UNIT-III

Dynamic Programming: General method, Multi stage graph, applications - Matrix chain multiplication, Optimal binary search trees, 0/1 knapsack problem, All pairs shortest path problem, Travelling sales person problem.

****

Dynamic programming

Dynamic programming is a name, coined by Richard Bellman in 1955. Dynamic programming, as greedy method, is a powerful algorithm design technique that can be used when the solution to the problem may be viewed as the **result of a sequence of decisions**. In the greedy method we make irrevocable decisions one at a time, using a greedy criterion. However, in dynamic programming we examine the decision sequence to see whether an optimal decision sequence contains optimal decision subsequence.

When optimal decision sequences contain optimal decision subsequences, we can establish recurrence equations, called dynamic-programming recurrence equations, that enable us to solve the problem in an efficient way.

Dynamic programming is **based on the principle of optimality** (also coined by Bellman). The principle of optimality states that no matter whatever the initial state and initial decision are, the remaining decision sequence must constitute an optimal decision sequence with regard to the state resulting from the first decision. The principle implies that an optimal decision sequence is comprised of optimal decision subsequences. Since the principle of optimality may not hold for some formulations of some problems, it is necessary to verify that it does hold for the problem being solved. Dynamic programming cannot be applied when this principle does not hold.

The steps in a dynamic programming solution are:

- Verify that the principle of optimality holds
- Set up the dynamic-programming recurrence equations
- Solve the dynamic-programming recurrence equations for the value of the optimal solution.
- Perform a trace back step in which the solution itself is constructed.

Dynamic programming differs from the greedy method since the greedy method produces only one feasible solution, which may or may not be optimal, while dynamic programming produces all possible sub-problems at most once, one of which guaranteed to be optimal. Optimal solutions to sub-problems are retained in a table, **thereby avoiding the work of re computing** the answer every time a sub-problem is encountered.

The divide and conquer principle solve a large problem, by breaking it up into smaller problems which can be solved independently. In dynamic programming this principle is carried to an extreme: when we don't know exactly which smaller problems to solve, we simply solve them all, and then store the answers away in a table to be used later in solving larger problems. Care is to be taken to avoid re computing previously computed values, otherwise the recursive program will have prohibitive complexity.

In some cases, the solution can be improved and in other cases, the dynamic programming technique is the best approach.

Let's understand this approach through an example.

Consider an example of the Fibonacci series. The following series is the Fibonacci series:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ,...

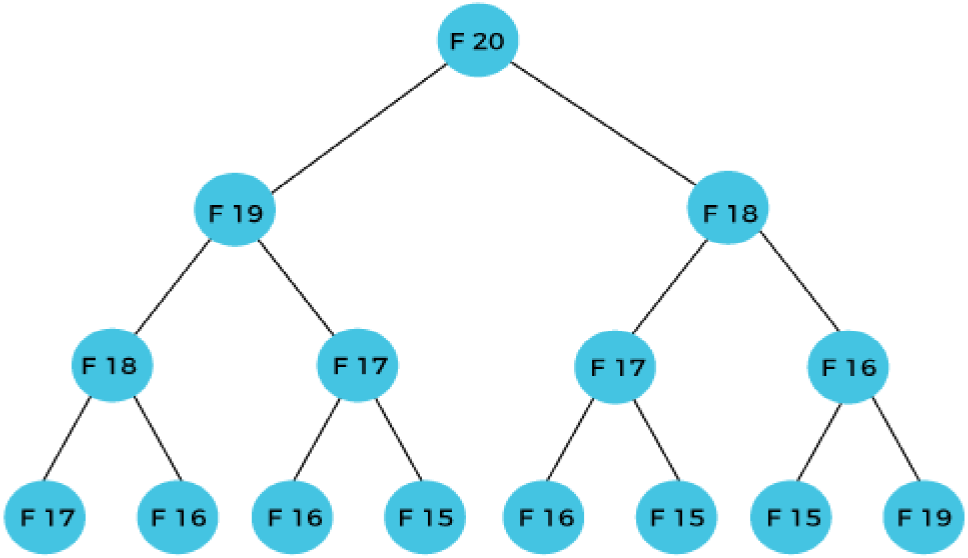
The numbers in the above series are not randomly calculated. Mathematically, we could write each of the terms using the below formula:

F(n) = F(n-1) + F(n-2),

With the base values F(0) = 0, and F(1) = 1. To calculate the other numbers, we follow the above relationship. For example, F(2) is the sum **f(0)** and **f(1),** which is equal to 1.

How can we calculate F(20)?

The F(20) term will be calculated using the nth formula of the Fibonacci series. The below figure shows that how F(20) is calculated.



In the above example, if we calculate the F(18) in the right subtree, then it leads to the tremendous usage of resources and decreases the overall performance.

The solution to the above problem is to save the computed results in an array. First, we calculate F(16) and F(17) and save their values in an array. The F(18) is calculated by summing the values of F(17) and F(16), which are already saved in an array. The computed value of F(18) is saved in an array. The value of F(19) is calculated using the sum of F(18), and F(17), and their values are already saved in an array. The computed value of F(19) is stored in an array. The value of F(20) can be calculated by adding the values of F(19) and F(18), and the values of both F(19) and F(18) are stored in an array. The final computed value of F(20) is stored in an array.

How does the dynamic programming approach work?

The following are the steps that the dynamic programming follows:

- It breaks down the complex problem into simpler subproblems.
- It finds the optimal solution to these sub-problems.
- It stores the results of subproblems (memoization). The process of storing the results of subproblems is known as memorization.
- It reuses them so that same sub-problem is calculated more than once.
- Finally, calculate the result of the complex problem.

The above five steps are the basic steps for dynamic programming. The dynamic programming is applicable that are having properties such as:

Those problems that are having overlapping subproblems and optimal substructures. Here, optimal substructure means that the solution of optimization problems can be obtained by simply combining the optimal solution of all the subproblems.

In the case of dynamic programming, the space complexity would be increased as we are storing the intermediate results, but the time complexity would be decreased.

Two difficulties may arise in any application of dynamic programming:

1. It may not always be possible to combine the solutions of smaller problems to form the solution of a larger one.
2. The number of small problems to solve may be un-acceptably large.

There is no characterized precisely which problems can be effectively solved with dynamic programming; there are many hard problems for which it does not seen to be applicable, as well as many easy problems for which it is less efficient than standard algorithms.

Divide and Conquer Method	**Dynamic Programming**
1. It deals (involves) three steps at each level of recursion: • **Divide** the problem into a number of subproblems. • **Conquer** the subproblems by solving them recursively. • **Combine** the solution to the subproblems into the solution for original subproblems.	1. It involves the sequence of four steps: • Characterize the structure of optimal solutions. • Recursively defines the values of optimal solutions. • Compute the value of optimal solutions in a Bottom-up minimum. • Construct an Optimal Solution from computed information.
2. It is Recursive.	**2.** It is non Recursive.
3. It does more work on subproblems and hence has more time consumption.	**3.** It solves subproblems only once and then stores in the table.
4. It is a top-down approach.	**4.** It is a Bottom-up approach.
5. In this subproblems are independent of each other.	**5.** In this subproblems are interdependent.
6. For example: Merge Sort & Binary Search etc.	**6. For example:** Matrix Multiplication.

Multi stage graph

The goal of multistage graph problem is to find minimum cost path from source to destination vertex. The input to the algorithm is a k-stage graph, n vertices are indexed in increasing order of stages.

What is Multi Stage Graph?

A multistage graph G = (V, E) is a directed graph in which the vertices are partitioned into k > 2 disjoint sets Vi, 1 < i < k. In addition, if <u, v> is an edge in E, then u ∈ Vi and v ∈ Vi+1 for some i, 1 < i < k.

- Let the vertex **‘s’** is the source, and **‘t’** the sink (The sink vertex, often denoted as **'t'**, is the endpoint of the flow network where the flow is directed to or consumed). Let c (i, j) be the cost of edge <i, j>.
- The cost of a path from ‘s’ to ‘t’ is the sum of the costs of the edges on the path. The multistage graph problem is to find a minimum cost path from ‘s’ to ‘t’.
- Each set Vi defines a stage in the graph. Because of the constraints on E, every path from ‘s’ to ‘t’ starts in stage 1, goes to stage 2, then to stage 3, then to stage 4, and so on, and eventually terminates in stage k.

A dynamic programming formulation for a k-stage graph problem is obtained by first noticing that every s to t path is the result of a sequence of k – 2 decisions. The ith decision involves determining which vertex in vi+1, 1 < i < k - 2, is to be on the path.

Let c (i, j) be the cost of the path from source to destination. Then using the forward approach, we obtain:

$$\mathbf{Cost\ (i,j) = min\ \{C\ (j,l) + Cost\ (i+1,l)\ \}\ ,\quad l \in Vi + 1\ ,\quad (j,l) \in E}$$

The multistage graph problem can be solved in two ways using dynamic programming :

1. Forward approach
2. Backward approach

Forward approach:

In the forward approach, we assume that there are k stages in the graph. We start from the last stage and find out the cost of each and every node to the first stage. We then find out the minimum cost path from the source to destination (i.e from stage 1 to stage k).

Procedure:

1. Maintain a cost array cost[n] which stores the distance from any vertex to the destination.
2. If a vertex has more than one path, then the path with the minimum distance is chosen and the intermediate vertex is stored in the distance array d[n]. This will give us a minimum cost path from each and every vertex.
3. Finally, the cost from 1st vertex cost(1,k) gives the minimum cost of the shortest path from source to destination.

4. For finding the path, start from vertex-1 then the distance array D(1) will give the minimum cost neighbour vertex which in turn gives the next nearest vertex and proceed in this way till we reach the destination. For a ‘k’ stage graph, there will be ‘k’ vertex in the path.
5. For forward approach,

   Cost (i,j) = min {C (j,l) + Cost (i+1,l) } , l∈Vi + 1 , (j,l)∈E

Example-1:

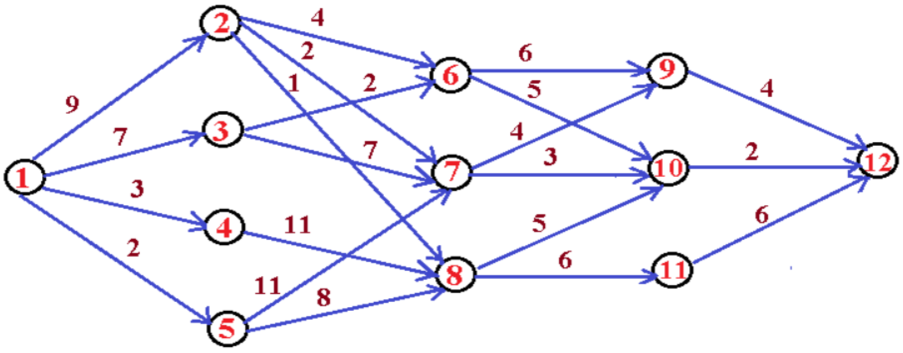


Fig-1

This problem is solved by using tabular method, So we draw a table contain all vertices(v), cost© and destination(d).

V	1	2	3	4	5	6	7	8	9	10	11	12
Cost												
D												

We use the following equation to find the minimum cost path from s to t:

Cost (i,j) = min {C (j,l) + Cost (i+1,l) }

Here our main objective is to select those paths which have minimum cost. So we can say that it is an optimization problem. It ca be solved by the principal of optimal which says the sequence of decisions. That means in every stage we have to take decisions.

In This problem we will start from 12 so its distance is 12 and cost is 0.

Now calculate for V(12) and stage 5.

