0-1 Knapsack

Question : 0-1 Knapsack
Given two integer arrays **A** and **B** of size **N** each which represent **values** and **weights** associated with **N** items respectively.

Also given an integer **C** which represents knapsack capacity.

Find out the maximum value subset of **A** such that sum of the weights of this subset is smaller than or equal to **C**.

NOTE:

- You cannot break an item, either pick the complete item, or don't pick it (0-1 property).

Approach1:
maxVal(K, len) : The maximum value that we can make using the elements [0…. len-1] with a knapsack of capacity K.

Note: Here, we choose elements such that the sum of weights of the elements chosen <= K (capacity of knapsack).
The sum of weights need not be equal to K. They should be <= K.

$$\text{maxVal}(K, \text{len}) = \max(\text{maxVal}(K, \text{len} - 1),\ \text{val}[\text{len}-1] + \text{maxVal}(K - \text{wt}[\text{len}-1], \text{len} - 1))$$

Base Cases:

1. len = 0
   maxVal(K, 0) represents the maximum value that we can make using an empty array with a knapsack of capacity K.
   RETURN 0
2. K = 0
   maxVal(0, len) represents the maximum value that we can make with a knapsack of capacity 0.
   RETURN 0
3. K < 0
   maxVal(-3, len) represents the maximum value that we can make with a knapsack of capacity -3. This is an invalid case and ideally we should never land in this case. Negative capacity does NOT make any sense. Hence, return a huge negative number.
   RETURN MIN

Variation1: An extra if condition to make sure we don't land in the case of negative capacity!

```
int maxVal(int K, int len, vector<int> &val, vector<int> &wt){
    if(len == 0 || K == 0)
        return 0;

    if(dp[K][len] != -1) return dp[K][len];

    if(wt[len - 1] <= K)
        dp[K][len] = max(maxVal(K, len - 1, val, wt),
                         val[len-1] + maxVal(K - wt[len-1], len - 1, val, wt));
    else
        dp[K][len] = maxVal(K, len - 1, val, wt);

    return dp[K][len];
}

int Solution::solve(vector<int> &A, vector<int> &B, int C) {
    memset(dp, -1, sizeof(dp));
    return maxVal(C, A.size(), A, B);
}
```

Variation2 (If you're lazy like me :P): Put an extra base case to return a huge negative value when we land in the case of negative capacity!

```
int maxVal(int K, int len, vector<int> &val, vector<int> &wt){
    if(K < 0)
        return MIN;
    if(len == 0 || K == 0)
        return 0;

    if(dp[K][len] != -1) return dp[K][len];

    dp[K][len] = max(maxVal(K, len - 1, val, wt),
                     val[len-1] + maxVal(K - wt[len-1], len - 1, val, wt));
    return dp[K][len];
}

int Solution::solve(vector<int> &A, vector<int> &B, int C) {
    memset(dp, -1, sizeof(dp));
    return maxVal(C, A.size(), A, B);
}
```

Approach2:
maxVal(K, len) : The maximum value that we can make using the elements [0…. len-1] such that the sum of weights of items chosen is exactly equal to K.

Note: In the previous approach, we chose elements whose sum of weights was less than or equal to K. But, in this approach we want it to be exactly equal to K.

The recursive relation remains the same!
maxVal(K, len) = max(maxVal(K, len - 1), val[len-1] + maxVal(K - wt[len-1], len - 1))

Base Cases :

1. len = 0
   maxVal(K, 0) represents the maximum value that we can make using an empty array such that the sum of weights of items is exactly equal to K. If K is non-zero, this is an impossible case. We can never make a non-zero sum of weights with an empty array
   a. If K != 0
      RETURN MIN
   b. If K = 0
      RETURN 0

   RETURN 0
2. K = 0
   maxVal(0, len) represents the maximum value that we can make such that the sum of weights of items is equal to 0. This will be 0
   RETURN 0
3. K < 0
   maxVal(-3, len) represents the maximum value that we can make such that the sum of weights is equal to -3. This is an invalid case and ideally we should never land at this case. Negative weight does NOT make any sense. Hence, return a huge negative number.
   RETURN MIN

The problem is not completely done yet!
maxVal(K, len) only tells us the maximum value that we can make using the elements [0…. len-1] such that the sum of weights of items chosen is exactly equal to K.

If we cannot make sum of weights equal to K, the above function will return MIN. Hence, the answer to the question is not just dp[Capacity][Length of A]!

We are only concerned with maximising the value and least concerned about how much weight we have filled in the knapsack.

Iterate the last row of the dp table and find the maximum value!

Variation1: An extra if condition to make sure we don't land in the case of negative weight!

```
int maxVal(int K, int len, vector<int> &val, vector<int> &wt){
    if(len == 0){
        if(K == 0) return 0;
        return MIN;
    }

     if(K == 0) return 0;

    if(dp[K][len] != -1) return dp[K][len];

    if(wt[len - 1] <= K)
        dp[K][len] = max(maxVal(K, len - 1, val, wt),
                val[len-1] + maxVal(K - wt[len-1], len - 1, val, wt));
    else
        dp[K][len] = maxVal(K, len - 1, val, wt);


    return dp[K][len];
}

int Solution::solve(vector<int> &A, vector<int> &B, int C) {
    memset(dp, -1, sizeof(dp));

    int ans = INT_MIN;
    for(int K = C; K >= 0; K--)
        ans = max(ans, maxVal(K, A.size(), A, B));
    return ans;
}
```

Variation2 : Put an extra base case to return a huge negative value when we land in the case of negative weight!

```
int maxVal(int K, int len, vector<int> &val, vector<int> &wt){
    if(len == 0){
        if(K == 0) return 0;
        return MIN;
    }
     if(K == 0) return 0;
     if(K < 0) return MIN;

    if(dp[K][len] != -1) return dp[K][len];

    dp[K][len] = max(maxVal(K, len - 1, val, wt),
                     val[len-1] + maxVal(K - wt[len-1], len - 1, val, wt));
   return dp[K][len];
}

int Solution::solve(vector<int> &A, vector<int> &B, int C) {
    memset(dp, -1, sizeof(dp));

    int ans = INT_MIN;
    for(int K = C; K >= 0; K--)
        ans = max(ans, maxVal(K, A.size(), A, B));
    return ans;
}
```