

Computer Engineering - Sem IV

NCMPC 41 : Design and Analysis of Algorithms

Module - 4 : Dynamic Programming (09 Hours)

Instructor : Mrs. Lifna C S

Topics to be covered

- **General Method**
- **Multistage graphs,**
- **Single source shortest path: Bellman Ford Algorithm,**
- **All pair shortest path: Floyd Warshall Algorithm,**
- **Matrix Chain Multiplication,**
- **Longest common subsequence,**
- **Optimal Binary Search Trees,**
- **0/1 knapsack Problem**

Dynamic Programming

- Algorithm design [to solve optimization problems](#)
- Breaks down the given problem into smaller overlapping subproblems and storing their solutions to avoid redundant computations.
- Dynamic programming [problems are usually discrete, not continuous](#)
- **Key Concepts of Dynamic Programming**
 1. **Optimal Substructure** : Deriving the solution to the problem from the solutions of its subproblems.
 2. **Overlapping Subproblems** : smaller problems are solved multiple times.
 3. **Memoization (Top-Down Approach)** : Store the results of previously solved subproblems in a table (hashmap or array) to avoid recomputation.
 4. **Tabulation (Bottom-Up Approach)** : Solve smaller subproblems first and use their results to build up the solution to larger subproblems.

Steps involved in the Dynamic Programming Strategy

1. Identify DP characteristics (Optimal Substructure & Overlapping Subproblems).
2. Define a **Recurrence Relation**.
3. Choose **Memoization (Top-Down)** or **Tabulation (Bottom-Up)**.
4. **Construct the DP Table** iteratively.
5. **Traceback** (if required) to reconstruct the solution.
6. **Optimize Space Complexity** where possible.

Advantages of DP

- ✓ **Efficient** – Reduces redundant calculations and improves performance.
- ✓ **Optimized for Large Inputs** – Works well for problems where recursion leads to exponential complexity.
- ✓ **Solves a Wide Range of Problems** – Used in optimization, sequence alignment, pathfinding, and more.

Dynamic Programming - Principle of Optimality

- In an optimal sequence of decisions or choices, each subsequence must also be optimal.
- In some problems, an optimal sequence may be found by making decisions one at a time and never making a mistake (True for greedy algorithms)
- For many problems it's not possible to make stepwise decisions based only on local information so that the sequence of decisions is optimal.
 - One way to solve such problems is to enumerate all possible decision sequences and choose the best
 - **Dynamic programming** can drastically reduce the amount of computation by avoiding sequences that cannot be optimal by the “principle of optimality”

Memoization Vs Tabulation

Feature	Memoization (Top-Down)	Tabulation (Bottom-Up)
Approach	Recursive (solves the main problem by breaking it into smaller subproblems)	Iterative (solves all subproblems first and builds the final solution)
Computation Order	Solves only required subproblems	Solves all subproblems, even if not needed
Storage	Uses a hashmap or matrix for caching recursive calls	Uses a 2D DP table for storing subproblem results
Function Calls	Uses recursion (may cause stack overflow for deep recursion)	Uses loops, avoiding recursion overhead
Efficiency	Faster for problems where only a few subproblems are needed	Efficient for problems requiring all subproblems
Space Complexity	$O(mn)$ for LCS (due to recursion stack)	$O(mn)$ but can be optimized to $O(n)$

Examples of Dynamic Programming Problems

1. Fibonacci Sequence

- Recursive Approach: $O(2^n)$ (Exponential time)
- DP Approach: $O(n)$ using Memoization or Tabulation

2. Knapsack Problem (0/1 Knapsack)

- Problem: Given weights and values of items, find the maximum value that fits in a given weight capacity.
- Complexity: $O(nW)$ using DP, where n is the number of items and W is the weight capacity.

3. Longest Common Subsequence (LCS)

- Problem: Given two strings, find the longest subsequence common to both.
- Complexity: $O(mn)$ using DP, where m and n are the lengths of the two strings.

4. Coin Change Problem

- Problem: Given denominations and a target amount, find the minimum number of coins needed.
- Complexity: $O(nT)$, where n is the number of denominations and T is the target amount.

5. Edit Distance (Levenshtein Distance)

- Problem: Find the minimum number of operations to convert one string into another.
- Complexity: $O(mn)$, where m and n are the lengths of the two strings.

Topics to be covered

- General Method
- Multistage graphs,
- Single source shortest path: Bellman Ford Algorithm,
- All pair shortest path: Floyd Warshall Algorithm,
- Matrix Chain Multiplication,
- Longest common subsequence,
- Optimal Binary Search Trees,
- 0/1 knapsack Problem

A **multistage graph** is a **directed graph** partitioned into multiple stages (say, k stages), where:

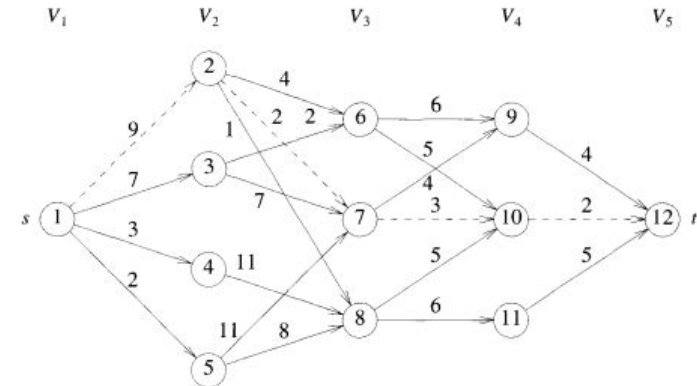
- Nodes are divided into **stages**.
- Edges go **only from one stage to the next** (i.e., stage i to stage $i+1$).
- There's a **source node** in stage 1 and a **destination (sink)** node in stage k .
- The goal is to find the **shortest (or longest) path** from source to destination.

Solved using 2 Approaches

1. Forward Approach (calculate to front (to sink))
2. Backward Approach (calculate to back (from source))

Dynamic Programming (DP) helps to

- solve the **shortest path problem** efficiently by **solving subproblems once** and **reusing the results**.



Multistage Graphs using Dynamic Programming

Input:

n : number of nodes

$\text{cost}[u][v]$: cost from node u to v (only if there is an edge)

Output:

$\text{minCost}[1]$: minimum cost from source (node 1) to destination (node n)

path: array representing the path from source to destination

1. Initialize $\text{minCost}[n] = 0$ (cost from destination to itself)

2. For i from $n-1$ down to 1:

$\text{minCost}[i] = \text{INF}$

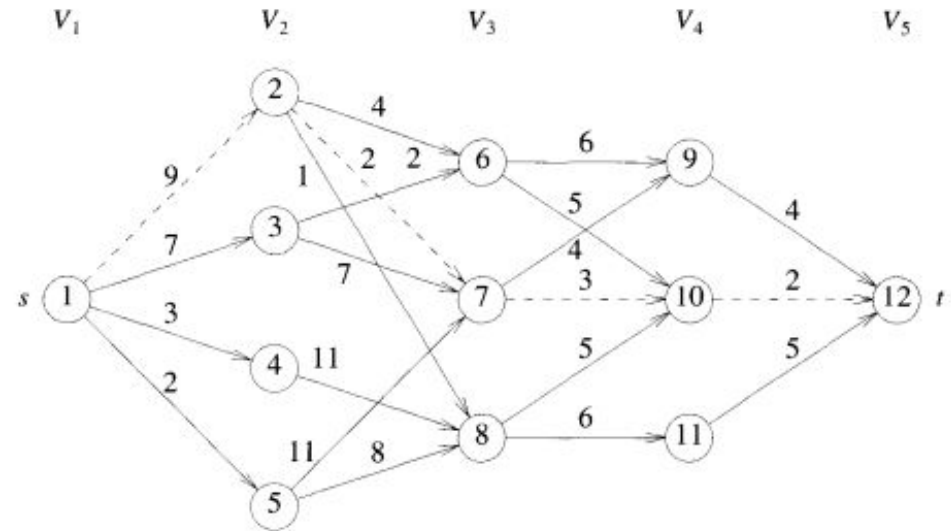
For each j where edge $(i \rightarrow j)$ exists:

if $\text{minCost}[i] > \text{cost}[i][j] + \text{minCost}[j]$:

$\text{minCost}[i] = \text{cost}[i][j] + \text{minCost}[j]$

$\text{nextNode}[i] = j$

3. Reconstruct the path using $\text{nextNode}[]$



Multistage Graphs using Dynamic Programming - Complexity

Time Complexity

- Each edge is processed once $\rightarrow O(E)$
- Reconstructing the path $\rightarrow O(n)$

Overall Time Complexity:

- $O(n^2)$ if using adjacency matrix,
- $O(n + e)$ if using adjacency list.

Multistage Graphs - Project selection (Classical Example)

Suppose we have:

- \$4 million budget
- 3 possible projects (e.g. flood control)
 - Each funded at \$1 million increments from \$0 to \$4 million
 - Each increment produces a different marginal benefit
- We want to find the plan that produces the maximum benefit
- **Stages** : number of decisions to be made
 - We have 3 stages, since we have 3 projects
- **States** : number of distinct possibilities
 - At each stage there are 5 states (\$0, 1, 2, 3, 4 million)

Project 0		Project 1		Project 2	
Investment	Benefit	Investment	Benefit	Investment	Benefit
0	0	0	0	0	0
1	6	1	5	1	1
2	8	2	11	2	4
3	8	3	16	3	5
4	10	4	17	4	6

Courtesy : [MIT Courseware](#)

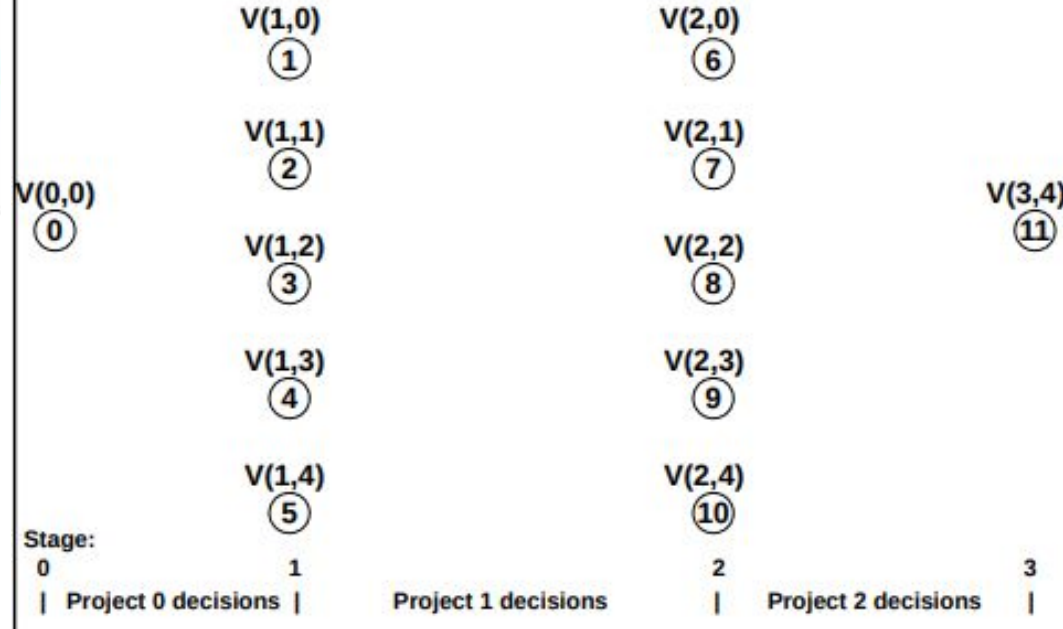
Multistage Graphs - Project selection Formulation

- We build a multistage graph to represent this problem:
 - Source node at start of graph, representing 'null' initial stage
 - Set of nodes at each stage for each state
 - Sink node at end of graph, which is a collapsed representation of the final state
- Each node characterized by $V(i,j)$:
 - $V(i,j)$ is value (benefit) obtained up to (but not including) stage i by committing j resources
 - Each node also stores its predecessor node in $P(i)$
- Each arc is characterized by $E(m,n)$:
 - $E(m,n)$ is value obtained by spending n resources on project m

Courtesy : [MIT Courseware](#)