Here **cost (5,12)=0** i.e cost(stage Number, Vertex)

Now update distance and cost in table for stage 5

V	1	2	3	4	5	6	7	8	9	10	11	12
Cost												0
D												12

Now calculate for V(9,10,11) and stage 4

cost(4,9) = min {c (9, 12) + cost (5, 12)} = min {4 + 0} =4

cost(4,10) = min {c (10, 12) + cost (5, 12)} =2

cost(4, 11) = min {c (11, 12) + cost (5, 12)} =6

Here the distance is 12

Now update new cost and distance in table for stage 4

V	1	2	3	4	5	6	7	8	9	10	11	12
Cost									4	2	6	0
D									12	12	12	12

Now calculate for V(6,7,8) and stage 3

For calculate for V(6) we must find out there connecting link in previous stage i.e 4 which is 9 and **10.**

cost(3,6) = min {c (6, 9) + cost (4, 9), c (6, 10) + cost (4,10)}

= min {6 + cost (4, 9), 5 + cost (4,10)}

= min {6 + 4, 5 + 2} ={10,7}=**7**

Now we find minimum from {10,7} which is 7 and the vertex give the minimum cost is 10

cost(3,7) = min {c (7, 9) + cost (4, 9) , c (7, 10) + cost (4,10)}

= min {4 + cost (4, 9), 3 + cost (4,10)}

= min {4 + 4, 3 + 2} = min {8, 5} =**5**

Now we find minimum from { 8,5 } which is 5 and the vertex give the minimum cost is 10

cost(3,8) = min {c (8, 10) + cost (4, 10), c (8, 11) + cost (4,11)}

= min {5 + cost (4, 10), 6 + cost (4 +11)}

= min {5 + 2, 6 + 6} = min {7, 12} =**7**

Now we find minimum from { 7,11 } which is 5 and the vertex give the minimum cost is 10

Now update new cost and distance in table for stage 3

V	1	2	3	4	5	6	7	8	9	10	11	12
Cost						7	5	7	4	2	6	0
D						10	10	10	12	12	12	12

Now calculate for V(2,3,4, 5) and stage 2

For calculate for V(2) we must find out there connecting link in previous stage i.e 3 which is 6, 7 and **8**

cost (2, 2) = min{c (2, 6) + cost (3, 6), c (2, 7) + cost (3, 7), c (2, 8) + cost (3,8)}

= min {4 + cost (3, 6), 2 + cost (3, 7), 1 + cost (3,8)}

= min {4 + 7, 2 + 5, 1 + 7} = min {11, 7, 8} **=7**

Now we find minimum from { 11,7,8 } which is 7 and the vertex give the minimum cost is 7

cost (2, 3) = min {c (3, 6) + cost (3, 6), c (3, 7) + cost (3,7)}

= min {2 + cost (3, 6), 7 + cost (3,7)}

= min {2 + cost (3, 6), 7 + cost (3,7)} = min {2 + 7, 7 + 5} = min {9, 12} =**9**

Now we find minimum from { 9,12 } which is 9 and the vertex give the minimum cost is 6

cost (2, 4) = min {c (4, 8) + cost (3, 8)}
= min {11 + 7} =**18**

Now we find minimum 18 and the vertex give the minimum cost is 8

cost (2, 5) = min {c (5, 7) + cost (3, 7), c (5, 8) + cost (3,8)}

= min {11 + 5, 8 + 7} = min {16, 15} =**15**

Now we find minimum from { 16,15 } which is 15 and the vertex give the minimum cost is 8

Now update new cost and distance in table for stage 2

V	1	2	3	4	5	6	7	8	9	10	11	12
Cost		7	9	18	15	7	5	7	4	2	6	0
D		7	6	8	8	10	10	10	12	12	12	12

Now calculate for V(1) and stage 1

For calculate for V(1) we must find out there connecting link in previous stage i.e 2 which is 2,3,4 and 5

cost (1, 1) = min {c (1, 2) + cost (2, 2), c (1, 3) + cost (2, 3), c (1, 4) + cost (2,4), c (1, 5) + cost (2,5)}

= min {9 + **cost (2, 2),** 7 + **cost (2, 3),** 3 + **cost (2, 4**), 2 + **cost (2,5**)}

= min {9 + 7, 7 + 9, 3 + 18, 2 +15} = min {16, 16, 21, 17} =16

Now we find minimum from {16, 16, 21, 17} **which is 16 and the vertex give the minimum cost is 2,3 because here we get 16 twice so we consider both vertices.**

So Now update the new cost and distance in table for stage-1

V	1	2	3	4	5	6	7	8	9	10	11	12
Cost	16	7	9	18	15	7	5	7	4	2	6	0
D	2/3	7	6	8	8	10	10	10	12	12	12	12

Now we apply dynamic programming, we know dynamic programming is a sequence of decision on the basis of available data

Now here data is available in the above table

For solving let we start **vertex 1 to onward**. Here decision is taken in forward direction

First we find d(stage, vertex) from taking decision such as

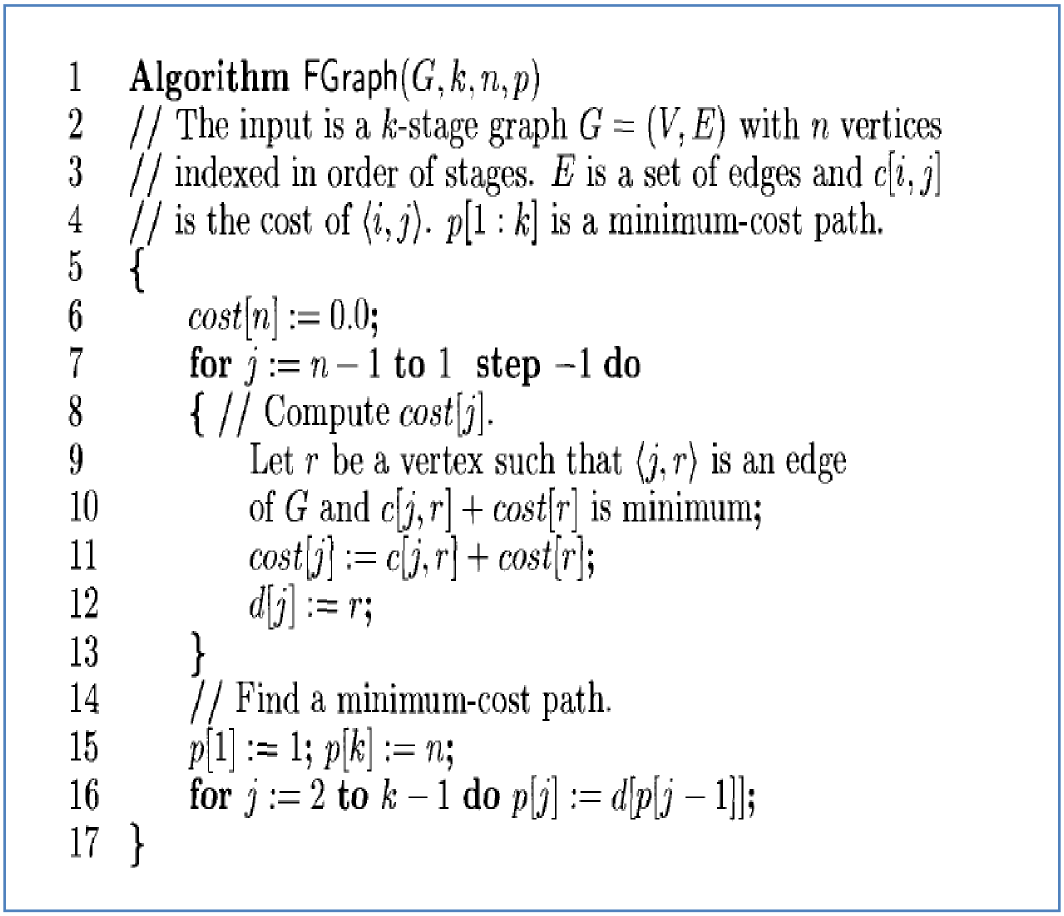
Here for 1 we have two value of d (2/3) so we will take 2 first

So

The path is : 1->2->7->10->12

Now we take decision by taking 3 for vertex 1.

The path is : 1->3->6->10->12



```
Algorithm FGraph($G, k, n, p$)
// The input is a $k$-stage graph $G = (V, E)$ with $n$ vertices
// indexed in order of stages. $E$ is a set of edges and $c[i, j]$
// is the cost of $\langle i, j \rangle$. $p[1 : k]$ is a minimum-cost path.
{
    $cost[n] := 0.0;$
    for $j := n - 1$ to 1 step $-1$ do
    { // Compute $cost[j]$.
        Let $r$ be a vertex such that $\langle j, r \rangle$ is an edge
        of $G$ and $c[j, r] + cost[r]$ is minimum;
        $cost[j] := c[j, r] + cost[r];$
        $d[j] := r;$
    }
    // Find a minimum-cost path.
    $p[1] := 1; p[k] := n;$
    for $j := 2$ to $k - 1$ do $p[j] := d[p[j - 1]];$
}
```

Algorithm: Multistage graph pseudo code corresponding to the forward approach

Explanation:

The pseudo code you've provided describes an algorithm to find a minimum-cost path in a directed graph where the graph is partitioned into stages. It seems to solve a path finding problem on a **k-stage graph** where:

- **n** is the number of vertices.
- **k** is the number of stages in the graph.
- **E** is a set of edges, where each edge (i, j) has an associated cost c[i,j].
- The algorithm computes the minimum-cost path from vertex 1 (stage 1) to vertex n (stage k) using dynamic programming.

Let me explain the algorithm step by step:

Step-by-Step Breakdown:

1. **Input Details**:
   - **G** represents a k-stage graph with n vertices.
   - **c[i, j]** represents the cost of an edge from vertex i to vertex j.
   - **p[l:k]** represents the path from vertex l to vertex k (from stage 1 to stage k).
2. **Initialization** (Lines 6-7):
   - **cost[n] := 0.0**: Set the cost of reaching the final vertex n to 0, as no cost is required to stay at the destination.
   - The loop in line 7 processes vertices in reverse order, starting from vertex n-1 to vertex 1. This ensures that the algorithm computes the minimum cost for each vertex while considering future vertices.
3. **Dynamic Programming Loop** (Lines 8-12):
   - For each vertex j, the algorithm looks for an edge (j, r) and computes the cost to reach vertex r. The cost is calculated as the cost of the edge (j, r) plus the cost to reach vertex r (cost[r]).
   - The algorithm selects the vertex r that minimizes this total cost, updating:
     - **cost[j]**: The minimum cost to reach the destination vertex from vertex j.
     - **d[j]**: The vertex r that gives the minimum cost for the transition from j.
4. **Reconstructing the Minimum-Cost Path** (Lines 15-16):
   - **p[l] := 1; p[k] := n**: Set the start of the path at vertex 1 and the end of the path at vertex n.
   - The loop in line 16 reconstructs the path from p[l] to p[k] by backtracking through the **d** array. The path is stored in p, where p[j] is the next vertex in the optimal path starting from vertex p[j-1].

Result:

- The algorithm computes the minimum-cost path and stores the path in the array p (from vertex 1 to vertex n).

Pseudo code Clarifications:

- **cost[]**: An array that stores the minimum cost to reach each vertex.
- **d[]**: An array that stores the next vertex for the optimal path.
- **p[]**: The array that holds the path from vertex 1 to vertex n.

Example :

Consider a directed graph with 5 vertices (n = 5) and 3 stages (k = 3). The edges and their costs are as follows:

- **Vertices (Stages)**: 1 → 2 → 3 → 4 → 5 (with the stages 1, 2, and 3)
- **Edges and Costs**:
  - (1 → 2) with cost 3
  - (1 → 3) with cost 2
  - (2 → 4) with cost 1
  - (2 → 5) with cost 5
  - (3 → 4) with cost 4
  - (3 → 5) with cost 1
  - (4 → 5) with cost 2

The goal is to find the **minimum-cost path** from **vertex 1** to **vertex 5** over **3 stages**.

Step-by-Step Walkthrough:

1. **Initialization**:
   - cost[n] := 0.0; → cost[5] = 0 because no cost is required to stay at vertex 5.
   - d[] array will store the vertex that gives the minimum cost for each vertex.
   - Start by setting the cost[] for all vertices to infinity, except for vertex 5 (the destination).

   Initially:

   - cost[5] = 0
   - cost[4] = ∞
   - cost[3] = ∞
   - cost[2] = ∞
   - cost[1] = ∞
   - d[] is empty.
2. **Dynamic Programming Loop**:
   - Start with vertex 4 (since we are iterating in reverse order):

   For vertex 4:

   - There is an edge (4 → 5) with cost 2. So, cost[4] = cost[5] + 2 = 0 + 2 = 2.
   - Update d[4] = 5 because vertex 5 gives the minimum cost for vertex 4.

   For vertex 3:

   - There are two edges:
     - (3 → 4) with cost 4, so cost[4] = cost[4] + 4 = 2 + 4 = 6.

    - (3 → 5) with cost 1, so cost[5] = cost[5] + 1 = 0 + 1 = 1.
  - Choose the minimum: cost[3] = 1 and update d[3] = 5.

For vertex 2:

  - There are two edges:
    - (2 → 4) with cost 1, so cost[4] = cost[4] + 1 = 2 + 1 = 3.
    - (2 → 5) with cost 5, so cost[5] = cost[5] + 5 = 0 + 5 = 5.
  - Choose the minimum: cost[2] = 3 and update d[2] = 4.

For vertex 1:

  - There are two edges:
    - (1 → 2) with cost 3, so cost[2] = cost[2] + 3 = 3 + 3 = 6.
    - (1 → 3) with cost 2, so cost[3] = cost[3] + 2 = 1 + 2 = 3.
  - Choose the minimum: cost[1] = 3 and update d[1] = 3.

