Dynamic Programming

By: Ashok Basnet

Course Outline

- Dynamic Programming
- Multistage Graphs
- All Pair Shortest Paths
- Single Source Shortest Path: General Weights
- Optimal Binary Search Trees
- String Editing Problem
- 0/1-Knapsack Problem
- Reliability Design
- TSP

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

- Dynamic programming is based on the **principle of optimality.**
- 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.

Dynamic Programming

- 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 recomputing the answer every time a sub-problem is encountered.***

Dynamic Programming

- 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, then store the answers away in a table to be used later in solving larger problems.*
- Care is to be taken to avoid recomputing 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.

Dynamic Programming

Two difficulties may arise in any application of dynamic programming:

- ***It may not always be possible to combine the solutions of smaller problems to form the solution of a larger one.***
- ***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.

Multi Stage Graphs

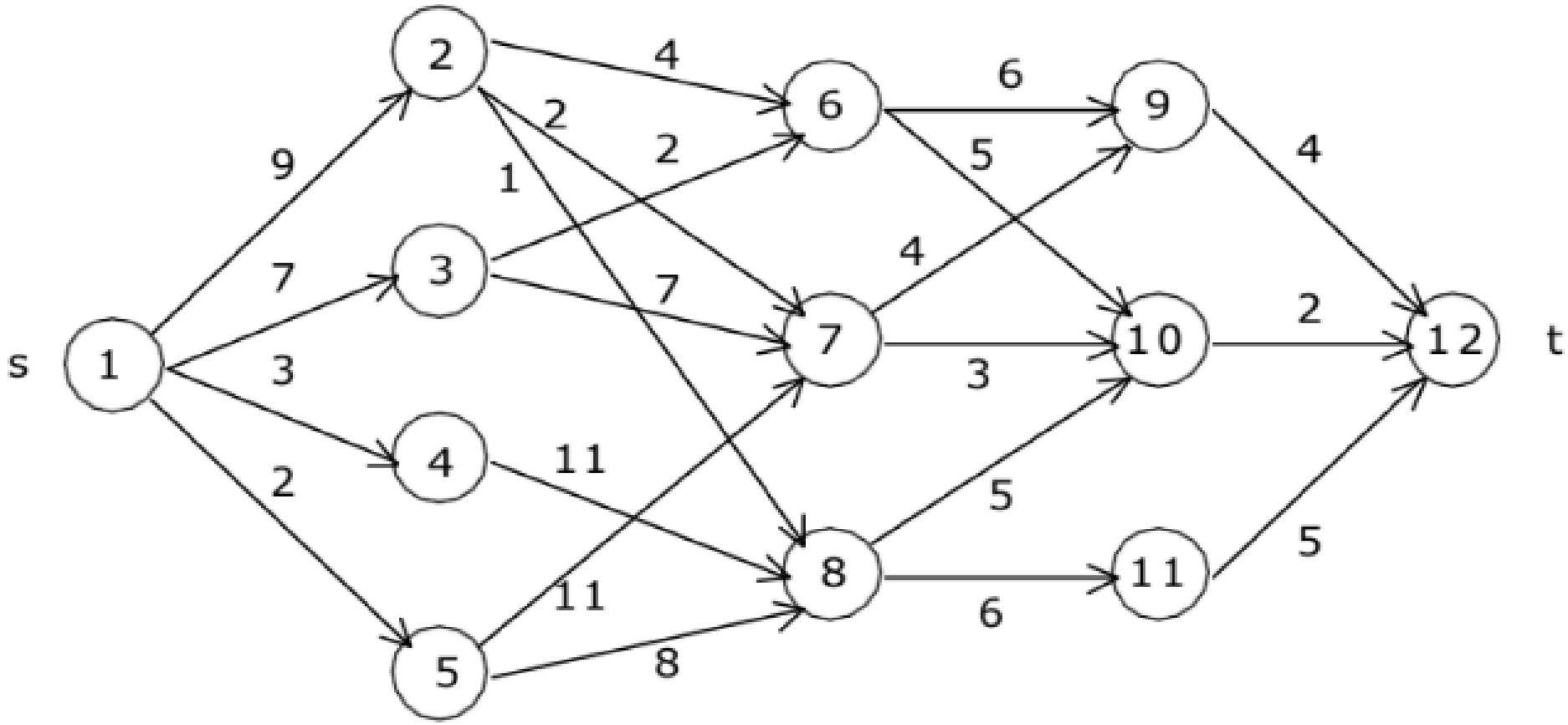
- A multistage graph ***G = (V, E)*** is a directed graph in which the vertices are partitioned into k ≥ 2 disjoint sets V_i, $1 \le i \le k$. In addition, if ***<u, v>*** is an edge in ***E***, then $u \in V_i$ and $v \in V_{i+1}$ for some ***i, 1 ≤ i ≤ k.***
- Let the vertex ***'s'*** is the source, and ***'t'*** the sink. 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 V_i 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.

Multi Stage Graphs

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

$$\text{cost}(i, j) = \min_{\substack{l \in V_{i+1} \\ \langle j, l \rangle \in E}} \{c(j, l) + \text{cost}(i + 1, l)\}$$

Multi Stage Graphs



Multi Stage Graphs

- Cost can be represented as: ***cost(stage_number, vertext_number)***

V	1	2	3	4	5	6	7	8	9	10	11	12
cost												
d												

Multi Stage Graphs

- For fifth stage

V	1	2	3	4	5	6	7	8	9	10	11	12
cost												0
d												12

- Cost(5, 12) = 0

Multi Stage Graphs

- for fourth stage

V	1	2	3	4	5	6	7	8	9	10	11	12
cost									4	2	5	0
d									12	12	12	12

- Cost(4, 9) = 4
- Cost(4, 10) = 2
- Cost(4, 11) = 5

Multi Stage Graphs

- for third stage

V	1	2	3	4	5	6	7	8	9	10	11	12
cost						7			4	2	5	0
d						10			12	12	12	12

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

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

Multi Stage Graphs

- for third stage

V	1	2	3	4	5	6	7	8	9	10	11	12
cost						7	5		4	2	5	0
d						10	10		12	12	12	12

- Cost(3, 6) = min{ c(6,9) + cost(4, 9), c(6, 10) + cost(4, 10) }
  = min { 6+4, 5+2 } = 7
- Cost(3, 7) = min{ c(7,9) + cost(4, 9) , c(7, 10) + cost(4, 10) }
  = min{ 4 + 4, 3 + 2 } = 5

Multi Stage Graphs

- for third stage

V	1	2	3	4	5	6	7	8	9	10	11	12
cost						7	5	7	4	2	5	0
d						10	10	10	12	12	12	12

- Cost(3, 6) = min{ c(6,9) + cost(4, 9), c(6, 10) + cost(4, 10) }
  = min { 6+4, 5+2 } = 7
- Cost(3, 7) = min{ c(7,9) + cost(4, 9) , c(7, 10) + cost(4, 10) }
  = min{ 4 + 4, 3 + 2 } = 5
- Cost(3, 8) = min { c(8, 10) + cost (4, 10) , c(8, 11) + cost(4, 11)
  = min { 5+2, 6+5 } = 7

Multi Stage Graphs

- For second stage

V	1	2	3	4	5	6	7	8	9	10	11	12
cost		7				7	5	7	4	2	5	0
d		7				10	10	10	12	12	12	12

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

  = { 4+7, 2+5, 1+7 } = 7

Multi Stage Graphs

- for second stage

V	1	2	3	4	5	6	7	8	9	10	11	12
cost		7	9			7	5	7	4	2	5	0
d		7	6			10	10	10	12	12	12	12

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

  = { 4+7, 2+5, 1+7 } = 7
- Cost(2, 3) = min{ c(3, 6) + cost (3, 6), c(3, 7) + cost(3, 7) }

  = { 2+7, 7+5 } = 9

Multi Stage Graphs

- for second stage

V	1	2	3	4	5	6	7	8	9	10	11	12
cost		7	9	18		7	5	7	4	2	5	0
d		7	6	8		10	10	10	12	12	12	12

- Cost(2, 2) = min{ c(2, 6) + cost (3, 6), c(2, 7) + cost(3, 7), c(2, 8) + cost(3, 8) }
  = { 4+7, 2+5, 1+7 } = 7
- Cost(2, 3) = min{ c(3, 6) + cost (3, 6), c(3, 7) + cost(3, 7) }
  = { 2+7, 7+5 } = 9
- Cost(2, 4) = min{ c(4, 8) + cost (3, 8) }
  = { 11+7 } = 18

Multi Stage Graphs

- for second stage

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

- Cost(2, 2) = min{ c(2, 6) + cost (3, 6), c(2, 7) + cost(3, 7), c(2, 8) + cost(3, 8) }
  = { 4+7, 2+5, 1+7 } = 7
- Cost(2, 3) = min{ c(3, 6) + cost (3, 6), c(3, 7) + cost(3, 7) }
  = { 2+7, 7+5 } = 9
- Cost(2, 4) = min{ c(4, 8) + cost (3, 8) }
  = { 11+7 } = 18
- Cost(2, 5) = min{ c(5, 7) + cost (3, 7), c(5, 8) + cost(3, 8) }
  = { 11+5, 8+7 } = 15

Multi Stage Graphs

- for first stage

V	1	2	3	4	5	6	7	8	9	10	11	12
cost	16	7	9	18	15	7	5	7	4	2	5	0
d	2 or 3	7	6	8	8	10	10	10	12	12	12	12

- 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+7, 7+9, 3+18, 2+15 } = min(16, 16, 21, 17)

Multi Stage Graphs

V	1	2	3	4	5	6	7	8	9	10	11	12
cost	16	7	9	18	15	7	5	7	4	2	5	0
d	2 or 3	7	6	8	8	10	10	10	12	12	12	12

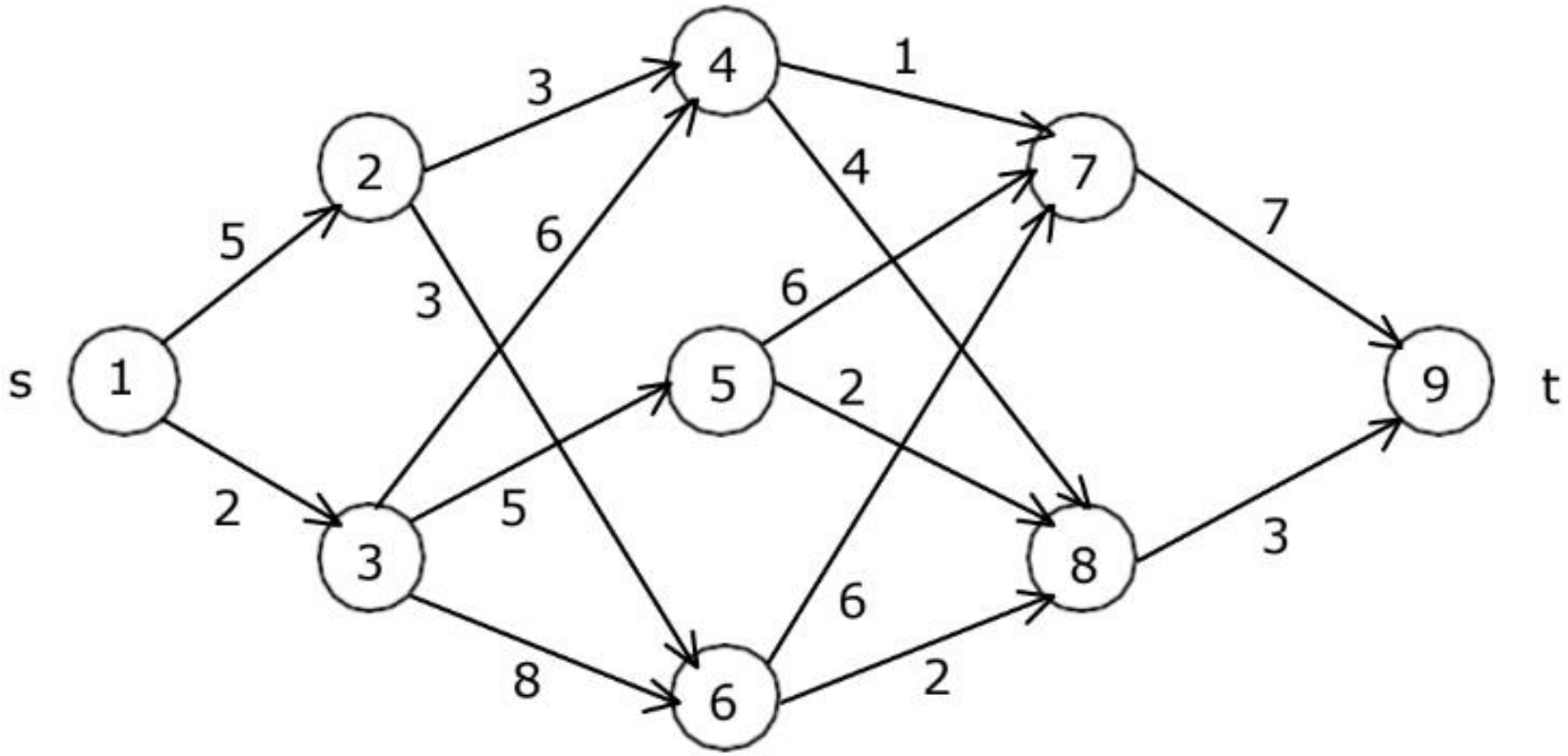
- We have complete data. Now we can implement the dynamic programming to make decision.
- For forward approach, we will move from source to sink.
- d(stage, vertex) = d(1, 1) = 2
- d(2, 2) = 7
- d(3, 7) = 10
- d(4, 10) = 12
- **Path = 1, 2, 7, 10, 12**

