1. **Consider the Knapsack problem algorithm we covered in class i.e.**

**max {V[i-1,j], vi + V[i-1,j- wi]} if j- wi ³ 0**

**V[i,j] =**

**V[i-1,j] if j- wi < 0**

**Initial conditions: V[0,j] = 0 and V[i,0] = 0.**

**This does not list the items selected when a final optimized solution is obtained. Use this algorithm and modify it by adding a new array Keep[i,w] which will keep track of the items selected. Write a pseudo code for this updated algorithm. Also, make sure you print the results showing items selected [Hint: use similar approach as in Print-Cut\_Rod\_Soln].**

**Ans:**

Given n items with weights[1..n] and values[1..n], and knapsack capacity W:

Create tables V[0..n][0..W] and Keep[1..n][0..W]

Initialize V[0][j] = 0 and V[i][0] = 0

Fill tables:

for i = 1 to n:

for j = 0 to W:

if weights[i] ≤ j and values[i] + V[i-1][j-weights[i]] > V[i-1][j]:

V[i][j] = values[i] + V[i-1][j-weights[i]]

Keep[i][j] = 1

else:

V[i][j] = V[i-1][j]

Keep[i][j] = 0

Backtrack for selected items:

j = W

selected = []

for i = n down to 1:

if Keep[i][j] == 1:

selected.append(i)

j -= weights[i]

Result: Maximum value = V[n][W], Items = selected

Time complexity: O(nW), Space complexity: O(nW)

**Q2. Determine the running time complexity of your pseudo code. Has Dynamic Programming approach of solving Knapsack problem changed the exponential time complexity of the original brute force solution?**

The running time complexity of the knapsack algorithm is O(nW), where n represents the number of items and W is the maximum capacity of the knapsack. This complexity arises because the algorithm uses two nested loops - the outer loop iterates through all n items while the inner loop processes each possible capacity value from 0 to W. Since each operation inside these loops takes constant time, the total number of operations is proportional to n × W.

What makes this dynamic programming approach valuable is the significant improvement over the brute force solution, which would require examining all possible combinations of items with a time complexity of O(2ⁿ). For problems with many items, this reduction from exponential to polynomial time complexity represents a dramatic performance enhancement, making previously intractable problems solvable in reasonable time.

**Q3. Use the answer of Q2 to explain how Knapsack problem is exponential (Hint - consider both inputs and see how they [or at least one] can cause Knapsack problem to be exponential).**

The original brute-force solution for the Knapsack problem has exponential complexity because:

For n items, there are 2ⁿ possible subsets of items

Each subset must be evaluated to check if it fits within capacity W and to calculate its value

Dynamic programming improves this significantly by:

Breaking the problem into smaller overlapping subproblems

Storing intermediate results to avoid redundant calculations

Reducing time complexity to O(nW)

Pseudo-Polynomial Classification

Despite the improvement, the Knapsack problem remains "pseudo-polynomial" because:

The capacity W directly influences the algorithm's time complexity

The number of items n affects the number of iterations

While O(nW) is better than O(2ⁿ), it is not truly polynomial in terms of input size

This is because W is not polynomial in terms of the number of input bits but grows linearly with the numeric value of W itself. In formal complexity analysis, a truly polynomial algorithm should depend only on the number of bits needed to represent the input, not on the numeric value of the input.