After this loop, the **cost[]** array is:

  - cost[1] = 3
  - cost[2] = 3
  - cost[3] = 1
  - cost[4] = 2
  - cost[5] = 0

And the **d[]** array (which stores the next vertex in the path):

  - d[1] = 3
  - d[2] = 4
  - d[3] = 5
  - d[4] = 5

3. **Reconstructing the Minimum-Cost Path**: Now, let's reconstruct the path from vertex 1 to vertex 5 by backtracking using the d[] array.
   - **p[1] = 1** (starting point)
   - **p[3] = 5** (destination point)
   - Now, backtrack: p[2] = d[p[1]] = d[1] = 3

   So, the path is:

   - p[1] = 1
   - p[2] = 3
   - p[3] = 5

   The minimum-cost path is **1 → 3 → 5**.
4. **Cost of the Minimum-Cost Path**:

- From the cost[] array, we know that the minimum cost to go from vertex 1 to vertex 5 is cost[1] = 3.

Final Results:

- **Minimum-Cost Path**: 1 → 3 → 5
- **Minimum Cost**: 3

Time Complexity Analysis

The function FGraph computes the minimum-cost path through a k-stage graph. The input graph GGG is represented by adjacency lists, and we are primarily concerned with how long it takes to run the algorithm and the space it uses.

1. **Outer Loop (Line 7 - Line 13)**:
   - The **for loop** in line 7 iterates through the vertices from n−1n-1n−1 to 1, which means it runs O(n) times (where n is the number of vertices).
   - For each vertex j, the function checks all outgoing edges (neighbors) of j in line 9. In an adjacency list representation of the graph, the time to check the outgoing edges of vertex j is proportional to its **degree** (the number of outgoing edges for vertex j).
   - The total time to process all edges is proportional to the total number of edges in the graph, E. Thus, the time complexity for this part of the algorithm is O(V+E), where:
     - V is the number of vertices (due to the loop over all vertices),
     - E is the total number of edges (since checking all edges requires examining each outgoing edge for all vertices).
2. **Reconstructing the Path (Line 16 - Line 17)**:
   - In line 16, the algorithm iterates through the stages (from stage 2 to stage k−1), so this loop runs O(k) times. This is because k is the number of stages, and the number of stages determines how many vertices are involved in the path reconstruction.
   - Path reconstruction (filling in the path array p[ ]) takes linear time in terms of the number of stages k, which gives a time complexity of O(k).

Total Time Complexity

Summing the time complexities from the two major parts:

- The first part, processing vertices and edges, contributes O(V+E).
- The second part, reconstructing the path, contributes O(k).

Thus, the **total time complexity** of the algorithm is: **O(V+E+k)**

Where:

- V is the number of vertices,
- E is the number of edges,
- k is the number of stages.

In most cases, k (the number of stages) is much smaller than V and E, so the overall complexity is dominated by the number of vertices and edges, i.e., O(V+E).

Space Complexity

For the space complexity, let's break it down based on the storage requirements:

- The adjacency list representation of the graph requires O(E+V) space because each edge and vertex is stored.
- The algorithm uses three arrays:
  - cost[ ] of size O(V),
  - d[ ] of size O(V) (for storing the "parent" or the next vertex in the path),
  - p[ ] of size O(k) (for storing the reconstructed path).

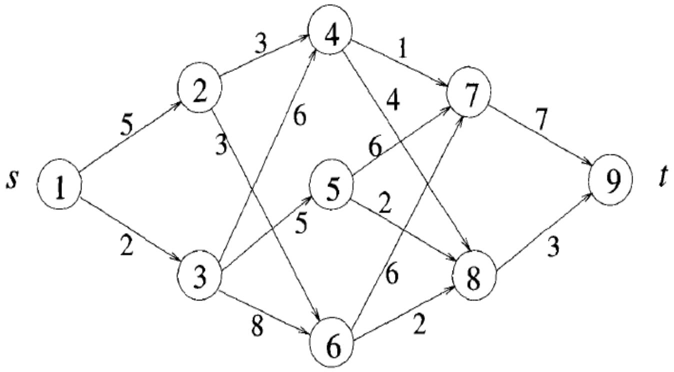
So, the **total space complexity** is: **O(E+V+k)**

Where:

- O(E+V) comes from storing the graph in an adjacency list,
- O(V) comes from the cost[] and d[ ] arrays,
- O(k) comes from the p[ ] array.

Example-2:

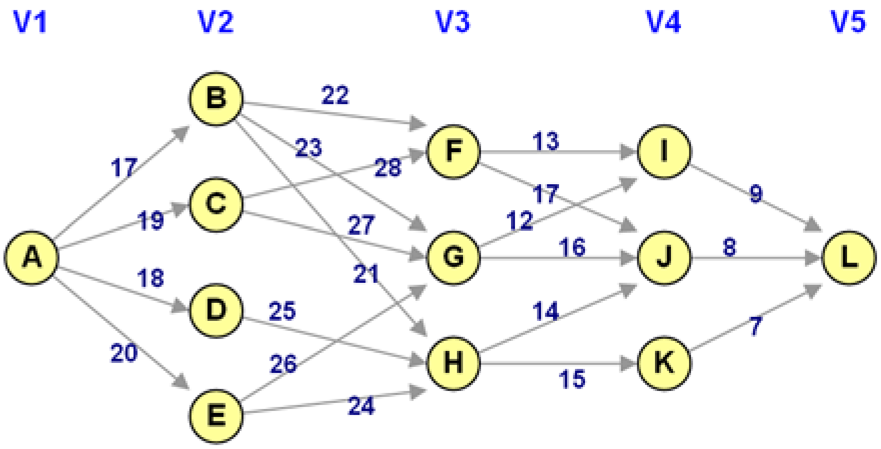
Find a minimum-cost path from s to t in the multi stage graph of below Figure. Do this first using the forward approach and then using the backward approach



Hence, the path having the minimum cost is 1→ 3→ 5→ 8→ 9.

Example-3:

Find the shortest path from node A to node L in the multistage graph below using the dynamic programming (backward) method and forward approach!



The Benefits of Using Multistage Graphs

Multistage graphs boast several advantages over other types of graphs:

- They offer superior efficiency, making them ideal for tackling complex problems.
- Implementing and scaling multistage graphs is relatively easier compared to other graph types.

The Drawbacks of Multistage Graphs

Despite their benefits, multistage graphs have a few disadvantages when compared to other graph representations:

- They may require more space than other graph representations.
- Interpreting them can be challenging as the information is distributed across various stages.

0/1 knapsack problem

Here I am going to explain about **0/1 Knapsack** problem using dynamic programming. We already know about what is knapsack problem? in greedy approach.

The main concept in both the approach is same that is we must select the item and fill it into sack (just like a bag) in such a way that we get more profit.

In greedy approach we can fill the item into the sack partially that means if the weight of item 20 and the space available in sack is 10 than can select 10 weight out of 20 weight and put into the sack that means factorial part is allowed here.

But in dynamic programming when we solving knapsack problem fractional part is not allowed. We can select either complete portion of item that means 1 or select nothing from the item that means 0 , that's why it is known as 0/1 knapsack problem.

Here the item / object are not divisible or breakable such as mobiles, laptop, mouse, keyboard etc.

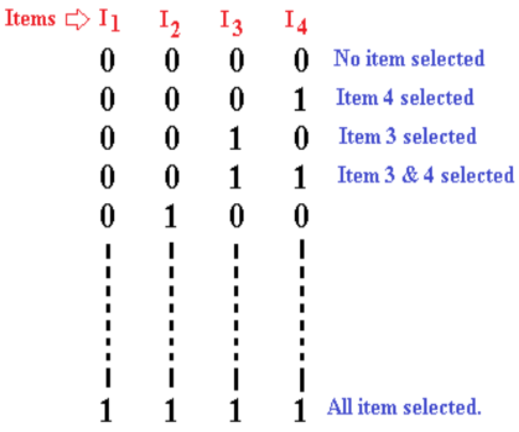
So here our objective is to maximise profit:

$$\max \Sigma P_i \; X_i$$

$$\text{where: } \Sigma \, w_i \; x_i < m$$

Let us understand how we solve 0/1 knapsack problem by using dynamic programming. This method is used to solve optimisation problem. Here we take sequence of decisions for solving any problem. In this method we should try all the possible solutions and pickup the best solution.

Before start 0/1 knapsack let we understand the all possibility to select items if we have 4 items below figure show some possibilities but not all because if we talk about all possibilities than it become 2 power 4.Same as if we have n items than 2 power n possibilities are there. If we have large number of items than finding all possibility are time consuming which is bigo of(2 power n).



Items ⇨	I_1	I_2	I_3	I_4	
	0	0	0	0	No item selected
	0	0	0	1	Item 4 selected
	0	0	1	0	Item 3 selected
	0	0	1	1	Item 3 & 4 selected
	0	1	0	0	
	⋮	⋮	⋮	⋮	
	1	1	1	1	All item selected.

So above method is time consuming if we are finding all the possibilities for the selection of items.

Now we will discuss how we do same thing in easy way so that we save time and effort. Let we understand dynamic programming by the help of an example.

Example:1

suppose we have

P = { 1, 2, 5, 6 }

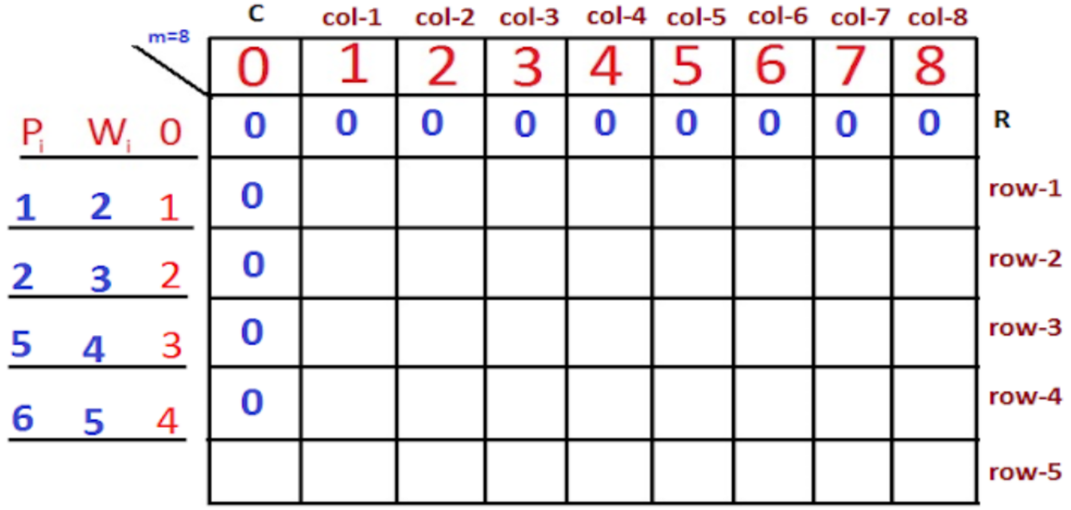
W= { 2, 3, 4, 5 }

and m=8 , n=4

For solving the above method we are using tabulation method.

Create column according to the capacity of sack / bag i.e here 8.

Taking all object at a time is not possible , So here one by one we will consider the object/item like 1, 2, 3, ...,8.



			C	col-1	col-2	col-3	col-4	col-5	col-6	col-7	col-8	
		m=8	0	1	2	3	4	5	6	7	8	
P_i	W_i	0	0	0	0	0	0	0	0	0	0	R
1	2	1	0									row-1
2	3	2	0									row-2
5	4	3	0									row-3
6	5	4	0									row-4
												row-5

Table- 1

Here we create table of 9 columns which is depend on the value of m i.e m=8. We have 4 items so we take 5 rows.

Initially R row and C column according to the above table will be 0.

Note: (According to this table)

If we consider row-2 for item-2 than we also consider item 1 with their (P,W).

If we consider row-3 for item-3 than we consider item-2 and item-1 with their (P,W).

If we consider row-4 for item-4 than we consider item-3,item-2 and item-1 with their (P,W).

If we consider row-5 than we consider item-4,item-3,item-2 and item-1 with their (P,W).

So we can say that if we are in ith row than we consider all the previous rows also.