Multistage Graphs - Project selection Formulation

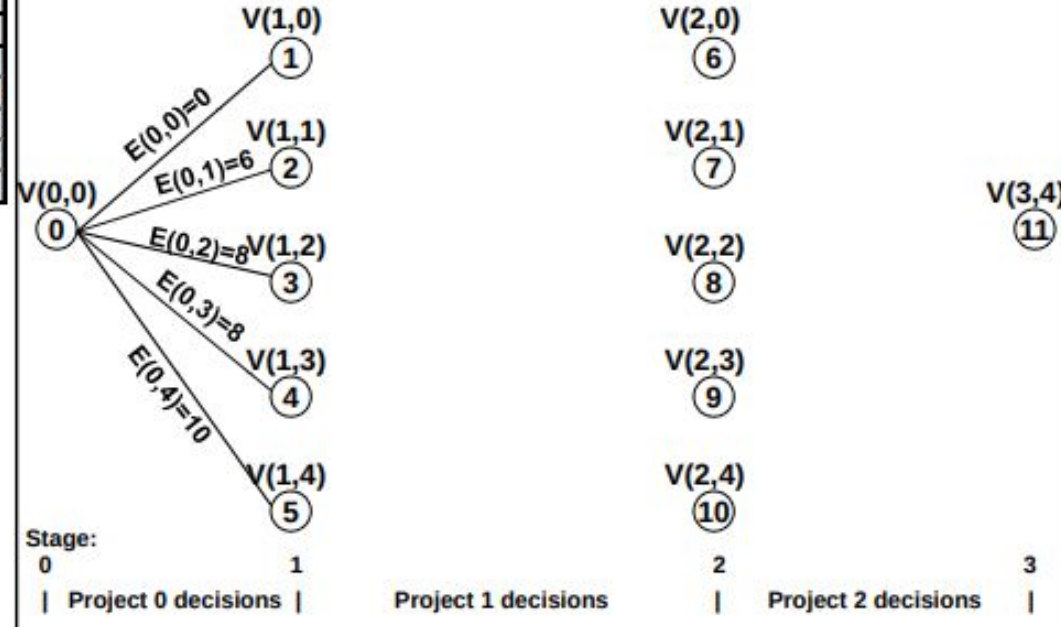
Project 0		Project 1		Project 2	
Investment	Benefit	Investment	Benefit	Investment	Benefit
0	0	0	0	0	0
1	6	1	5	1	1
2	8	2	11	2	4
3	8	3	16	3	5
4	10	4	17	4	6



Courtesy : [MIT Courseware](#)

Multistage Graphs - Project selection Formulation

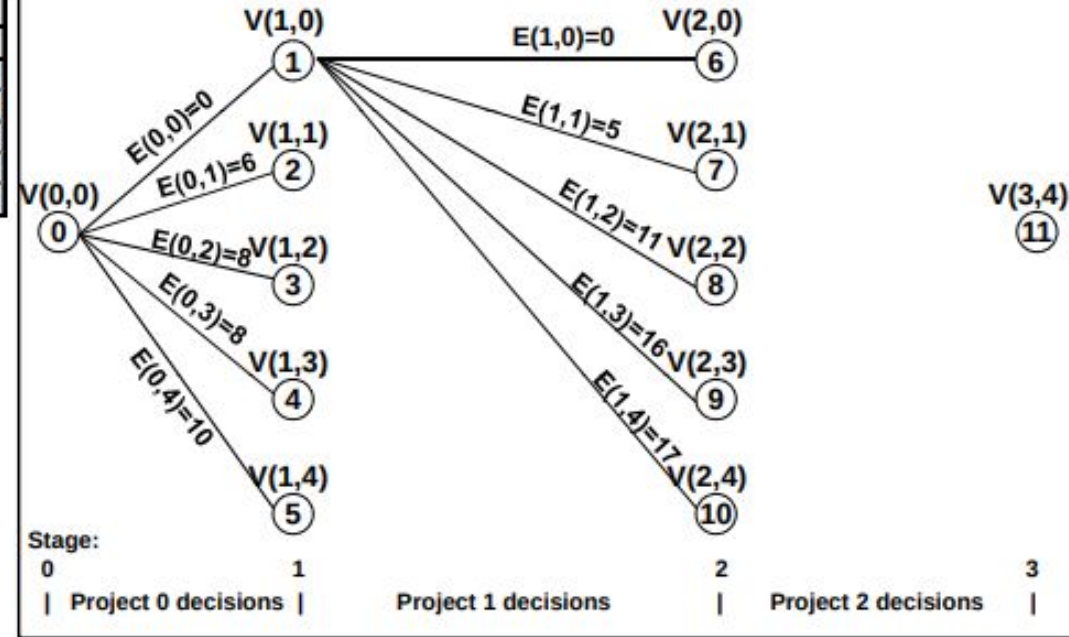
Project 0		Project 1		Project 2	
Investment	Benefit	Investment	Benefit	Investment	Benefit
0	0	0	0	0	0
1	6	1	5	1	1
2	8	2	11	2	4
3	8	3	16	3	5
4	10	4	17	4	6



Courtesy : [MIT Courseware](#)

Multistage Graphs - Project selection Formulation

Project 0		Project 1		Project 2	
Investment	Benefit	Investment	Benefit	Investment	Benefit
0	0	0	0	0	0
1	6	1	5	1	1
2	8	2	11	2	4
3	8	3	16	3	5
4	10	4	17	4	6

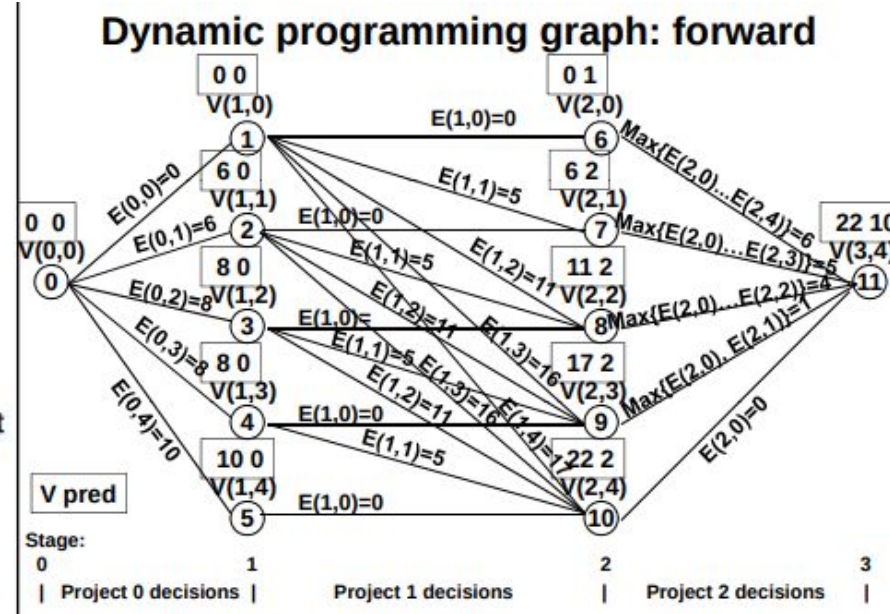


Courtesy : [MIT Courseware](#)

Multistage Graphs - Project selection Formulation

Project 0		Project 1		Project 2	
Investment	Benefit	Investment	Benefit	Investment	Benefit
0	0	0	0	0	0
1	6	1	5	1	1
2	8	2	11	2	4
3	8	3	16	3	5
4	10	4	17	4	6

- Generate graph in forward direction:
 - Start at source node
 - Compute $V(i,j)$ and $E(m,n)$ as graph is built
 - Keep track of predecessor $P(i)$ of each node that yields highest $V(i,j)$
 - This eliminates non-optimal subsequences ("pruning")
 - Eliminate infeasible arcs and nodes as graph is built
 - Rule is easy: Check budget constraint at each node; do not generate arcs or nodes that would violate it
 - End when sink node is reached from all nodes of previous stage
- Construct solution by tracing back from sink to source using predecessor variable

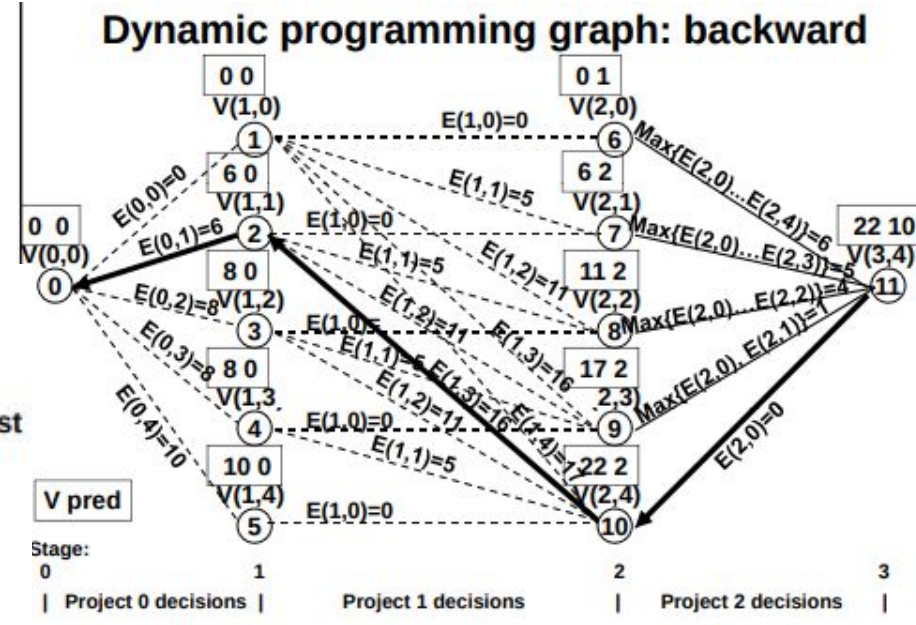


Courtesy : [MIT Courseware](#)

Multistage Graphs - Project selection Formulation

Project 0		Project 1		Project 2	
Investment	Benefit	Investment	Benefit	Investment	Benefit
0	0	0	0	0	0
1	6	1	5	1	1
2	8	2	11	2	4
3	8	3	16	3	5
4	10	4	17	4	6

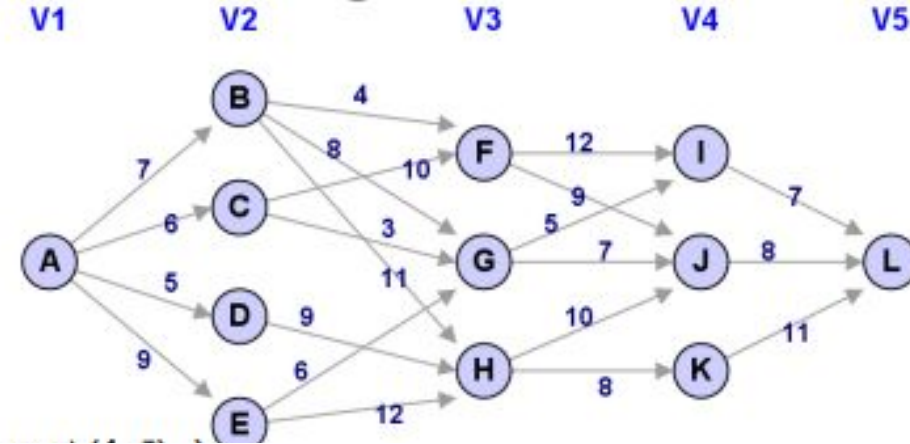
- Generate graph in forward direction:
 - Start at source node
 - Compute $V(i,j)$ and $E(m,n)$ as graph is built
 - Keep track of predecessor $P(i)$ of each node that yields highest $V(i,j)$
 - This eliminates non-optimal subsequences ("pruning")
 - Eliminate infeasible arcs and nodes as graph is built
 - Rule is easy: Check budget constraint at each node; do not generate arcs or nodes that would violate it
 - End when sink node is reached from all nodes of previous stage
- Construct solution by tracing back from sink to source using predecessor variable



Courtesy : [MIT Courseware](#)

Multistage Graphs using Dynamic Programming - Example

$\text{cost}(4, I) = c(I, L) = 7$
 $\text{cost}(4, J) = c(J, L) = 8$
 $\text{cost}(4, K) = c(K, L) = 11$



Multistage Graphs - Forward Direction

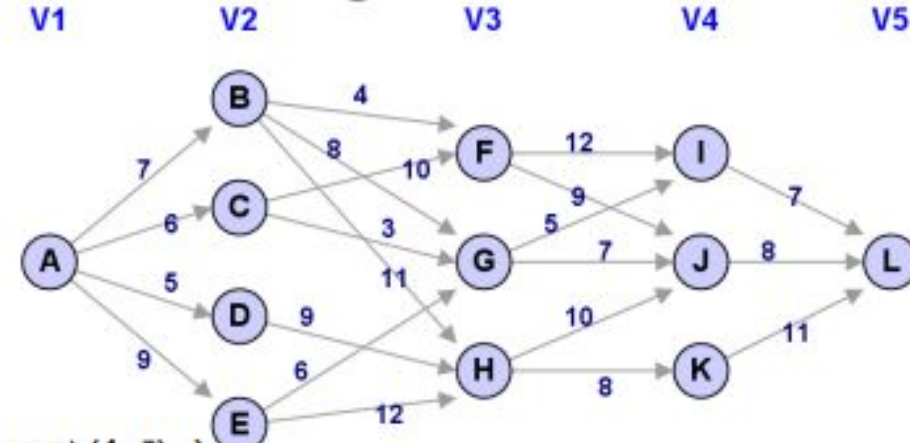
- Analysis by calculating path from a node to sink
- $\text{cost}(i, j) = \min\{c(j, k) + \text{cost}(i+1, k)\}$
- Calculation starts from nodes in stage $k-2$
- $\text{cost}(i, j)$ is distance of path from node j in stage i to sink (t)
- $c(j, l)$ is distance of path from node j to node l