Multi Stage Graphs

V	1	2	3	4	5	6	7	8	9	10	11	12
cost	16	7	9	18	15	7	5	7	4	2	5	0
d	2 or 3	7	6	8	8	10	10	10	12	12	12	12

- decision.
- For forward approach, we will move from source to sink.
- d(stage, vertex) = d(1, 1) = 3
- d(2, 3) = 6
- d(3, 6) = 10
- d(4, 10) = 12
- **Path = 1, 3, 6, 10, 12**

Multi Stage Graphs



• Algorithm for Multistage Graph (Shortest Path):

- **Initialization:**
  - Create a table to store the optimal costs for each node in each stage.
  - Initialize the last stage's costs as the weights of the corresponding nodes.
- **Bottom-Up Computation:**
  - Starting from the penultimate stage and moving backward, compute the optimal costs for each node in each stage based on the optimal costs of the next stage.
  - Use the following recurrence relation for each node in a stage `i`:

    cost[i] = min(weight(i, j) + cost[j]) for all j in the next stage

    Here, `weight(i, j)` represents the weight of the edge from node `i` to node `j`.
- **Traceback:**
  - After completing the bottom-up computation, the optimal cost of the starting node is obtained.
  - Trace back through the computed costs to reconstruct the optimal path.
- **Optimal Solution:**
  - The optimal solution is the path from the starting node to the ending node with the minimum total cost.


All pairs shortest paths

All pairs shortest paths (Floyd-Warshall)

- In the all pairs shortest path problem, we are to find ***a shortest path between every pair of vertices in a directed graph G.***
- That is, for every pair of vertices ***(i, j)***, we are to find a shortest path from ***i to j*** as well as one from ***j to i.***
- These two paths are the same when G is undirected.
- The all pairs shortest path problem is to determine a matrix A such that *A (i, j) is the length of a shortest path from i to j.*
- The matrix A can be obtained by solving n single-source problems using the algorithm shortest Paths.

All pairs shortest paths

- The shortest *i to j path in G, i ≠ j originates at vertex i and goes through some intermediate vertices (possibly none) and terminates at vertex j.*
- If k is an intermediate vertex on this shortest path, then the sub paths from ***i to k and from k to j must be shortest paths from i to k and k to j,*** respectively.
- Otherwise, the i to j path is not of minimum length. So, the principle of optimality holds.
- Let $A^k(i, j)$ represent the length of a shortest path from i to j going through no vertex of index greater than k, we obtain:

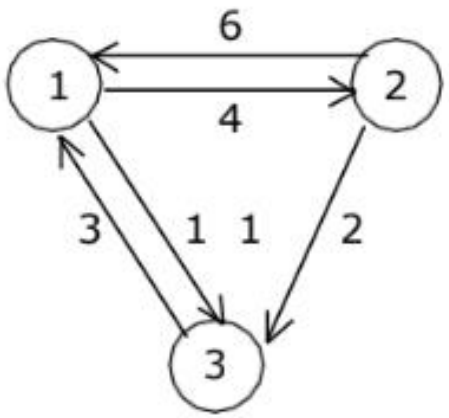
$$A^k(i, j) = \{\min_{1 \le k \le n} \{\min \{A^{k-1}(i, k) + A^{k-1}(k, j)\}, c(i, j)\}$$

All pairs shortest paths

```
Algorithm All Paths (Cost, A, n)
// cost [1:n, 1:n] is the cost adjacency matrix of a graph which
// n vertices; A [I, j] is the cost of a shortest path from vertex
// i to vertex j. cost [i, i] = 0.0, for 1 ≤ i ≤ n.
{
        for i := 1 to n do
                for j:= 1 to n do
                        A [i, j] := cost   [i, j];                    // copy cost into A.
        for k := 1 to n do
                for i := 1 to n do
                        for j := 1 to n do
                                A [i, j] := min (A [i, j], A [i, k] + A [k, j]);
}
```

All pairs shortest paths

Given a weighted digraph G = (V, E) with weight. Determine the length of the shortest path between all pairs of vertices in G. Here we assume that there are no cycles with zero or negative cost.



Cost adjacency matrix $(A^0) = \begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \infty & 0 \end{bmatrix}$

General formula: $\min_{1 \le k \le n} \{A^{k-1}(i, k) + A^{k-1}(k, j)\}, c(i, j)\}$

Solve the problem for different values of k = 1, 2 and 3

All pairs shortest paths

$$A^1(1,1) = \min\{(A^0(1,1) + A^0(1,1)), c(1,1)\} = \min\{0+0, 0\} = 0$$

$$A^1(1,2) = \min\{(A^0(1,1) + A^0(1,2)), c(1,2)\} = \min\{(0+4), 4\} = 4$$

$$A^1(1,3) = \min\{(A^0(1,1) + A^0(1,3)), c(1,3)\} = \min\{(0+11), 11\} = 11$$

$$A^1(2,1) = \min\{(A^0(2,1) + A^0(1,1)), c(2,1)\} = \min\{(6+0), 6\} = 6$$

$$A^1(2,2) = \min\{(A^0(2,1) + A^0(1,2)), c(2,2)\} = \min\{(6+4), 0)\} = 0$$

$$A^1(2,3) = \min\{(A^0(2,1) + A^0(1,3)), c(2,3)\} = \min\{(6+11), 2\} = 2$$

$$A^1(3,1) = \min\{(A^0(3,1) + A^0(1,1)), c(3,1)\} = \min\{(3+0), 3\} = 3$$

$$A^1(3,2) = \min\{(A^0(3,1) + A^0(1,2)), c(3,2)\} = \min\{(3+4), \infty\} = 7$$

$$A^1(3,3) = \min\{(A^0(3,1) + A^0(1,3)), c(3,3)\} = \min\{(3+11), 0\} = 0$$

$$A^{(1)} = \begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$$

All pairs shortest paths

Step 2: Solving the equation for, K = 2;

$A^2(1, 1) = \min\{(A^1(1, 2) + A^1(2, 1), c(1, 1)\} = \min\{(4 + 6), 0\} = 0$

$A^2(1, 2) = \min\{(A^1(1, 2) + A^1(2, 2), c(1, 2)\} = \min\{(4 + 0), 4\} = 4$

$A^2(1, 3) = \min\{(A^1(1, 2) + A^1(2, 3), c(1, 3)\} = \min\{(4 + 2), 11\} = 6$

$A^2(2, 1) = \min\{(A(2, 2) + A(2, 1), c(2, 1)\} = \min\{(0 + 6), 6\} = 6$

$A^2(2, 2) = \min\{(A(2, 2) + A(2, 2), c(2, 2)\} = \min\{(0 + 0), 0\} = 0$

$A^2(2, 3) = \min\{(A(2, 2) + A(2, 3), c(2, 3)\} = \min\{(0 + 2), 2\} = 2$

$A^2(3, 1) = \min\{(A(3, 2) + A(2, 1), c(3, 1)\} = \min\{(7 + 6), 3\} = 3$

$A^2(3, 2) = \min\{(A(3, 2) + A(2, 2), c(3, 2)\} = \min\{(7 + 0), 7\} = 7$

$A^2(3, 3) = \min\{(A(3, 2) + A(2, 3), c(3, 3)\} = \min\{(7 + 2), 0\} = 0$

$$A^{(2)} = \begin{bmatrix} 0 & 4 & 6 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$$

All pairs shortest paths

Step 3: Solving the equation for, k = 3;

$A^3(1, 1) = \min \{A^2(1, 3) + A^2(3, 1), c(1, 1)\} = \min \{(6 + 3), 0\} = 0$

$A^3(1, 2) = \min \{A^2(1, 3) + A^2(3, 2), c(1, 2)\} = \min \{(6 + 7), 4\} = 4$

$A^3(1, 3) = \min \{A^2(1, 3) + A^2(3, 3), c(1, 3)\} = \min \{(6 + 0), 6\} = 6$

$A^3(2, 1) = \min \{A^2(2, 3) + A^2(3, 1), c(2, 1)\} = \min \{(2 + 3), 6\} = 5$

$A^3(2, 2) = \min \{A^2(2, 3) + A^2(3, 2), c(2, 2)\} = \min \{(2 + 7), 0\} = 0$

$A^3(2, 3) = \min \{A^2(2, 3) + A^2(3, 3), c(2, 3)\} = \min \{(2 + 0), 2\} = 2$

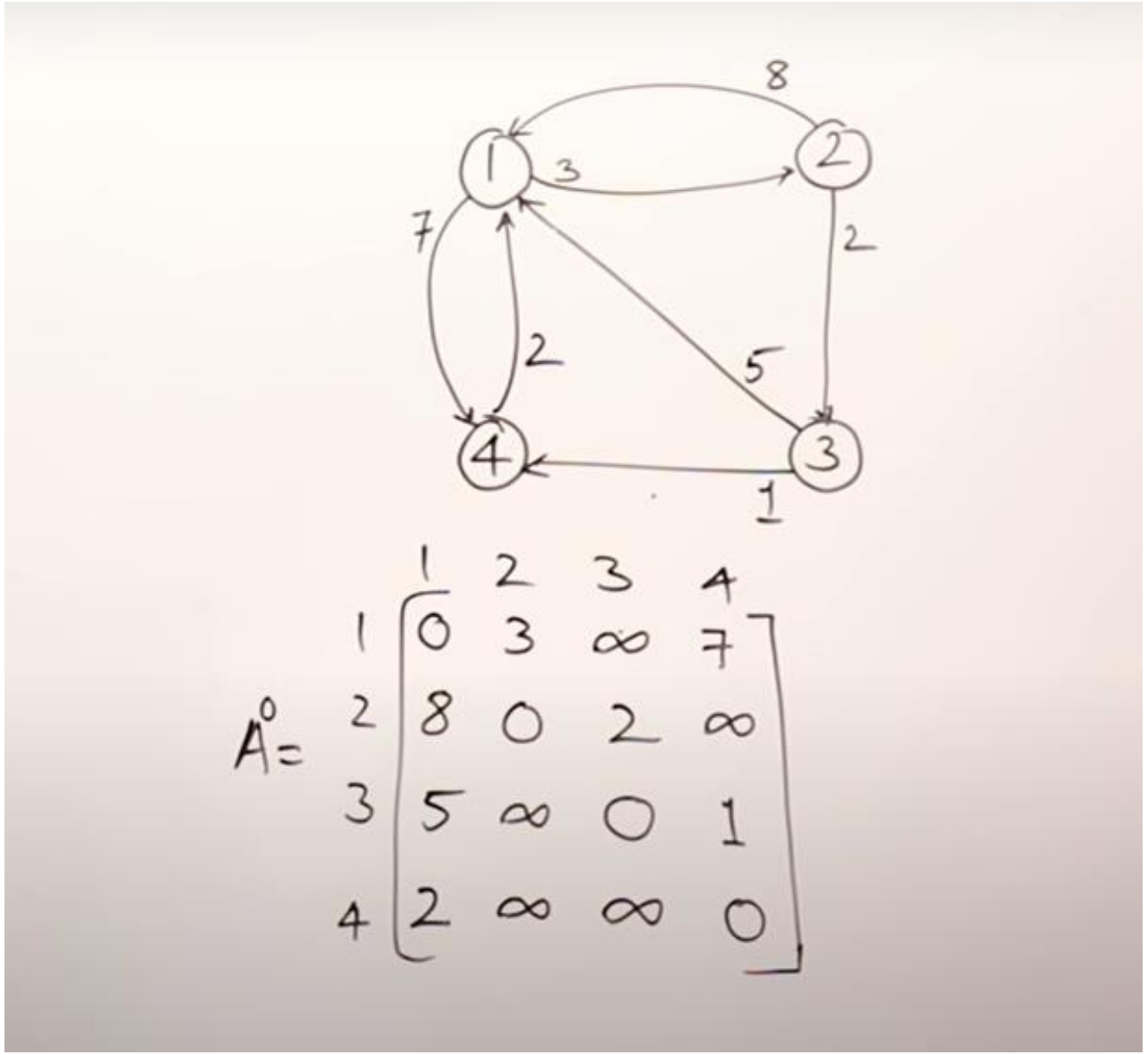
$A^3(3, 1) = \min \{A^2(3, 3) + A^2(3, 1), c(3, 1)\} = \min \{(0 + 3), 3\} = 3$