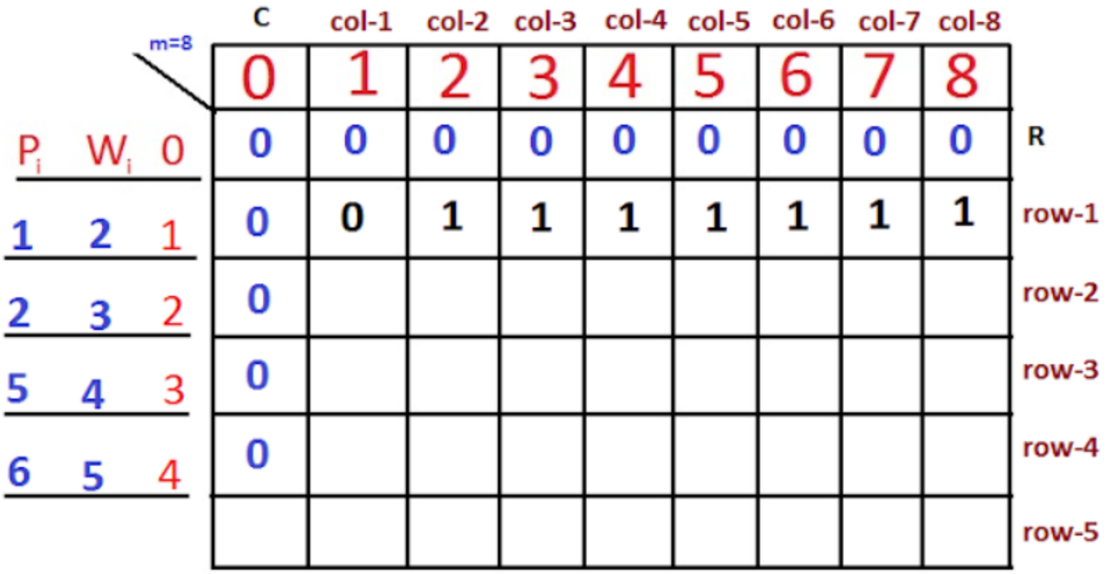
Now Let we start:

If we select item-1 than we see what is the profit i.e 1, So put 1 in col-2 column.

Now understand the logic for putting profit in columns suppose here for item-1 the weight is 2, Now imagine can you put suppose 2 kg of item in 1 kg of bag the answer is no so we put 0(profit) in column 1,

Now check for next can you put suppose 2 kg of item in 2 kg of bag the answer is yes,so we put 1(profit) in column 2, In the same way answer is yes for all column 3,4,5,6,7,8 so we put 1 in all column.

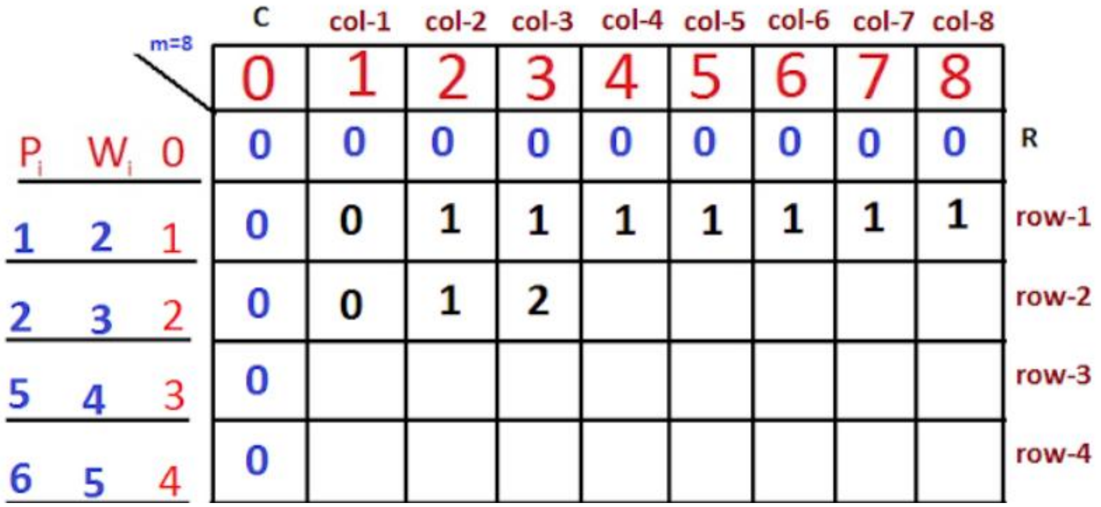
Now the table will be



			C	col-1	col-2	col-3	col-4	col-5	col-6	col-7	col-8	
		m=8	0	1	2	3	4	5	6	7	8	
P_i	W_i	0	0	0	0	0	0	0	0	0	0	R
1	2	1	0	0	1	1	1	1	1	1	1	row-1
2	3	2	0									row-2
5	4	3	0									row-3
6	5	4	0									row-4
												row-5

Table- 2

Second case we select item-2 whose weight W=3 & P=2 , so put their profit 2 into column-3 and all cells left side of column-3 will be remain same because we cannot put suppose 3 kg of item in 1 or 2 kg of bag.



			C	col-1	col-2	col-3	col-4	col-5	col-6	col-7	col-8	
		m=8	0	1	2	3	4	5	6	7	8	
P_i	W_i	0	0	0	0	0	0	0	0	0	0	R
1	2	1	0	0	1	1	1	1	1	1	1	row-1
2	3	2	0	0	1	2						row-2
5	4	3	0									row-3
6	5	4	0									row-4

Now we talk about right side of column-3. what we put here? So I already told you when we take row-2 than we also consider row-1 with (P,W). Now here we have two choice whether we can select 2 or 3 of weight.

Let we calculate profit

For row-2 & col-3

step-1 : 2 is profit of weight=3 of item-2

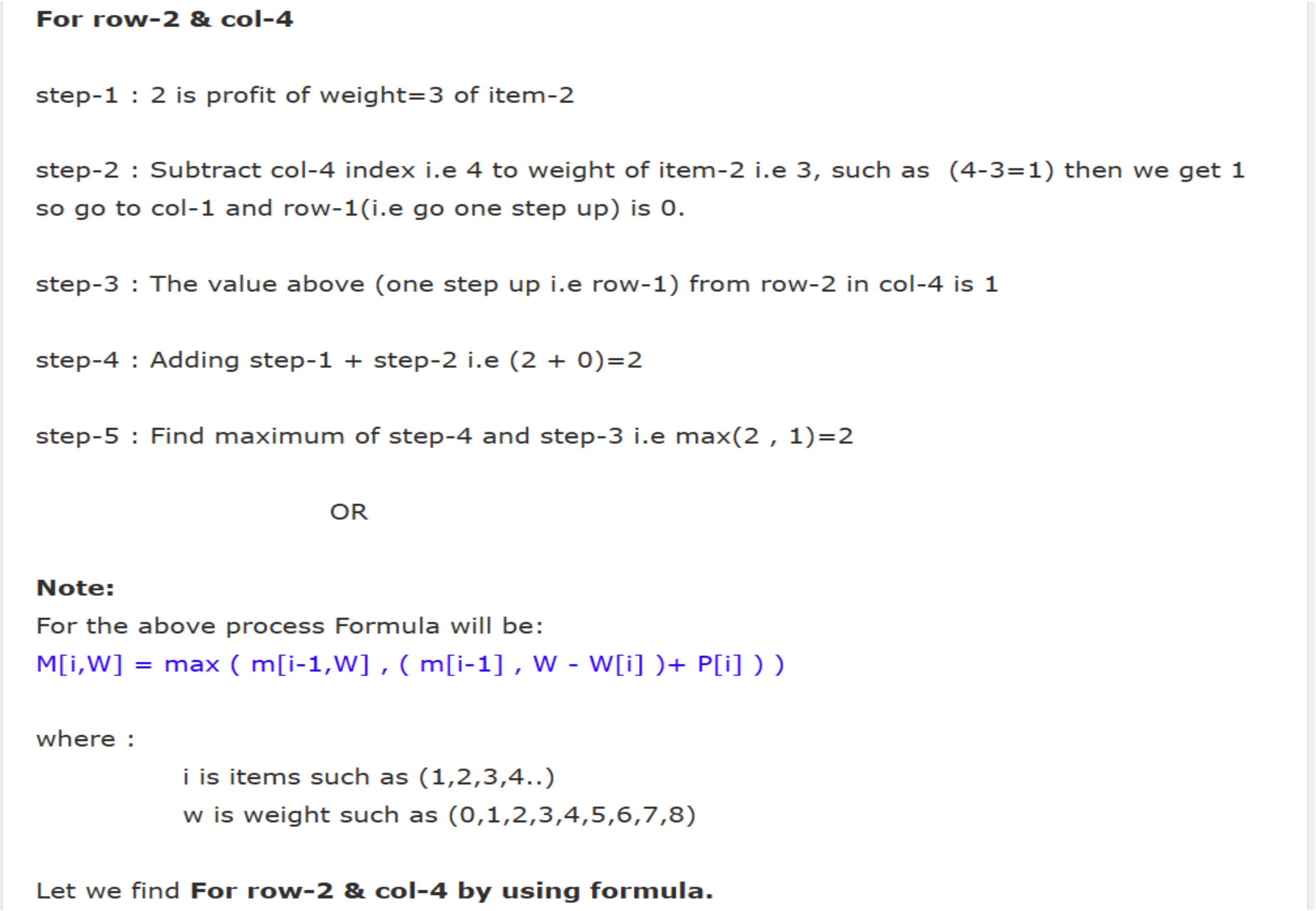
step-2 : Subtract col-3 index i.e 3 to weight of item-2 i.e 3, such as (3-3=0) then we get 0 so go to col-0 and row-1(i.e go one step up) is 0.

step-3 : The value above (one step up i.e row-1) from row-2 in col-3 is 1

step-4 : adding step-1 + step-2 i.e (2 + 0)=2

step-5 : Find maximum of step-4 and step-3 i.e max(2 , 1)=2

so write 2 col-3 row-2 as we show in above Now next



For row-2 & col-4

step-1 : 2 is profit of weight=3 of item-2

step-2 : Subtract col-4 index i.e 4 to weight of item-2 i.e 3, such as (4-3=1) then we get 1 so go to col-1 and row-1(i.e go one step up) is 0.

step-3 : The value above (one step up i.e row-1) from row-2 in col-4 is 1

step-4 : Adding step-1 + step-2 i.e (2 + 0)=2

step-5 : Find maximum of step-4 and step-3 i.e max(2 , 1)=2

OR

Note:

For the above process Formula will be:

M[i,W] = max (m[i-1,W] , (m[i-1] , W - W[i])+ P[i]))

where :

i is items such as (1,2,3,4..)

w is weight such as (0,1,2,3,4,5,6,7,8)

Let we find **For row-2 & col-4 by using formula.**

here i=2 w=4

M(2,4) = max (m(1,4), (m(1, 4-3)+2))

M(2,4)= max (1,(m(1,1)+2))

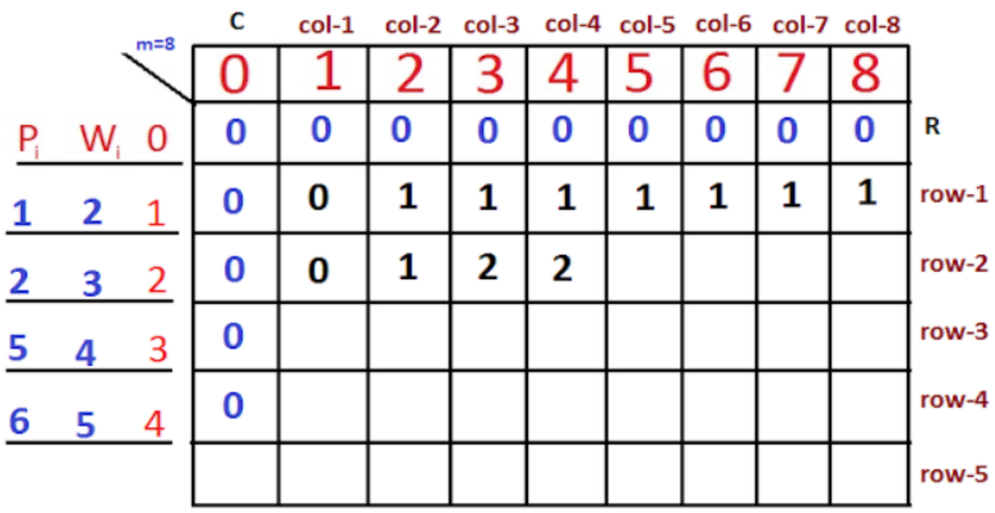
M(2,4)= max (1,(0+2))

M(2,4)= max (1,2)

M(2,4)= 2

so for item-2 col-4 value is 2

so write 2 col-4 row-2



		m=8	C	col-1	col-2	col-3	col-4	col-5	col-6	col-7	col-8	
			0	1	2	3	4	5	6	7	8	
P_i	W_i	0	0	0	0	0	0	0	0	0	0	R
1	2	1	0	0	1	1	1	1	1	1	1	row-1
2	3	2	0	0	1	2	2					row-2
5	4	3	0									row-3
6	5	4	0									row-4
												row-5

Table- 3

For row-2 & col-5

step-1 : 2 is profit of weight=3 of item-2

step-2 : Subtract col-5 index i.e 5 to weight of item-2 i.e 3, such as (5-3=2) then we get 2 so go to col-2 and row-1(i.e go one step up) is 1.

