**Dynamic Programming Beginner’s Tutorial**

Dynamic programming is a method for solving a complex problem by breaking it down into simpler subproblems, solving each of those subproblems just once, and storing their solutions – in an array(usually).

Now, every time the same sub-problem occurs, instead of recomputing its solution, the previously calculated solutions are used, thereby saving computation time at the expense of storage space.

Dynamic programming can be implemented in two ways –

* Memoization
* Tabulation

**Memoization** – Memoization uses the top-down technique to solve the problem i.e. it begin with original problem then breaks it into sub-problems and solve these sub-problems in the same way.

In this approach, you assume that you have already computed all subproblems. You typically perform a recursive call (or some iterative equivalent) from the main problem. You ensure that the recursive call never recomputes a subproblem because you cache the results, and thus duplicate sub-problems are not recomputed.

**Tabulation** – Tabulation is the typical Dynamic Programming approach. Tabulation uses the bottom up approach to solve the problem, i.e., by solving all related sub-problems first, typically by storing the results in an array. Based on the results stored in the array, the solution to the “top” / original problem is then computed.

Memoization and tabulation are both storage techniques applied to avoid recomputation of a subproblem

# 0 1 Knapsack Problem – Dynamic Programming Solutions

This is a C++ Program that Solves 0 1 Knapsack Problem using Dynamic Programming technique.

**Problem Description**

Given weights and values of n items, put these items in a knapsack of capacity M to get the maximum total value in the knapsack.

Note that, you can select items, the sum of whose weight is less than or equal to the capacity of knapsack, W.

# C++ Program for the Fractional Knapsack Problem

**Pre-requisite:** Fractional Knapsack Problem

Given two arrays **weight[ ]** and **profit[ ]** the *weights* and profit of **N** items, we need to put these items in a knapsack of capacity **W** to get the *maximum* total value in the knapsack.  
**Note:** Unlike 0/1 knapsack, you are allowed to break the item.

**Examples:**

***Input:****weight[] = {10, 20, 30}, profit[] = {60, 100, 120}, N= 50****Output:****Maximum profit earned = 240****Explanation:*** *Decreasing p/w ratio[] = {6, 5, 4}  
Taking up the weight values 10, 20, (2 / 3) \* 30   
Profit = 60 + 100 + 120 \* (2 / 3) = 240*

***Input:****weight[ ] = {10, 40, 20, 24}, profit[ ] = {100, 280, 120, 120}, N = 60****Output:****Maximum profit earned = 440****Explanation:****Decreasing p/w ratio[] = {10, 7, 6, 5}  
Taking up the weight values 10, 40, (1 / 2) \* 120   
Profit = 100 + 280 + (1 / 2) \* 120 = 440*

**Method 1 – without using STL:** The idea is to use Greedy Approach. Below are the steps:

1. Find the ratio **value/weight** for each item and sort the item on the basis of this ratio.
2. Choose the item with the highest ratio and add them until we can’t add the next item as a whole.
3. In the end, add the next item as much as we can.
4. Print the maximum profit after the above steps.

Below is the implementation of the above approach:

**Output:**

Maximum profit earned = 440

**Method 2 – using STL:**

1. Create a map with **profit[i] / weight[i]** as first and i as Second element for each element.
2. Define a variable **max\_profit = 0**.
3. Traverse the map in reverse fashion:
   * Create a variable named fraction whose value is equivalent to remaining\_weight / weight[i].
   * If **remaining\_weight** is greater than or equals to zero and its value is greater than **weight[i]** add current profit to **max\_profit** and reduce the remaining weight by **weight[i]**.
   * Else if remaining weight is less than weight[i] add **fraction \* profit[i]** to **max\_profit** and break.
4. Print the **max\_profit**.

 Below is the implementation of the above approach:

**Output:**

Maximum profit earned is:440