0-1 Knapsack

P.S. → Given an integer W and two integer arrays val[0...N-1] and wt[0...N-1] which represent knapsack capacity, values and weight associated with N items ~~#ep~~ respectively. Find out the maximum value subset of val[] such that the sum of weights of this subset is smaller than or equal to W.

→ We have only one quantity of each item

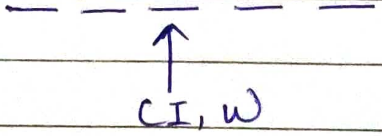
→ Either pick the complete item or don't pick it. (You cannot break an item).

We are given values array which can be thought of as the profit we are getting after putting the item in the knapsack. We have to find out the maximum profit we will get after putting weights such that

overall weight <= W

Also, we cannot use an item more than once.

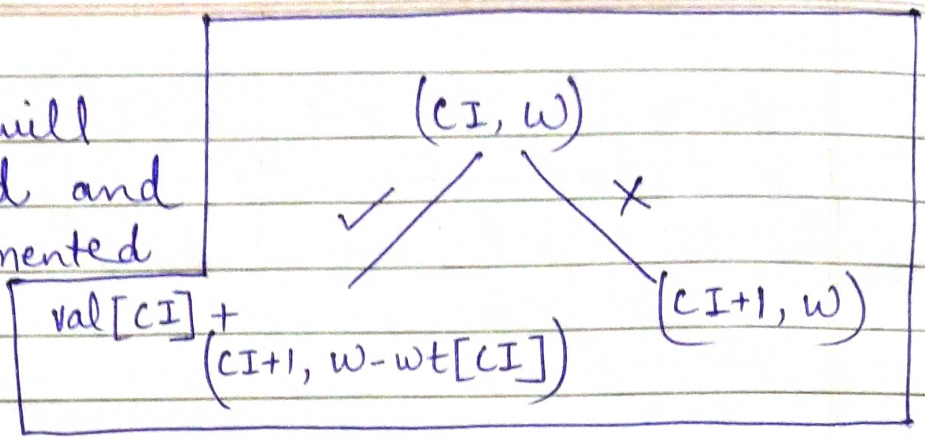
For every item, we have a choice either to include the item or not. Suppose we are at currentIndex CI and we are left with W amount of weight that can be put in the knapsack.



CI = current Index
W = weight left

If we chose to include CI, we will move forward (because we have to include an item only once) and W will become **W - wt[CI]**

If we chose to exclude CI, W will remain unchanged and CI will be incremented by 1.



If we include the item, we will add the value (profit) of that item.

