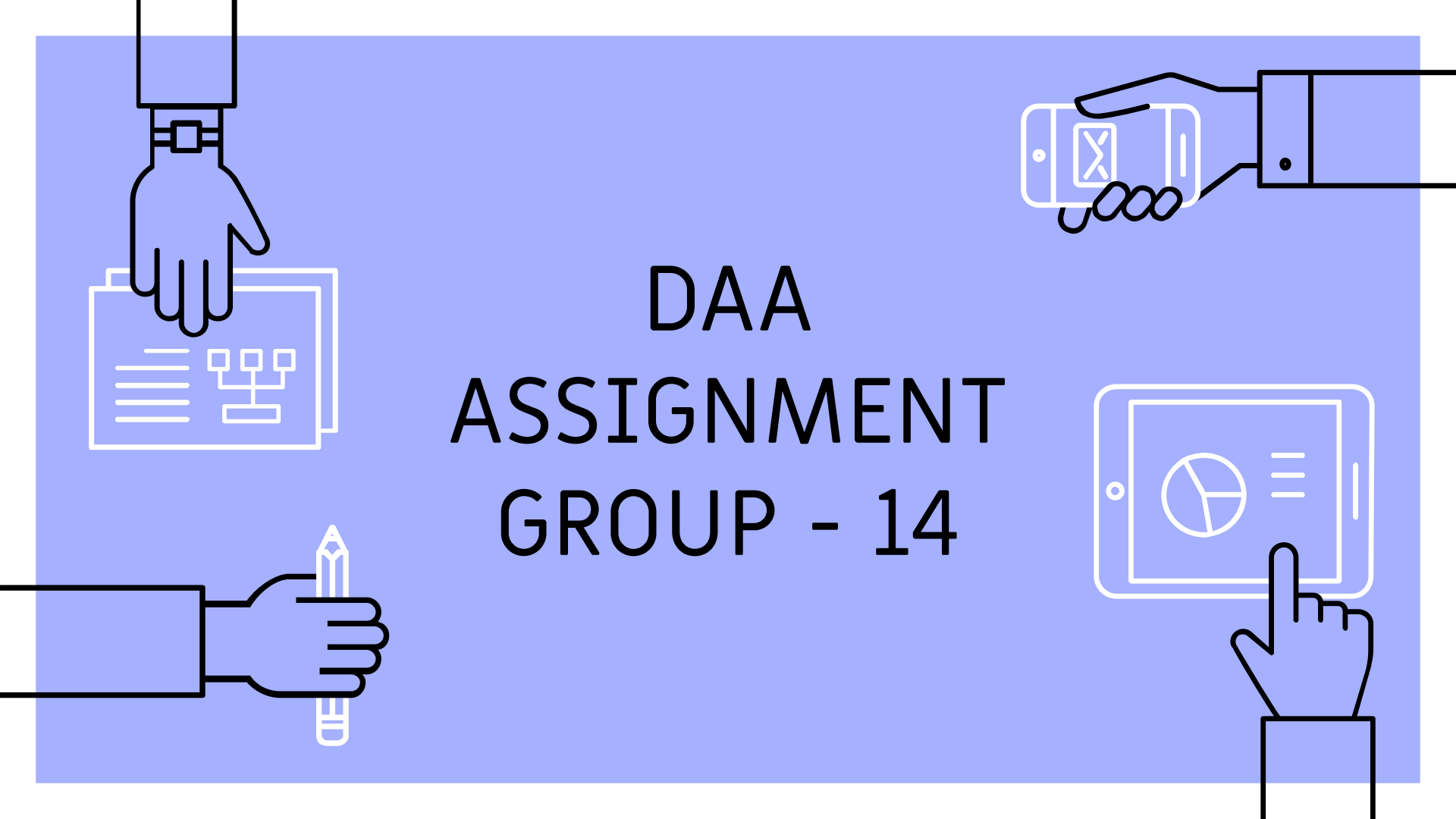
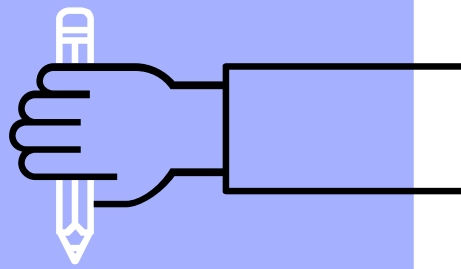
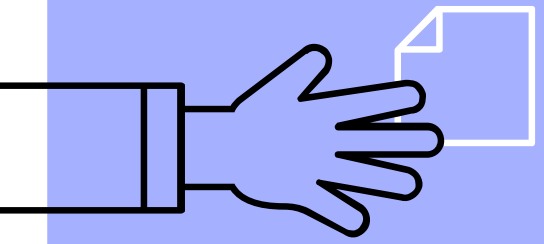