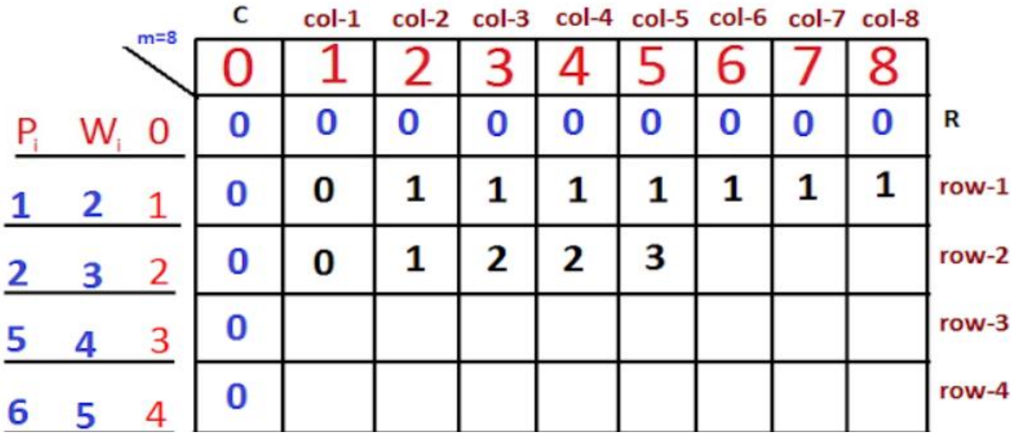
step-3 : The value above (one step up i.e row-1) from row-2 in col-5 is 1

step-4 : Adding step-1 + step-2 i.e (2 + 1)=3

step-5 : Find maximum of step-4 and step-3 i.e max(3 , 1)=3

M(2,5)= 3

so write 3 col-5 row-2



			C	col-1	col-2	col-3	col-4	col-5	col-6	col-7	col-8	
		m=8	0	1	2	3	4	5	6	7	8	
P_i	W_i	0	0	0	0	0	0	0	0	0	0	R
1	2	1	0	0	1	1	1	1	1	1	1	row-1
2	3	2	0	0	1	2	2	3				row-2
5	4	3	0									row-3
6	5	4	0									row-4

For row-2 & col-6

step-1 : 2 is profit of weight=3 of item-2

step-2 : Subtract col-6 index i.e 6 to weight of item-2 i.e 3, such as (6-3=3) then we get 2 so go to col-2 and row-1(i.e go one step up) is 1.

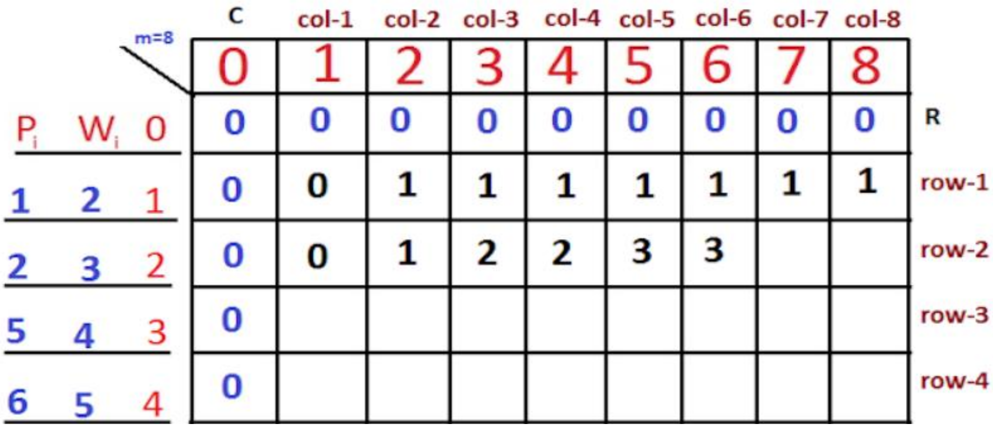
step-3 : The value above (one step up i.e row-1) from row-2 in col-6 is 1

step-4 : Adding step-1 + step-2 i.e (2 + 1)=3

step-5 : Find maximum of step-4 and step-3 i.e max(3 , 1)=3

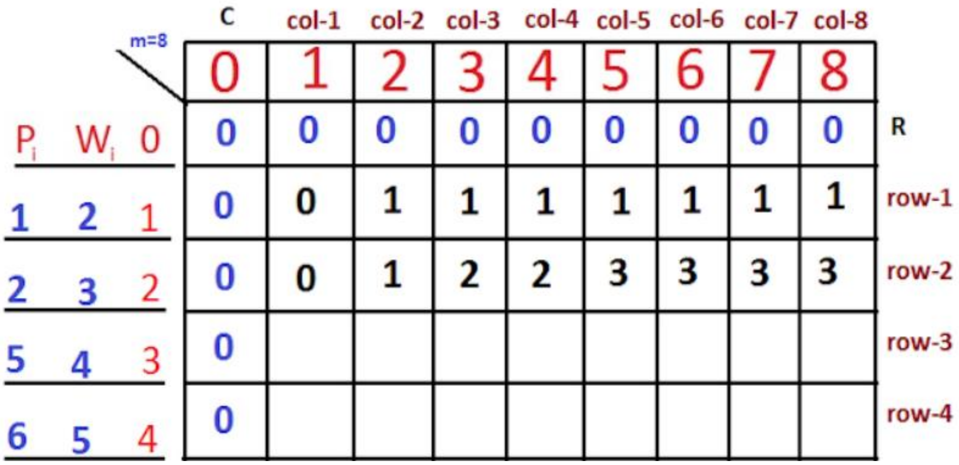
M(2,6)= 3

so write 3 col-6 row-2



			C	col-1	col-2	col-3	col-4	col-5	col-6	col-7	col-8	
		m=8	0	1	2	3	4	5	6	7	8	
P_i	W_i	0	0	0	0	0	0	0	0	0	0	R
1	2	1	0	0	1	1	1	1	1	1	1	row-1
2	3	2	0	0	1	2	2	3	3			row-2
5	4	3	0									row-3
6	5	4	0									row-4

Same way for col 7 & col 8

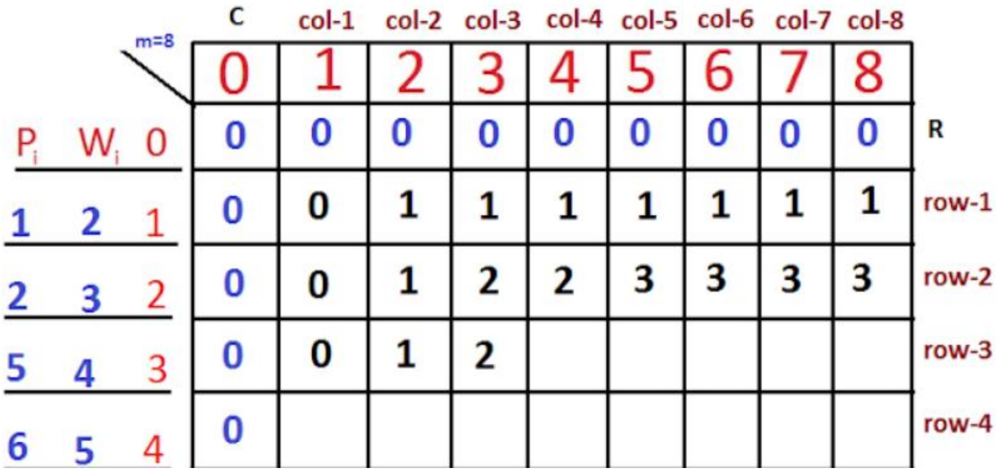


			C	col-1	col-2	col-3	col-4	col-5	col-6	col-7	col-8	
		m=8	0	1	2	3	4	5	6	7	8	
P_i	W_i	0	0	0	0	0	0	0	0	0	0	R
1	2	1	0	0	1	1	1	1	1	1	1	row-1
2	3	2	0	0	1	2	2	3	3	3	3	row-2
5	4	3	0									row-3
6	5	4	0									row-4

Now Calculate for row-3

Here weight of item is 4 and profit is 5

Write as it is value for col-1,col-2,col-3 of row-3



			C	col-1	col-2	col-3	col-4	col-5	col-6	col-7	col-8	
		m=8	0	1	2	3	4	5	6	7	8	
P_i	W_i	0	0	0	0	0	0	0	0	0	0	R
1	2	1	0	0	1	1	1	1	1	1	1	row-1
2	3	2	0	0	1	2	2	3	3	3	3	row-2
5	4	3	0	0	1	2						row-3
6	5	4	0									row-4

Now calculate for row-3 & col-4

step-1 : 5 is profit of weight=4 of item-3

step-2 : Subtract col-4 index i.e 4 to weight of item-3 i.e 4, such as (4-4=0) then we get 0 so go to col-0 and row-2(i.e go one step up) is 0.

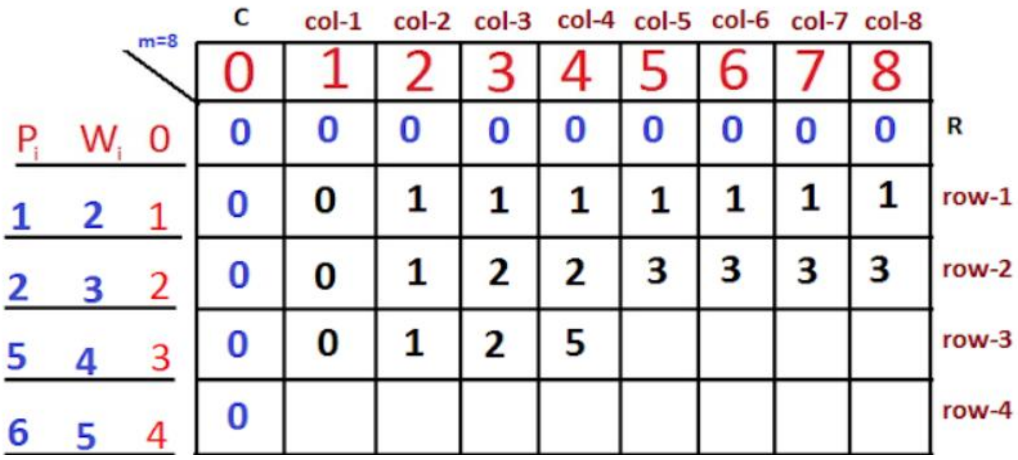
step-3 : The value above (one step up i.e row-2) from row-3 in col-4 is 2

step-4 : Adding step-1 + step-2 i.e (5 + 0)=5

step-5 : Find maximum of step-4 and step-3 i.e max(5 , 2)=5

M(3,4)= 5

so write 5 in col-4 row-3



			C	col-1	col-2	col-3	col-4	col-5	col-6	col-7	col-8	
		m=8	0	1	2	3	4	5	6	7	8	
P_i	W_i	0	0	0	0	0	0	0	0	0	0	R
1	2	1	0	0	1	1	1	1	1	1	1	row-1
2	3	2	0	0	1	2	2	3	3	3	3	row-2
5	4	3	0	0	1	2	5					row-3
6	5	4	0									row-4

Now calculate for row-3 & col-5

step-1 : 5 is profit of weight=4 of item-3

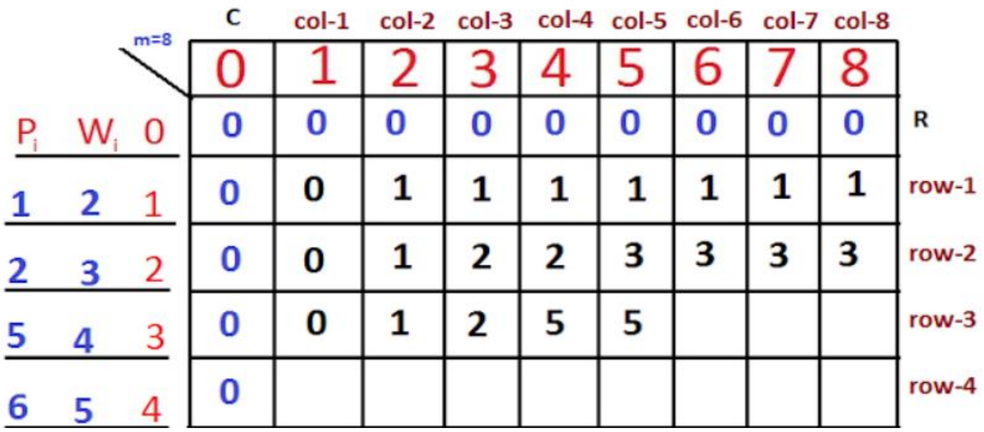
step-2 : Subtract col-5 index i.e 5 to weight of item-3 i.e 4, such as (5-4=1) then we get 0 so go to col-0 and row-2(i.e go one step up) is 0.