Multistage Graphs using Dynamic Programming - Example

Solution in Forward Direction

```

cost(4,I) = c(I,L) = 7
cost(4,J) = c(J,L) = 8
cost(4,K) = c(K,L) = 11
cost(3,F) = min { c(F,I) + cost(4,I) | c(F,J) + cost(4,J) }
cost(3,F) = min { 12 + 7 | 9 + 8 } = 17
cost(3,G) = min { c(G,I) + cost(4,I) | c(G,J) + cost(4,J) }
cost(3,G) = min { 5 + 7 | 7 + 8 } = 12
cost(3,H) = min { c(H,J) + cost(4,J) | c(H,K) + cost(4,K) }
cost(3,H) = min { 10 + 8 | 8 + 11 } = 18
    
```



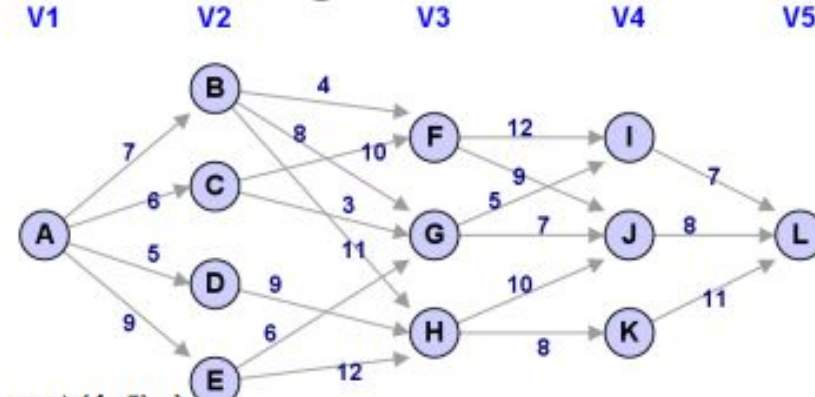
Courtesy : [Binus University - PPT](#)

Multistage Graphs using Dynamic Programming - Example

Solution in Forward Direction

```

cost(4,I) = c(I,L) = 7
cost(4,J) = c(J,L) = 8
cost(4,K) = c(K,L) = 11
cost(3,F) = min { c(F,I) + cost(4,I) | c(F,J) + cost(4,J) }
cost(3,F) = min { 12 + 7 | 9 + 8 } = 17
cost(3,G) = min { c(G,I) + cost(4,I) | c(G,J) + cost(4,J) }
cost(3,G) = min { 5 + 7 | 7 + 8 } = 12
cost(3,H) = min { c(H,J) + cost(4,J) | c(H,K) + cost(4,K) }
cost(3,H) = min { 10 + 8 | 8 + 11 } = 18
cost(2,B) = min { c(B,F) + cost(3,F) | c(B,G) + cost(3,G) | c(B,H) + cost(3,H) }
cost(2,B) = min { 4 + 17 | 8 + 12 | 11 + 18 } = 20
cost(2,C) = min { c(C,F) + cost(3,F) | c(C,G) + cost(3,G) }
cost(2,C) = min { 10 + 17 | 3 + 12 } = 15
cost(2,D) = min { c(D,H) + cost(3,H) }
cost(2,D) = min { 9 + 18 } = 27
cost(2,E) = min { c(E,G) + cost(3,G) | c(E,H) + cost(3,H) }
cost(2,E) = min { 6 + 12 | 12 + 18 } = 18
    
```



Courtesy : [Binus University - PPT](#)

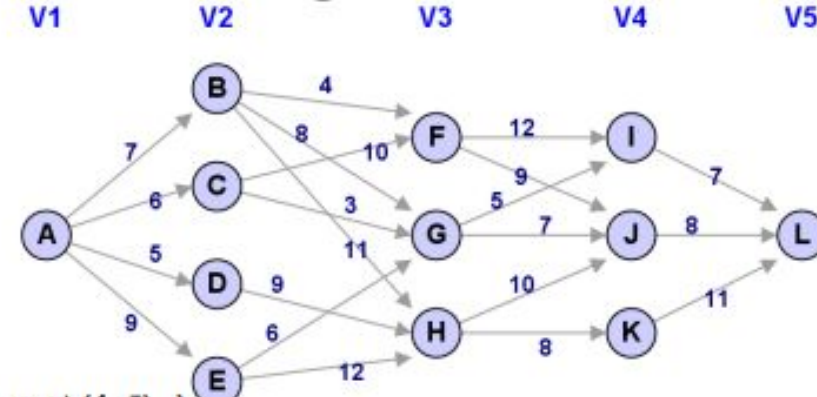
Multistage Graphs using Dynamic Programming - Example

Solution in Forward Direction

```

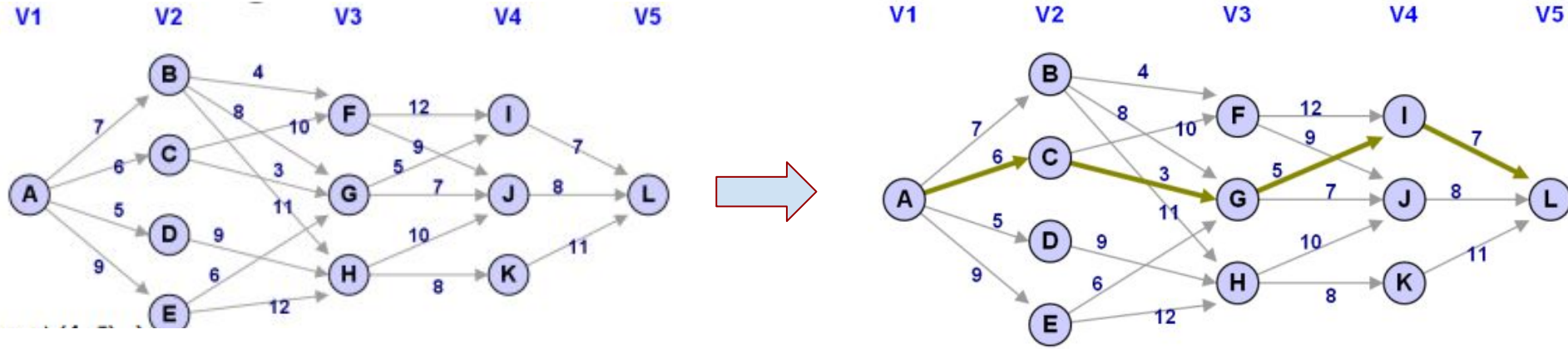
cost(4,I) = c(I,L) = 7
cost(4,J) = c(J,L) = 8
cost(4,K) = c(K,L) = 11
cost(3,F) = min { c(F,I) + cost(4,I) | c(F,J) + cost(4,J) }
cost(3,F) = min { 12 + 7 | 9 + 8 } = 17
cost(3,G) = min { c(G,I) + cost(4,I) | c(G,J) + cost(4,J) }
cost(3,G) = min { 5 + 7 | 7 + 8 } = 12
cost(3,H) = min { c(H,J) + cost(4,J) | c(H,K) + cost(4,K) }
cost(3,H) = min { 10 + 8 | 8 + 11 } = 18
cost(2,B) = min { c(B,F) + cost(3,F) | c(B,G) + cost(3,G) | c(B,H) + cost(3,H) }
cost(2,B) = min { 4 + 17 | 8 + 12 | 11 + 18 } = 20
cost(2,C) = min { c(C,F) + cost(3,F) | c(C,G) + cost(3,G) }
cost(2,C) = min { 10 + 17 | 3 + 12 } = 15
cost(2,D) = min { c(D,H) + cost(3,H) }
cost(2,D) = min { 9 + 18 } = 27
cost(2,E) = min { c(E,G) + cost(3,G) | c(E,H) + cost(3,H) }
cost(2,E) = min { 6 + 12 | 12 + 18 } = 18
cost(1,A) = min { c(A,B) + cost(2,B) | c(A,C) + cost(2,C) | c(A,D) + cost(2,D) | c(A,E) + cost(2,E) }
cost(1,A) = min { 7 + 20 | 6 + 15 | 5 + 27 | 9 + 18 } = 21
    
```

Shortest path is A-C-G-I-L with distance 21



Courtesy : [Binus University - PPT](#)

Multistage Graphs using Dynamic Programming - Example

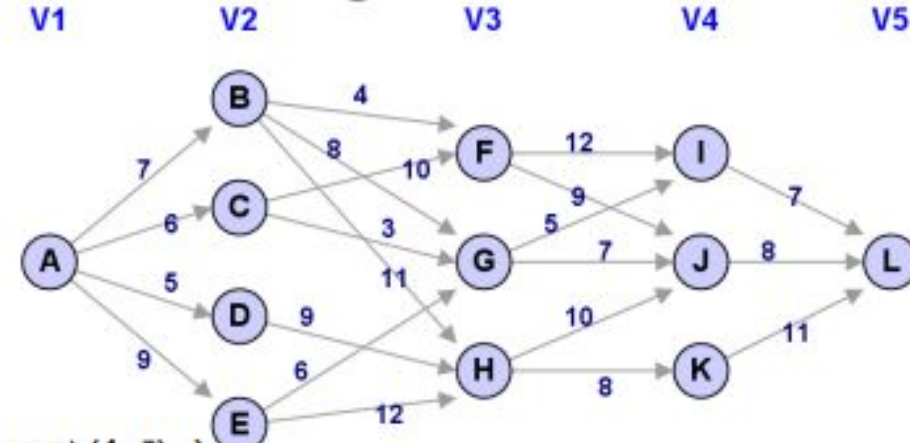


Courtesy : [Binus University - PPT](#)

Multistage Graphs using Dynamic Programming - Example

Solution in Backward Direction

$\text{bcost}(2, B) = c(A, B) = 7$
 $\text{bcost}(2, C) = c(A, C) = 6$
 $\text{bcost}(2, D) = c(A, D) = 5$
 $\text{bcost}(2, E) = c(A, E) = 9$



Multistage Graphs - Backward Direction

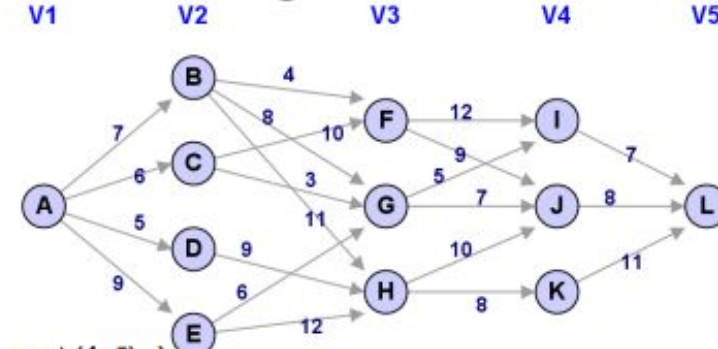
- Analysis by calculating path from source to a node
- $\text{bcost}(i, j) = \min\{\text{bcost}(i-1, l) + c(l, j)\}$
- Calculation starts from nodes in stage 3
- $\text{bcost}(i, j)$ is distance of path backward from source (s) to node j in stage i
- $c(j, l)$ is distance of path from node j to node l

Multistage Graphs using Dynamic Programming - Example

Solution in Backward Direction

```

bcost(2,B) = c(A,B) = 7
bcost(2,C) = c(A,C) = 6
bcost(2,D) = c(A,D) = 5
bcost(2,E) = c(A,E) = 9.
bcost(3,F) = min { c(B,F) + bcost(2,B) | c(C,F) + bcost(2,C) }
bcost(3,F) = min { 4 + 7 | 10 + 6 } = 11
bcost(3,G) = min { c(B,G) + bcost(2,B) | c(C,G) + bcost(2,C) | c(E,G) + bcost(2,E) }
bcost(3,G) = min { 8 + 7 | 3 + 6 | 6 + 9 } = 9
bcost(3,H) = min { c(B,H) + bcost(2,B) | c(D,H) + bcost(2,D) | c(E,H) + bcost(2,E) }
bcost(3,H) = min { 11 + 7 | 9 + 5 | 12 + 9 } = 14
    
```



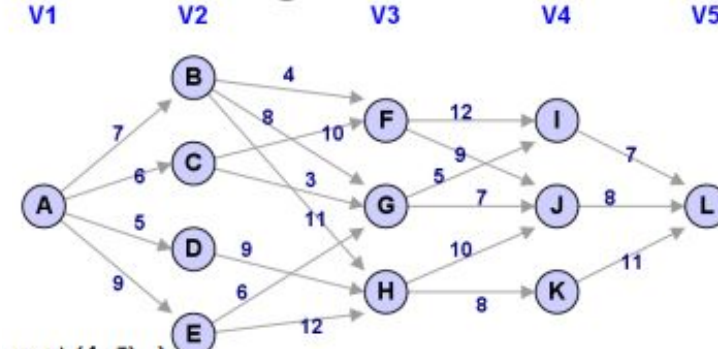
Courtesy : [Binus University - PPT](#)

