0 - 1 Knapsack Problem

0 - 1 Knapsack Problem

Difficulty: **Medium** Accuracy: **31.76%** Submissions: **474K+** Points: **4**

Given **n** items, each with a specific **weight** and **value**, and a knapsack with a capacity of **W**, the task is to put the items in the knapsack such that the **sum of weights of the items <= W** and the **sum of values** associated with them is **maximized**.

Note: You can either place an item entirely in the bag or leave it out entirely. Also, each item is available in **single** quantity.

Examples :

Input: W = 4, val[] = [1, 2, 3], wt[] = [4, 5, 1]
Output: 3
Explanation: Choose the last item, which weighs 1 unit and has a value of 3.

Input: W = 3, val[] = [1, 2, 3], wt[] = [4, 5, 6]
Output: 0
Explanation: Every item has a weight exceeding the knapsack's capacity (3).

Input: W = 5, val[] = [10, 40, 30, 50], wt[] = [5, 4, 2, 3]
Output: 80
Explanation: Choose the third item (value 30, weight 2) and the last item (value 50, weight 3) for a total value of 80.

Constraints:
$2 \le val.size() = wt.size() \le 10^3$
$1 \le W \le 10^3$
$1 \le val[i] \le 10^3$
$1 \le wt[i] \le 10^3$

Try more examples

eg: W = 4
val[] = [1,2,3]
wt[] = [4,5,1] → O/p: 3

we can either include an element or entirely exclude it.

parameters to be maintained.

i, W

include :- help(i-1, w-w[i]) + v[i]
exclude :- help(i-1, w)

Base Condition :-

if(i<0) return 0;

Brute Force Solution :- T.C. $O(2^n)$

```
class Solution {
  public:
    int knapsack(int W, vector<int> &val, vector<int> &wt) {
        // code here
        int n=val.size();
        return help(val, wt, n-1, W);
    }
    int help(vector<int> &val, vector<int> &wt, int i, int w){
        if(i<0 )return 0;
        int include=0;
        if(w>=wt[i])include=val[i]+help(val, wt, i-1, w-wt[i]);
        int exclude=help(val, wt, i-1, w);
        return max(include, exclude);
    }
};
```

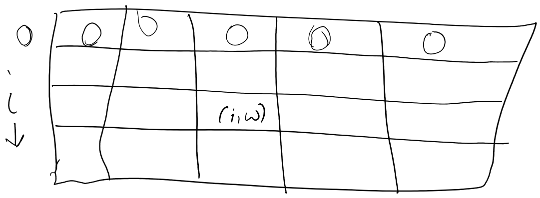
✓ Optimized Solution :- Using DP + memoization :-

```cpp
class Solution {
  public:
    int knapsack(int W, vector<int> &val, vector<int> &wt) {
        // code here
        int n=val.size();
        vector<vector<int>> dp(n, vector<int>(W+1, -1));
        return help(val, wt, n-1, W, dp);
    }
    int help(vector<int> &val, vector<int> &wt, int i, int w, vector<vector<int>> &dp){
        if(i<0 )return 0;
        if(dp[i][w]!=-1)return dp[i][w];
        int include=0;
        if(w>=wt[i])include=val[i]+help(val, wt, i-1, w-wt[i], dp);
        int exclude=help(val, wt, i-1, w, dp);
        return dp[i][w]=max(include, exclude);
    }
};
```

T.C : $O(n \times W)$

S.C : $O(n \times W)$

Bottom-Up DP :-



include :-

for(int w = wt[i] ; w <= W ; w++){

dp[i][w] = max(dp[i][w],
val + dp[i-1][w-wt[i]])

}

exclude :-

dp[i][w] = dp[i-1][w];

[1, 2, 3]

[4, 5, 1]

w = 4

4, 5

	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	1
2	0	0	0	0	0
3	0	3	3	3	3

return dp[n][W];

✓

```cpp
class Solution {
  public:
    int knapsack(int W, vector<int> &val, vector<int> &wt) {
        // code here
        int n=val.size();
        vector<vector<int>> dp(n+1, vector<int>(W+1));
        for(int i=1;i<=n;i++){
            for(int j=1;j<=W;j++){
                if(j>=wt[i-1]){
                    dp[i][j]=max(dp[i][j], val[i-1]+dp[i-1][j-wt[i-1]]);
                }
                dp[i][j]=max(dp[i][j], dp[i-1][j]);
            }
        }
        return dp[n][W];
    }
};
```

Bottom-Up Soln

we val[i-1] and wt[i-1]