DAA
ASSIGNMENT
GROUP - 14

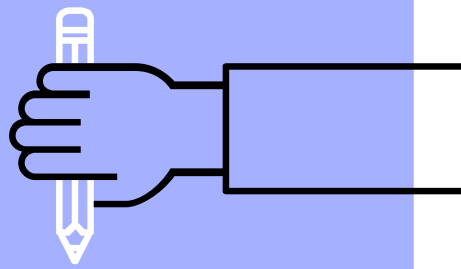
GROUP MEMBERS

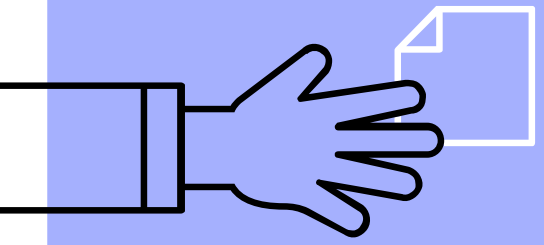


KISHAN TRIPATHI - IIT2019225
MUKUL MOHMARE - IIT2019226
ANSHUMAN BHRADWAJ - IIT2019227



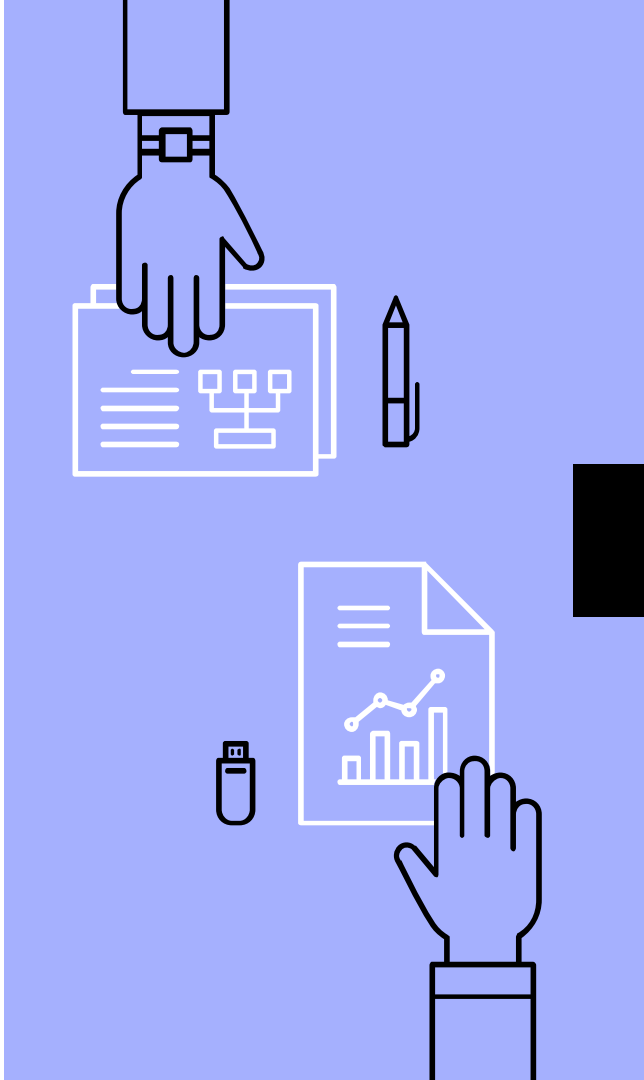
Contents





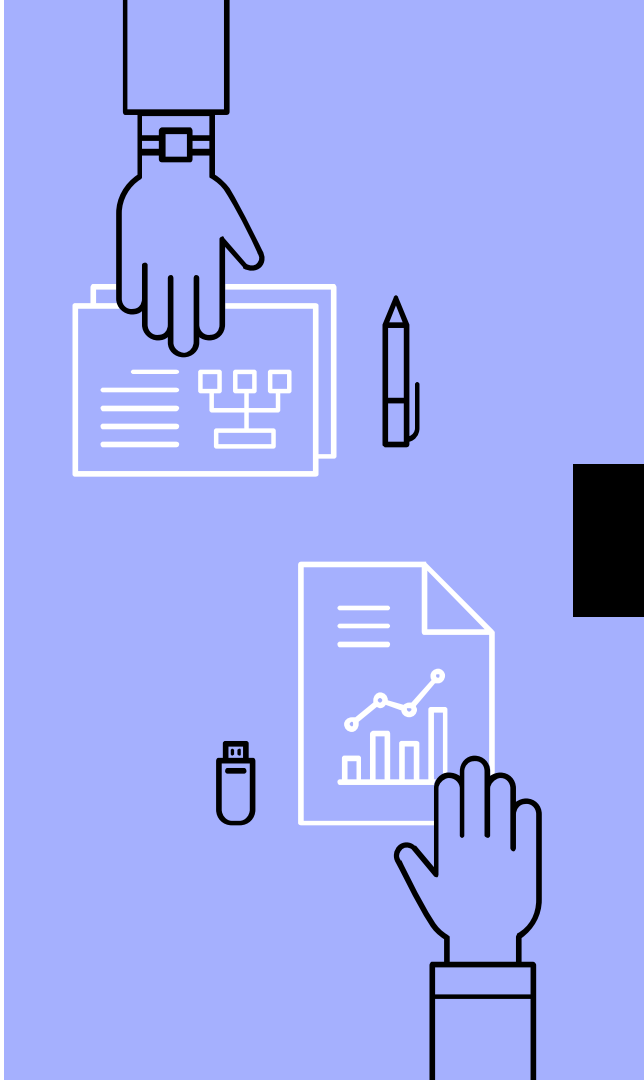
Abstract :

Illustrate the performance of the 0/1 Knapsack Problem and propose a parallel algorithm for the same.