Multistage Graphs using Dynamic Programming - Example

Solution in Backward Direction

```

bcost(2,B) = c(A,B) = 7
bcost(2,C) = c(A,C) = 6
bcost(2,D) = c(A,D) = 5
bcost(2,E) = c(A,E) = 9.
bcost(3,F) = min { c(B,F) + bcost(2,B) | c(C,F) + bcost(2,C) }
bcost(3,F) = min { 4 + 7 | 10 + 6 } = 11
bcost(3,G) = min { c(B,G) + bcost(2,B) | c(C,G) + bcost(2,C) | c(E,G) + bcost(2,E) }
bcost(3,G) = min { 8 + 7 | 3 + 6 | 6 + 9 } = 9
bcost(3,H) = min { c(B,H) + bcost(2,B) | c(D,H) + bcost(2,D) | c(E,H) + bcost(2,E) }
bcost(3,H) = min { 11 + 7 | 9 + 5 | 12 + 9 } = 14
bcost(4,I) = min { c(F,I) + bcost(3,F) | c(G,I) + bcost(3,G) }
bcost(4,I) = min { 12 + 11 | 5 + 9 } = 14
bcost(4,J) = min { c(F,J) + bcost(3,F) | c(G,J) + bcost(3,G) | c(H,J) + bcost(3,H) }
bcost(4,J) = min { 9 + 11 | 7 + 9 | 10 + 14 } = 16
bcost(4,K) = min { c(H,K) + bcost(3,H) }
bcost(4,K) = min { 8 + 14 } = 22
    
```



Courtesy : [Binus University - PPT](#)

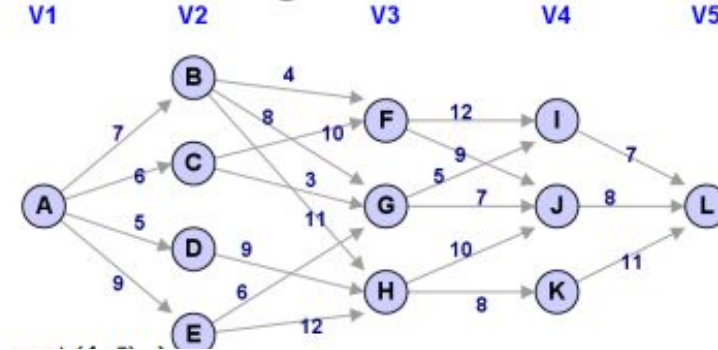
Multistage Graphs using Dynamic Programming - Example

Solution in Backward Direction

```

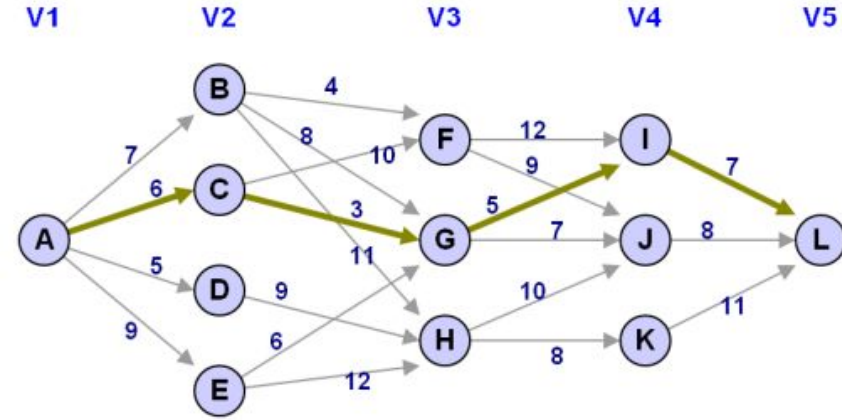
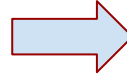
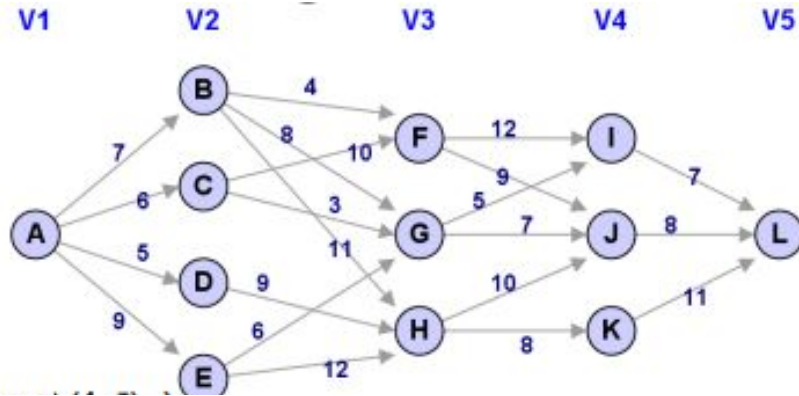
bcost(2,B) = c(A,B) = 7
bcost(2,C) = c(A,C) = 6
bcost(2,D) = c(A,D) = 5
bcost(2,E) = c(A,E) = 9.
bcost(3,F) = min { c(B,F) + bcost(2,B) | c(C,F) + bcost(2,C) }
bcost(3,F) = min { 4 + 7 | 10 + 6 } = 11
bcost(3,G) = min { c(B,G) + bcost(2,B) | c(C,G) + bcost(2,C) | c(E,G) + bcost(2,E) }
bcost(3,G) = min { 8 + 7 | 3 + 6 | 6 + 9 } = 9
bcost(3,H) = min { c(B,H) + bcost(2,B) | c(D,H) + bcost(2,D) | c(E,H) + bcost(2,E) }
bcost(3,H) = min { 11 + 7 | 9 + 5 | 12 + 9 } = 14
bcost(4,I) = min { c(F,I) + bcost(3,F) | c(G,I) + bcost(3,G) }
bcost(4,I) = min { 12 + 11 | 5 + 9 } = 14
bcost(4,J) = min { c(F,J) + bcost(3,F) | c(G,J) + bcost(3,G) | c(H,J) + bcost(3,H) }
bcost(4,J) = min { 9 + 11 | 7 + 9 | 10 + 14 } = 16
bcost(4,K) = min { c(H,K) + bcost(3,H) }
bcost(4,K) = min { 8 + 14 } = 22
bcost(5,L) = min { c(I,L) + bcost(4,I) | c(J,L) + bcost(4,J) | c(K,L) + bcost(4,K) }
bcost(5,L) = min { 7 + 14 | 8 + 16 | 11 + 22 } = 21
    
```

Shortest path is A-C-G-I-L with distance 21



Courtesy : [Binus University - PPT](#)

Multistage Graphs using Dynamic Programming - Example

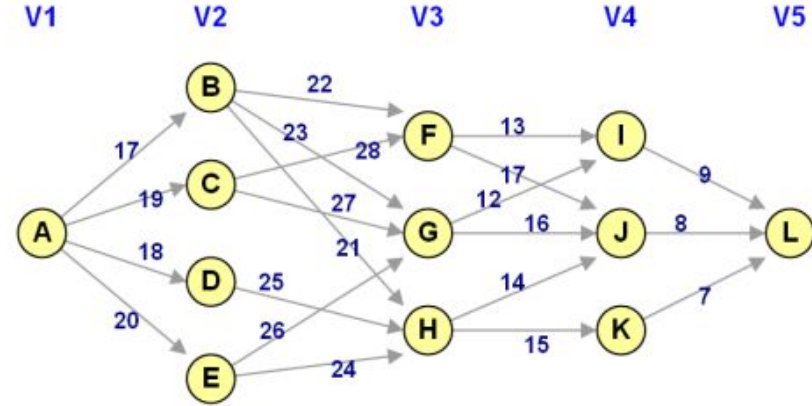
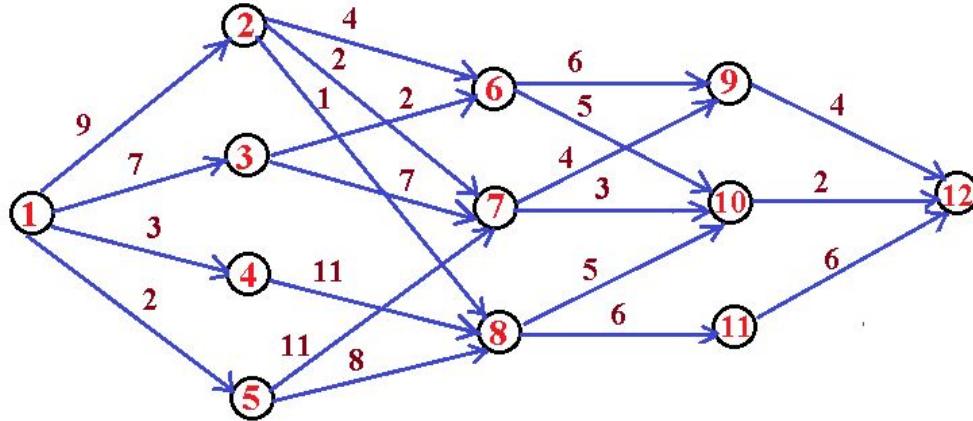


Courtesy : [Binus University - PPT](#)

Multistage Graphs - Problems

Find shortest path from source to sink using Dynamic Programming (forward method and backward method)

Project 0		Project 1		Project 2	
Investment	Benefit	Investment	Benefit	Investment	Benefit
0	0	0	0	0	0
1	6	1	5	1	1
2	8	2	11	2	4
3	8	3	16	3	5
4	10	4	17	4	6

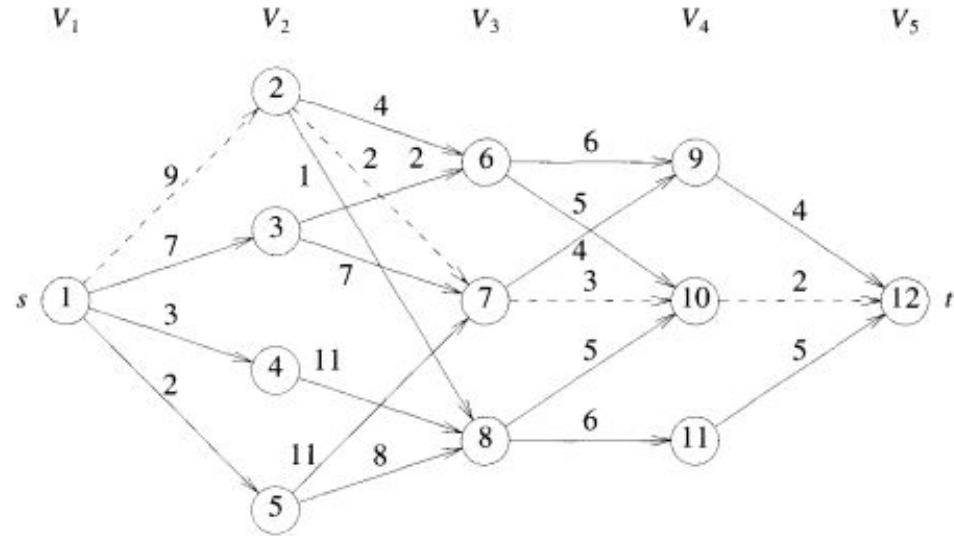
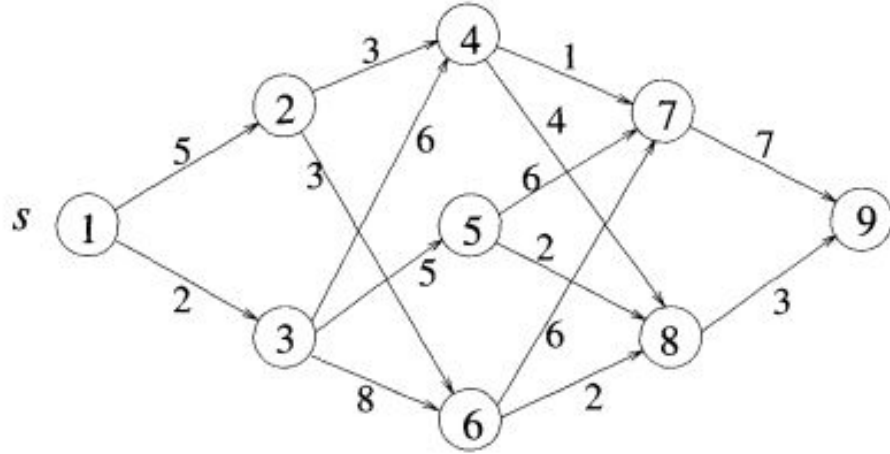


<https://kailash392.wordpress.com/wp-content/uploads/2019/02/fundamentals-of-computer-algorithms-by-ellis-horowitz.pdf>

<https://www.vrakashacademy.com/2021/06/multistage-graph-dynamic-programming-ada.html>

Multistage Graphs - Problems

Find shortest path from source to sink using Dynamic Programming (forward method and backward method)



Topics to be covered

- General Method
- Multistage graphs,
- **Single source shortest path: Bellman Ford Algorithm,**
- All pair shortest path: Floyd Warshall Algorithm,
- Matrix Chain Multiplication,
- Longest common subsequence,
- Optimal Binary Search Trees,
- 0/1 knapsack Problem

Single Source Shortest Path - Bellman Ford Algorithm

- To find the **single-source shortest path** in a graph (**negative edge weights allowed**)
- widely used in
 - **Google Maps & GPS Navigation** – Finding shortest travel routes.
 - **Network Routing Protocols** – Used in OSPF (Open Shortest Path First).
 - *AI Pathfinding (A)** – Used in games for optimal movement.