Example –

W = 5

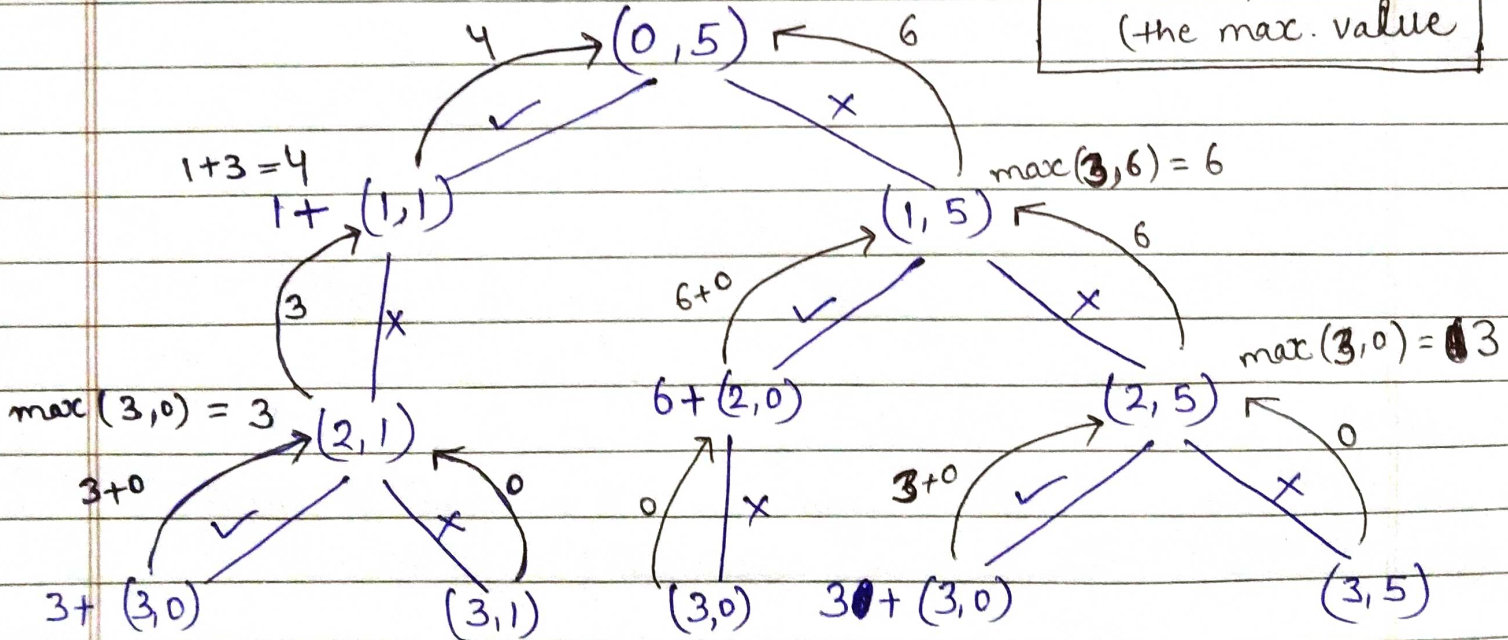
wt[ ] = {4, 5, 1}

val[ ] = {1, 6, 3}.

Since, we are starting from 0 index, CI = 0

W = 5

max(4, 6) = 6
(the max. value



Time Complexity :-

For every item, we have 2 choices

∴ TC → $O(2^n)$

Space Complexity :-

we're going till last index

∴ SC → $O(n)$

✓ = item included

X = item not included

From the recursive tree diagram, we can easily identify the base case of our recursive code.

Some conclusion from the diagram are :-

→ If the weight left (w) is less than the weight of CI, donot need to include the item and therefore we have only one branch at that point.

→ If the CI is equal to or greater than the size of the val / wt array, i.e. we have iterated through all the items, we can return 0 from there (because there is no item left to be considered).

```cpp
// } Driver Code Ends

class Solution
{
    public:
    int knapsackSol(int W, int wt[], int val[], int n, int ind) {
        if(W==0)
        return 0;
        if(ind>=n)
        return 0;
        int consider = 0;
        if(W>=wt[ind]) {
            consider = val[ind] + knapsackSol(W-wt[ind], wt, val, n, ind+1);
        }
        int notconsider = knapsackSol(W, wt, val, n, ind+1);
        return max(consider, notconsider);
    }

    //Function to return max value that can be put in knapsack of capacity W.
    int knapSack(int W, int wt[], int val[], int n)
    {
       // Your code here
       return knapsackSol(W, wt, val, n, 0);
    }
};

} // } Driver Code Ends
```

consider = item (at index *ind*) is included

notconsider = item (at index *ind*) is not included

The above recursive code will give TLE because there will be many overlapping cases where the evaluation will be repetitive. To avoid this situation, we prefer to make a key out of the parameters which are changing while calling the recursive function and store the answer of the key. We can do this by using 2-D array or vector in C++.

Here, ind and weight W is changing and therefore 2-D array of ind and W are used to resolve the the overlapping issue.

Optimised and Accepted Code -

```
// } Driver Code Ends

class Solution
{
    public:
    int v[1001][1001];
    int knapsackSol(int W, int wt[], int val[], int n, int ind) {
        if(W==0)
        return 0;
        if(ind>=n)
        return 0;
        if(v[ind][W] != -1)
        return v[ind][W];
        int consider = 0;
        if(W>=wt[ind]) {
            consider = val[ind] + knapsackSol(W-wt[ind], wt, val, n, ind+1);
        }
        int notconsider = knapsackSol(W, wt, val, n, ind+1);
        v[ind][W] = max(consider, notconsider);
        return max(consider, notconsider);
    }


    //Function to return max value that can be put in knapsack of capacity W.
    int knapSack(int W, int wt[], int val[], int n)
    {
       // Your code here
       memset(v,-1,sizeof(v));
       return knapsackSol(W, wt, val, n, 0);
    }
};
```