0/1 Knapsack

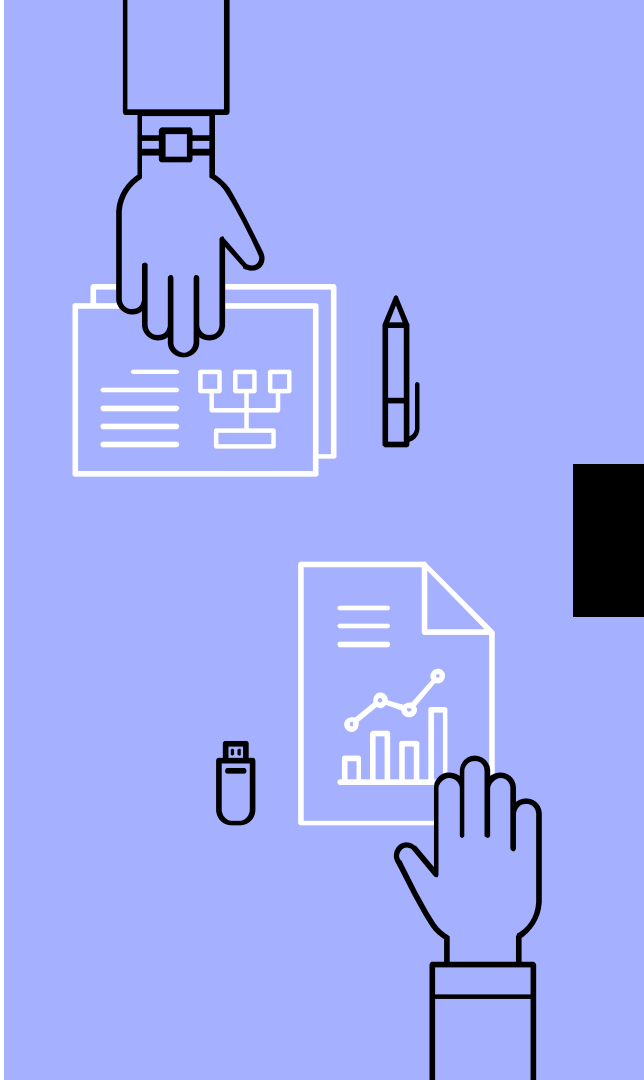
Given weights and values of n items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack.



Lets See An Example.....

Given a set of items each with fixed weight capacities and values. Find a number of these items that will maximize value but whose total weight does not exceed the given capacity

Total knapsack profit = 15 by picking weights {4,1,1,2}

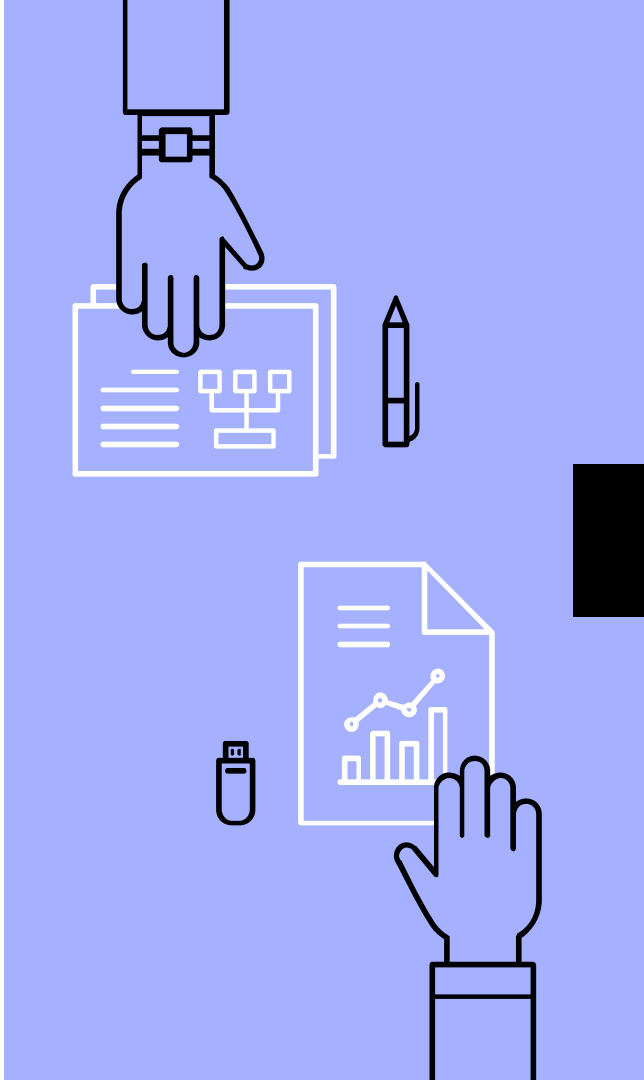


Dynamic Programming

Dynamic Programming is optimisation over plain recursion. The idea is to simply store the results of subproblems, so that we do not have to re-compute them when needed later.