Input : Given a **weighted, directed graph** $G=(V,E)$ and a **source vertex** s

Goal : Find the **shortest path** from s to all other vertices in V .

The **Bellman-Ford algorithm** solves the SSSP problem and is particularly useful when:

- The graph contains **negative weight edges**
- You need **to detect negative weight cycles**

It uses a bottom-up dynamic programming approach.

Single Source Shortest Path - Bellman Ford Algorithm

Input : $\text{dist}[v]$ = shortest distance from source s to vertex v

Initialize $\text{dist}[s] = 0$, and $\text{dist}[v] = \infty$ for all other vertices

We repeat the **edge relaxation** process $|V| - 1$ times:

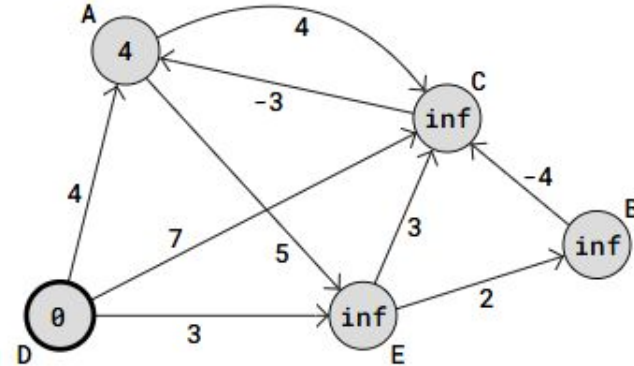
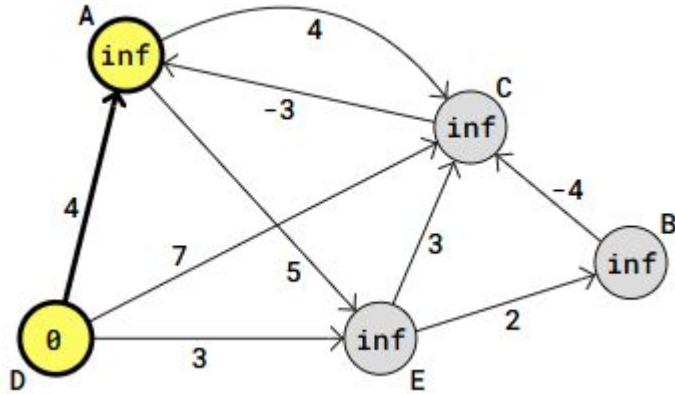
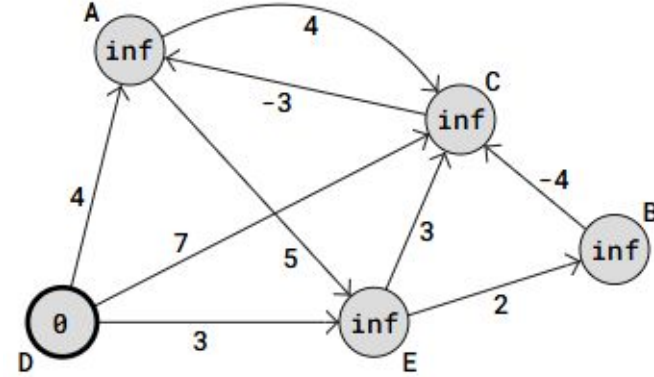
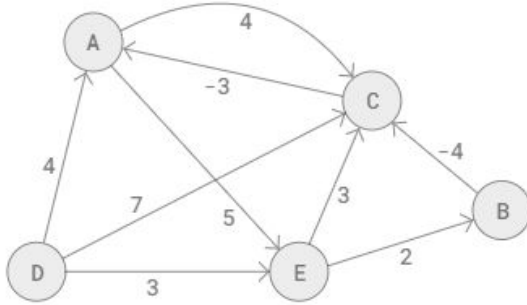
1. For each edge $(u, v) \in E$
 - If $\text{dist}[u] + w(u, v) < \text{dist}[v]$ then
$$\text{dist}[v] = \text{dist}[u] + w(u, v)$$
2. After $|V| - 1$ iterations, perform one more pass over all edges:
 - If any edge can still be relaxed, a **negative weight cycle** exists.

Note :

- Define $\text{dp}[i][v]$ = shortest distance from source s to vertex v using at most i edges
- update it during the edge relaxation

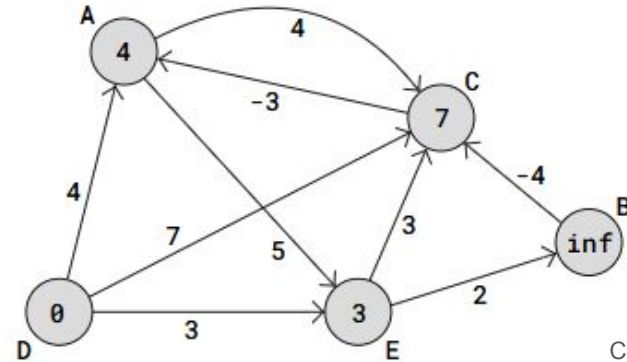
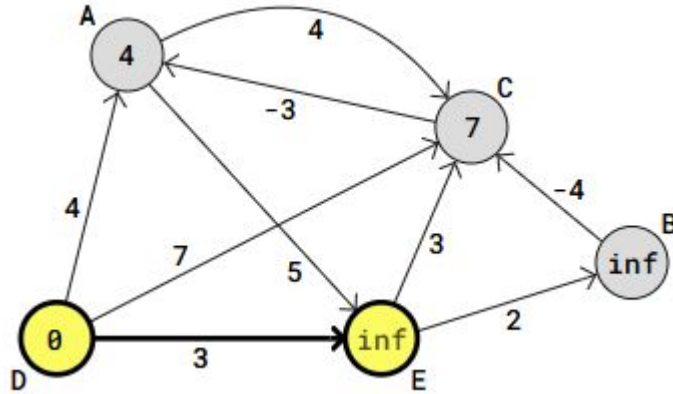
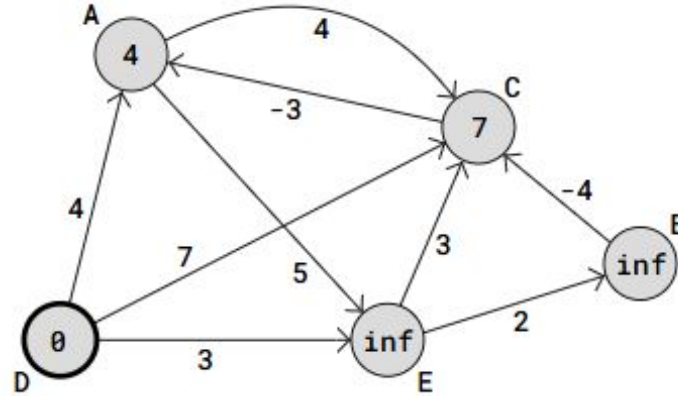
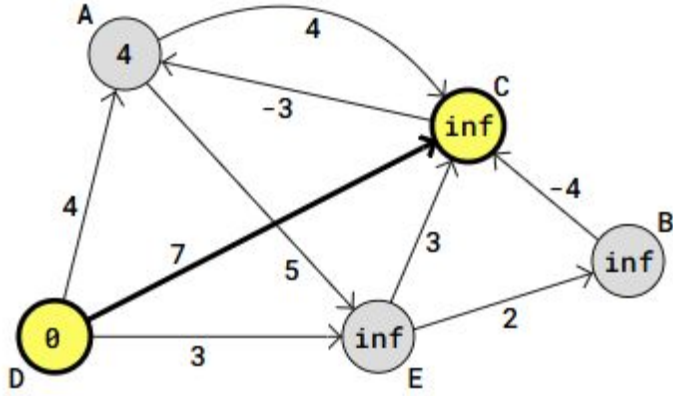
$$\text{dp}[i][v] = \min \left(\text{dp}[i-1][v], \min_{(u,v) \in E} (\text{dp}[i-1][u] + w(u, v)) \right)$$

Single Source Shortest Path - Bellman Ford Algorithm (Example)



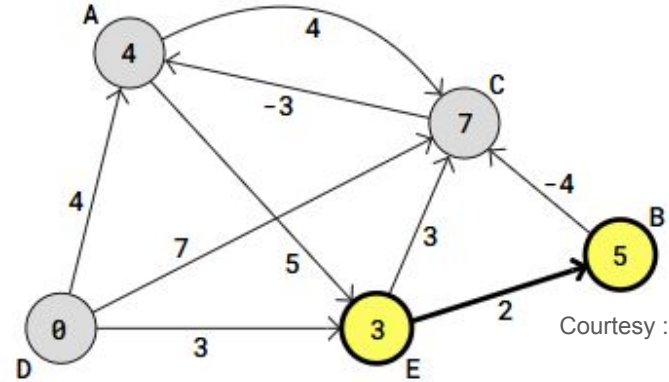
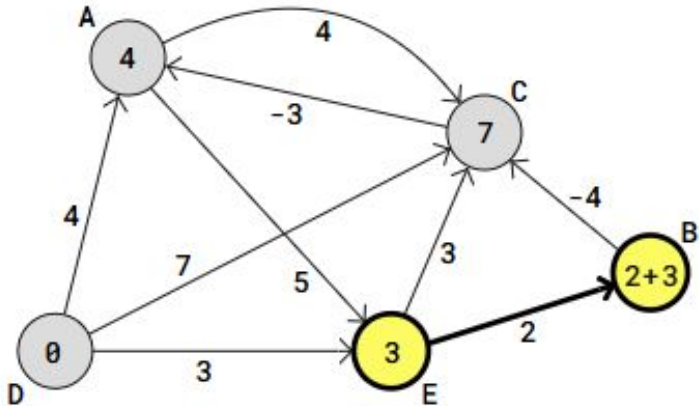
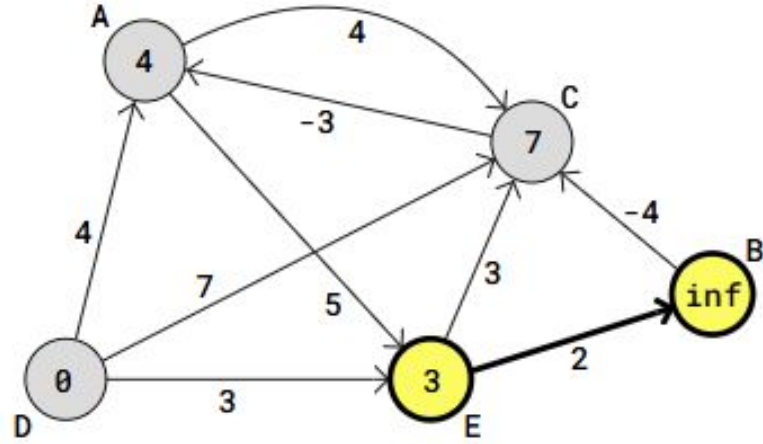
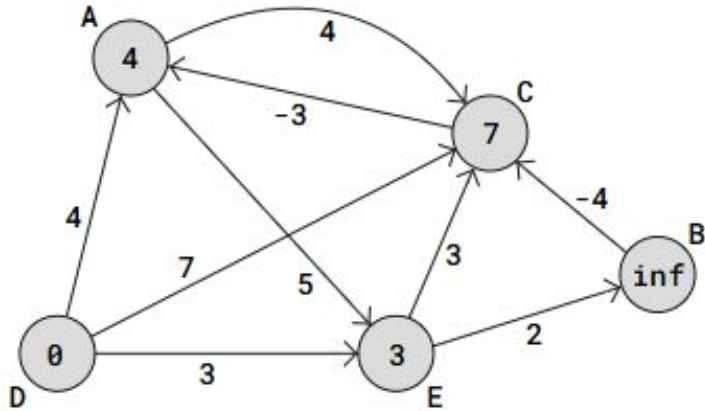
Courtesy : [w3schools](https://www.w3schools.com)