$A^3(3, 2) = \min \{A^2(3, 3) + A^2(3, 2), c(3, 2)\} = \min \{(0 + 7), 7\} = 7$

All pairs shortest paths

$$A^{(3)} = \begin{bmatrix} 0 & 4 & 6 \\ 5 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$$

All pairs shortest paths



Final Output

- **A4 =**

Vertices	1	2	3	4
1	0	3	5	6
2	5	0	2	3
3	3	6	0	1
4	2	5	7	0

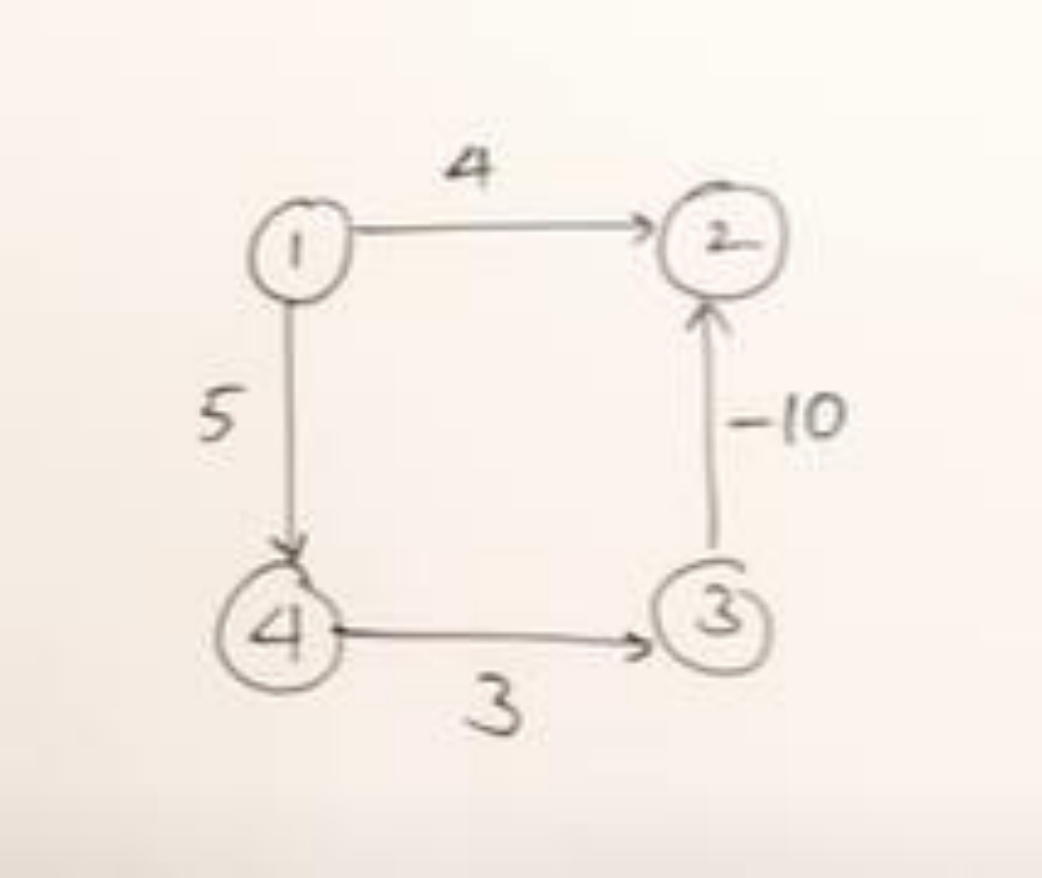
Single Source Shortest Path: General Weights (Bellman Ford Algorithm)

- The ***Single-Source Shortest Path (SSSP)*** algorithm is used to find the shortest paths from a single source vertex to all other vertices in a weighted graph.
- There are several algorithms to solve this problem, and two of the most commonly used ones are ***Dijkstra's algorithm and Bellman-Ford algorithm.***
- Given a graph and a source vertex ***src*** in the graph, find the shortest paths from ***src*** to all vertices in the given graph. The graph may contain negative weight edges.
- Dijkstra doesn’t work for Graphs with negative weights, Bellman-Ford works for such graphs.
- Bellman-Ford is also simpler than Dijkstra and suites well for distributed systems.

- **Initialization:**
  - Create an array ***dist[]*** to store the shortest distances from the source to each vertex.
  - Initialize ***dist[]*** with infinity for all vertices except the source, and set ***dist[source]*** to 0.
- **Relaxation Loop:**
  - Iterate ***|V| - 1*** times, where ***|V|*** is the number of vertices.
  - For each edge ***(u, v)*** in the graph, check if the distance from the source to ***v*** can be shortened by going through ***u***.
  - If ***dist[u] + weight(u, v) < dist[v]***, update ***dist[v]*** with the new, shorter distance.
- **Negative Cycle Check:**
  - After the relaxation loop, check for the presence of negative cycles.
  - If there is a vertex **v** such that ***dist[u] + weight(u, v) < dist[v]***, then the graph contains a negative cycle.
- **Result:**
  - The final distances in the ***dist[]*** array represent the shortest paths from the source to all other vertices, or the presence of a negative cycle is detected.

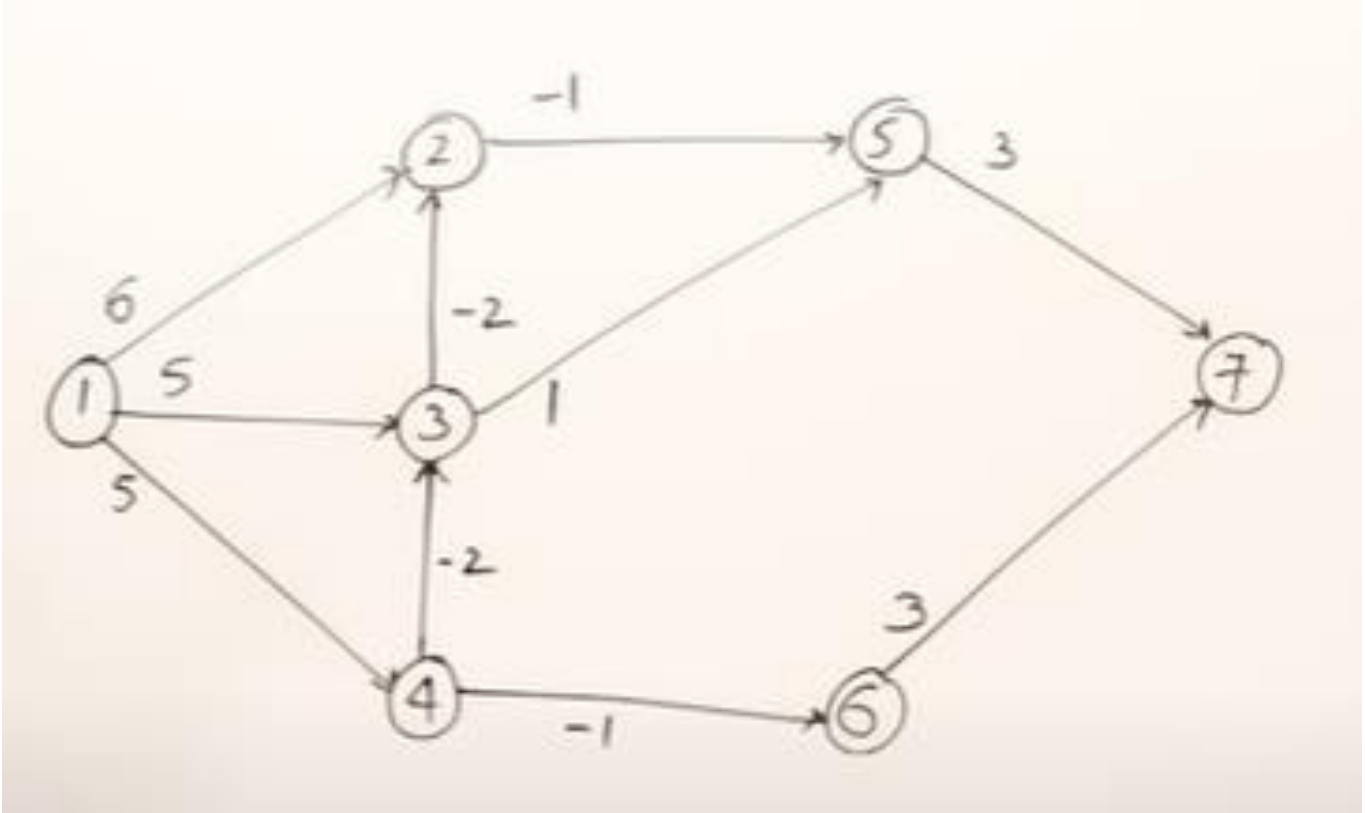
```
function BellmanFord(graph, source):
    create an array dist[] to store the shortest distances
    initialize dist[] with infinity for all vertices except the source
    set distance[source] to 0
    for i from 1 to |V| - 1:
        for each edge (u, v) in graph:
            alt = dist[u] + weight(u, v)
            if alt < dist[v]:
                dist[v] = alt
    for each edge (u, v) in graph:
        if dist[u] + weight(u, v) < dist[v]:
            raise Error("Graph contains a negative cycle")
    return dist
```