Any Dynamic Programming problem has two properties

1. Optimal Substructure
2. Overlapping

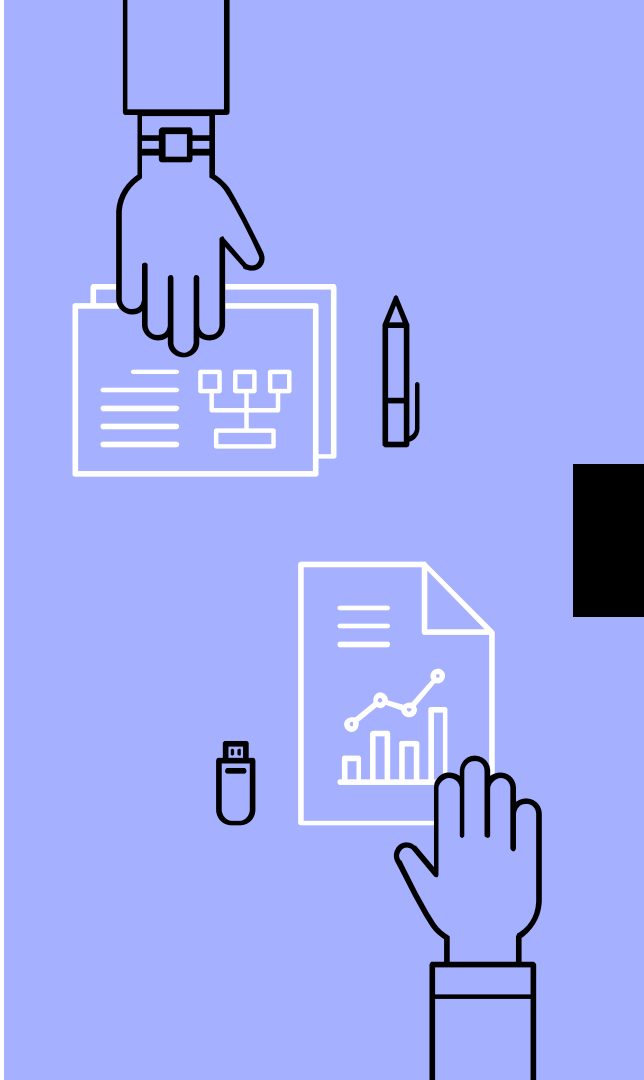


Algorithm Design

We want to maximise the total value of all the items that we are going to put into the knapsack such that the total weight of items is less than or equal to the knapsack's capacity.

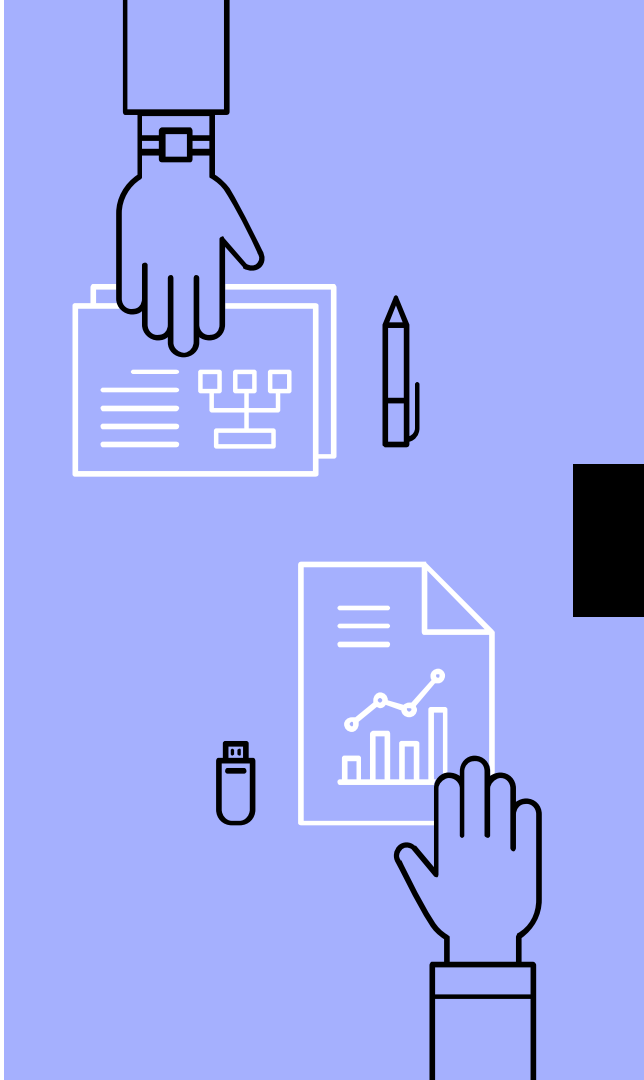
There can be two cases for every item:

1. The item is included in the optimal subset,
2. Not included in the optimal set



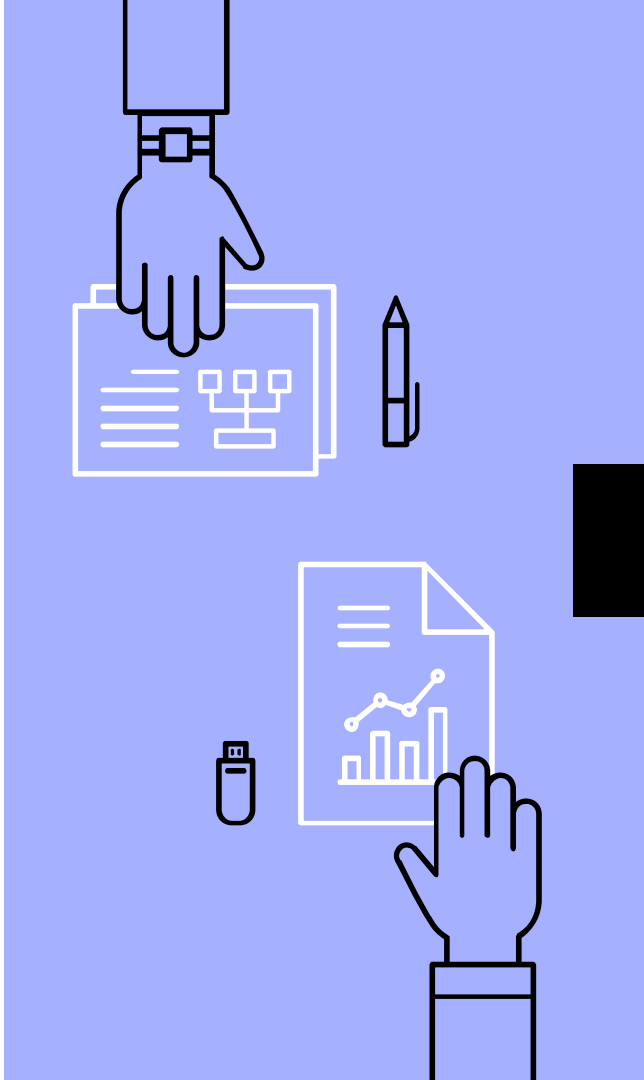
Recurrence Relation :

```
if (n=0 or W=0)
      return 0
if (weight[ n ] > W)
      return solve(n-1, W)
else
return max{ solve(n-1, W), solve(n-1,
W-weight[ n ])
```