Single Source Shortest Path - Bellman Ford Algorithm (Example)



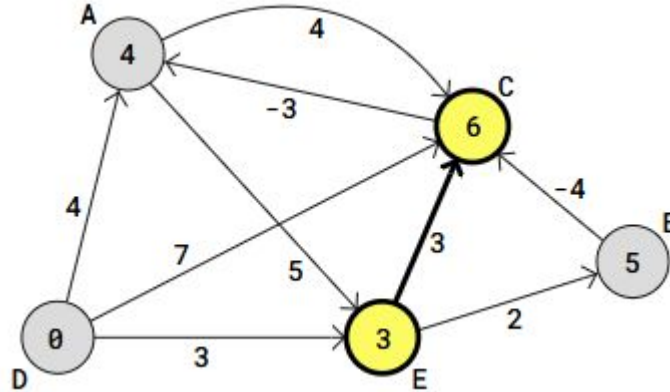
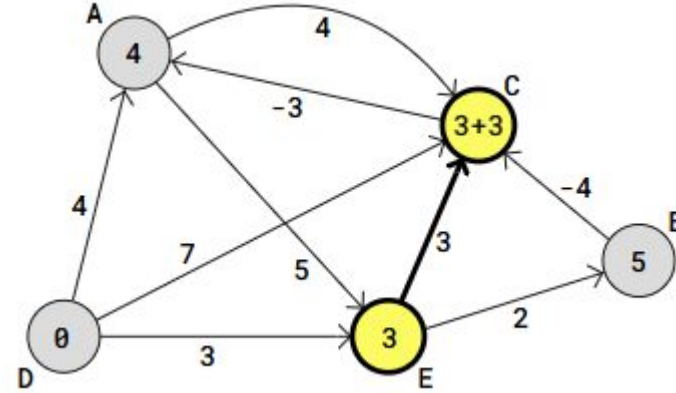
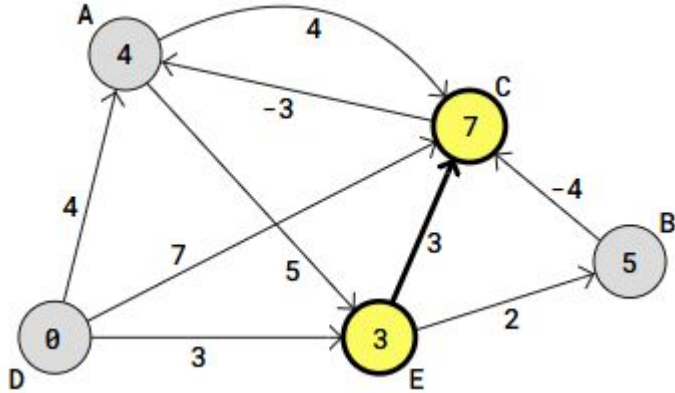
Courtesy : [w3schools](https://www.w3schools.com)

Single Source Shortest Path - Bellman Ford Algorithm (Example)



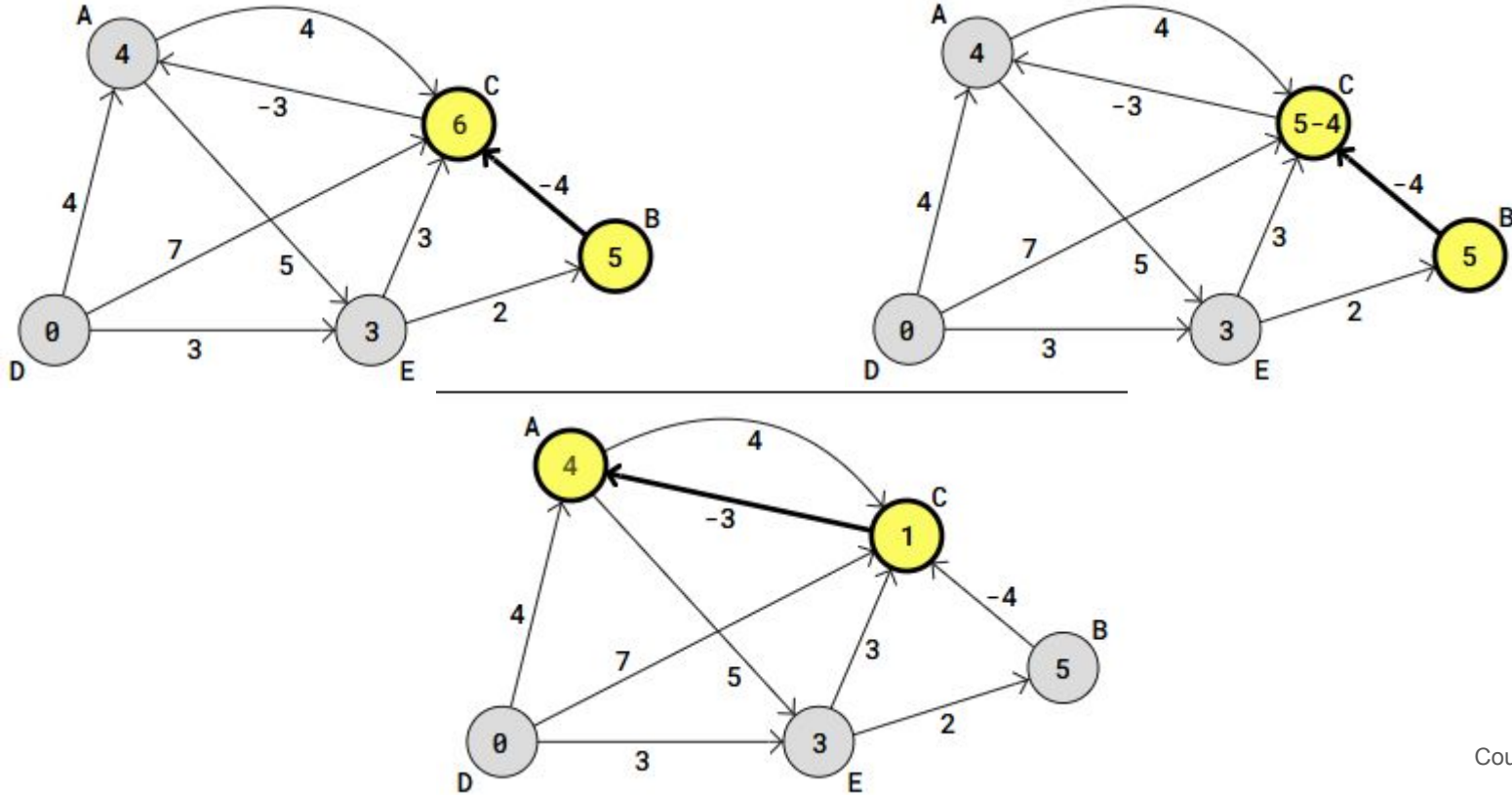
Courtesy : [w3schools](https://www.w3schools.com)

Single Source Shortest Path - Bellman Ford Algorithm (Example)



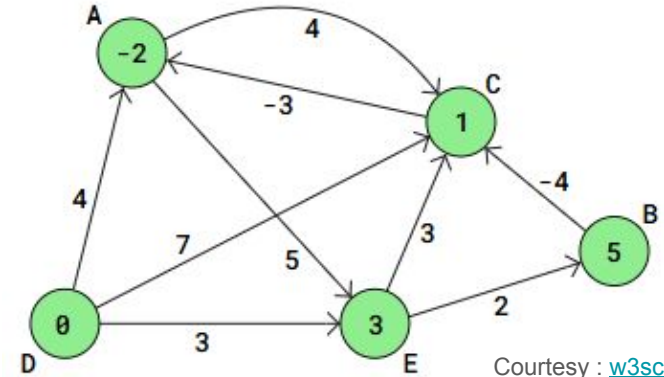
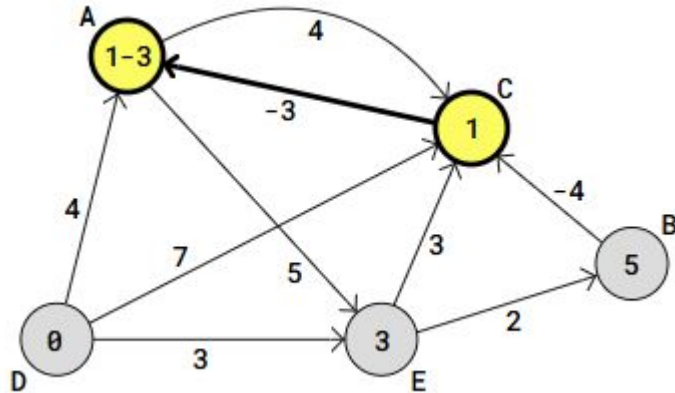
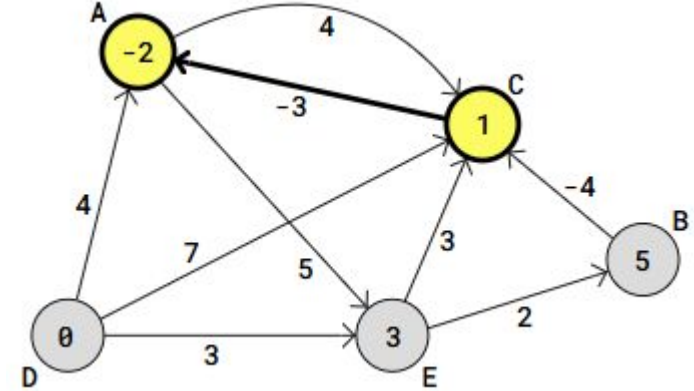
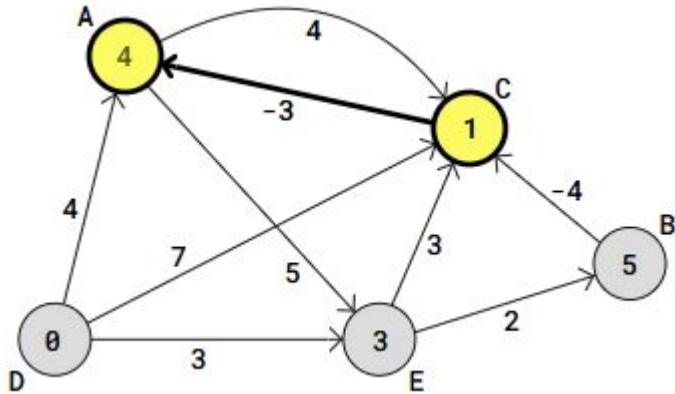
Courtesy : [w3schools](https://www.w3schools.com)

Single Source Shortest Path - Bellman Ford Algorithm (Example)



Courtesy : [w3schools](https://www.w3schools.com)

Single Source Shortest Path - Bellman Ford Algorithm (Example)

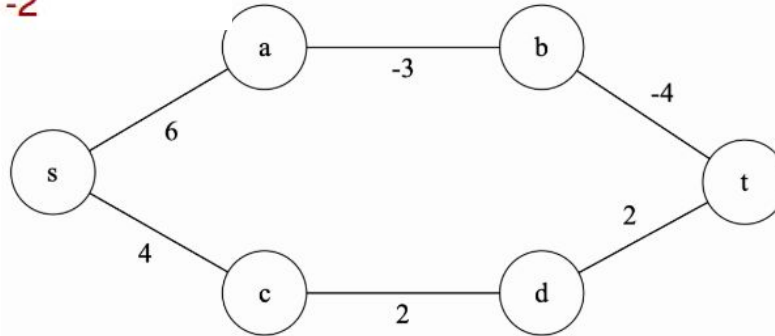
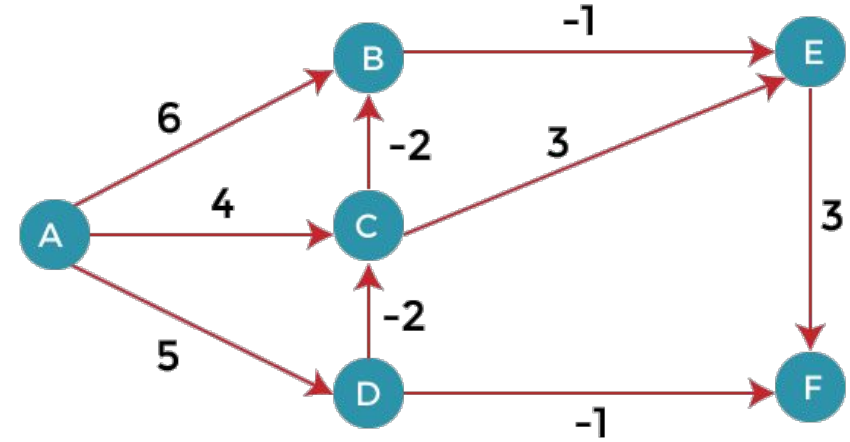
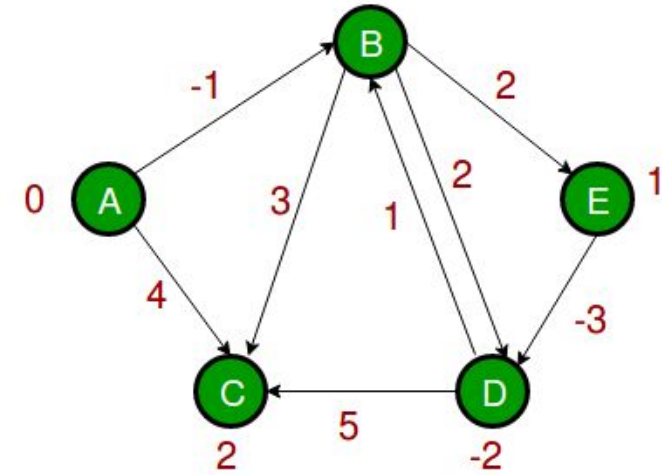