step-3 : The value above (one step up i.e row-2) from row-3 in col-4 is 2

step-4 : Adding step-1 + step-2 i.e (5 + 0)=5

step-5 : Find maximum of step-4 and step-3 i.e max(5 , 2)=5

M(3,5)= 5



			C	col-1	col-2	col-3	col-4	col-5	col-6	col-7	col-8	
		m=8	0	1	2	3	4	5	6	7	8	
P_i	W_i	0	0	0	0	0	0	0	0	0	0	R
1	2	1	0	0	1	1	1	1	1	1	1	row-1
2	3	2	0	0	1	2	2	3	3	3	3	row-2
5	4	3	0	0	1	2	5	5				row-3
6	5	4	0									row-4

Now calculate for row-3 & col-6

step-1 : 5 is profit of weight=4 of item-3

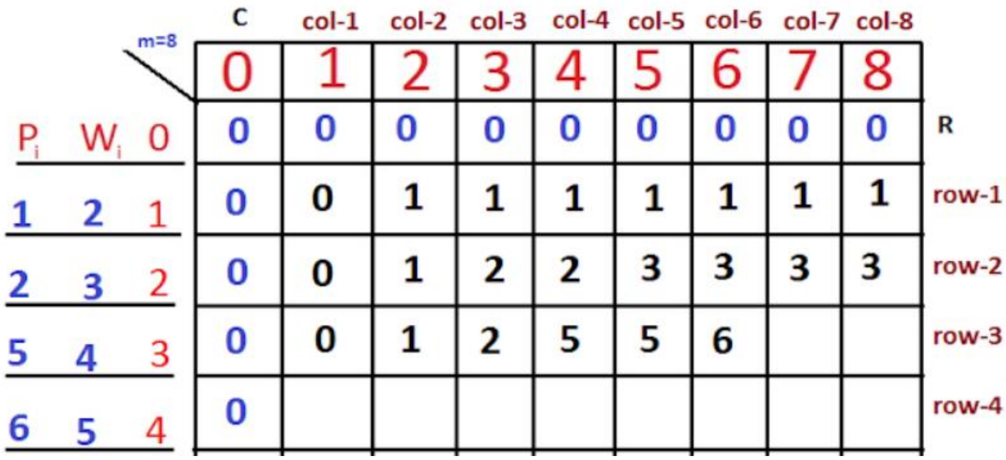
step-2 : Subtract col-6 index i.e 6 to weight of item-3 i.e 4, such as (6-4=2) then we get 2 so go to col-2 and row-2(i.e go one step up) is 1.

step-3 : The value above (one step up i.e row-2) from row-3 in col-6 is 3

step-4 : Adding step-1 + step-2 i.e (5 + 1)=6

step-5 : Find maximum of step-4 and step-3 i.e max(6 , 3)=6

M(3,6)= 6



			C	col-1	col-2	col-3	col-4	col-5	col-6	col-7	col-8	
		m=8	0	1	2	3	4	5	6	7	8	
P_i	W_i	0	0	0	0	0	0	0	0	0	0	R
1	2	1	0	0	1	1	1	1	1	1	1	row-1
2	3	2	0	0	1	2	2	3	3	3	3	row-2
5	4	3	0	0	1	2	5	5	6			row-3
6	5	4	0									row-4

Now calculate for row-3 & col-7

step-1 : 5 is profit of weight=4 of item-3

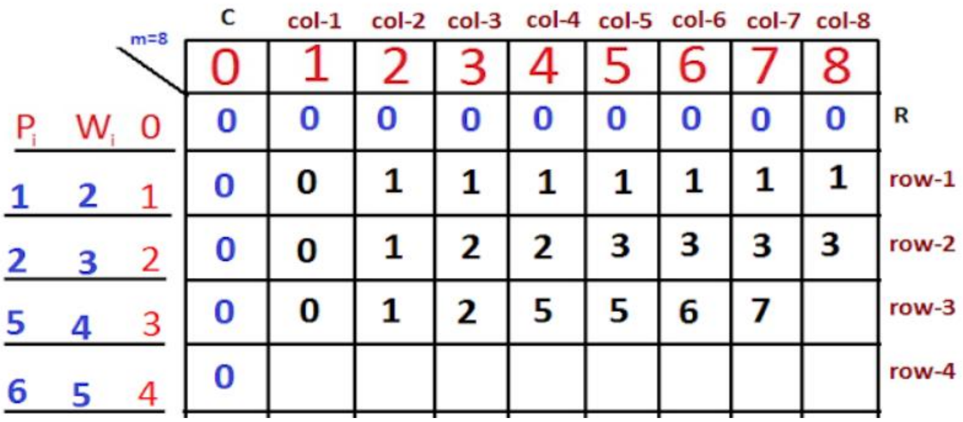
step-2 : Subtract col-7 index i.e 7 to weight of item-3 i.e 4, such as (7-4=3) then we get 3 so go to col 3 and row-2(i.e go one step up) is 2.

step-3 : The value above (one step up i.e row-2) from row-3 in col-7 is 3

step-4 : Adding step-1 + step-2 i.e (5 + 2)=7

step-5 : Find maximum of step-4 and step-3 i.e max(7 , 3)=7

M(3,7)= 7



			C	col-1	col-2	col-3	col-4	col-5	col-6	col-7	col-8	
		m=8	0	1	2	3	4	5	6	7	8	
Pi	Wi	0	0	0	0	0	0	0	0	0	0	R
1	2	1	0	0	1	1	1	1	1	1	1	row-1
2	3	2	0	0	1	2	2	3	3	3	3	row-2
5	4	3	0	0	1	2	5	5	6	7		row-3
6	5	4	0									row-4

Now calculate for row-3 & col-8

step-1 : 5 is profit of weight=4 of item-3

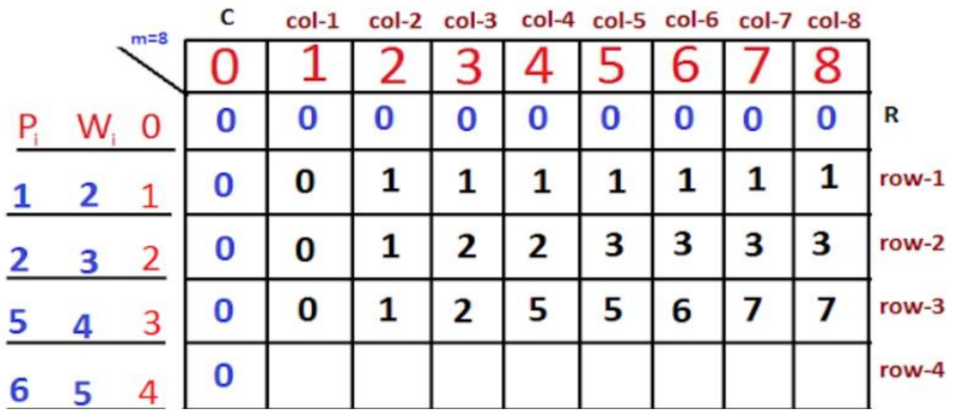
step-2 : Subtract col-8 index i.e 8 to weight of item-3 (i.e 4), such as (8-4=4) then we get 4 so go to col 4 and row-2(i.e go one step up) is 2.

step-3 : The value above (one step up i.e row-2) from row-3 in col-8 is 3

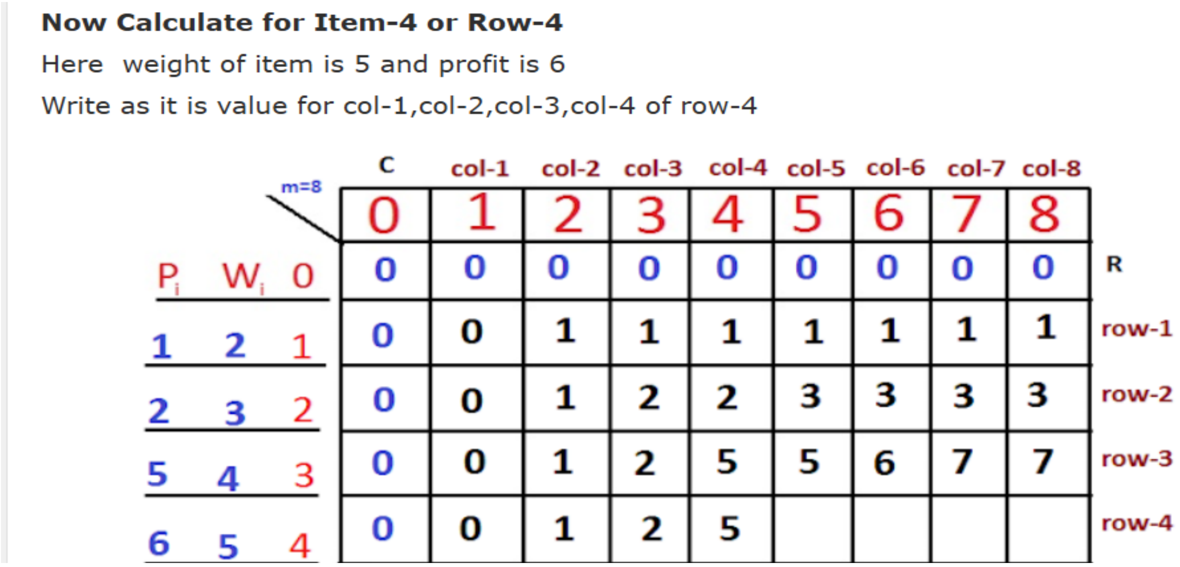
step-4 : Adding step-1 + step-2 i.e (5 + 2)=7

step-5 : Find maximum of step-4 and step-3 i.e max(7 , 3)=7

M(3,8)= 7



			C	col-1	col-2	col-3	col-4	col-5	col-6	col-7	col-8	
		m=8	0	1	2	3	4	5	6	7	8	
Pi	Wi	0	0	0	0	0	0	0	0	0	0	R
1	2	1	0	0	1	1	1	1	1	1	1	row-1
2	3	2	0	0	1	2	2	3	3	3	3	row-2
5	4	3	0	0	1	2	5	5	6	7	7	row-3
6	5	4	0									row-4

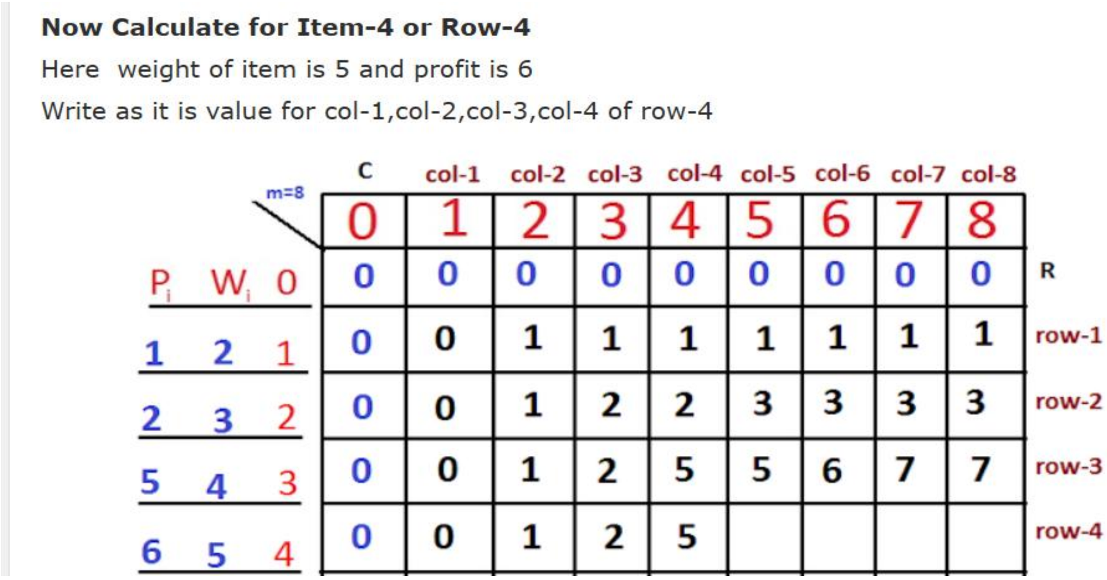


Now Calculate for Item-4 or Row-4

Here weight of item is 5 and profit is 6

Write as it is value for col-1,col-2,col-3,col-4 of row-4

			C	col-1	col-2	col-3	col-4	col-5	col-6	col-7	col-8	
		m=8	0	1	2	3	4	5	6	7	8	
P_i	W_i	0	0	0	0	0	0	0	0	0	0	R
1	2	1	0	0	1	1	1	1	1	1	1	row-1
2	3	2	0	0	1	2	2	3	3	3	3	row-2
5	4	3	0	0	1	2	5	5	6	7	7	row-3
6	5	4	0	0	1	2	5					row-4

