**EX - 8: Dynamic Programming**

**Aim:** Implementation of different real time problems using Dynamic Greedy approach.

1. All Pairs shortest paths
2. Single source shortest path for general weights
3. 0/1 Knapsack problem
4. Optimal binary search tree
5. Travelling sales person problem

**Program Description:**

**Dynamic programming** are a class of algorithms solve complex problems by breaking them down into simpler subproblems. By solving each subproblem only once and storing the results, it avoids redundant computations, leading to more efficient solutions for a wide range of problems. It guarantees to get optimal solutions for the given problem.

**Approach of Dynamic programming:**

* **dentify Subproblems:** Divide the main problem into smaller, independent subproblems.
* **Store Solutions:**Solve each subproblem and store the solution in a table or array.
* **Build Up Solutions:** Use the stored solutions to build up the solution to the main problem.
* **Avoid Redundancy:** By storing solutions, DP ensures that each subproblem is solved only once, reducing computation time.

Dynamic programming can be achieved using two approaches:

**1. Top-down approach:** In computer science, problems are resolved by recursively formulating solutions, employing the answers to the problems’ subproblems. If the answers to the subproblems overlap, they may be memoized or kept in a table for later use. The top-down approach follows the strategy of memorization. The memoization process is equivalent to adding the recursion and caching steps. The difference between recursion and caching is that recursion requires calling the function directly, whereas caching requires preserving the intermediate results.

**2.Bottom-up approach:** In the bottom-up method, once a solution to a problem is written in terms of its subproblems in a way that loops back on itself, users can rewrite the problem by solving the smaller subproblems first and then using their solutions to solve the larger subproblems.  Unlike the top-down approach, the bottom-up approach removes the recursion. Thus, there is neither stack overflow nor overhead from the recursive functions. It also allows for saving memory space. Removing recursion decreases the time complexity of recursion due to recalculating the same values.

**Applications of Dynamic programming approach;**

1. **Implementation of All pairs shortest paths problem (Floyd Warshall algorithm):**

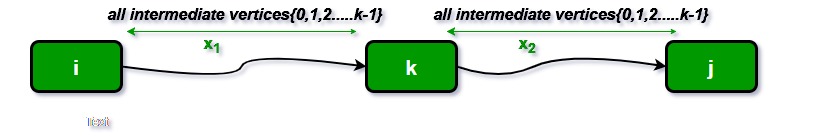
The Floyd-Warshall algorithm, named after its creators Robert Floyd and Stephen Warshall, is a fundamental algorithm in computer science and graph theory. It is used to find the shortest paths between all pairs of nodes in a weighted graph. This algorithm is highly efficient and can handle graphs with both positive and negative edge weights, making it a versatile tool for solving a wide range of network and connectivity problems.

**Idea Behind Floyd Warshall Algorithm:**

Suppose we have a graph **G[][]** with **V** vertices from **1** to **N**. Now we have to evaluate a **shortestPathMatrix[][]** where s**hortestPathMatrix[i][j]** represents the shortest path between vertices **i** and **j**.

Obviously the shortest path between **i** to **j** will have some **k** number of intermediate nodes. The idea behind floyd warshall algorithm is to treat each and every vertex from **1** to **N** as an intermediate node one by one.

The following figure shows the above optimal substructure property in floyd warshall algorithm:



**Floyd Warshall Algorithm:**

* Initialize the solution matrix same as the input graph matrix as a first step.
* Then update the solution matrix by considering all vertices as an intermediate vertex.
* The idea is to pick all vertices one by one and updates all shortest paths which include the picked vertex as an intermediate vertex in the shortest path.
* When we pick vertex number **k** as an intermediate vertex, we already have considered vertices **{0, 1, 2, .. k-1}**as intermediate vertices.
* For every pair**(i, j)** of the source and destination vertices respectively, there are two possible cases.
  + **k** is not an intermediate vertex in shortest path from**i**to**j**. We keep the value of**dist[i][j]**as it is.
  + **k** is an intermediate vertex in shortest path from **i** to**j**. We update the value of**dist[i][j]**as **dist[i][k] + dist[k][j],** if **dist[i][j] > dist[i][k] + dist[k][j]**