Problem 1

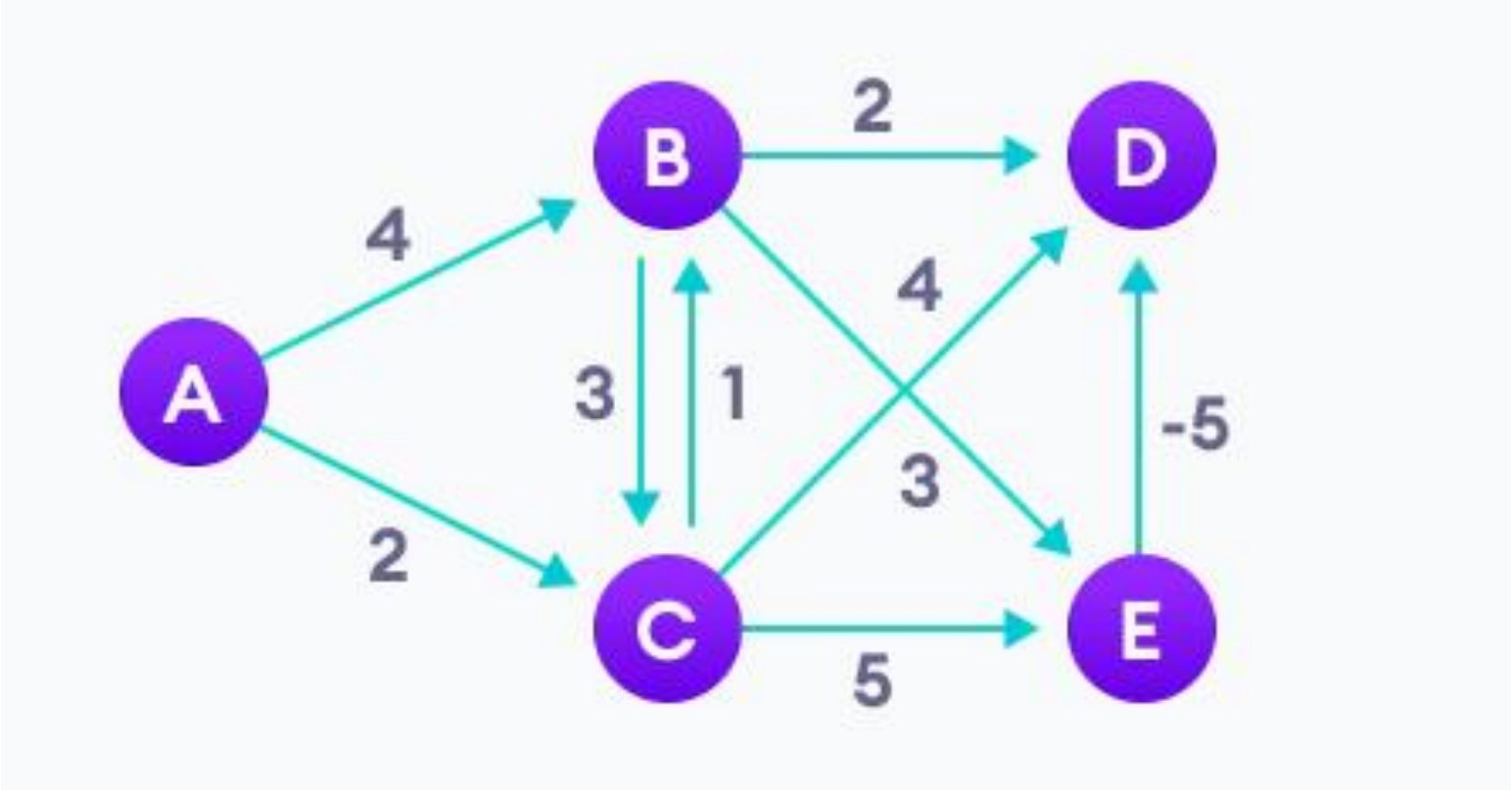


Problem 2

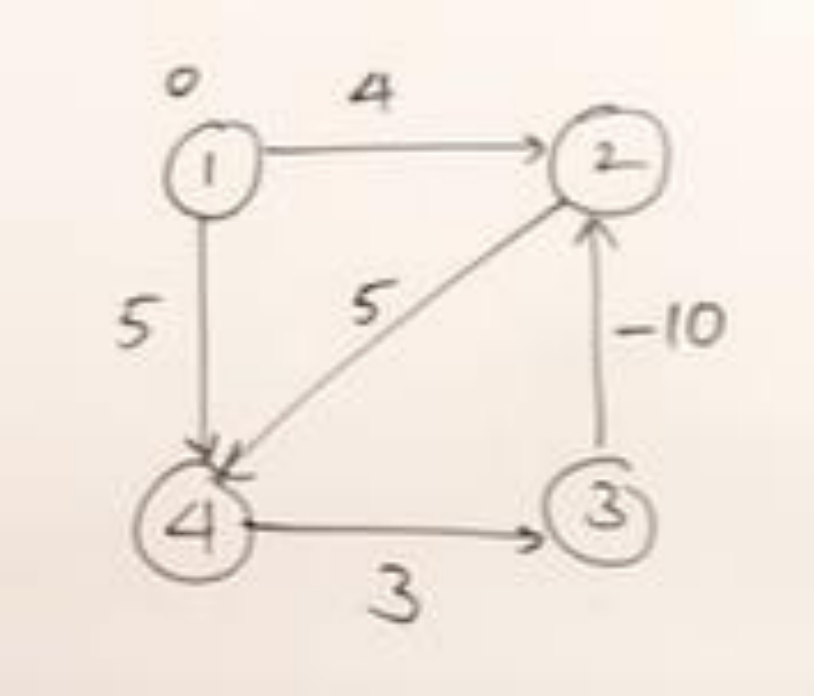
- *Let the given source vertex be 1. Initialize all distances as infinite, except the distance to the source itself. Total number of vertices in the graph is 7, so all edges must be processed 6 times.*

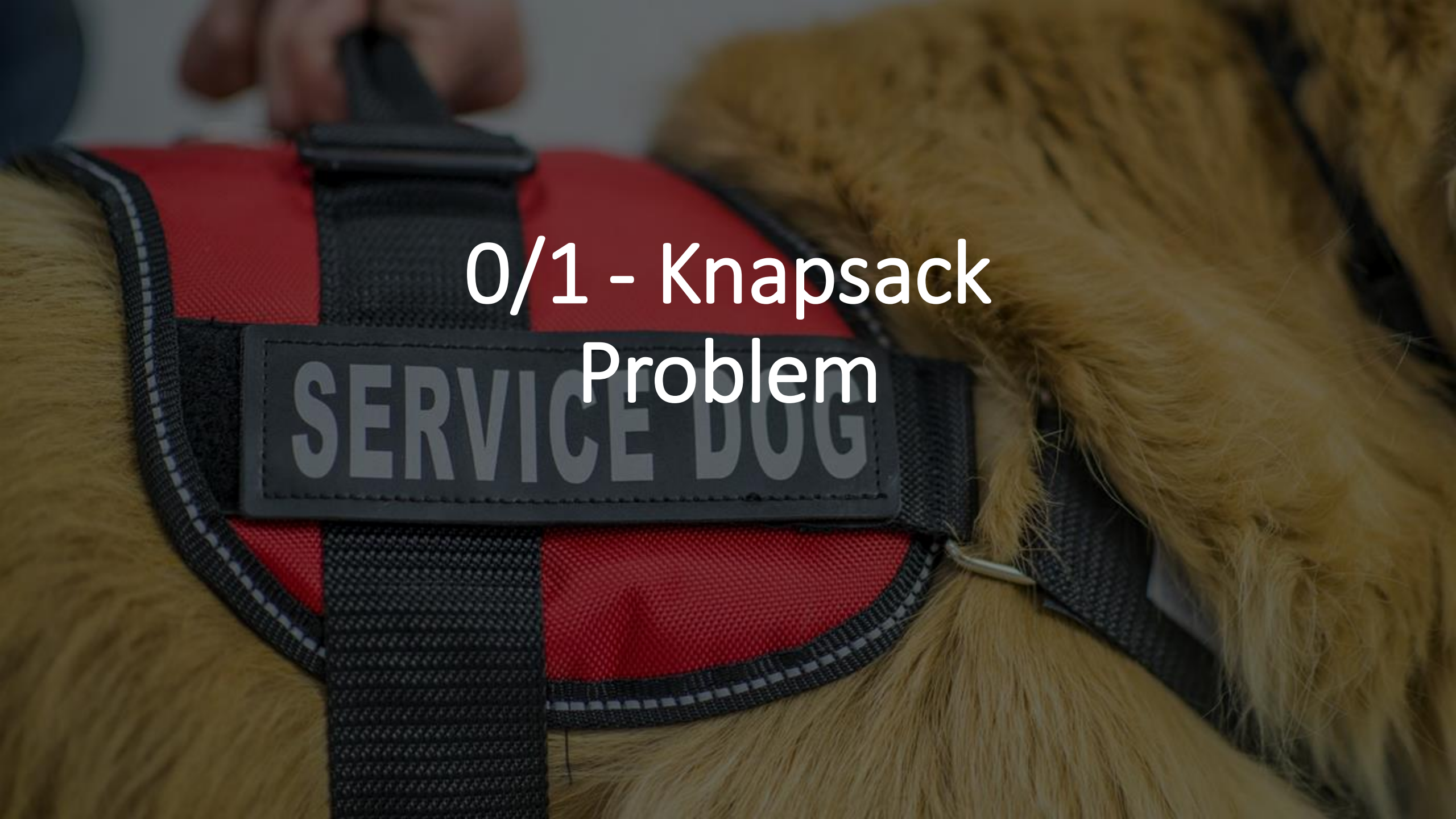


Problem 3



Problem 4





0/1 - Knapsack Problem

0/1 - Knapsack Problem

- **Knapsack Problem** algorithm is a very helpful problem in combinatorics. In the supermarket there are n packages (n ≤ 100) the package i has weight ***W[i] ≤ 100*** and value ***V[i] ≤ 100.*** A thief breaks into the supermarket, the thief cannot carry weight exceeding ***M (M ≤ 10)***.
- The problem to be solved here is: which packages the thief will take away to get the highest value?
- **Input**
  - *Maximum weight M and the number of packages n.*
  - *Array of weight W[i] and corresponding value V[i].*
- **Output:**
  - *Maximize value and corresponding weight in capacity.*
  - *Which packages the thief will take away.*

0/1 - Knapsack Problem

- The 0/1 Knapsack problem using dynamic programming. In this Knapsack algorithm type, each package can be taken or not taken. Besides, the *thief cannot take a fractional amount of a taken package or take a package more than once*. This type can be solved by Dynamic Programming Approach.
- The basic idea of Knapsack dynamic programming is to use a table to store the solutions of solved subproblems.
- When analyzing 0/1 Knapsack problem using Dynamic programming, you can find some noticeable points.
- The value of the knapsack algorithm depends on two factors:
  - ***How many packages are being considered?***
  - ***The remaining weight which the knapsack can store?***

0/1 - Knapsack Problem

- Therefore, you have two variable quantities. With dynamic programming, you have useful information:
  - ***the objective function will depend on two variable quantities***
  - ***the table of options will be a 2-dimensional table.***
- If calling B[i][j] is the maximum possible value by selecting in packages {1, 2, …, i} with weight limit j.
- The maximum value when selected in n packages with the weight limit M is B[n][M]. In other words: When there are i packages to choose, B[i][j] is the optimal weight when the maximum weight of the knapsack is j.
- The optimal weight is always less than or equal to the maximum weight: B[i][j] ≤ j.

0/1 - Knapsack Problem

- For example: B[4][10] = 8. It means that in the optimal case, the total weight of the selected packages is 8, when there are 4 first packages to choose from (1st to 4th package) and the maximum weight of the knapsack is 10. It is not necessary that all 4 items are selected.
- W[i], V[i] are in turn the weight and value of package i, in which i ∈ {1, …, n}.
- M is the maximum weight that the knapsack can carry.
- ***If (j < wt[i]) :***

  `B[i][j] = B[i – 1][j]`
- ***Else :***

  `B[i][j]= max(V[i]+B[i – 1][j – W[i]], B[i – 1][j])`

0/1 - Knapsack Problem

- Let's say we have 4 items and a knapsack of capacity 7.

Weight	1	3	4	5
Value	1	4	5	7

- Hence, m = 7

0/1 - Knapsack Problem

Weight	1	3	4	5
Value	1	4	5	7

V	Wt.	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0	0
1	1	0							
4	3	0							
5	4	0							
7	5	0							

0/1 - Knapsack Problem

V	Wt.		j = 0	1	2	3	4	5	6	7
0	0	i = 0	0	0	0	0	0	0	0	0
1	1	1	0	1						
4	3	2	0							
5	4	3	0							
7	5	4	0							

Here, $B[i][j] = max(V[i]+B[i-1][j-W[i]], B[i-1][j])$

$B[1][1] = max(1+B[0][0], B[0][1])$

$B[1][1] = max(1, 0) = 1$

0/1 - Knapsack Problem

V	Wt.		j = 0	1	2	3	4	5	6	7
0	0	i = 0	0	0	0	0	0	0	0	0
1	1	1	0	1	1	1	1	1	1	1
4	3	2	0							
5	4	3	0							
7	5	4	0							

0/1 - Knapsack Problem

V	Wt.		j = 0	1	2	3	4	5	6	7
0	0	i = 0	0	0	0	0	0	0	0	0
1	1	1	0	1	1	1	1	1	1	1
4	3	2	0	1	1					
5	4	3	0							
7	5	4	0							

0/1 - Knapsack Problem

V	Wt.		j = 0	1	2	3	4	5	6	7
0	0	i = 0	0	0	0	0	0	0	0	0
1	1	1	0	1	1	1	1	1	1	1
4	3	2	0	1	1					
5	4	3	0							
7	5	4	0							

Here, $B[i][j] = max(V[i]+B[i-1][j-W[i]], B[i-1][j])$

$B[2][3] = max(4+B[1][0], B[1][3])$

$B[2][3] = max(4+0, 1) = 4$

0/1 - Knapsack Problem

V	Wt.		j = 0	1	2	3	4	5	6	7
0	0	i = 0	0	0	0	0	0	0	0	0
1	1	1	0	1	1	1	1	1	1	1
4	3	2	0	1	1	4				
5	4	3	0							
7	5	4	0							