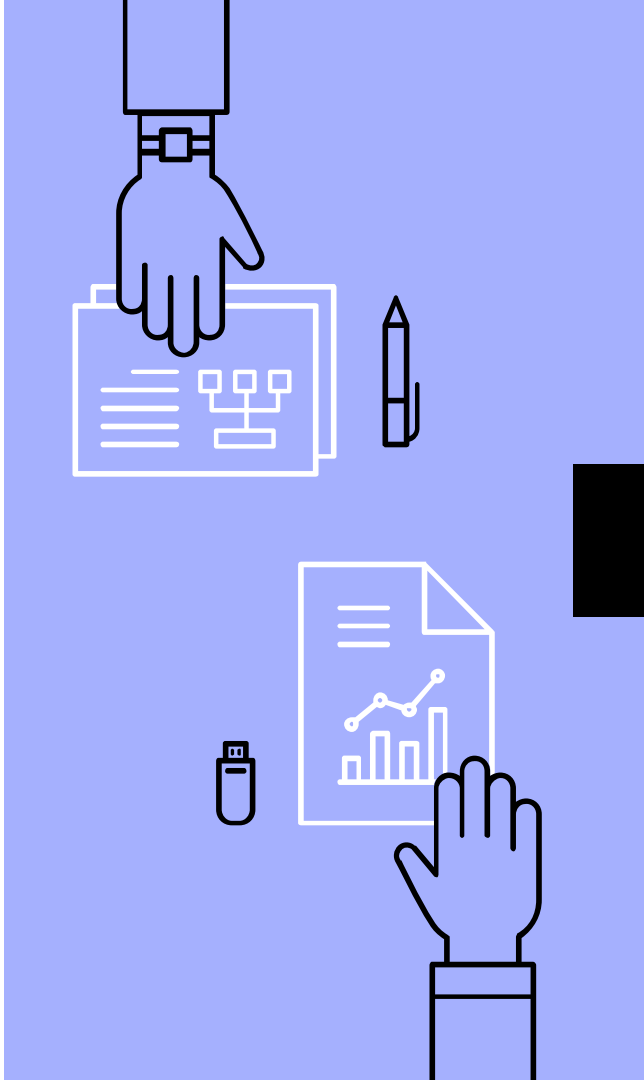
We use two approaches to solve this problem

- Bottom up approach
- Top down approach



Bottom Up Approach

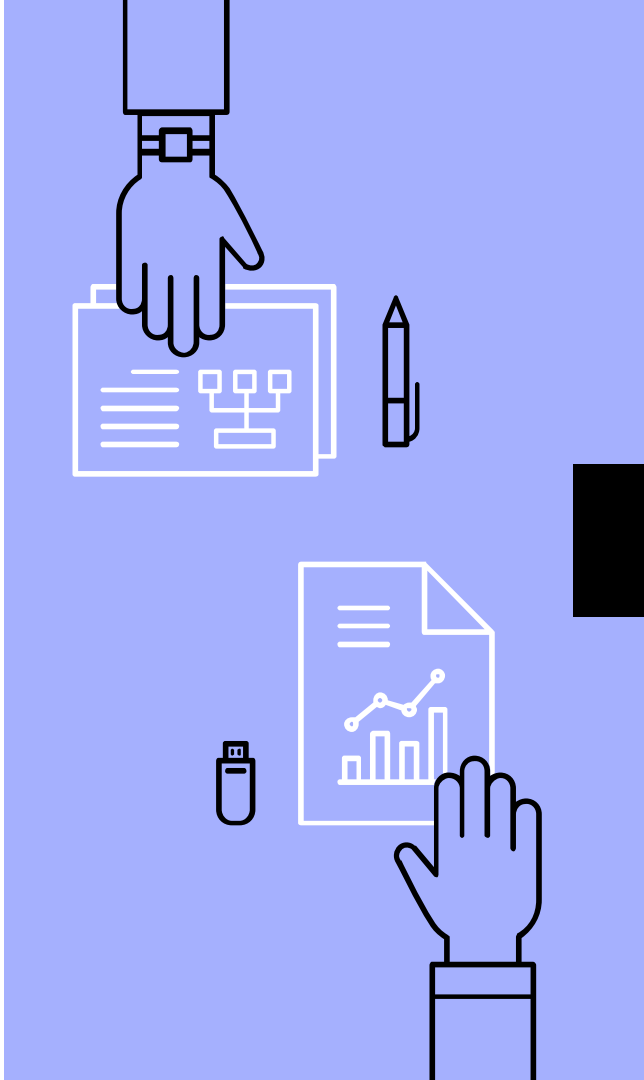
Analyze the problem and see the order in which the sub-problems are solved and start solving from the trivial subproblem, up towards the given problem.