**Recurrence relation:**

*Distance[i][j] = minimum (Distance[i][j], (Distance from i to****A****) + (Distance from****A****to j ))*

**Pseudo-Code of Floyd Warshall’s Algorithm :**

*For k = 0 to n – 1   
 For i = 0 to n – 1   
 For j = 0 to n – 1   
 Distance[i, j] = min(Distance[i, j], Distance[i, k] + Distance[k, j])*

Time efficiency: O(n3)

Space efficiency: Matrices can be written over their predecessors

Program Code:

Input and Output:

1. **Single source shortest path for general weights (Bellman Ford algorithm):**

The **Bellman–Ford algorithm**  helps to find the shortest path from one city to all other cities, even if some roads have negative lengths. It’s like a **GPS** for computers, useful for figuring out the quickest way to get from one point to another in a network.

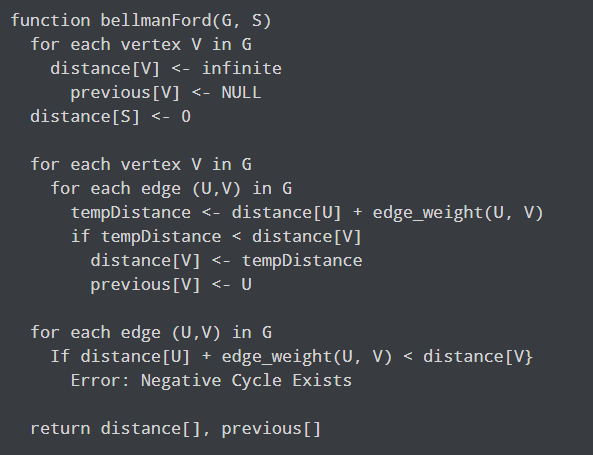
**The idea behind Bellman Ford Algorithm:**

The Bellman-Ford algorithm’s primary principle is that it starts with a single source and calculates the distance to each node. The distance is initially unknown and assumed to be infinite, but as time goes on, the algorithm relaxes those paths by identifying a few shorter paths. The algorithm repeatedly relaxes all edges in the graph, checking if the current shortest path to a destination can be improved by taking that edge. If so, the algorithm updates the shortest path.Hence it is said that Bellman-Ford is based on “**Principle of Relaxation**“.

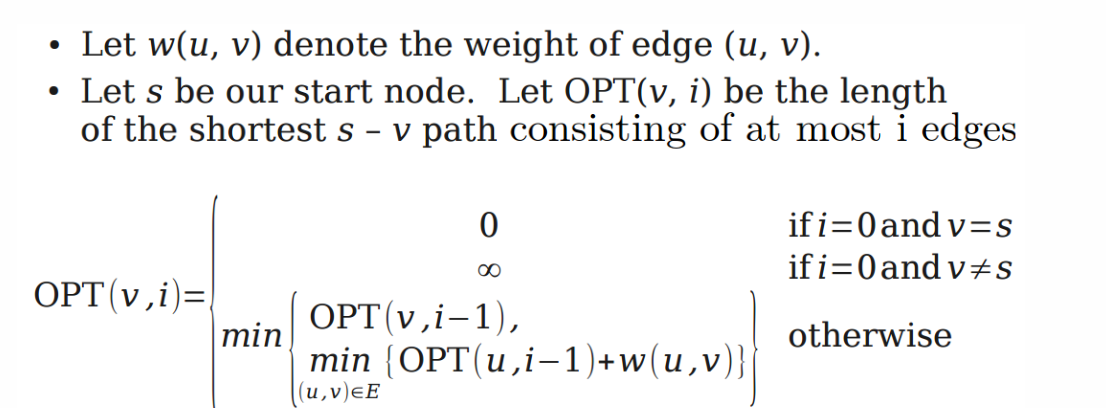
**Principle of Relaxation of Edges for Bellman-Ford:**

* It states that for the graph having **N** vertices, all the edges should be relaxed **N-1**times to compute the single source shortest path.
* In order to detect whether a negative cycle exists or not, relax all the edge one more time and if the shortest distance for any node reduces then we can say that a negative cycle exists.
* In short if we relax the edges **N** times, and there is any change in the shortest distance of any node between the **N-1th** and **Nth** relaxation than a negative cycle exists, otherwise not exist.

