Problem Statement:

The store consists of N number of items, each weighing w_i with a value of v_i. Given the maximum weight(W) that can be carried out, the task is to find the maximum value goods that a thief can carry out.

Solution:

1. **Recursive:**

   The most obvious solution to this problem is brute-force recursive. This solution is brute-force because it evaluates the total weight and value of all possible subsets, then selects the subset with the highest value that's still under the weight limit.

   If there are **n** items to choose from, then there will be **2n** possible combinations of items for the knapsack. An item is either chosen or not chosen. A bit string of 0's and 1's is generated which is of length n. If the ith symbol of a bit string is 0, then the ith item is not chosen and if it is 1, the ith item is chosen.

   Pseudo-Code:

```
Method: knapSack(int W, int wt[], int val[], int n)
    {

        if (n == 0 || W == 0)
            return 0;

   //If weight of the nth item more than Knapsack capacity W,
   // then this item cannot be included in the optimal
       solution
        if (wt[n - 1] > W)
            return knapSack(W, wt, val, n - 1);

        // Return the maximum of two cases:
        // (1) nth item included
        // (2) not included
        else
            return
     max(val[n - 1]+knapSack(W - wt[n - 1], wt,val, n - 1),
                        knapSack(W, wt, val, n - 1));
    }
```

- **Optimal Substructure:** To consider all subsets of items, there can be two cases for every item.
  1. Case 1: The item is included in the optimal subset.
  2. Case 2: The item is not included in the optimal set.

- Therefore, the maximum value that can be obtained from 'n' items is the max of the following two values.
  1. Maximum value obtained by n-1 items and W weight (excluding nth item).

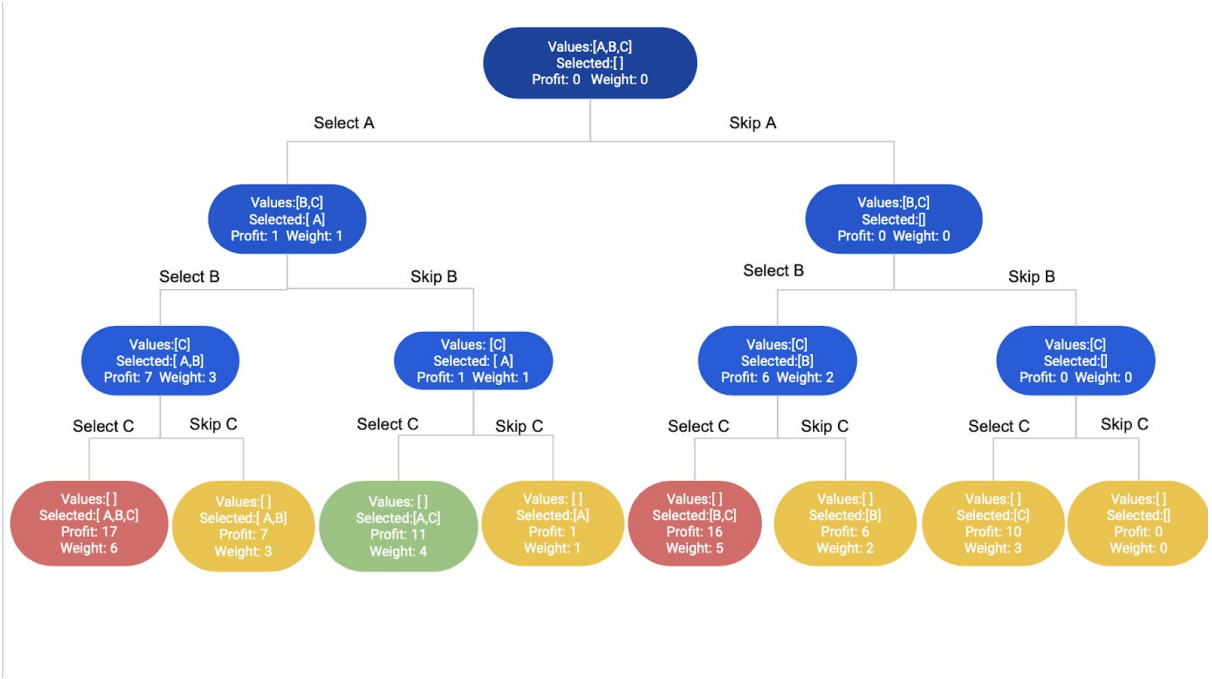
2. Value of nth item plus maximum value obtained by n-1 items and W minus the weight of the nth item (including nth item).