Courtesy : [w3schools](https://www.w3schools.com)

Single Source Shortest Path - Comparison

Feature	Dijkstra's Algorithm	Bellman-Ford Algorithm
Graph Type	Works with non-negative edge weights only	Works with negative weights (but no negative cycles)
Time Complexity	$O(V^2)$ (with simple implementation), $O((V + E) \log V)$ (with min-heap + adjacency list)	$O(V \times E)$
Edge Relaxation	Relaxes each edge once , in a greedy manner	Relaxes all edges V-1 times
Handling of Negative Weights	✗ Cannot handle negative weight edges	✓ Can handle negative weight edges
Negative Weight Cycles	✗ Cannot detect them	✓ Can detect negative weight cycles
Approach Type	Greedy	Dynamic Programming
Shortest Path Guarantee	Only valid if no negative weights	Works with negative weights (if no negative cycles)
Best for	Faster for graphs with all positive weights	More versatile; works in more general scenarios

Single Source Shortest Path - Bellman Ford Problems



<https://strncat.github.io/jekyll/update/2019/04/04/bellman-ford.html>

<https://www.shiksha.com/online-courses/articles/introduction-to-bellman-ford-algorithm/#:~:text=Bellman%20for d%20is%20a%20single,nodes%20in%20a%20weighte d%20graph.>

Topics to be covered

- General Method
- Multistage graphs,
- Single source shortest path: Bellman Ford Algorithm,
- All pair shortest path: Floyd Warshall Algorithm,
- Matrix Chain Multiplication,
- Longest common subsequence,
- Optimal Binary Search Trees,
- 0/1 knapsack Problem

All pair shortest path: Floyd Warshall Algorithm

- **Dynamic Programming** algorithm used to find the **shortest paths between all pairs of vertices** in a **weighted graph** (both directed and undirected).
- useful when the graph has **negative weights** but **no negative weight cycles**.
- Progressively improves the solution by considering each vertex as an **intermediate** point in the path.
- **Algorithm**

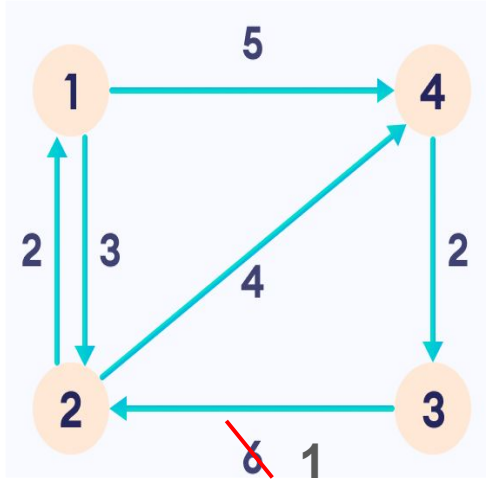
Input : Given a graph G : represented by an adjacency matrix $\text{dist}[][]$,

where $\text{dist}[i][j]$: wt of the edge from i to j.

If there is no direct edge, then $\text{dist}[i][j] = \infty$ (infinity).

```
for k from 0 to V-1:
    for i from 0 to V-1:
        for j from 0 to V-1:
            if  $\text{dist}[i][k] + \text{dist}[k][j] < \text{dist}[i][j]$ :
                 $\text{dist}[i][j] = \text{dist}[i][k] + \text{dist}[k][j]$ 
```

All pair shortest path: Floyd Warshall Algorithm

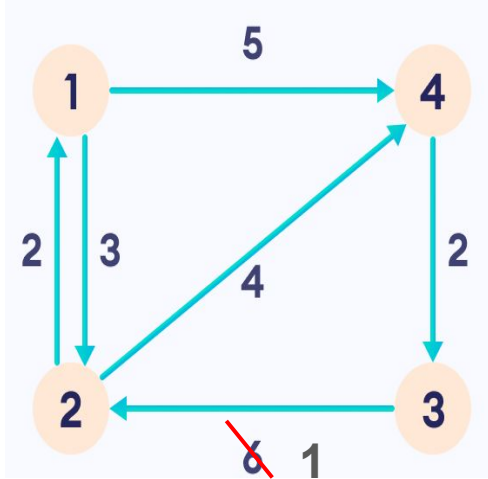


$$A^0 = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 0 & 3 & \infty & 5 \\ 2 & 0 & \infty & 4 \\ \infty & 1 & 0 & \infty \\ \infty & \infty & 2 & 0 \end{bmatrix} \end{matrix}$$

Fill each cell with the distance between i th and j th vertex

Courtesy : programiz.com

All pair shortest path: Floyd Warshall Algorithm

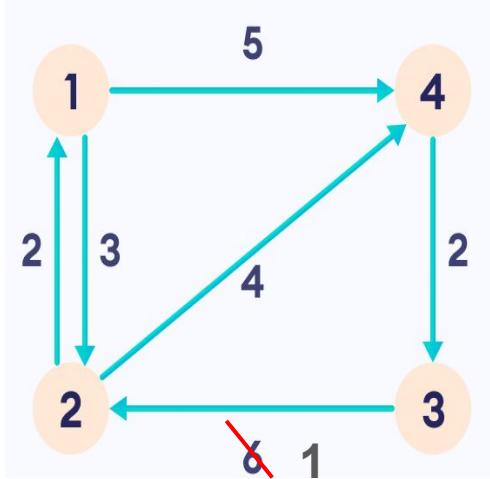


$$A^1 = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 0 & 3 & \infty & 5 \\ 2 & 0 & & \\ \infty & & 0 & \\ \infty & & & 0 \end{bmatrix} \end{matrix} \rightarrow \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 0 & 3 & \infty & 5 \\ 2 & 0 & \infty & 4 \\ \infty & 1 & 0 & \infty \\ \infty & \infty & 2 & 0 \end{bmatrix} \end{matrix}$$

Calculate the distance from the source vertex to destination vertex through this vertex 1

Courtesy : [programiz.com](https://www.programiz.com)

All pair shortest path: Floyd Warshall Algorithm

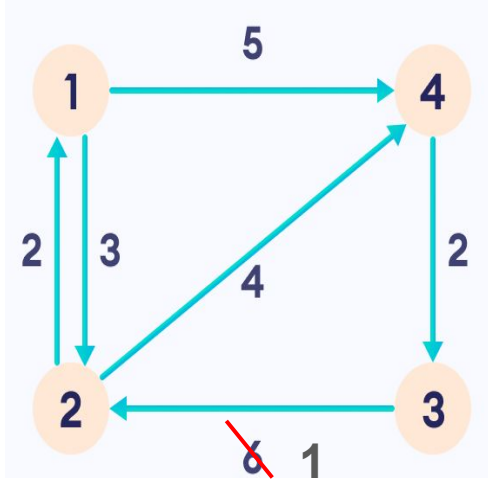


$$A^2 = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 0 & 3 & \infty & 5 \\ 2 & 0 & \infty & 4 \\ 3 & 1 & 0 & 5 \\ 4 & \infty & 2 & 0 \end{bmatrix} \end{matrix} \rightarrow \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 0 & 3 & \infty & 5 \\ 2 & 0 & \infty & 4 \\ 3 & 3 & 1 & 0 \\ 4 & \infty & \infty & 2 \end{bmatrix} \end{matrix}$$

Calculate the distance from the source vertex to destination vertex through this vertex 2

Courtesy : programiz.com

All pair shortest path: Floyd Warshall Algorithm

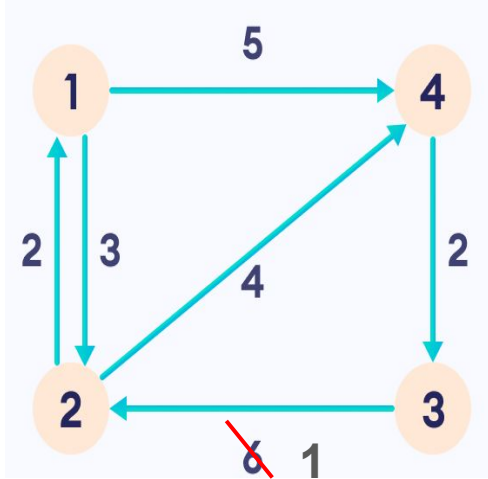


$$A^3 = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 0 & & \infty & \\ & 0 & \infty & \\ 3 & 1 & 0 & 5 \\ & & 2 & 0 \end{bmatrix} \end{matrix} \rightarrow \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 0 & 3 & \infty & 5 \\ 2 & 0 & \infty & 4 \\ 3 & 1 & 0 & 5 \\ 5 & 3 & 2 & 0 \end{bmatrix} \end{matrix}$$

Calculate the distance from the source vertex to destination vertex through this vertex 3

Courtesy : programiz.com

All pair shortest path: Floyd Warshall Algorithm

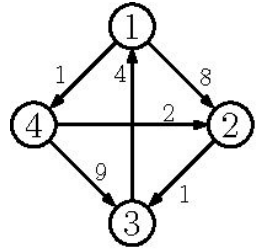


$$A^4 = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 0 & & & 5 \\ & 0 & & 4 \\ & & 0 & 5 \\ 5 & 3 & 2 & 0 \end{bmatrix} \end{matrix} \rightarrow \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 0 & 3 & 7 & 5 \\ 2 & 0 & 6 & 4 \\ 3 & 1 & 0 & 5 \\ 5 & 3 & 2 & 0 \end{bmatrix} \end{matrix}$$

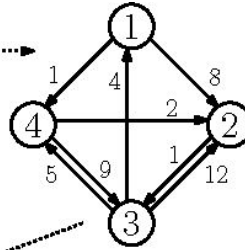
Calculate the distance from the source vertex to destination vertex through this vertex 4

Courtesy : programiz.com

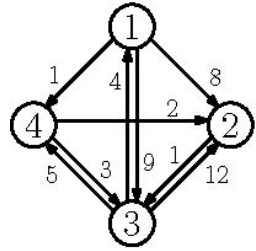
All pair shortest path: Floyd Warshall Algorithm



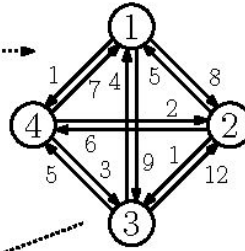
$$d^{(0)} = \begin{bmatrix} 0 & 8 & \infty & 1 \\ \infty & 0 & 1 & \infty \\ 4 & \infty & 0 & \infty \\ \infty & 2 & 9 & 0 \end{bmatrix}$$



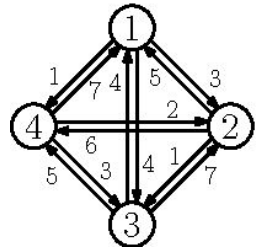
$$d^{(1)} = \begin{bmatrix} 0 & 8 & \infty & 1 \\ \infty & 0 & 1 & \infty \\ 4 & 12 & 0 & 5 \\ \infty & 2 & 9 & 0 \end{bmatrix}$$



$$d^{(2)} = \begin{bmatrix} 0 & 8 & 9 & 1 \\ \infty & 0 & 1 & \infty \\ 4 & 12 & 0 & 5 \\ \infty & 2 & 3 & 0 \end{bmatrix}$$



$$d^{(3)} = \begin{bmatrix} 0 & 8 & 9 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 12 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{bmatrix}$$