Algorithmic steps:



Recurrence relation:



Time complexity: **O(V \* E)**

**Space complexity: O(V)**

Program Code:

Input and Output:

1. **0/1 Knapsack problem:**

The 0/1 knapsack problem means that the items are either completely or no items are filled in a knapsack**.**



The 0/1 Knapsack Problem

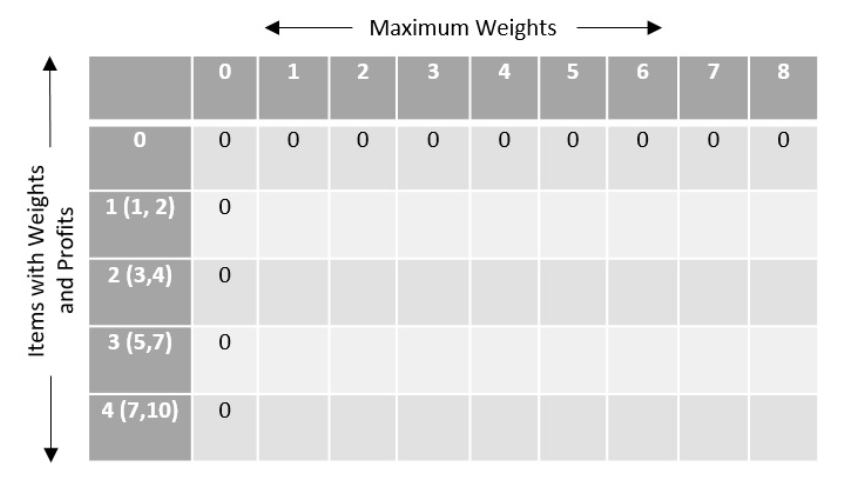
Rules:

* Every item has a weight and value.
* Your knapsack has a weight limit.
* Choose which items you want to bring with you in the knapsack.
* You can either take an item or not, you cannot take half of an item for example.

Goal:

* Maximize the total value of the items in the knapsack.

The memoization (Dynamic programming) technique stores the previous function call results in an array, so that previous results can be fetched from that array and does not have to be calculated again as shown below table:

****

****

**Recurrence relation:**

**F(I,j) = max{ F(i-1, j), profiti  + F(i-1, j-weighti) } if j - wi ≥ 0**

**Otherwise F(I - 1, j), j - wi <0**

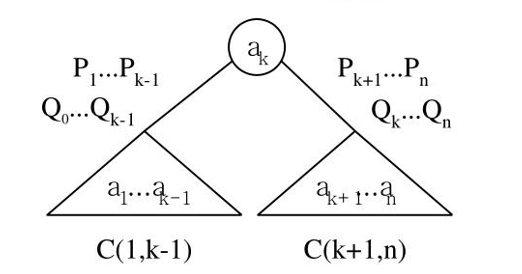
Time Complexity: O(N \* W). As redundant calculations of states are avoided.  
 Auxiliary Space: O(N \* W) + O(N)

**Program code:**

**Input and Output:**

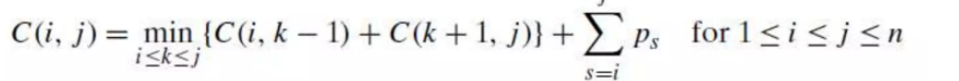
1. **Optimal Binary search tree (OBST):**

