**Name: Atharva Salitri**

**Roll No.: 029**

**Branch: TY CSAI-B**

**Batch: B2**

**PRN: 12310120**

# Experiment Number: 03

**Title:** Assignment Based on Dynamic programming strategy. (Implement 0-1 Knapsack problem using cpp or java)

|  |  |  |  |
| --- | --- | --- | --- |
| **Title of Experimentation** | **CO**  **Mapping** | **CO-Statements** | **PO**  **Mapping** |
| Assignment Based on Dynamic programming strategy. (Implement 0-1 Knapsack problem using cpp or java) | CO1, CO2,C03 | To analyze asymptotic complexity of the algorithm for a complex computational problem using suitable mathematical techniques | PO1, PO2, PO3, PO4, PO5, PO11 |

**Theory:**

**0-1 Knapsack Problem** is a classic optimization problem:

* Given n items, each with a weight w[i] and a profit p[i], and a knapsack with capacity W.
* The goal is to maximize total profit by selecting items without exceeding capacity W.
* Each item can either be **taken (1)** or **not taken (0)** → hence “0-1 Knapsack.”

Dynamic Programming (DP) is used to solve this problem efficiently by avoiding recomputation.  
The key recurrence relation is:

dp[i][w]=max(dp[i−1][w],profit[i−1]+dp[i−1][w−weight[i−1]])

where:

* dp[i][w] = max profit using first i items with knapsack capacity w.

### **Input:**

* Number of items n
* Profit array p[n]
* Weight array w[n]
* Knapsack capacity W

### **Output:**

* Maximum profit achievable within capacity W.

### **Objective of Experiment:**

To implement and demonstrate how **Dynamic Programming** can be applied to solve the **0-1 Knapsack Problem** efficiently compared to recursive brute force.

### **Algorithm (Dynamic Programming – Bottom-Up):**

1. Initialize a DP table dp[n+1][W+1].
2. For each item i (1…n):
   * For each capacity w (1…W):
     + If weight[i-1] <= w, compute:

dp[i][w]=max⁡(dp[i−1][w], profit[i−1]+dp[i−1][w−weight[i−1]])dp[i][w] = \max(dp[i-1][w], \, profit[i-1] + dp[i-1][w - weight[i-1]])dp[i][w]=max(dp[i−1][w],profit[i−1]+dp[i−1][w−weight[i−1]])

* + - Else, inherit previous value:

dp[i][w]=dp[i−1][w]dp[i][w] = dp[i-1][w]dp[i][w]=dp[i−1][w]

1. Result is stored in dp[n][W].

### **Pseudo Code:**

function knapsack(profit[], weight[], n, W):

create dp[n+1][W+1]

for i = 0 to n:

for w = 0 to W:

if i == 0 or w == 0:

dp[i][w] = 0

else if weight[i-1] <= w:

dp[i][w] = max(profit[i-1] + dp[i-1][w - weight[i-1]],

dp[i-1][w])

else:

dp[i][w] = dp[i-1][w]

return dp[n][W]

### **Java Implementation:**

import java.util.Scanner;

public class KnapsackDP {

public static int knapsack(int[] profit, int[] weight, int n, int W) {

int[][] dp = 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) {

dp[i][w] = 0;

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

dp[i][w] = Math.max(profit[i - 1] + dp[i - 1][w - weight[i - 1]],

dp[i - 1][w]);

} else {

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

}

}

}

return dp[n][W];

}

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

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

int n = sc.nextInt();

int[] profit = new int[n];

int[] weight = new int[n];

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

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

profit[i] = sc.nextInt();

}

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

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

weight[i] = sc.nextInt();

}

System.out.print("Enter knapsack capacity: ");

int W = sc.nextInt();

int maxProfit = knapsack(profit, weight, n, W);

System.out.println("Maximum profit = " + maxProfit);

sc.close();

}

}

**Input:**

Enter number of items: 3

Enter profits of items:

60 100 120

Enter weights of items:

10 20 30

Enter knapsack capacity: 50

**Output:**

Maximum profit = 220

### **Flowchart (suggested structure):**

* **Start**
* Input: n, profits[], weights[], W
* Initialize DP table
* For each item i = 1 to n  
  → For each capacity w = 1 to W  
  → If item fits → choose max(include, exclude)  
  → Else inherit previous
* End loops
* Output dp[n][W]

**Source Code, with description and with Output Need to be Uploaded to the VOLP**

**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);

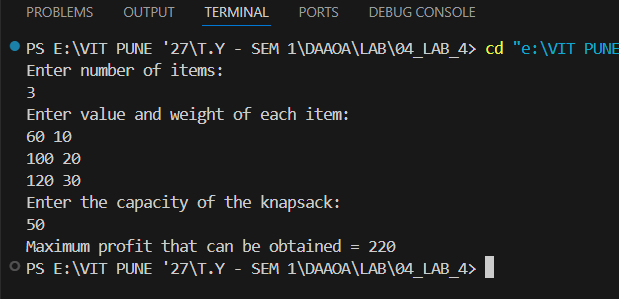
        System.out.println("Maximum profit that can be obtained = " + maxProfit);

        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.