Logic Behind Bottom Up Approach :

We create Dp table of size [n+1][w+1] where n is number of different weights and w is the knapsack capacity

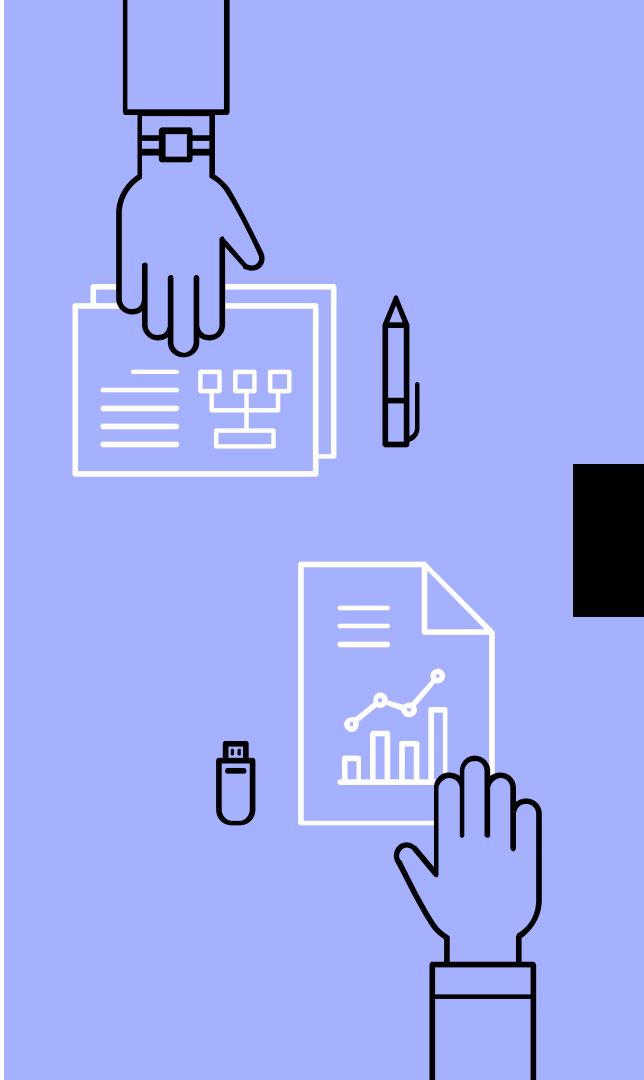
```
if (i == 0 || w == 0)
        K[i][w] = 0;
    else if (wt[i - 1] <= w)
        K[i][w] = max(val[i - 1] +
                      K[i - 1][w - wt[i - 1]],
                      K[i - 1][w]);
    else
        K[i][w] = K[i - 1][w];
```



Top Down Approach

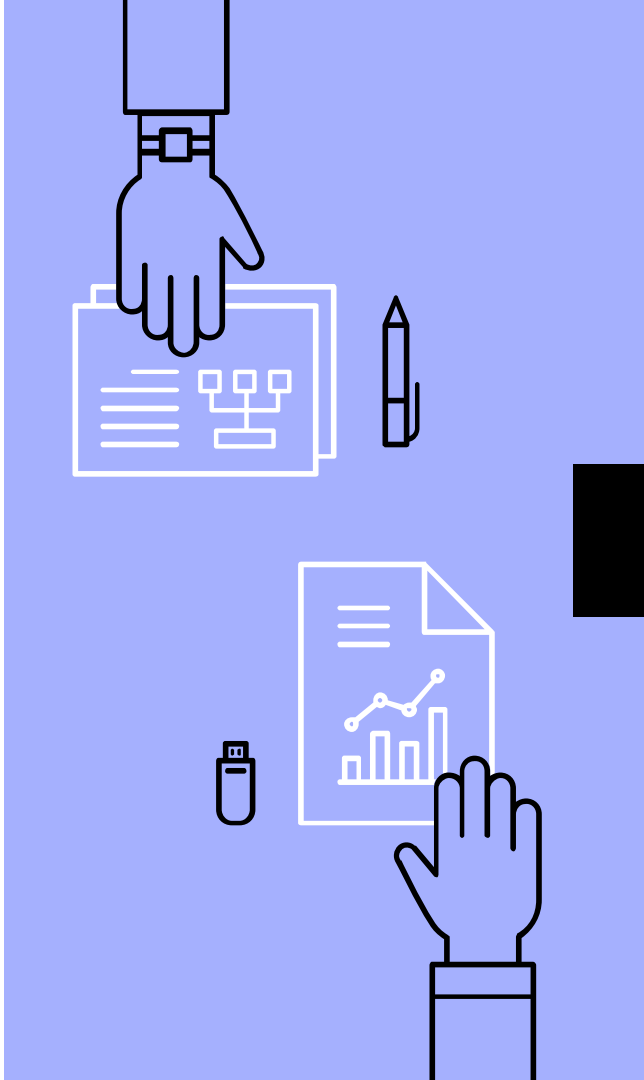
We solved the given problem by first breaking it down into simpler subproblems. If the problem is already solved then return the saved answer for that otherwise solve the subproblem and save the answer. This method is very intuitive.