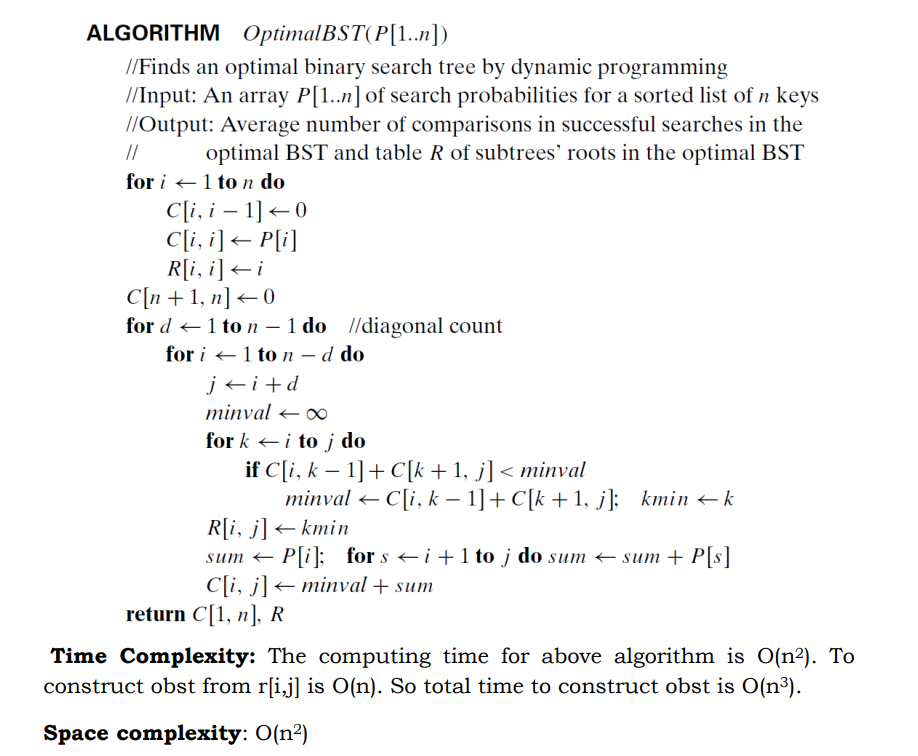
OBST is a binary search tree which provides the smallest possible search time (or expected search) for a given sequence of accesses (or access probabilities).

****

**OBST successful case:**

**Recurrence relation for successful search:**

****

****

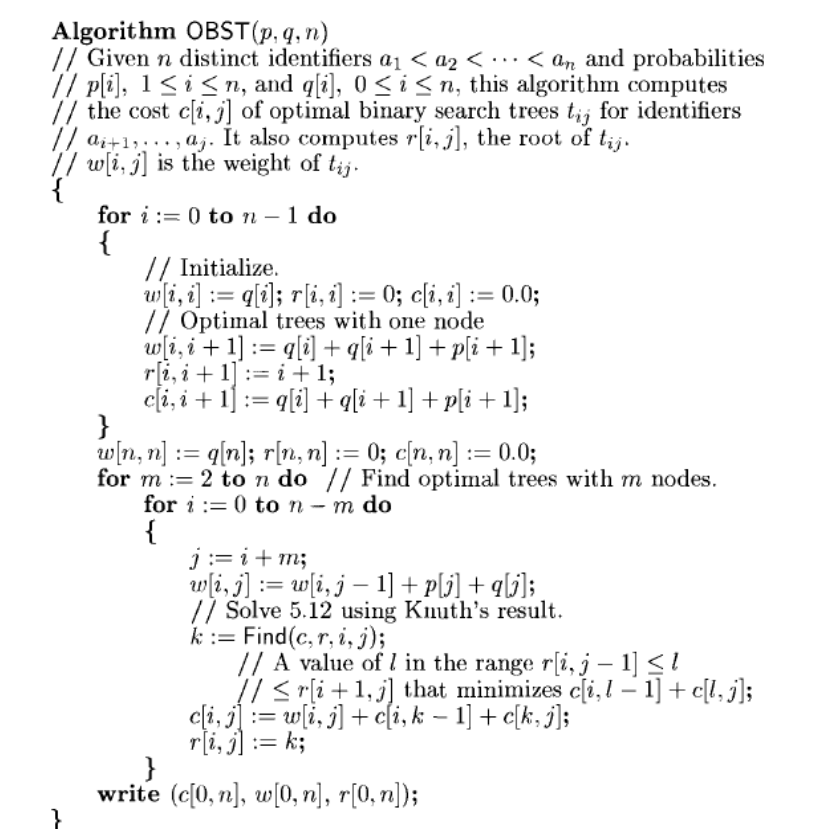
**OBST unsuccessful and successful case:**