Here, $B[i][j] = max(V[i]+B[i-1][j-W[i]], B[i-1][j])$

$B[2][3] = max(4+B[1][0], B[1][3])$

$B[2][3] = max(4+0, 1) = 4$

0/1 - Knapsack Problem

V	Wt.		j = 0	1	2	3	4	5	6	7
0	0	i = 0	0	0	0	0	0	0	0	0
1	1	1	0	1	1	1	1	1	1	1
4	3	2	0	1	1	4	5			
5	4	3	0							
7	5	4	0							

Here, $B[i][j] = max(V[i]+B[i-1][j-W[i]], B[i-1][j])$

$B[2][4] = max(4+B[1][1], B[1][4])$

$B[2][3] = max(4+1, 1) = 5$

0/1 - Knapsack Problem

V	Wt.		j = 0	1	2	3	4	5	6	7
0	0	i = 0	0	0	0	0	0	0	0	0
1	1	1	0	1	1	1	1	1	1	1
4	3	2	0	1	1	4	5	5	5	5
5	4	3	0							
7	5	4	0							

0/1 - Knapsack Problem

V	Wt.		j = 0	1	2	3	4	5	6	7
0	0	i = 0	0	0	0	0	0	0	0	0
1	1	1	0	1	1	1	1	1	1	1
4	3	2	0	1	1	4	5	5	5	5
5	4	3	0	1	1	4	5			
7	5	4	0							

Here, $B[i][j] = max(V[i]+B[i-1][j-W[i]], B[i-1][j])$

$B[4][4] = max(5+B[2][0], B[2][4])$

$B[2][3] = max(5+0, 4) = 5$

0/1 - Knapsack Problem

V	Wt.		j = 0	1	2	3	4	5	6	7
0	0	i = 0	0	0	0	0	0	0	0	0
1	1	1	0	1	1	1	1	1	1	1
4	3	2	0	1	1	4	4	4	4	4
5	4	3	0	1	1	4	5	6	6	9
7	5	4	0	1	1	4	5	7	8	9

0/1 - Knapsack Problem

V	Wt.		j = 0	1	2	3	4	5	6	7
0	0	i = 0	0	0	0	0	0	0	0	0
1	1	1	0	1	1	1	1	1	1	1
4	3	2	0	1	1	4	4	4	4	4
5	4	3	0	1	1	4	5	6	6	9
7	5	4	0	1	1	4	5	7	8	9

- 9 is the highest value we can get.
- It's coming from ***3rd row to 4th*** row.
- So, we select the 3rd row with weight 4 and value 5.
- Now, the weight of 4kg is full, so we take 2nd row and 3rd column. It has value of 4 and coming from same row.
- So, we take second column with weight 3 and value 4.

0/1 - Knapsack Problem

- Let's say we have 4 items and a knapsack of capacity 8.

Weight	2	3	4	5
Value	1	2	5	6

- Solve given problem.

0/1 - Knapsack Problem

Weight	2	3	4	5
Value	1	2	5	6

Value	Wt		J = 0	1	2	3	4	5	6	7	8
0	0	i = 0	0	0	0	0	0	0	0	0	0
1	2	1	0	0	1	1	1	1	1	1	1
2	3	2	0	0	1	2	2	3	3	3	3
5	4	3	0	0	1	2	5	5	6	7	7
6	5	4	0	0	1	2	5	6	6	7	8

String Editing Problem

- The String Editing Problem, also known as the ***Edit Distance or Levenshtein*** Distance problem, involves determining the minimum number of operations required to transform one string into another.
- The allowed operations typically include ***insertion, deletion, and substitution of characters.***
- The goal is to find the minimum cost or number of operations needed to make the two strings identical.
- **Example:**
  - Convert ***azced to abcdef***
  - Let's create a matrix to solve the String Editing Problem (Edit Distance problem) using dynamic programming.

String Editing Problem

	Null	a	b	c	d	e	f
Null	0	1	2	3	4	5	6
a	1						
z	2						
c	3						
e	4						
d	5						

String Editing Problem

	Null	a	b	c	d	e	f
Null	**0**	**1**	**2**	**3**	**4**	**5**	**6**
a	**1**	*0*	*1*	*2*	*3*	*4*	*5*
z	**2**	*1*	*1*	*2*	*3*	*4*	*5*
c	**3**	*2*	*2*	*1*	*2*	*3*	*4*
e	**4**	*3*	*3*	*2*	*2*	*2*	*3*
d	**5**	*4*	*4*	*3*	*2*	*3*	*3*

String Editing Problem

	Null	a	b	c	d	e	f
Null	**0**	**1**	**2**	**3**	**4**	**5**	**6**
a	**1**	***0***	*1*	*2*	*3*	*4*	*5*
z	**2**	*1*	***1***	*2*	*3*	*4*	*5*
c	**3**	*2*	*2*	***1***	***2***	*3*	*4*
e	**4**	*3*	*3*	*2*	*2*	***2***	*3*
d	**5**	*4*	*4*	*3*	*2*	*3*	***3***

Reliability Design

- The problem is to design a system that is composed of several devices connected in series.
- Let r_i be the reliability of device D_i (that is r_i is the probability that device i will function properly) then the reliability of the entire system is πr_i.
- Even if the individual devices are very reliable, the reliability of the system may not be very good.
- For example, if n=10 and $r_i = 0.99$, $1 \leq i \leq 10$, then $\pi r_i = 0.904$.
- Hence, it is desirable to duplicate devices. Multiple copies of the same device type are connected in parallel.

If stage i contains mi copies of device D_i. Then the probability that all m_i have a malfunction is $(1 - r_i)^{m_i}$. Hence the reliability of stage i becomes $1 – (1 - r_i)^{m_i}$.

The reliability of stage 'i' is given by a function $\phi_i (m_i)$.

Our problem is to use device duplication. This maximization is to be carried out under a cost constraint. Let c_i be the cost of each unit of device i and let c be the maximum allowable cost of the system being designed.

We wish to solve:

Maximize $\prod_{1 \leq i \leq n} \phi_i (m_i)$

Subject to $\sum_{1 \leq i \leq n} C_i m_i < C$

$m_i \geq 1$ and interger, $1 \leq i \leq n$

Reliability Design

Assume each $C_i > 0$, each m_i must be in the range $1 \leq m_i \leq u_i$, where

$$u_i = \left\lfloor \left(C + C_i - \sum_{1}^{n} C_j \right) \Big/ C_i \right\rfloor$$

- An optimal solution m1, m2 ……. mn is the result of a sequence of decisions, one decision for each mi.
- The dominance rule (f1, x1) dominate (f2, x2) if f1 ≥ f2 and x1 ≤ x2. Hence, dominated tuples can be discarded from S^i

Example 1:

Design a three stage system with device types D_1, D_2 and D_3. The costs are \$30, \$15 and \$20 respectively. The Cost of the system is to be no more than \$105. The reliability of each device is 0.9, 0.8 and 0.5 respectively.

Solution:

We assume that if if stage I has mi devices of type i in parallel, then $\phi_i(m_i) = 1 - (1-r_i)^{mi}$

Since, we can assume each $c_i > 0$, each m_i must be in the range $1 \leq m_i \leq u_i$. Where:

$$u_i = \left\lfloor \left(C + C_i - \sum_{1}^{n} C_j \right) \Big/ C_i \right\rfloor$$

Using the above equation compute u_1, u_2 and u_3.

$$u_1 = \frac{105 + 30 - (30 + 15 + 20)}{30} = \frac{70}{30} = 2$$

$$u_2 = \frac{105 + 15 - (30 + 15 + 20)}{15} = \frac{55}{15} = 3$$

$$u_3 = \frac{105 + 20 - (30 + 15 + 20)}{20} = \frac{60}{20} = 3$$

D_i	C_i	r_i	u_i
D1	**30**	**0.9**	**2**
D2	**15**	**0.8**	**3**
D3	**20**	**0.5**	**3**

We use $S_j^i \rightarrow i$: *stage number and* J: *no. of devices in stage* $i = m_i$

$S^o = \{f_o(x), x\}$ *initially* $f_o(x) = 1$ *and* $x = 0$, *so,* $S^o = \{1, 0\}$

Compute S^1, S^2 and S^3 as follows:

S^1 = depends on u_1 value, as $u_1 = 2$, so

$$S^1 = \{S_1^1, S_2^1\}$$

S^2 = depends on u_2 value, as $u_2 = 3$, so

$$S^2 = \left\{S_1^2, S_2^2, S_3^2\right\}$$

S^3 = depends on u_3 value, as $u_3 = 3$, so

$$S^3 = \left\{S_1^3, S_2^3, S_3^3\right\}$$

Now find, $S_1^1 = \left\{\left(f_1(x), x\right)\right\}$

$f_1(x) = \{\phi_1(1) f_o(), \phi_1(2) f_0()\}$ With devices $m_1 = 1$ and $m_2 = 2$

Compute $\phi_1(1)$ and $\phi_1(2)$ using the formula: $\phi_i\ (mi)) = 1 - (1 - r_i)^{mi}$

$$\phi_1(1) = 1 - (1 - r_1)^{m\,1} = 1 - (1 - 0.9)^1 = 0.9$$

$$\phi_1(2) = 1 - (1 - 0.9)^2 = 0.99$$

$$S_{1\ 1} = \{f_1(x),\ x\} = = (0.9, 30)$$

$$S_2^1 = \{0.99\ ,\ 30 + 30\} = (0.99, 60)$$

Therefore, $S^1 = \{(0.9, 30), (0.99, 60)\}$

Next find $S_1^2 = \{(f_2(x), x)\}$

$$f_2(x) = \{\phi_2(1) * f_1(\ ), \phi_2(2) * f_1(\ ), \phi_2(3) * f_1(\ )\}$$

$$\phi_2(1) = 1 - (1 - r_{I_{mi}}) = 1 - (1 - 0.8) = 1 - 0.2 = 0.8$$

$$\phi_2(2) = 1 - (1 - 0.8)^2 = 0.96$$

$$\phi_2(3) = 1 - (1 - 0.8)^3 = 0.992$$

$$S_1^2 = \{(0.8(0.9),30+15), (0.8(0.99),60+15)\} = \{(0.72, 45), (0.792, 75)\}$$

$$S_2^2 = \{(0.96(0.9),30+15+15), (0.96(0.99),60+15+15)\}$$
$$= \{(0.864, 60), (0.9504, 90)\}$$

$$S_3^2 = \{(0.992(0.9),30+15+15+15), (0.992(0.99),60+15+15+15)\}$$
$$= \{(0.8928, 75), (0.98208, 105)\}$$

$$S^2 = \left\{S_1^2, S_2^2, S_3^2\right\}$$

By applying Dominance rule to S^2:

Therefore, $S^2 = \{(0.72, 45), (0.864, 60), (0.8928, 75)\}$

$S_2 = \{(0.72, 45), (0.864, 60), (0.8928, 75)\}$

$$\phi_3(1) = 1 - (1 - r_I)^{mi} = 1 - (1 - 0.5)^1 = 1 - 0.5 = 0.5$$

$$\phi_3(2) = 1 - (1 - 0.5)^2 = 0.75$$

$$\phi_3(3) = 1 - (1 - 0.5)^3 = 0.875$$

$$S_1^3 = \{(0.5\,(0.72), 45 + 20), (0.5\,(0.864), 60 + 20), (0.5\,(0.8928), 75 + 20)\}$$

$$S_1^3 = \{(0.36, 65), (0.437, 80), (0.4464, 95)\}$$

$$S_2^3 = \{(0.75\,(0.72), 45 + 20 + 20), (0.75\,(0.864), 60 + 20 + 20),$$
$$(0.75\,(0.8928), 75 + 20 + 20)\}$$

$$= \{(0.54, 85), (0.648, 100), (0.6696, 115)\}$$

$$S_3^3 = \{ (0.875\,(0.72), 45 + 20 + 20 + 20), (0.875\,(0.864), 60 + 20 + 20 + 20),$$
$$(0.875\,(0.8928), 75 + 20 + 20 + 20) \}$$

$$S_3^3 = \{(0.63, 105), (1.756, 120), (0.7812, 135)\}$$

If cost exceeds 105, remove that tuples

$S^3 = \{(0.36, 65), (0.437, 80), (0.54, 85), (0.648, 100)\}$

The best design has a reliability of 0.648 and a cost of 100. Tracing back for the solution through S^i's we can determine that $m_3 = 2$, $m_2 = 2$ and $m_1 = 1$.