Also referred as **Memorization**.



Logic Behind Top Down Approach :

```
if (i < 0)
    return 0;
if (dp[i][W] != -1)
    return dp[i][W];
if (wt[i] > W)
dp[i][W] = knapSackRec(W, wt,val,i-1,dp);
return dp[i][W];
```



Sample Input :

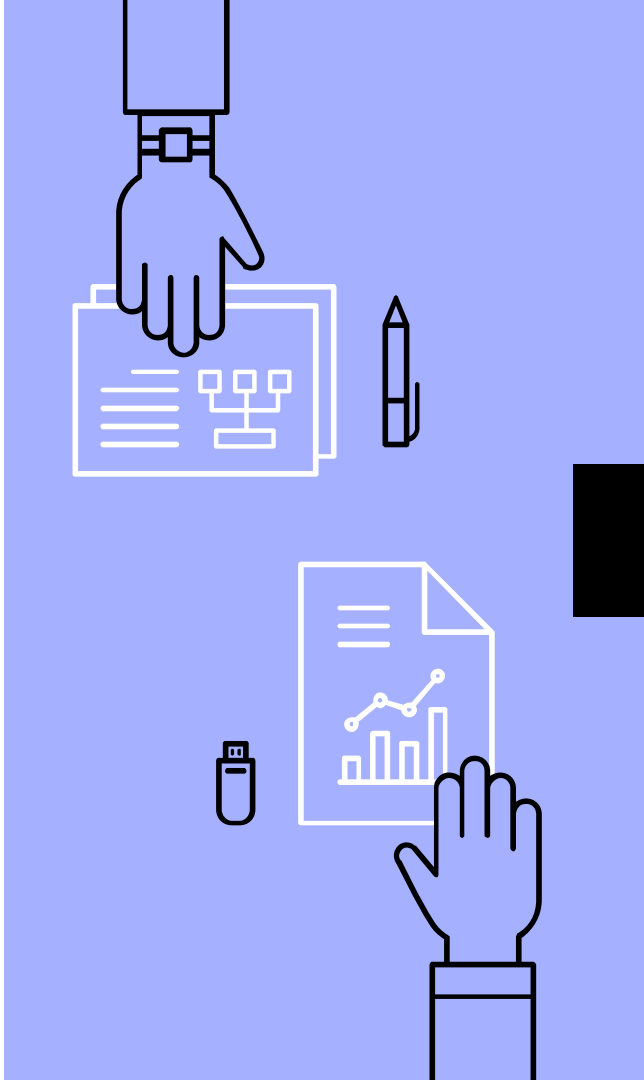
Input :

Knapsack Capacity (W) = 11

No. of weights (n) = 5

Weights (w[n]) = {3, 4, 5, 9, 4}

Value (v[n]) = {3, 4, 4, 10, 4}



Sample Output :

0 0 0 3 3 3 3 3 3 3 3
0 0 0 3 4 4 4 7 7 7 7
0 0 0 3 4 4 4 7 7 8 8
0 0 0 3 4 4 4 7 7 10 10 10
0 0 0 3 4 4 4 7 8 10 10 11

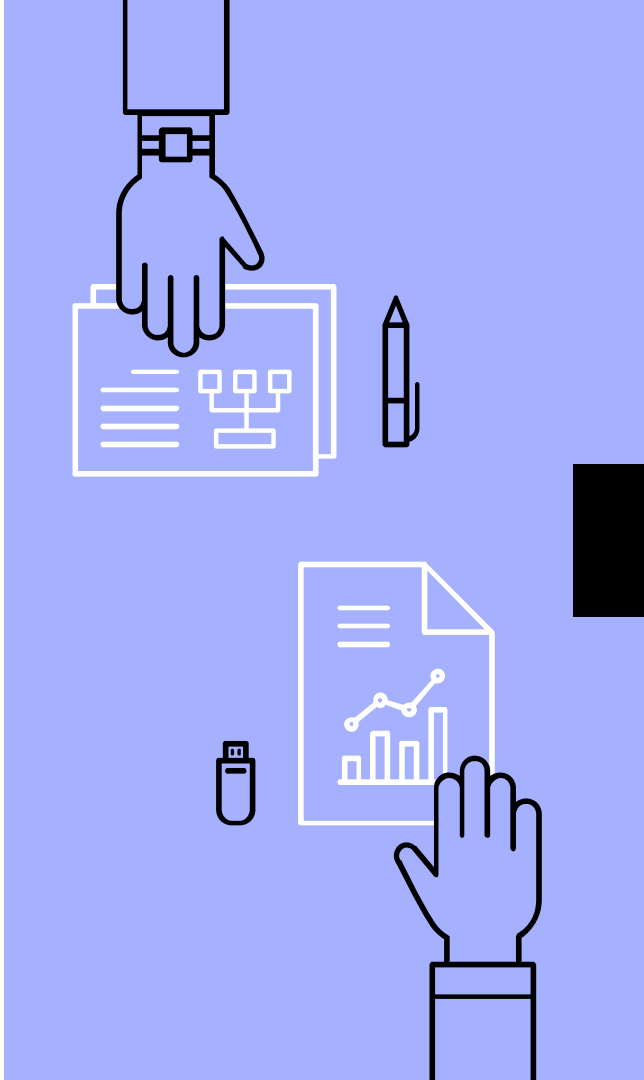
Max Value: 11

Selected Packs:

Package 5 with W = 4 and Value = 4

Package 2 with W = 4 and Value = 4

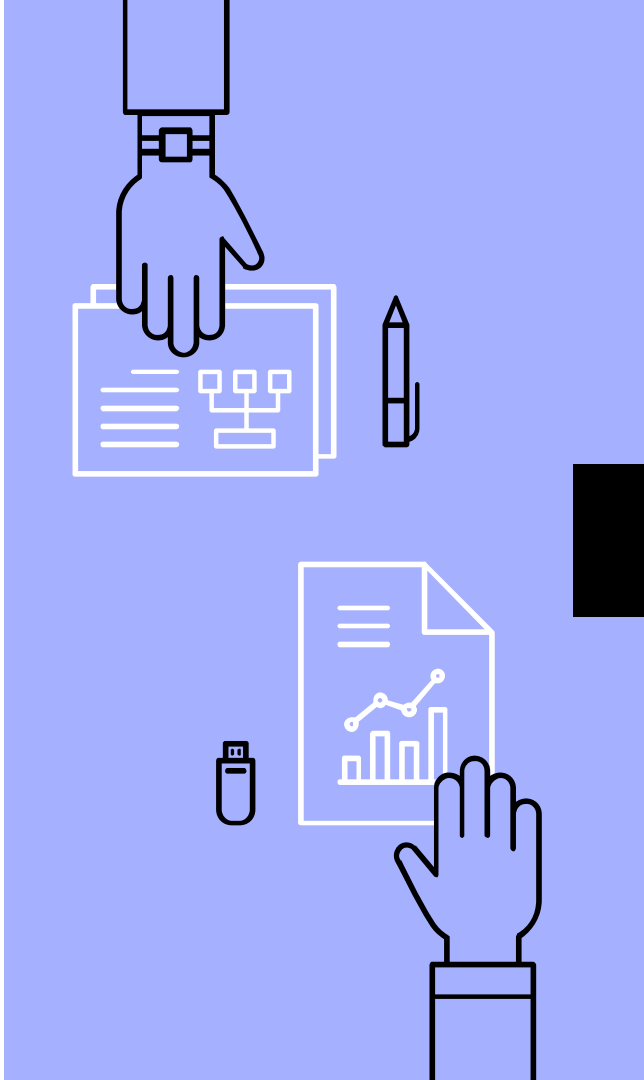
Package 1 with W = 3 and Value = 3



Space-Optimized DP Approach

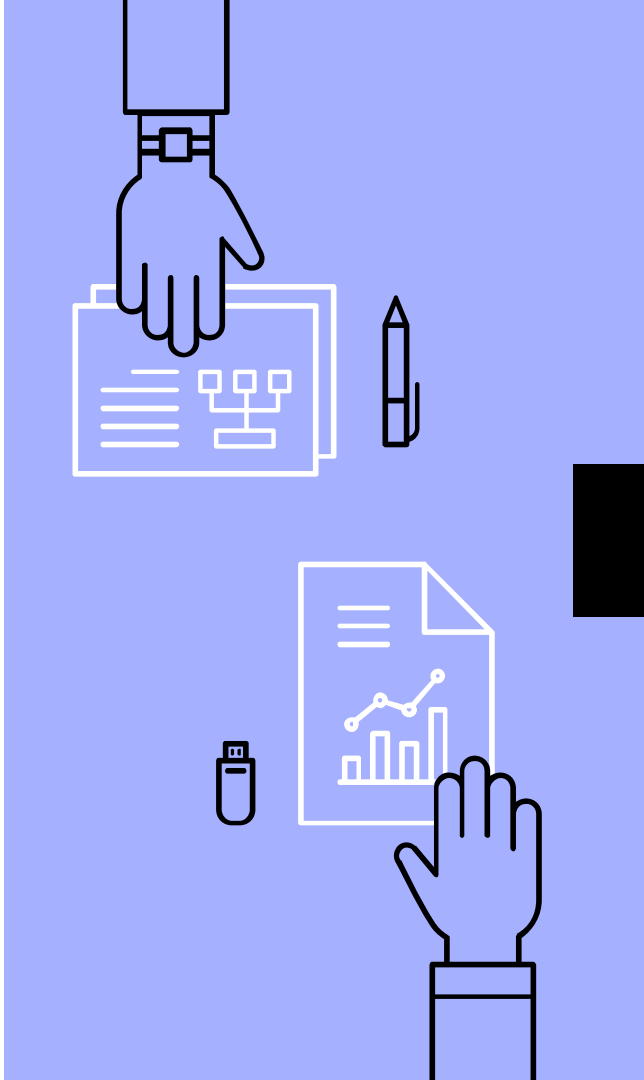
We used a n * W matrix. We can reduce the used extra space. The idea behind the optimization is, to compute mat[i][j], we only need solution of previous row.

Hence we create a matrix of size 2*W. If n is odd, then the final answer will be at mat[0][W] and if n is even then the final answer will be at mat[1][W].



Complexity Analysis

- Time complexity for both the algorithm is O(N*W).
- Space complexity for bottom up algorithm isO(N*W).
- Space complexity for space optimised dp solution is O(W)



THANK YOU