**Assignment 5**

**Name: Atharva Salitri**

**Roll No.: 029**

**Branch: TY CSAI-B**

**Batch: B2**

**PRN: 12310120**

**Title:**

Assignment Based on Dynamic Programming to Implement 0/1 Knapsack Problem (Determine Time and space complexity)

**Theory:**

**Objective**

To implement the 0/1 Knapsack problem using dynamic programming and analyze its time and space complexity.

**Theory and Explanation**

The 0/1 Knapsack problem involves selecting a subset of items with given weights and profits to maximize the total profit without exceeding the knapsack's capacity. Each item can either be included **completely (1)** or excluded **completely (0)** — partial selection is not allowed.

We aim to maximize the total profit ∑pixi\sum p\_i x\_i∑pixi subject to ∑wixi≤W\sum w\_i x\_i \leq W∑wixi≤W, where WWW is the knapsack capacity, pip\_ipi and wiw\_iwi are the profit and weight of item iii, and xi∈{0,1}x\_i \in \{0,1\}xi∈{0,1}.

Dynamic programming solves this by breaking down into smaller subproblems and storing intermediate results to avoid redundant computations. The subproblem is: the max profit using the first iii items with capacity jjj.

The recursive relation is:

K[i][j]={0if i=0 or j=0K[i−1][j]if wi>jmax⁡(K[i−1][j],pi+K[i−1][j−wi])if wi≤jK[i][j] = \begin{cases} 0 & \text{if } i=0 \text{ or } j=0 \\ K[i-1][j] & \text{if } w\_i > j \\ \max(K[i-1][j], p\_i + K[i-1][j - w\_i]) & \text{if } w\_i \leq j \end{cases}K[i][j]=⎩⎨⎧0K[i−1][j]max(K[i−1][j],pi+K[i−1][j−wi])if i=0 or j=0if wi>jif wi≤j

Where:

* K[i][j]K[i][j]K[i][j] = max profit with first iii items and capacity jjj
* wi,piw\_i, p\_iwi,pi are weight and profit of the ithi^{th}ith item

The bottom-up filling of the table KKK iteratively computes each value until the answer K[n][W]K[n][W]K[n][W] is found.

**Key Points**

* Items cannot be broken; either full item or none.
* Results of subproblems are stored to avoid recomputation.
* The solution explores both including and excluding each item.
* Final answer is K[n][W] — max profit for nnn items and capacity W.

**Pseudocode**

text

function knapsack(W, wt[], val[], n):

create 2D array K[n+1][W+1]

for i from 0 to n:

for j from 0 to W:

if i == 0 or j == 0:

K[i][j] = 0

else if wt[i-1] <= j:

K[i][j] = max(val[i-1] + K[i-1][j - wt[i-1]], K[i-1][j])

else:

K[i][j] = K[i-1][j]

return K[n][W]

**Code:**

import java.util.Scanner;

public class Knapsack {

    public static int knapsack(int W, int wt[], int val[], int n) {

        int[][] K = new int[n + 1][W + 1];

        for (int i = 0; i <= n; i++) {

            for (int w = 0; w <= W; w++) {

                if (i == 0 || w == 0) {

                    K[i][w] = 0;

                } else if (wt[i - 1] <= w) {

                    K[i][w] = Math.max(val[i - 1] + K[i - 1][w - wt[i - 1]], K[i - 1][w]);

                } else {

                    K[i][w] = K[i - 1][w];

                }

            }

        }

        return K[n][W];

    }

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        System.out.println("Enter number of items:");

        int n = sc.nextInt();

        int[] val = new int[n];

        int[] wt = new int[n];

        System.out.println("Enter value and weight of each item:");

        for (int i = 0; i < n; i++) {

            val[i] = sc.nextInt();

            wt[i] = sc.nextInt();

        }

        System.out.println("Enter the capacity of the knapsack:");

        int W = sc.nextInt();

        int maxProfit = knapsack(W, wt, val, n);

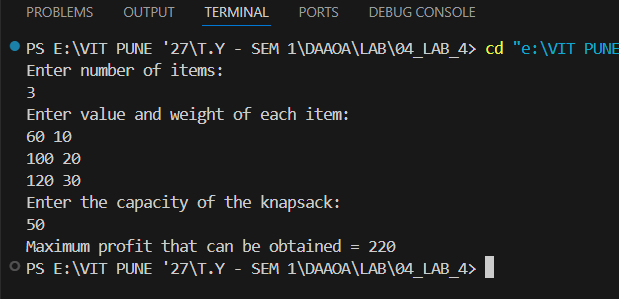
        System.out.println("Maximum profit that can be obtained = " + maxProfit);

        sc.close();

    }

}

**OUTPUT:**

****

**Time and Space Complexity Analysis:**

**Time Complexity**

* The algorithm fills an (n+1)×(W+1) table.
* Each cell computation takes constant time O(1)
* Total time complexity: **O(nW)** where n is item count and W is knapsack capacity.

**Space Complexity**

* Uses a 2D array Kof size (n+1)×(W+1)
* Requires space proportional to number of items and capacity.
* Total space complexity: **O(nW)**

**Pseudocode with Complexity Comments**

text

FUNCTION knapsack(W, wt, val, n)

DECLARE 2D array K of size (n+1) x (W+1) // Space: +(n+1)\*(W+1) = O(n\*W)

FOR i FROM 0 TO n // Time: +n+1

FOR w FROM 0 TO W // Time: +(W+1) per i; Total: \*n\*W

IF i == 0 OR w == 0 // Time: +1 per iteration

K[i][w] ← 0 // Time: +1

ELSE IF wt[i-1] <= w // Time: +1

K[i][w] ← MAX(val[i-1] + K[i-1][w - wt[i-1]], K[i-1][w]) // Time: +1 (max and addition)

ELSE

K[i][w] ← K[i-1][w] // Time: +1

ENDIF

ENDFOR

ENDFOR

RETURN K[n][W] // Time: +1 (return)

ENDFUNCTION

FUNCTION main

DECLARE scanner // Space: +1

PRINT "Enter number of items:" // Time: +1

INPUT n // Time: +1

DECLARE arrays val[n], wt[n] // Space: +n each = +2n total

PRINT "Enter value and weight of each item:" // Time: +1

FOR i FROM 0 TO n-1 // Time: +n

INPUT val[i], wt[i] // Time: +1 per read

ENDFOR

PRINT "Enter the capacity of the knapsack:" // Time: +1

INPUT W // Time: +1

maxProfit ← knapsack(W, wt, val, n) // Time: O(n\*W), Space: O(n\*W)

PRINT "Maximum profit that can be obtained = " + maxProfit // Time: +1

CLOSE scanner // Time: +1

ENDFUNCTION

**Complexity Explanation**

* **Time Complexity:** The nested loops iterate over each item (n) and capacity (W), so overall the time complexity is O(n×W))
* **Space Complexity:** The 2D DP table K requires O(n×W) space to store intermediate results.
* Input and output operations take linear time and constant extra space outside the storage arrays.
* All constant time operations (+1) occur within nested loops to build the solution table.

**Conclusion**

In this lab exercise, we learned how to implement 0-1 Knapsack problem using Dynamic Programming.