TSP

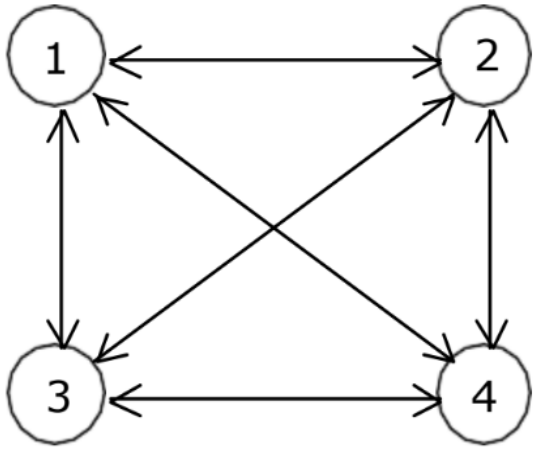
Let G = (V, E) be a directed graph with edge costs C_{ij}. The variable c_{ij} is defined such that $c_{ij} > 0$ for all I and j and $c_{ij} = \alpha$ if $< i, j> \notin E$. Let $|V| = n$ and assume $n > 1$. A tour of G is a directed simple cycle that includes every vertex in V. The cost of a tour is the sum of the cost of the edges on the tour. The traveling sales person problem is to find a tour of minimum cost. The tour is to be a simple path that starts and ends at vertex 1.

Generalizing equation 1, we obtain (for i $\notin$ S)

$$g(i, S) = \min_{j \in S} \{ c_{ij} + g(i, S - \{ j \}) \} \quad \text{-- 2}$$

The Equation can be solved for g (1, V – 1}) if we know g (k, V – {1, k}) for all choices of k.

For the following graph find minimum cost tour for the traveling salesperson problem:



The cost adjacency matrix = $\begin{bmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{bmatrix}$

Let us start the tour from vertex 1:

$$g(1, V - \{1\}) = \min_{2 \le k \le n} \{c_{1k} + g(k, V - \{1, K\})\} \quad - \quad (1)$$

More generally writing:

$$g(i, s) = \min \{c_{ij} + g(J, s - \{J\})\}$$

Clearly, $g(i, \Phi) = c_{i1}$, $1 \leq i \leq n$. So,

$$g(2, \Phi) = C_{21} = 5$$

$$g(3, \Phi) = C_{31} = 6$$

$$g(4, \Phi) = C_{41} = 8$$

$$g(1, \{2, 3, 4\}) = \min \{c_{12} + g(2, \{3, 4\}, c_{13} + g(3, \{2, 4\}), c_{14} + g(4, \{2, 3\})\}$$

$$g(2, \{3, 4\}) = \min \{c_{23} + g(3, \{4\}), c_{24} + g(4, \{3\})\}$$
$$= \min \{9 + g(3, \{4\}), 10 + g(4, \{3\})\}$$

$$g(3, \{4\}) = \min \{c_{34} + g(4, \Phi)\} = 12 + 8 = 20$$

$$g(4, \{3\}) = \min \{c_{43} + g(3, \Phi)\} = 9 + 6 = 15$$

Therefore, $g(2, \{3, 4\}) = \min\{9 + 20, 10 + 15\} = \min\{29, 25\} = 25$

$g(3, \{2, 4\}) = \min\{(c_{32} + g(2, \{4\}), (c_{34} + g(4, \{2\})\}$

$g(2, \{4\}) = \min\{c_{24} + g(4, \Phi)\} = 10 + 8 = 18$

$g(4, \{2\}) = \min\{c_{42} + g(2, \Phi)\} = 8 + 5 = 13$

Therefore, $g(3, \{2, 4\}) = \min\{13 + 18, 12 + 13\} = \min\{41, 25\} = 25$

Therefore, g (3, {2, 4}) = min {13 + 18, 12 + 13} = min {41, 25} = 25

g (4, {2, 3}) = min {c_{42} + g (2, {3}), c_{43} + g (3, {2})}

g (2, {3}) = min {c_{23} + g (3, Φ} = 9 + 6 = 15

g (3, {2}) = min {c_{32} + g (2, Φ} = 13 + 5 = 18

Therefore, g (4, {2, 3}) = min {8 + 15, 9 + 18} = min {23, 27} = 23

g (1, {2, 3, 4}) = min {c_{12} + g (2, {3, 4}), c_{13} + g (3, {2, 4}), c_{14} + g (4, {2, 3})}
= min {10 + 25, 15 + 25, 20 + 23} = min {35, 40, 43} = 35

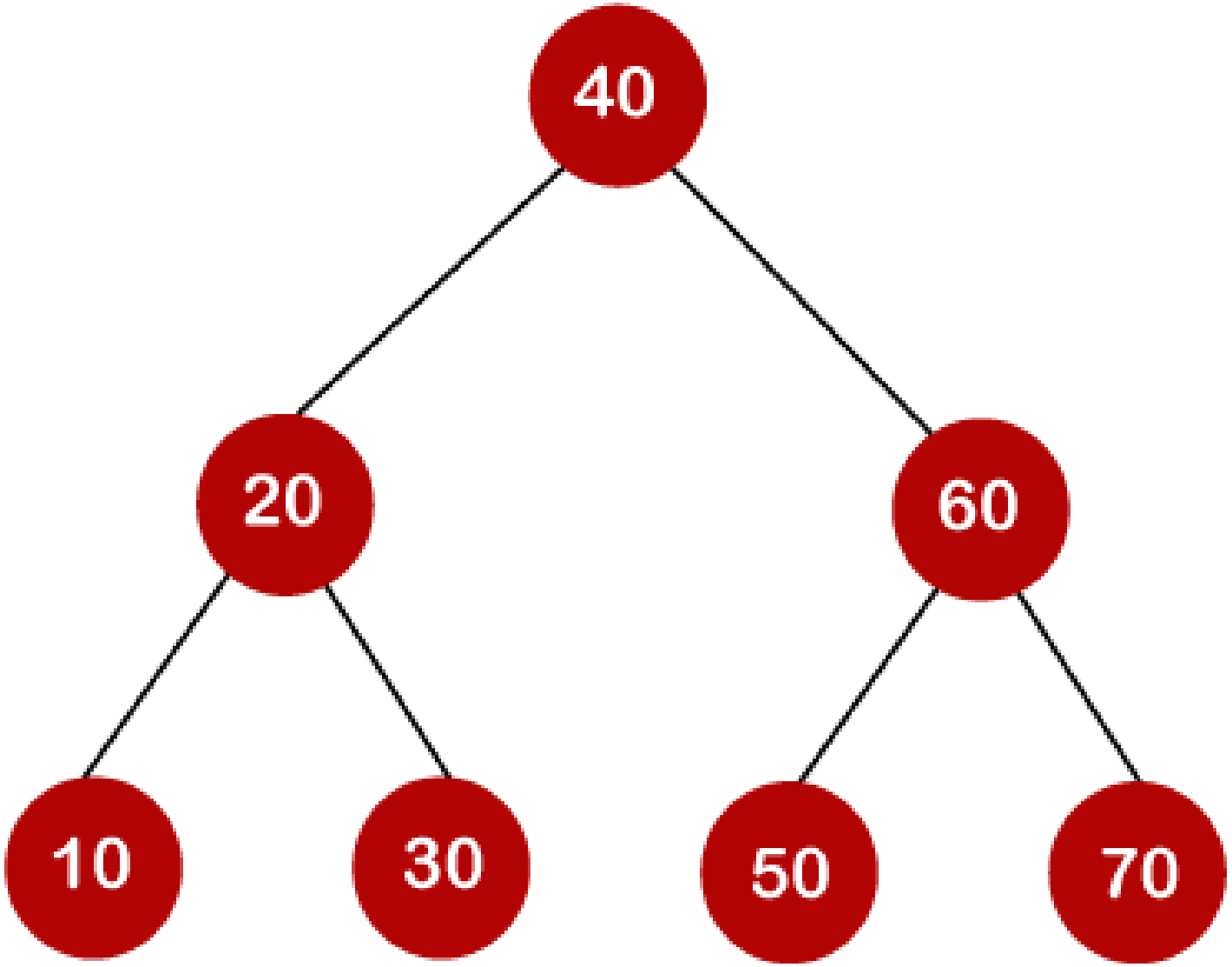
The optimal tour for the graph has length = 35

The optimal tour is: 1, 2, 4, 3, 1.

Optimal Binary Search Tree

- As we know that in binary search tree, the nodes in the left subtree have lesser value than the root node and the nodes in the right subtree have greater value than the root node.
- We know the key values of each node in the tree, and we also know the frequencies of each node in terms of searching means how much time is required to search a node.
- The frequency and key-value determine the overall cost of searching a node. The cost of searching is a very important factor in various applications.
- The overall cost of searching a node should be less. The time required to search a node in BST is more than the balanced binary search tree as a balanced binary search tree contains a lesser number of levels than the BST.
- There is one way that can reduce the cost of a binary search tree is known as an **optimal binary search tree**.

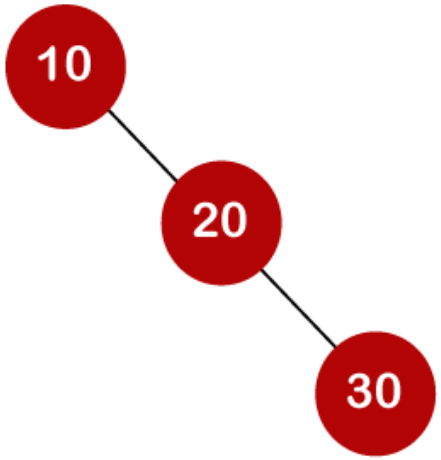
Optimal Binary Search Tree

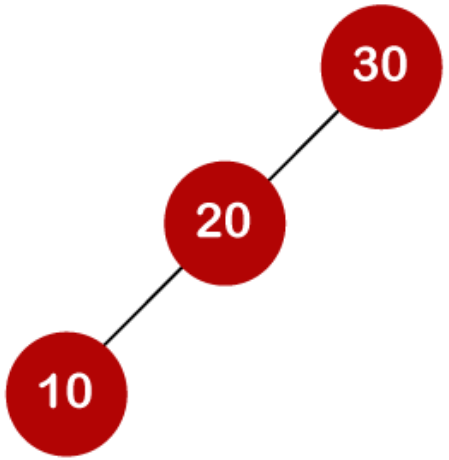


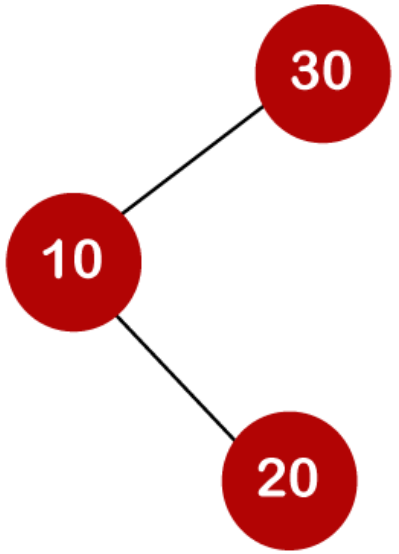
Optimal Binary Search Tree

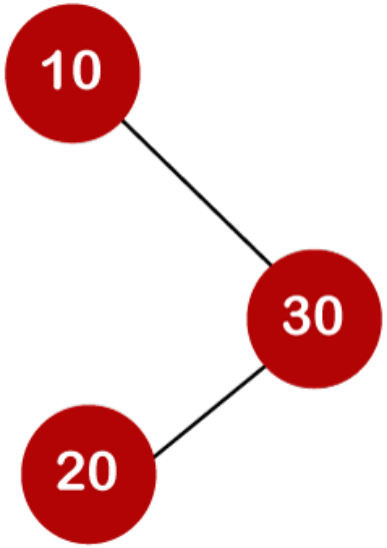
- In the above tree, all the nodes on the left subtree are smaller than the value of the root node, and all the nodes on the right subtree are larger than the value of the root node.
- The maximum time required to search a node is equal to the minimum height of the tree, equal to log(n).
- Now we will see how many binary search trees can be made from the given number of keys.
- For example: 10, 20, 30 are the keys, and the following are the binary search trees that can be made out from these keys.