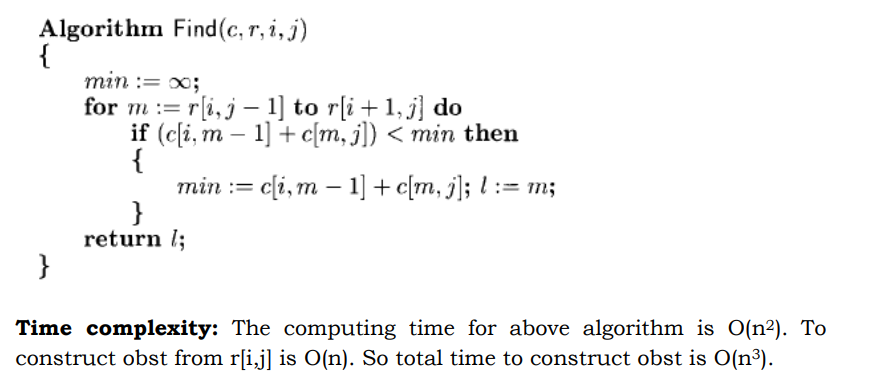
**Recurrence relation for successful and unsuccessful search:**

**W(I,j)= W(i,j-1) + P(j)+Q(j)**

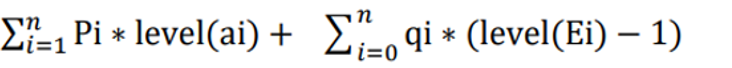
**C(i,j)= min { C(i,k-1)+C(k,j)} +W(i,j)**

**R(i,j) = minimum of k**

****

****

**Total cost of OBST is:**

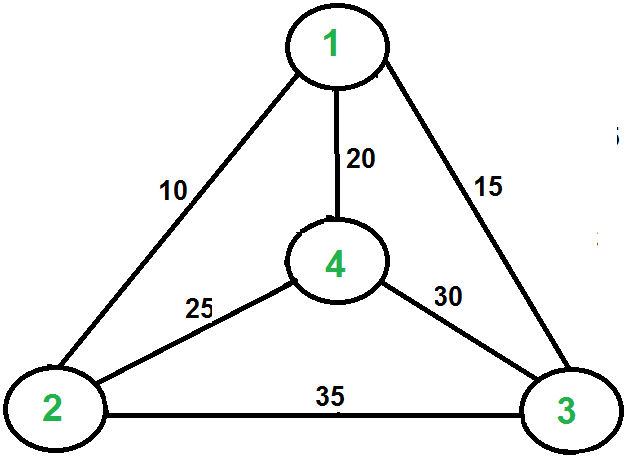
****

**Program code;**

**Input and Output:**

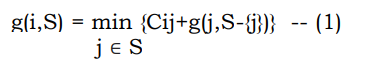
1. **Travelling sales person problem:**

Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point.



Let G=(V,E) be a directed graph with edge cost Cij. The variable Cij is define such that Cij > 0 every i,j and Cij = ∞ if (i,j) ∈ E. Let |V| = n and assume n>1. A tour of G is a directed simple cycle that include every vertex in V. The cost of a tour is the sum of the cost of the edges on the tour. The travelling salesperson problem is to find a tour of minimum cost without loss of generality, assume a tour is a simple path that starts and ends at vertex 1. Every tour consists of an edge (1,k) for some k ∈ V-{1} and a path from vertex k to vertex 1. The path from vertex k to vertex 1 goes through each vertex in V-{1,k} exactly once. It is easy to see that if the tour is optimal, then the path from k to 1 must be a shortest k to1 path going through all vertices in V-{1,k} Let g(i,S) be the length of a shortest path starting at vertex i, going through all vertices in S and terminating at vertex 1. The function g(1, V-{1}) is the length of an optimal salesperson tour.

The recurrence relation:



Time complexity: O(n22n) as the computation of g(i,S) with|S| = k requires k-1 comparisons when solving equation (1).

Space Complexity: O(n.2n)

Program code;

Input and Output: