base condition

Start from the end of the weight array, so for finding the base condition you have to choose the smallest valid input for this array. That is 0. And also the smallest value of capacity (suppose in question there is capacity of knapsack is 0. So we return 0.

Choice diagram on the basic of these two condition



Code For 0-1 knapsack problem

Base condition
if(capacity == 0 || N == 0) return 0;

Now according to the choice diagram
if(Wi<= capacity)
return max(val[n-1] + knapsack(weight , value , N-1 , capacity - weight[N-1] ,
knapsack(weight , value , N-1 , capacity);

Else if (Wi > capacity) return knapsack(weight , value , N-1 , capacity);

```
int knapSack(int W, int wt[], int val[], int n)
    {
       // Your code here
       if(n== 0 || W == 0) return 0;

       if(wt[n-1] <= W){
           return max(val[n-1]+knapSack(W-wt[n-1] , wt , val , n-1) ,
knapSack(W , wt , val , n-1));

       }
       else if(wt[n-1] > W){
           return knapSack(W , wt , val , n-1);
       }
    }
```

Complexity Analysis:

Time Complexity: O(2n).
As there are redundant subproblems.
Auxiliary Space :O(1) + O(N). No extra data structure has been used for storing values but O(N) auxiliary stack space(ASS) has been used for recursion stack.

----------O-1 knapsack Dp solution Memoization----------------

Why DP??
Because in this there is an overlapping sub- problem.
And what is an overlapping sub-promblem ?
Let's suppose we already calculate the value of t[x][y];
And after some recursive calls we got the same point, in recursive solution we recalculate the value of this point ,
but In the Dp solution we already store it in the matrix and we return that value.

How can you decide the size of the matrix ?
So , for deciding the size of the matrix we check the given input parameters , and those parameters change after every recursive call. We make them n and m.
And second is we look for the constraint in given question and make matrix of that size,

for example —

if N<= 100 and W <=1000
Then our matrix size is t[102][1002];

And after that we fill our matrix to -1;
Using two for loop or using memset function.

//code
In the recursive code there is only change in few lines of code

1. Initialize the matrix as global (int t[N][W])
2. And set -1 to every ith row and jth column. (memset(t , -1 , sizeof(t));
3. Now check if current point is already present in the matrix or not
   For that `if(t[n][W] != -1)`
   If present then return that value.
4. Before every recursive call store that value in the matrix .

Time complexity = 0(N*W)

```
 //vector<vector<int>> t(1002, vector<int>(1002 , -1));
 int t[1002][1002];
 //
class Solution
{
    public:
    //Function to return max value that can be put in knapsack of
capacity W.
    int solve(int W, int wt[], int val[], int n)
    {
       // Your code here
       if(n== 0 || W == 0) return 0;
       if(t[n][W] != -1) return t[n][W];

       if(wt[n-1] <= W){
           return t[n][W] = max(val[n-1]+solve(W-wt[n-1] , wt , val ,
n-1) ,
           solve(W , wt , val , n-1));

       }
       else if(wt[n-1] > W){
           return t[n][W] = solve(W , wt , val , n-1);
       }
      // return 0;
    }

    int knapSack(int W, int wt[], int val[], int n)
    {
        memset(t , -1 , sizeof(t));
       return solve(W , wt , val , n);
    }
};
```

----------O-1 knapsack Dp solution Top Down----------------

We already have bottom down approach then what is the need of top down approach
Because in the bottom down(memoization) approach we make recursive calls and
sometimes the recursive stack is full which might get errors like stack overflow.

That’s why we use the Top down approach and people also call it real Dp sometimes.

How can you decide the size of the matrix ?
So , for deciding the size of the matrix we check the given input parameters , and those parameters change after every recursive call. We make them i and j.
For ex-
    Let’s suppose n= 4 and W = 8
        Then our matrix i and j will be t[n+1][W+1];

Changes made to convert recursive code to top down.

1. Initialization of matrix (in this we fill the first row and first column to 0 or something according to the question.)
2. Change recursive call to iterative

Let's see how you can achieve these two targets.
Firstly change the base condition to the initialization step .

```
//here creating a new 2d vector to store the value of iterative calls
vector<vector<int>> t(n+1, vector<int>(W+1));
//converting first row and column to 0
       for(int i =0; i<n+1; i++){
           for(int j = 0; j<W+1; j++){
               if(n == 0|| W == 0){
                   t[i][j] = 0;
               }
           }
       }
```

And after that fill the rest of the matrix with help of previous values. And our ans found at the end of the matrix that is t[n][W].
So at the end we simply return the value of the last element in the matrix.

```
for(int i =1; i<n+1; i++){
           for(int j = 1; j<W+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];
               }
           }
       }
```

```
for(int i =1; i<n+1; i++){
           for(int j = 1; j<W+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];
               }
           }
       }
```
\
//Full code top down approach

```
class Solution
{
    public:
    int knapSack(int W, int wt[], int val[], int n)
    {
       // Your code here
       vector<vector<int>> t(n+1, vector<int>(W+1));
       for(int i =0; i<n+1; i++){
           for(int j = 0; j<W+1; j++){
               if(n == 0|| W == 0){
                   t[i][j] = 0;
               }
           }
       }

         for(int i =1; i<n+1; i++){
           for(int j = 1; j<W+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];
               }
           }
       }
       return t[n][W];
    }
};
```