0-1 Knapsack problem

An example of Dynamic programming

Question-

In a 0-1 knapsack problem, we are given a set of items, each with a weight and a value and we need to determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and total value is as large as possible.

In other words, given two integer arrays 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 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).

Example:

Value= {20, 5, 10, 40, 15, 25}

Weight= {1, 2, 3, 8, 7, 4}

W=10

The knapsack value is 60

Value= 20+40= 60

Weight= 1+8= 9<10

The idea is to use recursion to solve this problem. For each item there are 2 possibilities

1. We include the current item in knapsack and recur for remaining items with decreased capacity of knapsack. If capacity becomes negative do not recur or return -INFINITY.

2. We exclude current item and recur for remaining items.

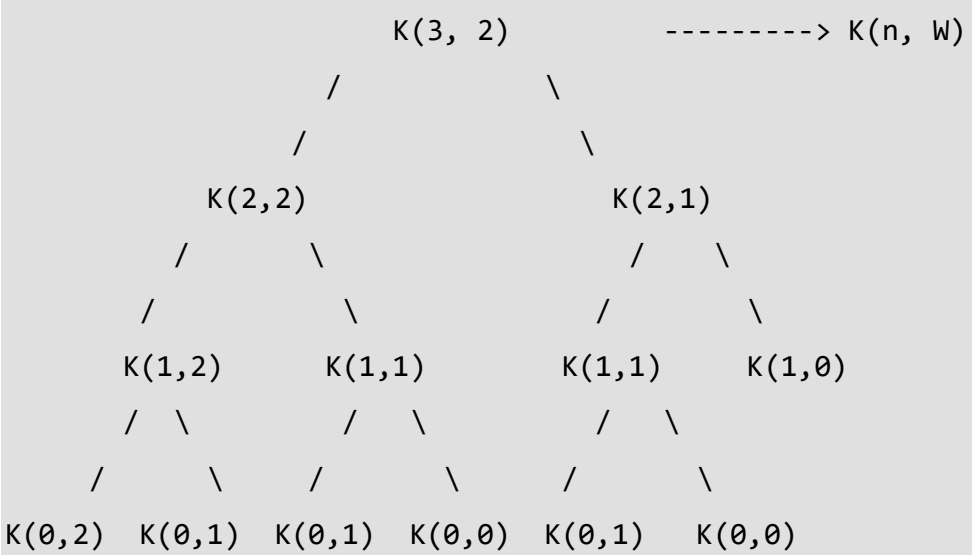
Recursion Algorithm:

```
// Values (stored in array v)
// Weights (stored in array w)
// Number of distinct items (n)
// Knapsack capacity W
int knapSack(int v[], int w[], int n, int W)
{
        // base case: Negative capacity
        if (W < 0)
                return INT_MIN;

        // base case: no items left or capacity becomes 0
        if (n < 0 || W == 0)
                return 0;

        // Case 1. include current item n in knapSack (v[n]) and recur for
        // remaining items (n - 1) with decreased capacity (W - w[n])
        int include = v[n] + knapSack(v, w, n - 1, W - w[n]);

        // Case 2. exclude current item n from knapSack and recur for
        // remaining items (n - 1)
        int exclude = knapSack(v, w, n - 1, W);

        // return maximum value we get by including or excluding current item
        return max (include, exclude);
}
```

The time complexity of the above solution is exponential as each recursive call is making n recursive calls. The problem has an optimal substructure as the problem can be broken down into yet smaller sub problems and so on. It also has an overlapping substructure so we will end up solving the same sub problem again and again. We can see that by drawing the recursion tree.

In the following recursion tree, K() refers to knapSack(). The two parameters indicated in the following recursion tree are n and W. The recursion tree is for following sample inputs.

wt[] = {1, 1, 1}, W = 2, val[] = {10, 20, 30}



Recursion tree for Knapsack capacity 2 units and 3 items of 1 unit weight.

Memoization algorithm:

```
unordered_map<string, int> lookup;

// Values (stored in array v)
// Weights (stored in array w)
// Number of distinct items (n)
// Knapsack capacity W
int knapSack(int v[], int w[], int n, int W)
{
	// base case: Negative capacity
	if (W < 0)
		return INT_MIN;

	// base case: no items left or capacity becomes 0
	if (n < 0 || W == 0)
		return 0;

	// construct a unique map key from dynamic elements of the input
	string key = to_string(n) + "|" + to_string(W);

	// if sub-problem is seen for the first time, solve it and
	// store its result in a map
	if (lookup.find(key) == lookup.end())
```

```
        {
                // Case 1. include current item n in knapSack (v[n]) & recur for
                // remaining items (n - 1) with decreased capacity (W - w[n])
                int include = v[n] + knapSack(v, w, n - 1, W - w[n]);

                // Case 2. exclude current item n from knapSack and recur for
                // remaining items (n - 1)
                int exclude = knapSack(v, w, n - 1, W);

                // assign max value we get by including or excluding current item
                lookup[key] = max (include, exclude);
        }

        // return solution to current sub-problem
        return lookup[key];
}
```

The time complexity of above solution is O(n x W) ,where W is the Knapsack capacity and n is the number of items in the input.