If the weight of ‘nth’ item is greater than ‘W’, then the nth item cannot be included and Case 1 is the only possibility.

Here is a visual representation of our algorithm:

Items	A	B	C
Profits	1	6	10
Weight	1	2	3

Maximum weight (W) = 4



All yellow boxes have a total weight that is less than or equal to the capacity (4), and all the red ones have a weight that is more than 4. The best solution we have is the green box with items [A, C] having a total profit of 11 and a total weight of 4.

Let’s visually draw the recursive calls to see if there are any overlapping subproblems. As we can see, in each recursive call, profits and weights arrays remain constant, and only capacity and currentIndex change. For simplicity, let’s denote capacity with ‘c’ and currentIndex with ‘i’:

We can clearly see that ‘c:4, i=3’ has been called twice, hence we have an overlapping subproblems pattern. As we already know, overlapping subproblems can be solved through Memoization.

Complexity Analysis:

- Time Complexity: $O(2^n)$ - As there are redundant subproblems.
- Auxiliary Space :O(n)- For the recursive stack.

2. Memoization:

This method is basically an extension to the recursive approach so that we can overcome the problem of calculating redundant cases and thus increased complexity. We can solve this problem by simply creating a 2-D array that can store a particular state (n, w) if we get it the first time. Now if we come across the same state (n, w) again instead of calculating it in exponential complexity we can directly return its result stored in the table in constant time. This method gives an edge over the recursive approach in this aspect.

Pseudo-Code:

```
// Declare a table dynamically
// N is the number of items and W is the max weight
   int dp[][] = new int[N + 1][W + 1];

 // Initialize the table with -1
   for(int i = 0; i < N + 1; i++)
      for(int j = 0; j < W + 1; j++)
         dp[i][j] = -1;
// Call the method knapSackRec(W, wt, val, N, dp)

Method: knapSackRec(int W, int wt[], int val[], int n,int [][]dp)
                  {    if (n == 0 || W == 0)
                         return 0;
                       if (dp[n][W] != -1)
                         return dp[n][W];

                       if (wt[n - 1] > W)
                        // Store the value of function call
                        // stack in table before return
                         return dp[n][W] = knapSackRec(W, wt, val,n - 1, dp);

                       else
                         // Return value of table after storing
                         return dp[n][W] = max((val[n - 1] +
                                  knapSackRec(W - wt[n - 1], wt,val, n - 1, dp)),
                                       knapSackRec(W, wt, val,n - 1, dp));
```

3. Dynamic Programming:

When analyzing 0/1 Knapsack problem using Dynamic programming, you can find some noticeable points. The value of the knapsack algorithm depends on two factors:

1. How many items are being considered
2. The remaining weight which the knapsack can store.

Therefore, you have two variable quantities.

With dynamic programming, you have useful information:

1. the objective function will depend on two variable quantities
2. the table of options will be a 2-dimensional table.

If calling B[i][j] is the maximum possible value by selecting in items {1, 2, ..., i} with weight limit j.

- The maximum value when selected in n items with the weight limit M is B[n][M]. In other words: When there are i items to choose, B[i][j] is the optimal weight when the maximum weight of the knapsack is j.
- The optimal weight(j) is always less than or equal to the maximum weight: B[i][j] ≤ j.

For example: B[4][10] = 8. It means that in the optimal case, the total weight of the selected items is 8, when there are 4 first items to choose from (1st to 4th item) and the maximum weight of the knapsack is 10. It is not necessary that all 4 items are selected.

Formula to Calculate B[i][j]

Input, you define:

- W[i], V[i] are in turn the weight and value of item i, in which i $\in$ {1, …, n}.
- M is the maximum weight that the knapsack can carry.

