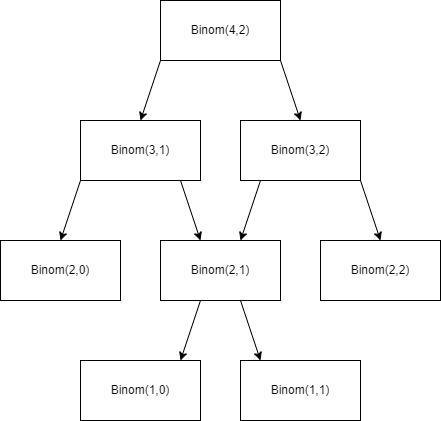
Diagram

Description automatically generated

The recursive formula for binomial coefficients is: , where the base case is whenor . The recursion tree for Binom(4,2) is:



Diagram

Description automatically generated

memoizedTable = {} *# This is a Python dictionary*

def binomial\_coefficient(n, k):

    assert n >= k

    if k == 0 or k == n:

        return 1

    key = (n, k)

    if key not in memoizedTable:

        memoizedTable[(n,k)] = binomial\_coefficient(n-1, k) + binomial\_coefficient(n-1, k-1)

        return memoizedTable[(n,k)]

    else:

        return memoizedTable[(n,k)]

print(binomial\_coefficient(4,2))

The memoized table saves the precomputed values. This table is in a form of Pascal’s triangle

Diagram

Description automatically generated

* Time Complexity of my algorithm:

Given that all precomputed (n,k) values are saved inside the dictionary, we can be sure that it will run rerun those values again if they are encountered again. In other words, at most configuration is calculated and we know that at the base case, it runs in . Therefore, this algorithm runs in 

* Memory complexity of my algorithm

Because the dictionary only saves keys (n, k) such that and n runs from 1 to n and k runs from 1 to k, there are n rows and (n-k) columns. Therefore, the memory complexity is 

Diagram

Description automatically generated

This is a DAG because there is no cycles in this graph.

Diagram

Description automatically generated with medium confidence

First, node 1 has only outcoming edges and node 5 has only incoming edges, so we can safely omit them when we search for cycles in this graph, because cycles cannot go through nodes that solely have outcoming or incoming edges.   
=> There are only nodes 2,3,4 left. We can see that node 4 goes to node 2, node 2 goes to node 3. But node 3 doesn’t go back to node 4. Therefore, there isn’t a cycle in the remaining nodes

Diagram

Description automatically generated

This is not a DAG because there is a cycle in this graph

A picture containing clock, watch

Description automatically generated

Like part i. above, we can omit nodes 1 and 5 because they only have either incoming or outcoming edges. There is an edge going from node 2 to node 3, node 3 to node 4, and node 4 to node 2. Therefore, it exists a cycle that goes around nodes 2-3-4 in this graph

Text

Description automatically generated  
Text, letter

Description automatically generatedFirst case:   
Interpretation of : if item k is not included in the knapsack, then the optimal subproblem value is the same as the one when we do not consider the item k at all with the same weight.   
Proof by contradition for this case: Suppose that  is correct. Then there must be another selection such that . From the subproblem value , we know that the optimal selection does not have the item k in the next iteration, so  does not increase. This is a contradiction and there does not exist any  such that  =>  is always correct.

Second case:   
Interpretation of : if item k is included in the knapsack, then the optimal subproblem value is the one when we do not consider the item k with the new weight subtracted by the weight of item k from the current weight, plus the value of the item k.   
Proof by contradition for this case: Suppose that  is correct, then there must be another selection with a higher value and it means that . When we add the kth item to , we have. Now we have a subset of the first k items with higher value than , which is the optimal value by definition. This is a contradiction   
=>  is always correct.

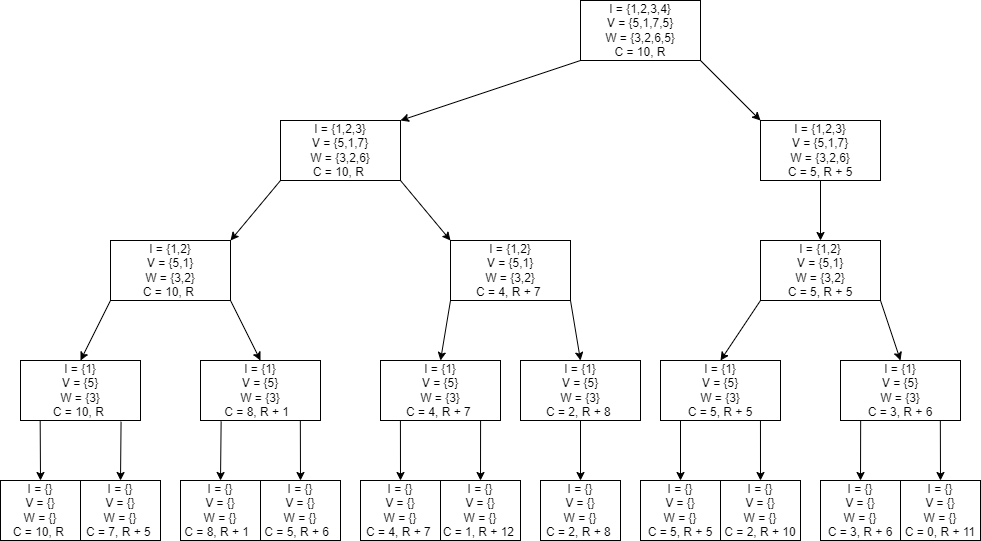
Table

Description automatically generated

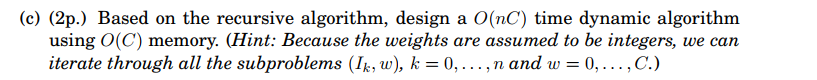
Graphical user interface, text, application, email

Description automatically generated

The recursion tree for the given items and capacity is as follows:



* Solution: the optimal packings is items = {1, 3}, with weights = 9 and maximum value = 12



In the “tradition” dynamic algorithm for 0/1 knapsack problem, a table of size n x C is required, where n is the number of items and C is the capacity. Upon closer analysis, the computation for the value of table[i][j] only depends on the solutions from the previous row. In other words, we can use only a table of one row with the size of (C + 1), where table[C] stores the final result. For each iteration in an n-for loop, we can overwrite previous values in the table, which is no longer needed for the current computation.

The dynamic algorithm in Python language is

*# values[] stores the values for each item*

*# weights[] stores the weights for each item*

*# n is the number of items*

*# C is the maximum capacity of bag*

*# table[C+1] to store final result*

def knapSack(values, weights, n, C):

    table = [0]\*(C+1);

    for i in range(n):

        for j in range(C, weights[i] - 1,-1):

            table[j] = max(table[j] , values[i] + table[j - weights[i]]);

    return table[C];

Time Complexity: there are two for-loops, one runs for n times and another for C times. In total, the dynamic algorithm runs in O(nC) time

Memory complexity of the dynamic algorithm: we only need a table of size (C + 1) to store the current latest results => The auxiliary table is required in O(C + 1) or O(C)