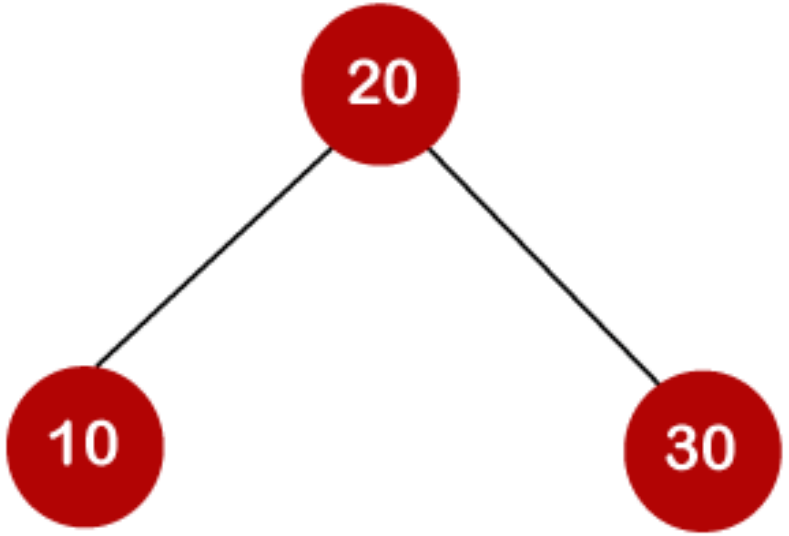
$$\frac{{}^{2n}C_n}{n+1}$$











- When we use the above formula, then it is found that total 5 number of trees can be created.
- The cost required for searching an element depends on the comparisons to be made to search an element. Now, we will calculate the average cost of time of the above binary search trees.

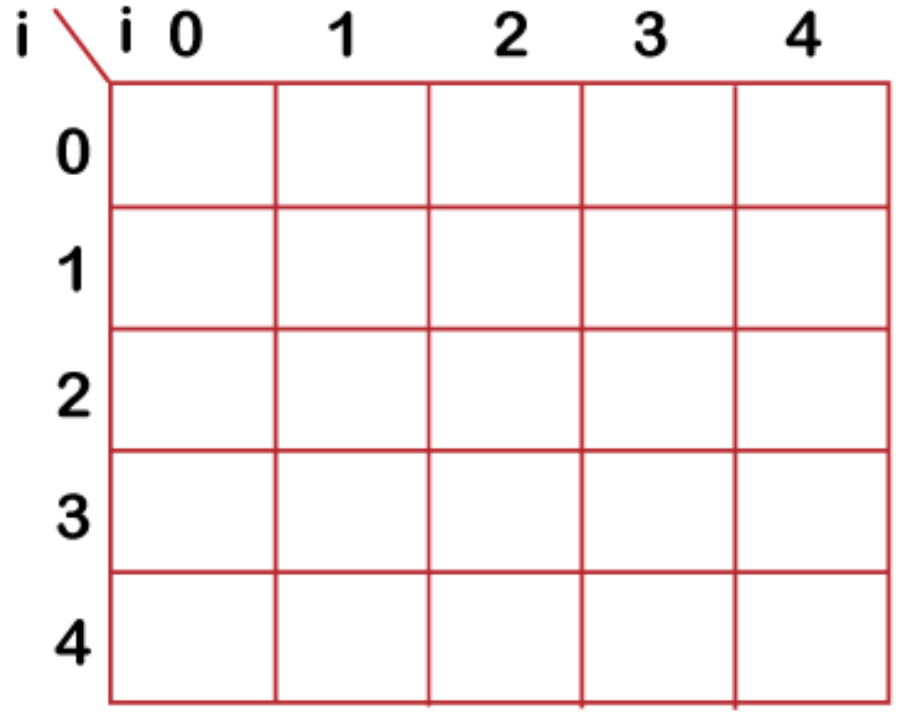
- Till now, we read about the height-balanced binary search tree.
- To find the optimal binary search tree, we will determine the frequency of searching a key.
- Let's assume that frequencies associated with the keys 10, 20, 30 are 3, 2, 5.

Keys	10	20	30
Frequency	3	2	5

- The above trees have different frequencies. The tree with the lowest frequency would be considered the optimal binary search tree. The tree with the frequency 17 is the lowest, so it would be considered as the optimal binary search tree.

Dynamic Approach Optimal Binary Search Trees

	1	2	3	4
Keys →	10	20	30	40
Frequency →	4	2	6	3



General formula for calculating the minimum cost is:

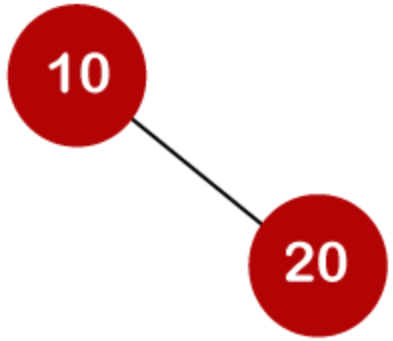
C[i,j] = min (I<k<=j){c[i, k-1] + c[k,j]} + w(i,j)

- **First, we will calculate the values where j-i is equal to zero.**
- When i=0, j=0, then j-i = 0
- When i = 1, j=1, then j-i = 0
- When i = 2, j=2, then j-i = 0
- When i = 3, j=3, then j-i = 0
- When i = 4, j=4, then j-i = 0
- Therefore, c[0, 0] = 0, c[1 , 1] = 0, c[2,2] = 0, c[3,3] = 0, c[4,4] = 0
- **Now we will calculate the values where j-i equal to 1.**
- When j=1, i=0 then j-i = 1
- When j=2, i=1 then j-i = 1
- When j=3, i=2 then j-i = 1
- When j=4, i=3 then j-i = 1

	0	1	2	3	4
0	0	4			
1		0	2		
2			0	6	
3				0	3
4					0

- Now to calculate the cost, we will consider only the jth value.
- The cost of c[0,1] is 4 (The key is 10, and the cost corresponding to key 10 is 4).
- The cost of c[1,2] is 2 (The key is 20, and the cost corresponding to key 20 is 2).
- The cost of c[2,3] is 6 (The key is 30, and the cost corresponding to key 30 is 6)
- The cost of c[3,4] is 3 (The key is 40, and the cost corresponding to key 40 is 3)

- **Now we will calculate the values where j-i = 2**
- When j=2, i=0 then j-i = 2
- When j=3, i=1 then j-i = 2
- When j=4, i=2 then j-i = 2
- In this case, we will consider two keys.
- When i=0 and j=2, then keys 10 and 20. There are two possible trees that can be made out from these two keys shown below:



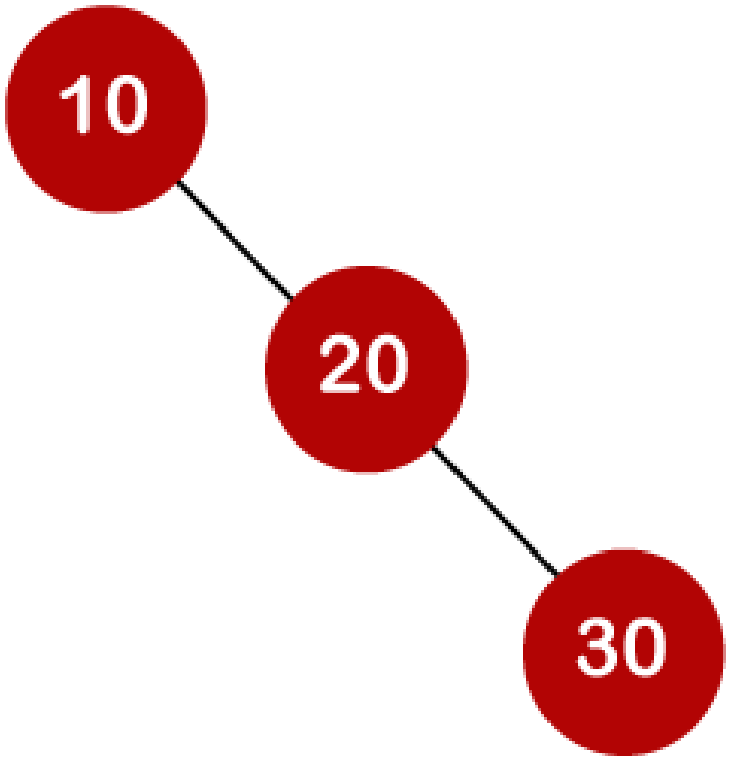
- In the first binary tree, cost would be: 4*1 + 2*2 = 8
- In the second binary tree, cost would be: 4*2 + 2*1 = 10
- The minimum cost is 8; therefore, c[0,2] = 8

	0	1	2	3	4
0	0	4	8		
1		0	2		
2			0	6	
3				0	3
4					0

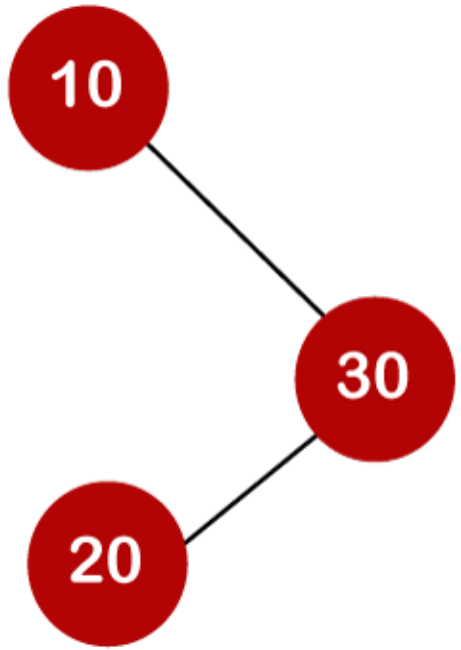
- When i=1 and j=3, then keys 20 and 30. There are two possible trees that can be made out from these two keys shown below:
  - In the first binary tree, cost would be: 1*2 + 2*6 = 14
  - In the second binary tree, cost would be: 1*6 + 2*2 = 10
  - The minimum cost is 10; therefore, c[1,3] = 10
- When i=2 and j=4, we will consider the keys at 3 and 4, i.e., 30 and 40. There are two possible trees that can be made out from these two keys shown as below:
  - In the first binary tree, cost would be: 1*6 + 2*3 = 12
  - In the second binary tree, cost would be: 1*3 + 2*6 = 15
  - The minimum cost is 12, therefore, c[2,4] = 12

i \ 1	0	1	2	3	4
0	0	4	8^{1}		
1		0	2	10^{3}	
2			0	6	12^{3}
3				0	3
4					0

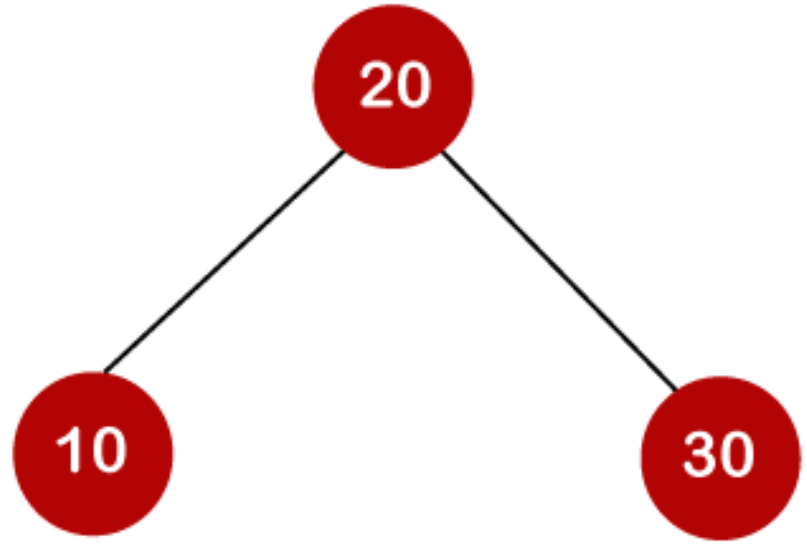
- Now we will calculate the values when j-i = 3
- When j=3, i=0 then j-i = 3
- When j=4, i=1 then j-i = 3
- When i=0, j=3 then we will consider three keys, i.e., 10, 20, and 30.
- The following are the trees that can be made if 10 is considered as a root node.
- In the above tree, 10 is the root node, 20 is the right child of node 10, and 30 is the right child of node 20.
- Cost would be: 1*4 + 2*2 + 3*6 = 26