In the case of simply having only 1 item to choose. You calculate B[1][j] for every j: which means the maximum weight of the knapsack ≥ the weight of the 1st item
B[1][j] = W[1]

With the weight limit j, the optimal selections among items {1, 2, ..., i – 1, i} to have the largest value will have two possibilities:

- If item i is not selected, B[i][j] is the maximum possible value by selecting among items {1, 2, ..., i – 1} with weight limit of j. You have:

B[i][j] = B[i – 1][j]

- If item i is selected (of course only consider this case when W[i] ≤ j) then B[i][j] is equal to the value V[i] of item i plus the maximum value can be obtained by selecting among items {1, 2, ..., i – 1} with weight limit (j – W[i]). That is, in terms of the value you have:

B[i][j] = V[i] + B[i – 1][j – W[i]]

So B[i][j] can hold the maximum of the two possible values: B[i-1][j] or V[i]+B[i – 1][j – W[i]].

Basis of Dynamic Programming

So, you have to consider if it is better to choose item i or not. From there you have the recursive formula as follows:
B[i][j]= max(B[i – 1][j], V[i]+B[i – 1][j – W[i]]

It is easy to see B[0][j] = maximum value possible by selecting from 0 item = 0.

Calculate the Table of Options

You build a table of options based on the above recursive formula.
Table of options B includes n + 1 lines, M + 1 columns,

- Firstly, on the basis of dynamic programming: Line 0 includes all zeros.
- Using recursive formulas, use line 0 to calculate line 1, use line 1 to calculate line 2, etc. ... until all lines are calculated.

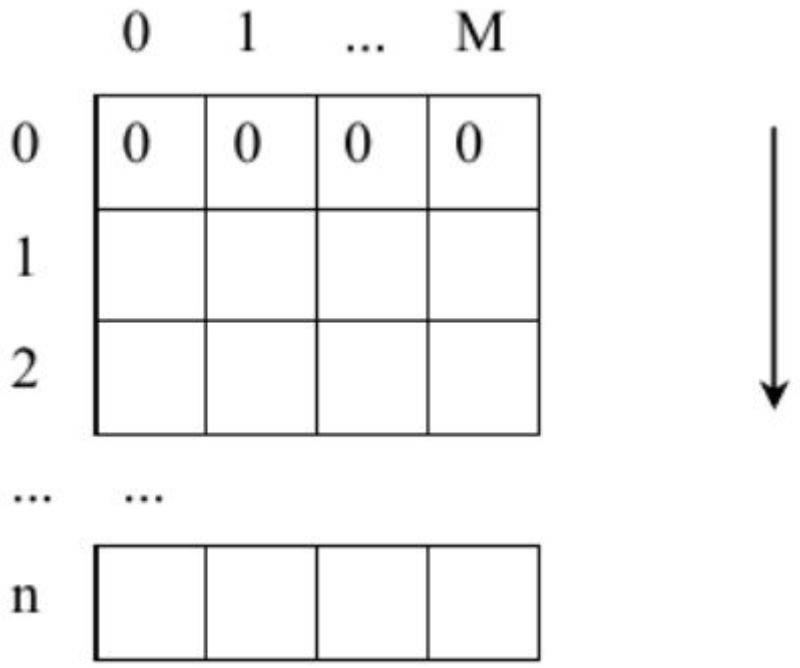


Table of Options

For example:
Using the same example discussed early.

Items	A	B	C
Profits	1	6	10
Weight	1	2	3

Maximum weight (W) = 4

Equivalent Knapsack Matrix =

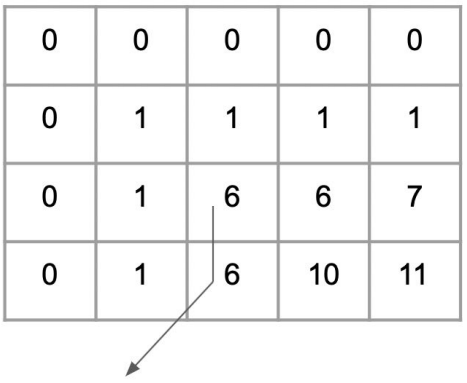
0	0	0	0	0
0	1	1	1	1
0	1	6	6	7
0	1	6	10	11

Trace

When calculating the table of options, you are interested in B[n][M] which is the maximum value obtained when selecting in all n items with the weight limit M.