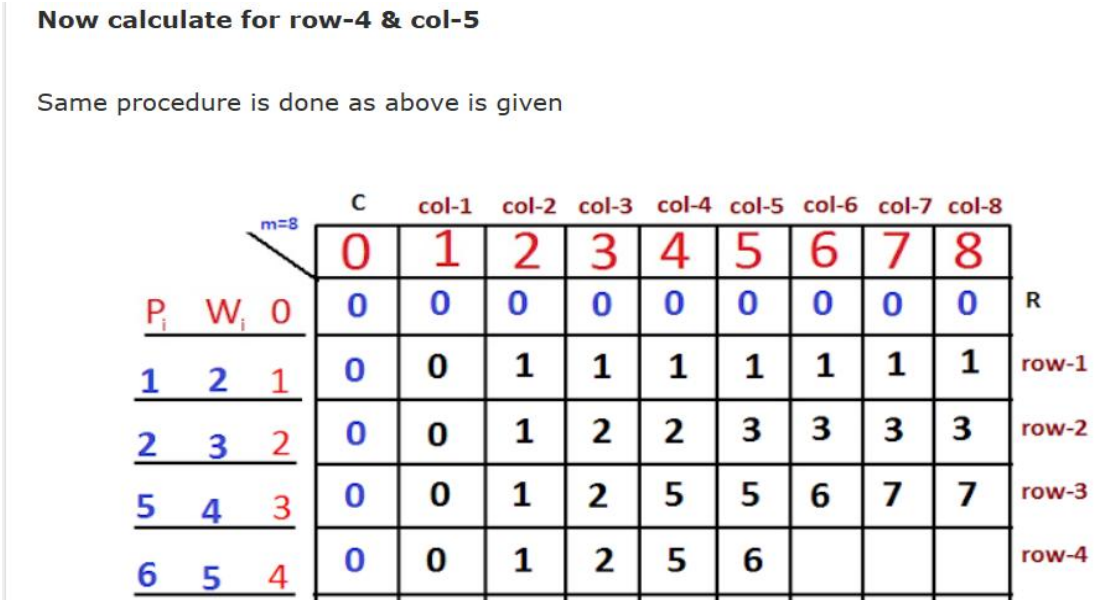


Now Calculate for Item-4 or Row-4

Here weight of item is 5 and profit is 6

Write as it is value for col-1,col-2,col-3,col-4 of row-4

			C	col-1	col-2	col-3	col-4	col-5	col-6	col-7	col-8	
		m=8	0	1	2	3	4	5	6	7	8	
P_i	W_i	0	0	0	0	0	0	0	0	0	0	R
1	2	1	0	0	1	1	1	1	1	1	1	row-1
2	3	2	0	0	1	2	2	3	3	3	3	row-2
5	4	3	0	0	1	2	5	5	6	7	7	row-3
6	5	4	0	0	1	2	5					row-4



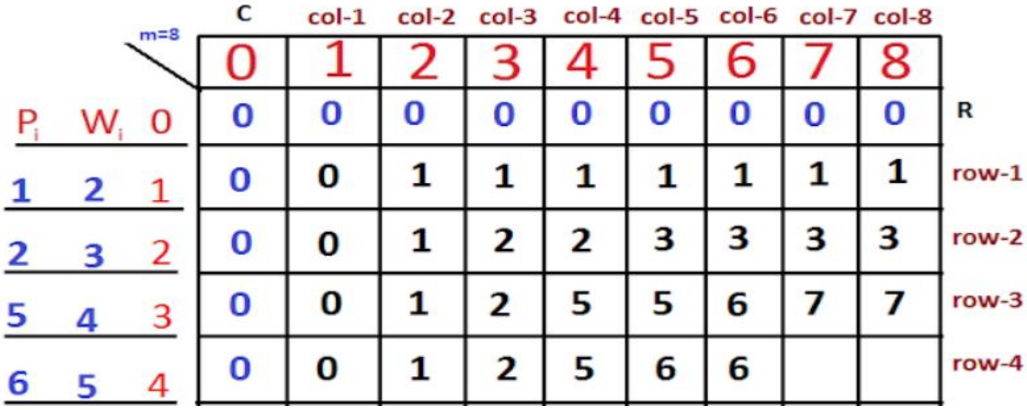
Now calculate for row-4 & col-5

Same procedure is done as above is given

			C	col-1	col-2	col-3	col-4	col-5	col-6	col-7	col-8	
		m=8	0	1	2	3	4	5	6	7	8	
P_i	W_i	0	0	0	0	0	0	0	0	0	0	R
1	2	1	0	0	1	1	1	1	1	1	1	row-1
2	3	2	0	0	1	2	2	3	3	3	3	row-2
5	4	3	0	0	1	2	5	5	6	7	7	row-3
6	5	4	0	0	1	2	5	6				row-4

Now calculate for row-4 & col-6

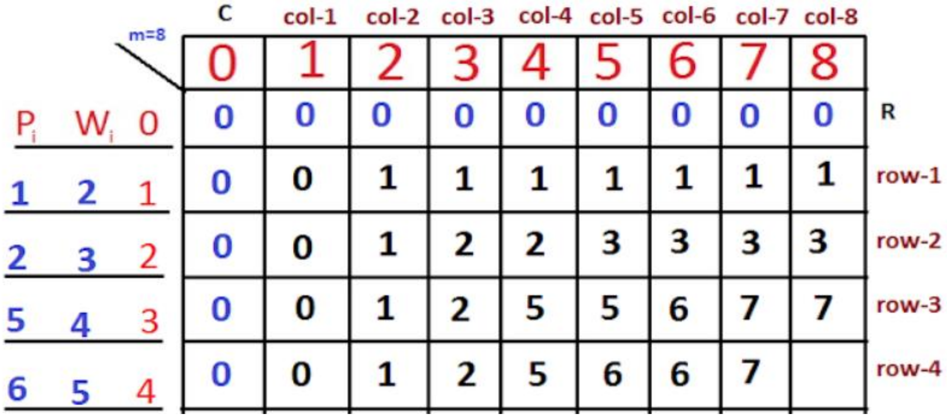
Same procedure is done as above is given



		m=8	C	col-1	col-2	col-3	col-4	col-5	col-6	col-7	col-8	
			0	1	2	3	4	5	6	7	8	
P_i	W_i	0	0	0	0	0	0	0	0	0	0	R
1	2	1	0	0	1	1	1	1	1	1	1	row-1
2	3	2	0	0	1	2	2	3	3	3	3	row-2
5	4	3	0	0	1	2	5	5	6	7	7	row-3
6	5	4	0	0	1	2	5	6	6			row-4

Now calculate for row-4 & col-7

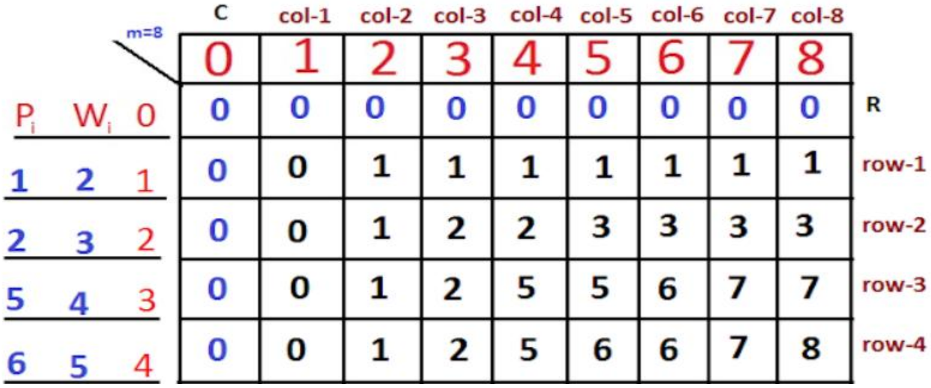
Same procedure is done as above is given



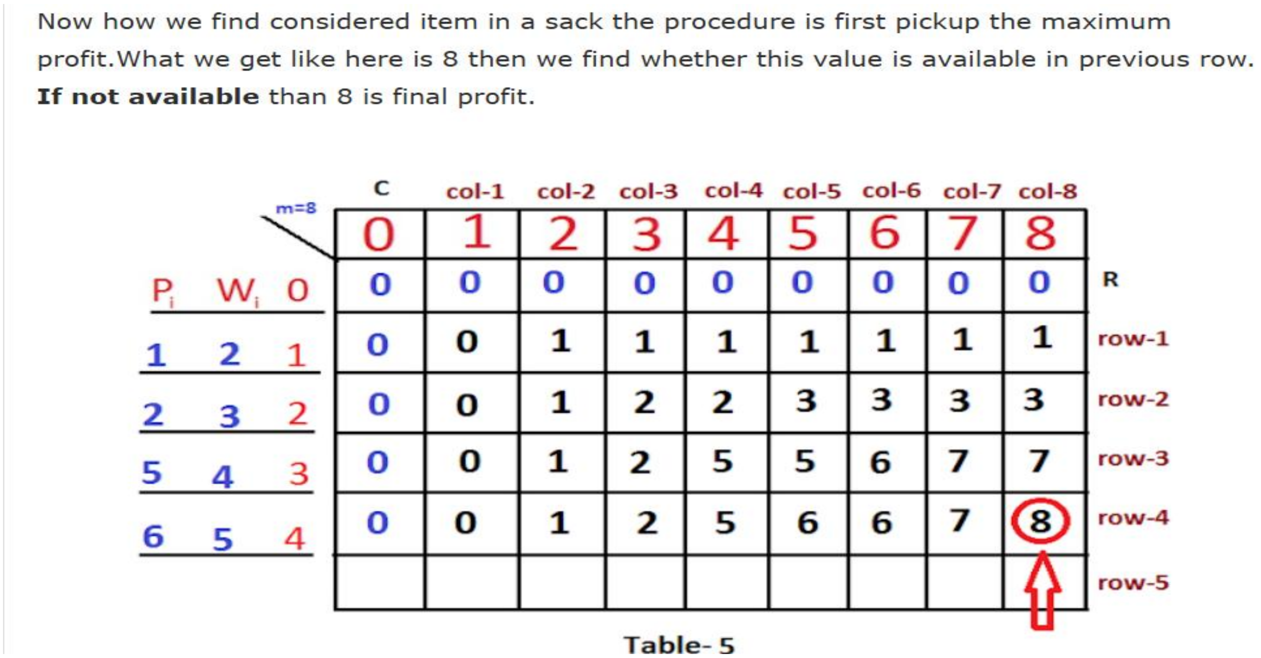
		m=8	C	col-1	col-2	col-3	col-4	col-5	col-6	col-7	col-8	
			0	1	2	3	4	5	6	7	8	
P_i	W_i	0	0	0	0	0	0	0	0	0	0	R
1	2	1	0	0	1	1	1	1	1	1	1	row-1
2	3	2	0	0	1	2	2	3	3	3	3	row-2
5	4	3	0	0	1	2	5	5	6	7	7	row-3
6	5	4	0	0	1	2	5	6	6	7		row-4

Now calculate for row-4 & col-8

Same procedure is done as above is given



		m=8	C	col-1	col-2	col-3	col-4	col-5	col-6	col-7	col-8	
			0	1	2	3	4	5	6	7	8	
P_i	W_i	0	0	0	0	0	0	0	0	0	0	R
1	2	1	0	0	1	1	1	1	1	1	1	row-1
2	3	2	0	0	1	2	2	3	3	3	3	row-2
5	4	3	0	0	1	2	5	5	6	7	7	row-3
6	5	4	0	0	1	2	5	6	6	7	8	row-4



Now how we find considered item in a sack the procedure is first pickup the maximum profit.What we get like here is 8 then we find whether this value is available in previous row. **If not available** than 8 is final profit.

		m=8	C	col-1	col-2	col-3	col-4	col-5	col-6	col-7	col-8	
			0	1	2	3	4	5	6	7	8	
P_i	W_i	0	0	0	0	0	0	0	0	0	0	R
1	2	1	0	0	1	1	1	1	1	1	1	row-1
2	3	2	0	0	1	2	2	3	3	3	3	row-2
5	4	3	0	0	1	2	5	5	6	7	7	row-3
6	5	4	0	0	1	2	5	6	6	7	8	row-4
												row-5

Table- 5

As we can observe in the above table that 8 is the maximum profit among all the entries. The pointer points to the last row and the last column having 8 value. Now we will compare 8 value with the **previous row;** if the previous row, i.e., i = 3 contains the same value 8 then the pointer will shift upwards. Since the previous row does not contain the value 5 so we will consider the row i=3, and the weight corresponding to the row is 4