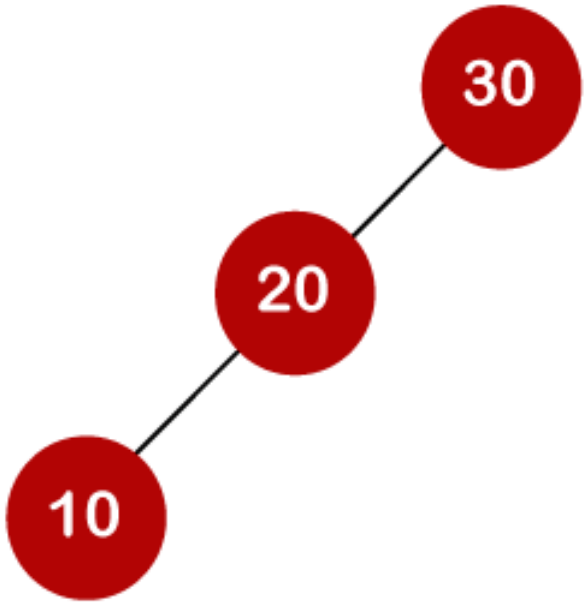
- In the above tree, 10 is the root node, 30 is the right child of node 10, and 20 is the left child of node 20.
- Cost would be: 1*4 + 2*6 + 3*2 = 22



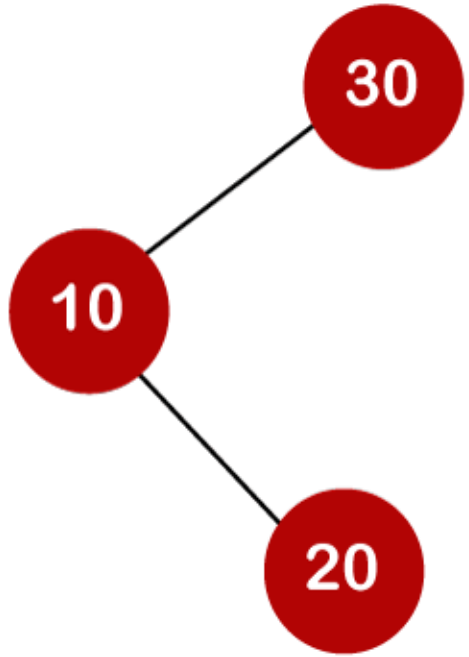
- In the above tree, 20 is the root node, 30 is the right child of node 20, and 10 is the left child of node 20.
- Cost would be: 1*2 + 4*2 + 6*2 = 22



- In the above tree, 30 is the root node, 20 is the left child of node 30, and 10 is the left child of node 20.
- Cost would be: 1*6 + 2*2 + 3*4 = 22



- In the above tree, 30 is the root node, 10 is the left child of node 30 and 20 is the right child of node 10.
- Cost would be: 1*6 + 2*4 + 3*2 = 20
- Therefore, the minimum cost is 20 which is the 3rd root. So, c[0,3] is equal to 20.



- When i=1 and j=4 then we will consider the keys 20, 30, 40
- c[1,4] = min{ c[1,1] + c[2,4], c[1,2] + c[3,4], c[1,3] + c[4,4] } + 11

  = min{0+12, 2+3, 10+0}+ 11

  = min{12, 5, 10} + 11
- The minimum value is 5; therefore, c[1,4] = 5+11 = 16

i \ 1	0	1	2	3	4
0	0	4	8^1	20^3	
1		0	2	10^3	16^3
2			0	6	12^3
3				0	3
4					0

- **Now we will calculate the values when j-i = 4**
- When j=4 and i=0 then j-i = 4
- In this case, we will consider four keys, i.e., 10, 20, 30 and 40. The frequencies of 10, 20, 30 and 40 are 4, 2, 6 and 3 respectively.
- w[0, 4] = 4 + 2 + 6 + 3 = 15
- If we consider 10 as the root node then

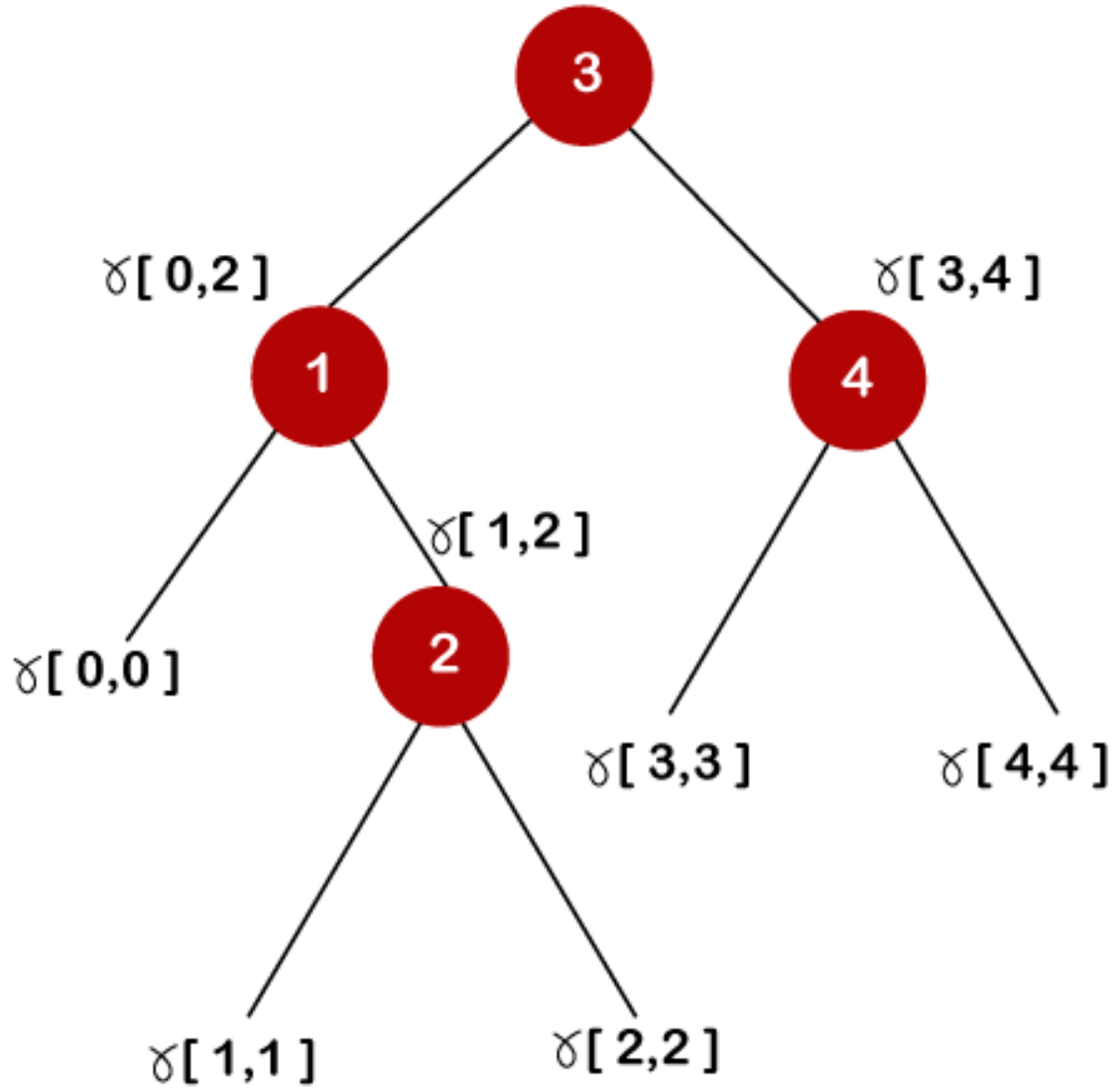
  $C[0, 4] = min \{c[0,0] + c[1,4]\}+ w[0,4] = min \{0 + 16\} + 15= 31$
- If we consider 20 as the root node then

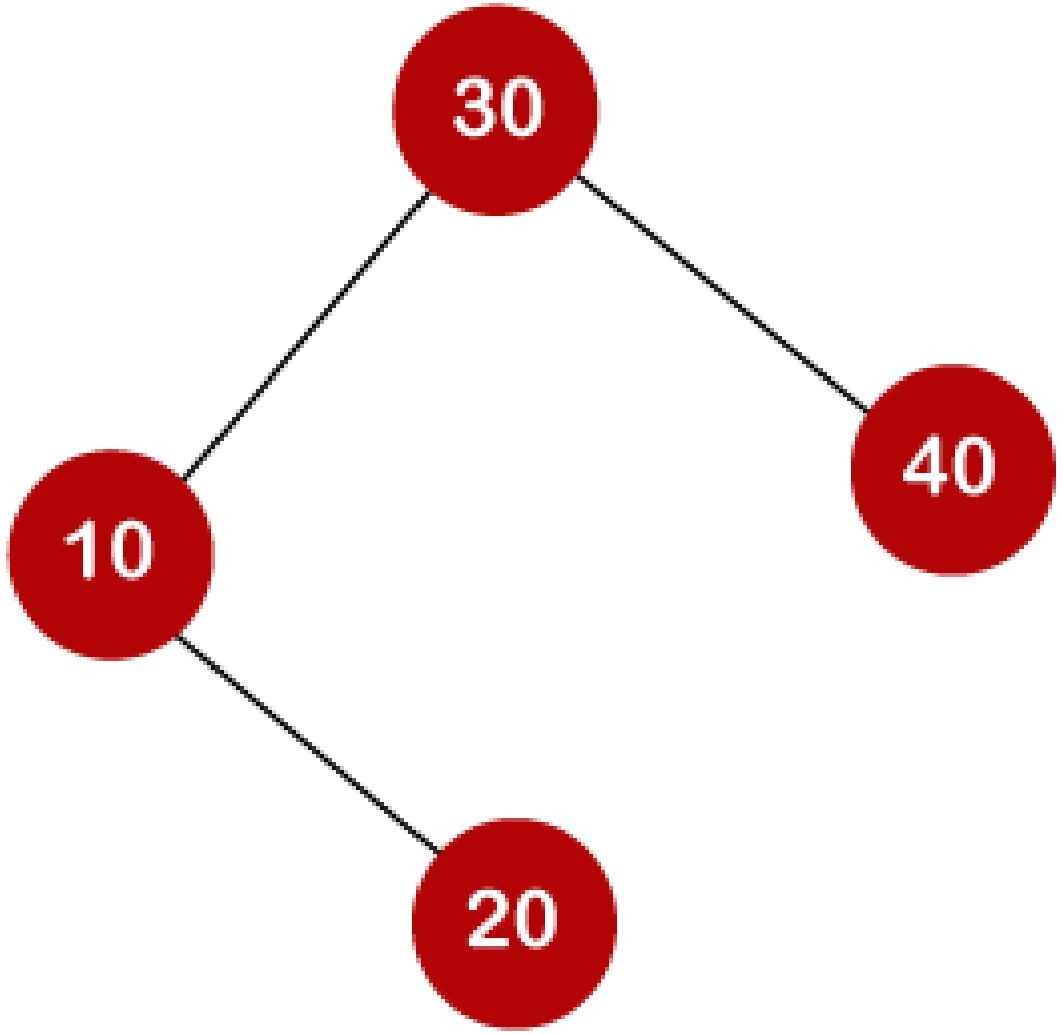
  $C[0,4] = min\{c[0,1] + c[2,4]\} + w[0,4] = min\{4 + 12\} + 15 = 16 + 15 = 31$
- If we consider 30 as the root node then,

  $C[0,4] = min\{c[0,2] + c[3,4]\} +w[0,4] = min \{8 + 3\} + 15 = 26$
- If we consider 40 as the root node then,

  $C[0,4] = min\{c[0,3] + c[4,4]\} + w[0,4] = min\{20 + 0\} + 15 = 35$

i \ 1	0	1	2	3	4
0	0	4	8^{1}	20^{3}	26^{3}
1		0	2	10^{3}	16^{3}
2			0	6	12^{3}
3				0	3
4					0


3
δ[0,2]
δ[3,4]
1
4
δ[1,2]
δ[0,0]
2
δ[3,3]
δ[4,4]
δ[1,1]
δ[2,2]


30
10
40
20