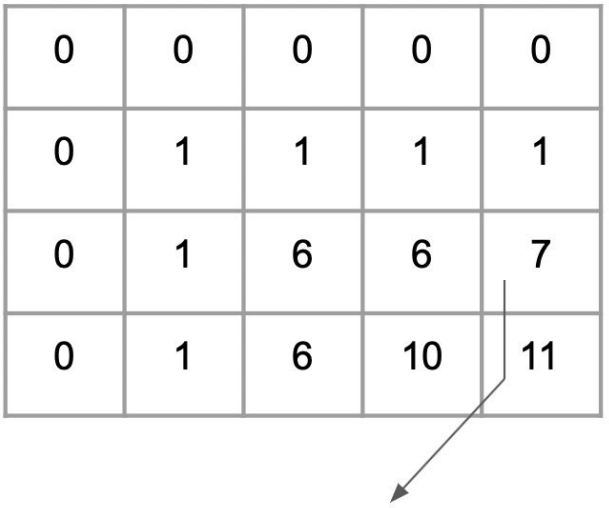
- If B[n][M] = B[n – 1][M] then item n is not selected, you trace B[n – 1][M].
- If B[n][M] ≠ B[n – 1][M], you notice that the optimal selection has the item n and trace B[n – 1][M – W[n]].

Continue to trace until reaching row 0 of the table of options.



0	0	0	0	0
0	1	1	1	1
0	1	6	6	7
0	1	6	10	11

Both the cells have same value. This means that the 3rd item C is not selected in this case.

- 

0	0	0	0	0
0	1	1	1	1
0	1	6	6	7
0	1	6	10	11

Both the cells have different value.
Hence item C is selected.

-

Pseudo-Code:

Method:

```
knapSack(int W, int wt[],int val[], int n)
    {
        int i, w;
        int B[][] = new int[n + 1][W + 1];

        // Build table B[][] in bottom up manner
```

```
                for (i = 0; i <= n; i++)
                {
                    for (w = 0; w <= W; w++)
                    {
                        if (i == 0 || w == 0)
                            B[i][w] = 0;
                        else if (wt[i - 1] <= w)
                            B[i][w]
                                = max(val[i - 1]
                                  + B[i - 1][w - wt[i -
        1]],
                                  B[i - 1][w]);
                        else
                            B[i][w] = B[i - 1][w];
                    }
                }

                return B[n][W];
            }
```

Complexity Analysis:

- Time Complexity: O(N*W).
  where ‘N’ is the number of weight element and ‘W’ is capacity. As for every weight element we traverse through all weight capacities 1<=w<=W.
- Auxiliary Space: O(N*W).
  The use of the 2-D array of size ‘N*W’.

4. **Space Optimized DP:**
   We have discussed a Dynamic Programming based solution. In the previous solution, we used a n * W matrix. We can reduce the used extra space. The idea behind the optimization is, to compute B[i][j], we only need a solution of the previous row. In 0-1 Knapsack Problem if we are currently on B[i][j] and we include ith element then we move j-wt[i] steps back in the previous row and if we exclude the current element we move on the jth column in the previous row. So here we can observe that at a time we are working only with 2 consecutive rows.
   In below solution, we create a matrix of size 2*W. If n is odd, then the final answer will be at B[0][W] and if n is even then the final answer will be at B[1][W] because index starts from 0.

   Method:
```
int KnapSack(int val[], int wt[], int n, int W)
    {
        // matrix to store final result
        int mat[][] = new int[2][W + 1];

        // iterate through all items
        int i = 0;
        while (i < n) // one by one traverse each element
        {
```

```
        int j = 0; // traverse all weights j <= W
        if (i % 2 != 0)
        {
            while (++j <= W) // check for each value
            {
          if (wt[i] <= j) // include element
          {
        B[1][j] = Math.max(val[i] + B[0][j - wt[i]],B[0][j]);
          }
        else // exclude element
                {
                    B[1][j] = B[0][j];
                }
            }

        }

        // if i is even that means till now
        // we have even number of elements
        // so we store result in 0th indexed row
        else
        {
            while (++j <= W)
            {
              if (wt[i] <= j)
              {
                B[0][j]= Math.max(val[i] +B[1][j - wt[i]],
                                                    B[1][j]);
                }
       else{
                 B[0][j] = B[1][j];
            }
           }
        }
        i++;
    }
    return (n % 2 != 0) ? B[0][W] : B[1][W];
}
```