Selection of objects/items

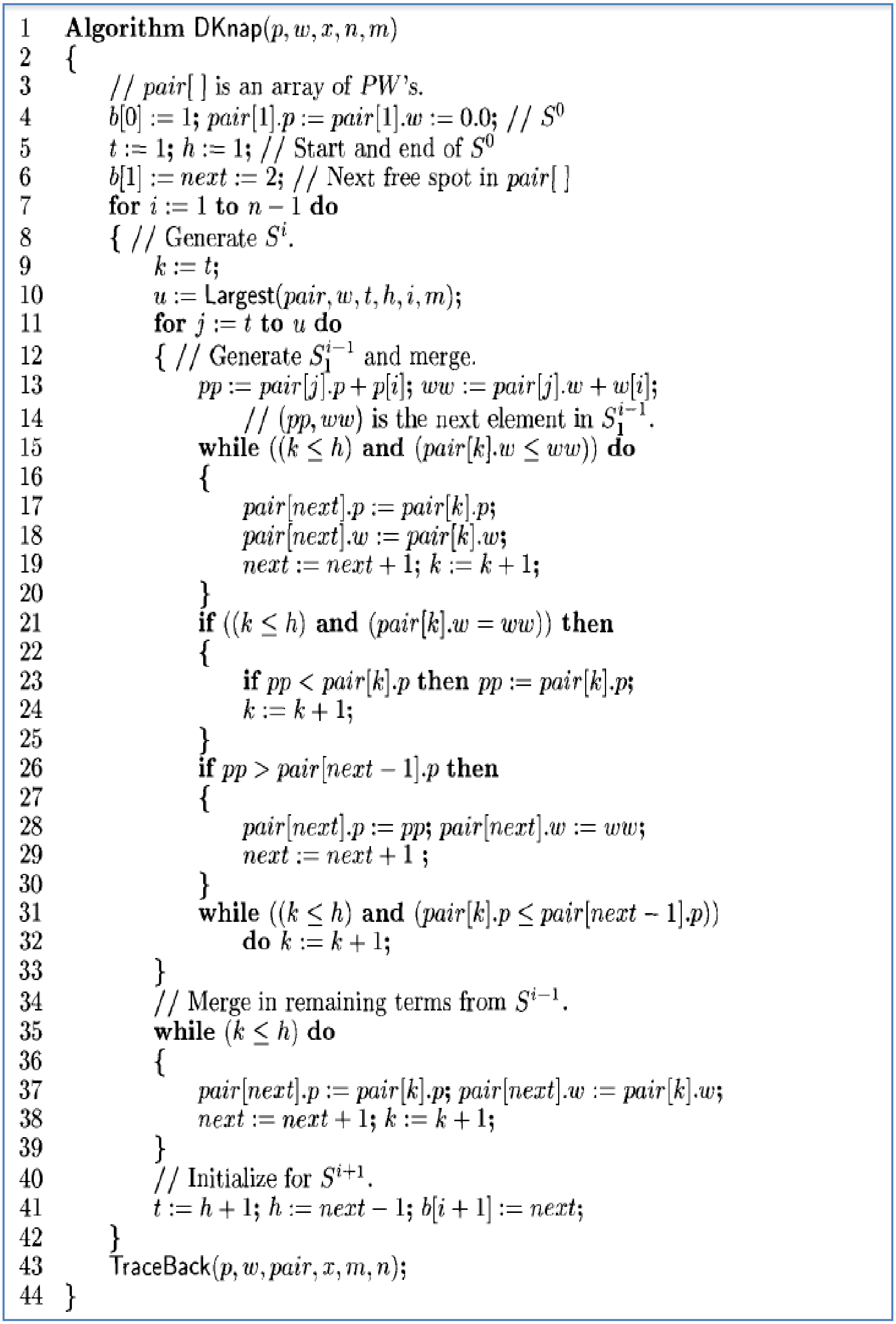
Backtracking:

- Start at dp[4][8] = 8.
- Compare it with dp[3][8] = 7. Since dp[4][8] != dp[3][8], **item 4 is selected** (weight 5, profit 6).
    - Now, reduce the capacity: w = 8 - 5 = 3. //
- Move to dp[3][3] = 2.
- Compare it with dp[2][3] = 2. Since dp[3][3] == dp[2][3], **item 3 is not selected**.
- Move to dp[2][3] = 2.
- Compare it with dp[1][3] = 1. Since dp[2][3] != dp[1][3], **item 2 is selected** (weight 3, profit 2).
    - Now, reduce the capacity: w = 3 - 3 = 0.
- Move to dp[1][0] = 0.
- Compare it with dp[0][0] = 0. Since dp[1][0] == dp[0][0], **item 1 is not selected**.

Selected Items:

- **Item 4** (Weight 5, Profit 6)
- **Item 2** (Weight 3, Profit 2)

(x1,x2,x3,x4) = (0,1,0,1)

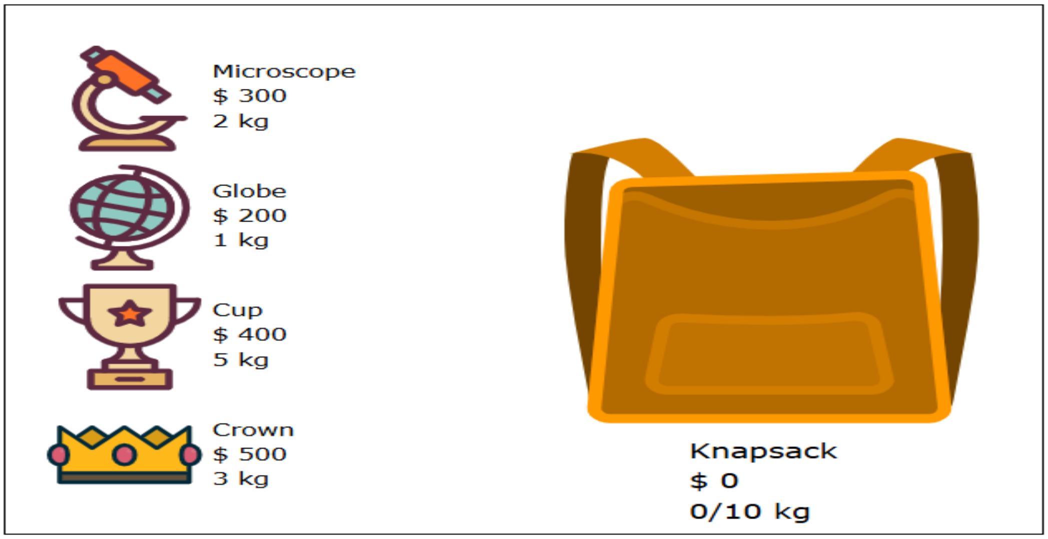


```
Algorithm DKnap(p, w, x, n, m)
{
    // pair[ ] is an array of PW's.
    b[0] := 1; pair[1].p := pair[1].w := 0.0; // $S^0$
    t := 1; h := 1; // Start and end of $S^0$
    b[1] := next := 2; // Next free spot in pair[ ]
    for i := 1 to n − 1 do
    { // Generate $S^i$.
        k := t;
        u := Largest(pair, w, t, h, i, m);
        for j := t to u do
        { // Generate $S_1^{i-1}$ and merge.
            pp := pair[j].p + p[i]; ww := pair[j].w + w[i];
                // (pp, ww) is the next element in $S_1^{i-1}$.
            while ((k ≤ h) and (pair[k].w ≤ ww)) do
            {
                pair[next].p := pair[k].p;
                pair[next].w := pair[k].w;
                next := next + 1; k := k + 1;
            }
            if ((k ≤ h) and (pair[k].w = ww)) then
            {
                if pp < pair[k].p then pp := pair[k].p;
                k := k + 1;
            }
            if pp > pair[next − 1].p then
            {
                pair[next].p := pp; pair[next].w := ww;
                next := next + 1 ;
            }
            while ((k ≤ h) and (pair[k].p ≤ pair[next − 1].p))
                do k := k + 1;
        }
        // Merge in remaining terms from $S^{i-1}$.
        while (k ≤ h) do
        {
            pair[next].p := pair[k].p; pair[next].w := pair[k].w;
            next := next + 1; k := k + 1;
        }
        // Initialize for $S^{i+1}$.
        t := h + 1; h := next − 1; b[i + 1] := next;
    }
    TraceBack(p, w, pair, x, m, n);
}
```

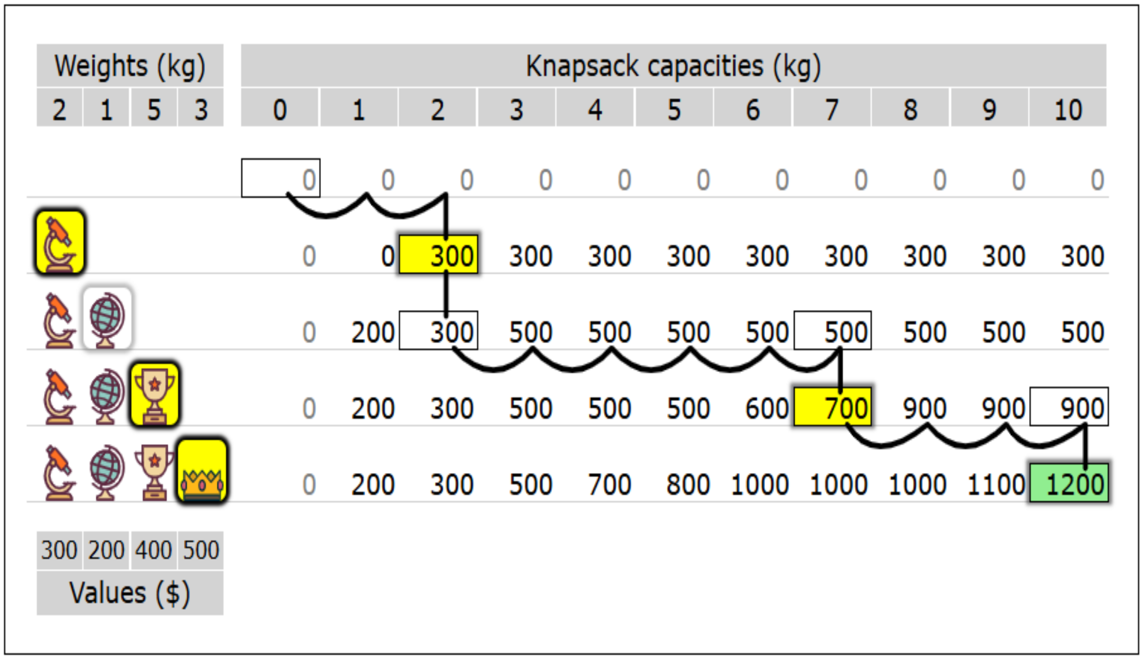
Algorithm- Algorithm for 0/1 knapsack problem

Example-2:

For the given set of items and knapsack capacity = 10 kg, find the optimal solution for the 0/1 knapsack problem making use of dynamic programming approach.



> Here is a drawing of how the included items are found, using the step-by-step method:



Weights (kg)				Knapsack capacities (kg)										
2	1	5	3	0	1	2	3	4	5	6	7	8	9	10
				0	0	0	0	0	0	0	0	0	0	0
				0	0	300	300	300	300	300	300	300	300	300
				0	200	300	500	500	500	500	500	500	500	500
				0	200	300	500	500	500	600	700	900	900	900
				0	200	300	500	700	800	1000	1000	1000	1100	1200
300	200	400	500											
Values ($)														

This is how the included items are found:

1. The bottom right value is 1200, and the cell above is 900. The values are different, which means the crown is included.
2. The next cell we go to is on the row above, and we move left as many times as the crown is heavy, so 3 places left to the cell with value 700.

3. The cell we are in now has value 700, and the cell above has value 500. The values are different, which means the item on the current row is included: the cup.
4. The cup weighs 5 kg, so the next cell we go to is on the row above, and 5 places to the left, to the cell with value 300, on the row were the globe is considered.
5. The cell above has the same value 300, which means the globe is not included, and the next cell we go to is the cell right above with value 300 where the microscope is considered.
6. Since the cell above is different than the current cell with value 300, it means the microscope is included.
7. The next cell we go to is on the line above, and two places to the left because the microscope is 2 kg.
8. We arrive at the upper leftmost cell. Since the value is 0, it means we are finished.

Our 0/1 Knapsack problem has maximum value when these items are included: the **crown, the cup, and the microscope.**

Example-3:

For the given set of items and knapsack capacity = 5 kg, find the optimal solution for the 0/1 knapsack problem making use of dynamic programming approach.

Item	**Weight**	**Value**
1	**2**	**3**
2	**3**	**4**
3	**4**	**5**
4	**5**	**6**

Example-4:

For the given set of items and knapsack capacity = 8 kg, find the optimal solution for the 0/1 knapsack problem making use of dynamic programming approach.

Item	**A**	**B**	**C**	**D**
Profit	2	4	7	10
Weight	1	3	5	7