



## Experiment 2.3

**Student Name:** Suraj Sagar

**Branch:** CSE

**Semester:** 5th

**Subject Name:** AI&ML

**UID:** 21BCS3388

**Section/Group:** 602-B

**Date of Performance:** 11/10/23

**Subject Code:** 21CSH-316

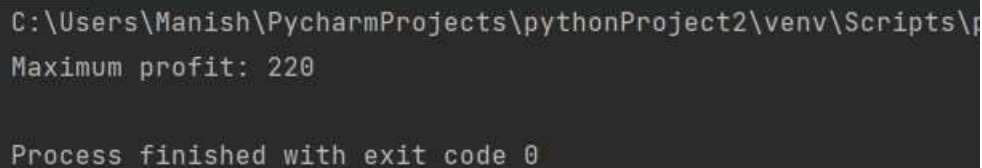
- 1. Aim:** Develop a program and analyze complexity to implement 0-1 Knapsack using Dynamic Programming
- 2. Objective:** The objective of this experiment is to find the optimal solution to a classic combinatorial optimization problem.

### 3. Sample Code-

```
def knapsack(profit, weights, capacity):  
    n = len(profit)  
    dp = [[0] * (capacity + 1) for _ in range(n + 1)]  
  
    for i in range(n + 1):  
        for w in range(capacity + 1):  
            if i == 0 or w == 0:  
                dp[i][w] = 0  
            elif weights[i - 1] <= w:  
                dp[i][w] =  
                max(dp[i - 1][w], dp[i - 1][w - weights[i - 1]] + profit[i - 1])  
            else:  
                dp[i][w] = dp[i - 1][w]  
  
    return dp[n][capacity]  
  
# Example usage
```

```
profit = [60, 100, 120]
weights = [10, 20, 30]
capacity = 50
result = knapsack(profit, weights, capacity)
print("Maximum profit:", result)
```

## 4. Outcome-



```
C:\Users\Manish\PycharmProjects\pythonProject2\venv\Scripts\p
Maximum profit: 220

Process finished with exit code 0
```

## 5. Code Explanation-

- The function `knapsack` takes three arguments: `values` (a list of values for the items), `weights` (a list of weights for the items), and `capacity` (the maximum weight the knapsack can hold).
- `n` is the number of items in the dataset.
- `dp` is a 2D list (a table) with  $(n + 1)$  rows and  $(\text{capacity} + 1)$  columns to store the dynamic programming results. It represents the maximum value that can be obtained with different subsets of items and various capacities.

- This is a nested loop structure that iterates through each item (from 0 to n) and each possible capacity (from 0 to the given capacity).
- For the base case, when there are no items ( $i == 0$ ) or when the capacity is zero ( $w == 0$ ), the maximum value is 0. These are the initial conditions for the dynamic programming table.
- For each item (indexed by  $i$ ) and each capacity (indexed by  $w$ ), the code checks two conditions:
  - If the weight of the current item ( $weights[i - 1]$ ) is less than or equal to the current capacity ( $w$ ), it considers two options:
    - Exclude the current item:  $dp[i - 1][w]$
    - Include the current item:  $dp[i - 1][w - weights[i - 1]] + values[i - 1]$  - The maximum value of these two options is stored in  $dp[i][w]$ .
  - If the weight of the current item is greater than the current capacity, it means the item cannot be included, so the value remains the same as the value obtained without this item.
- The function returns the maximum value that can be obtained with the entire set of items (n) and the given knapsack capacity (capacity).

Finally, the code provides an example of how to use the `knapsack` function by specifying `values`, `weights`, and `capacity`. The maximum value is then printed.

The **time complexity of this dynamic programming solution is  $O(n * capacity)$** , where  $n$  is the number of items and `capacity` is the maximum capacity of the knapsack.