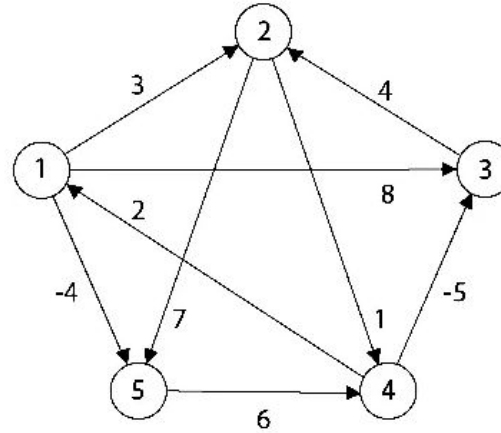
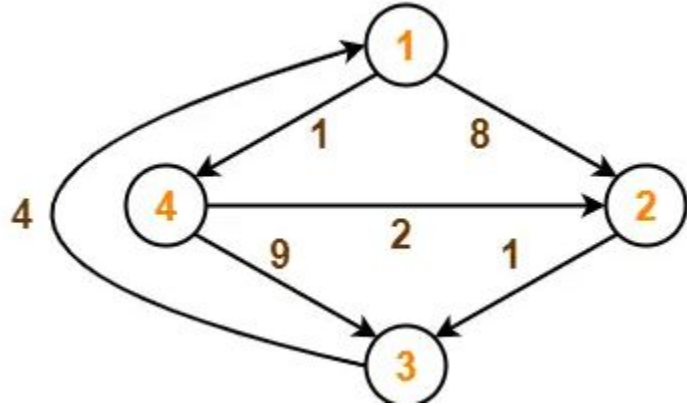


$$d^{(4)} = \begin{bmatrix} 0 & 3 & 4 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 7 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{bmatrix}$$

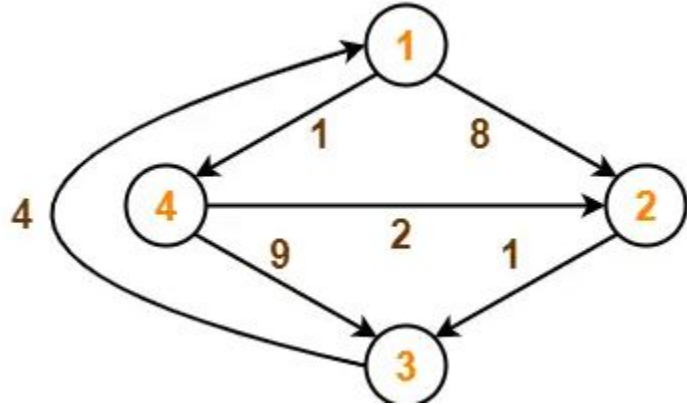
$$\text{final} = \begin{bmatrix} 0 & 3 & 4 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 7 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{bmatrix}$$

Courtesy : [semanticscholar](https://www.semanticscholar.org/)

All pair shortest path: Floyd Warshall Algorithm



All pair shortest path: Floyd Warshall Algorithm



$$D_3 =$$

	1	2	3	4
1	0	8	9	1
2	5	0	1	6
3	4	12	0	5
4	7	2	3	0

$$D_4 =$$

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

$$D_1 =$$

	1	2	3	4
1	0	8	∞	1
2	∞	0	1	∞
3	4	12	0	5
4	∞	2	9	0

$$D_2 =$$

	1	2	3	4
1	0	8	9	1
2	∞	0	1	∞
3	4	12	0	5
4	∞	2	3	0

Floyd Warshall Algorithm Example in Real Life

Problem 1: Network Latency Optimization

Scenario: A company wants to optimize data transmission between multiple data centers across different cities. Each city is connected to others with varying latencies. The goal is to find the shortest latency path between every pair of data centers to ensure efficient communication.

Solution: Use the Floyd-Warshall algorithm to compute the shortest path between all pairs of data centers, optimizing the overall network latency.

Problem 2: Urban Traffic Planning

Scenario: A city planner needs to design a traffic system that minimizes the travel time between various intersections. Each road has a different travel time, and the planner wants to ensure that the shortest routes between all intersections are identified.

Solution: Apply the Floyd-Warshall algorithm to find the shortest travel times between all intersections, aiding in traffic signal placement and route optimization.

Problem 3: Social Network Analysis

Scenario: A social network platform wants to analyze the strength of connections between users. The platform needs to determine the shortest path between every pair of users based on their interactions or mutual friends.

Solution: The Floyd-Warshall algorithm can be used to calculate the shortest paths between all pairs of users, allowing the platform to understand and visualize the network's structure.

All pair shortest path: Floyd Warshall Algorithm

Time and Space Complexity

- Time Complexity: $O(V^3)$ // (Three nested loops over all vertices)
- Space Complexity: $O(V^2)$ // (To store the distance matrix)

Applications

- Routing algorithms (like in network packets)
- Finding transitive closures
- Graph analysis for game theory or AI

Topics to be covered

- General Method
- Multistage graphs,
- Single source shortest path: Bellman Ford Algorithm,
- All pair shortest path: Floyd Warshall Algorithm,
- **Matrix Chain Multiplication,**
- Longest common subsequence,
- Optimal Binary Search Trees,
- 0/1 knapsack Problem

Matrix Chain Multiplication

Given a sequence of matrices A_1, A_2, \dots, A_n , where each matrix A_i has dimensions $p_{i-1} \times p_i$, the goal is to **determine the most efficient way to multiply these matrices together.**

Note: Matrix multiplication is associative, i.e., $(AB)C = A(BC)$, but the number of scalar multiplications needed can vary depending on the parenthesis placement.

$$A_1 A_2 A_3 = (A_1 A_2) A_3 = A_1 (A_2 A_3),$$

Note : Different parenthesis placements can lead to **different numbers of scalar multiplications.**

Goal : Find the ordering of the parenthesis with the **minimum number of scalar multiplications.**

Matrix Chain Multiplication

Given a **p x q** matrix A , a **q x r** matrix B and a **r x s** matrix C, ABC can be computed in two ways : **(AB)C** and **A(BC)**

The number of multiplications needed :
 $\text{mult}[(AB)C] = pqr + prs$
 $\text{mult}[A(BC)] = qrs + pqs.$

When $p = 5$, $q = 4$, $r = 6$ and $s = 2$, then

$$\begin{aligned}\text{mult}[(AB)C] &= 180, \\ \text{mult}[A(BC)] &= 88.\end{aligned}$$

A big difference!

Findings : The **multiplication “sequence”** (parenthesization) is important!!!

Courtesy : [CSE- Hong Kong University of Science & Technology](https://www.cse.hkust.edu.hk/)

Matrix Chain Multiplication

Given

dimensions p_0, p_1, \dots, p_n

corresponding to matrix sequence A_1, A_2, \dots, A_n

where A_i has dimension $p_{i-1} \times p_i$,

determine the “multiplication sequence” that minimizes the number of scalar multiplications in computing $A_1 A_2 \cdots A_n$. That is, determine how to parenthesize the multiplications.

$$\begin{aligned} A_1 A_2 A_3 A_4 &= (A_1 A_2)(A_3 A_4) \\ &= A_1(A_2(A_3 A_4)) = A_1((A_2 A_3)A_4) \\ &= ((A_1 A_2)A_3)(A_4) = (A_1(A_2 A_3))(A_4) \end{aligned}$$

Exhaustive search: $\Omega(4^n/n^{3/2})$.

Courtesy : [CSE- Hong Kong University of Science & Technology](https://www.cse.hkust.hk/~liffa/)

Matrix Chain Multiplication using Dynamic Programming

Step 1: . Finding an appropriate **optimal substructure property** and corresponding recurrence relation on table items.

$$m[i, j] = \min_{i \leq k < j} (m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j)$$

Step 2 : **Filling in the table properly.**

This requires finding an ordering of the table elements so that when a table item is calculated using the recurrence relation, all the table values needed by the recurrence relation have already been calculated.

Courtesy : [CSE- Hong Kong University of Science & Technology](#)

Matrix Chain Multiplication using Dynamic Programming

```
Matrix-Chain( $p, n$ )
{
  for ( $i = 1$  to  $n$ )  $m[i, i] = 0$ ;
  for ( $l = 2$  to  $n$ )
  {
    for ( $i = 1$  to  $n - l + 1$ )
    {
       $j = i + l - 1$ ;
       $m[i, j] = \infty$ ;
      for ( $k = i$  to  $j - 1$ )
      {
         $q = m[i, k] + m[k + 1, j] + p[i - 1] * p[k] * p[j]$ ;
        if ( $q < m[i, j]$ )
        {
           $m[i, j] = q$ ;
           $s[i, j] = k$ ;
        }
      }
    }
  }
  return  $m$  and  $s$ ; (Optimum in  $m[1, n]$ )
}
```

Complexity: The loops are nested three deep.

Each loop index takes on $\leq n$ values.

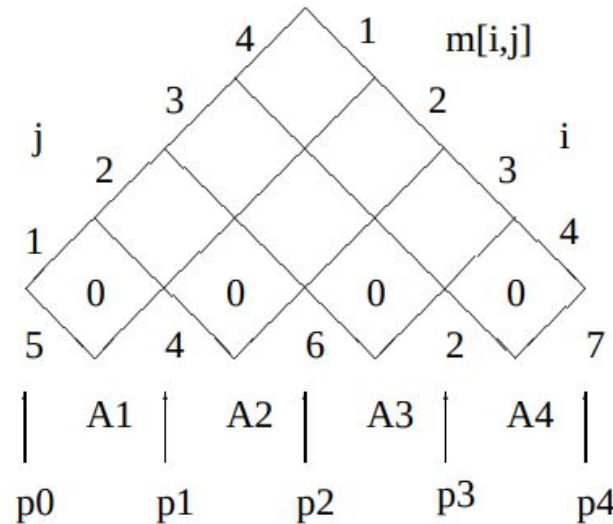
Hence the **time complexity** is $O(n^3)$. Space complexity $\Theta(n^2)$

Courtesy : [CSE- Hong Kong University of Science & Technology](https://www.hkust.edu.hk/)

Matrix Chain Multiplication using Dynamic Programming

Example: Given a chain of four matrices A_1, A_2, A_3 and A_4 , with $p_0 = 5, p_1 = 4, p_2 = 6, p_3 = 2$ and $p_4 = 7$. Find $m[1, 4]$.

S0 : Initialization



Courtesy : [CSE- Hong Kong University of Science & Technology](https://www.cse.hkust.hk/)

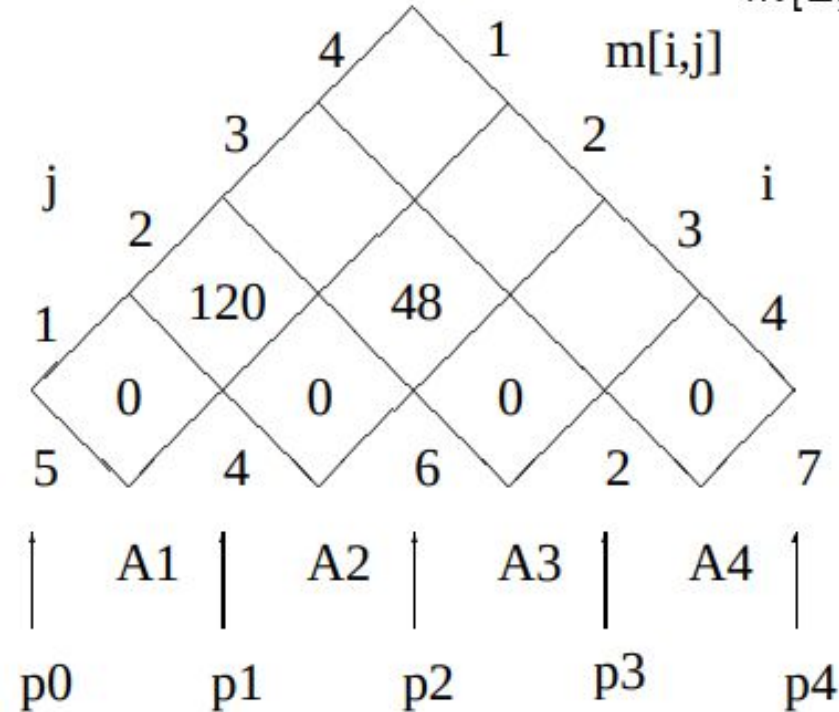
Mrs. Lifna C S

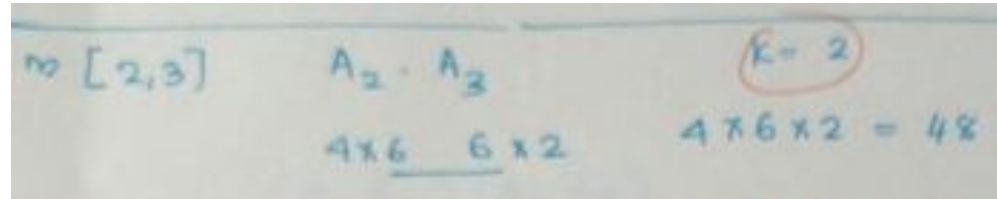
Matrix Chain Multiplication using Dynamic Programming

Step 1b: Computing $m[2,3]$ By definition

$$m[2,3] = \min_{2 \leq k < 3} (m[2,k] + m[k+1,3] + p_1 p_k p_3)$$

$$= m[2,2] + m[3,3] + p_1 p_2 p_3 = 48.$$





Handwritten calculation for $m[2,3]$ showing the multiplication of matrices A_2 and A_3 with dimensions 4×6 and 6×2 , resulting in $4 \times 6 \times 2 = 48$.

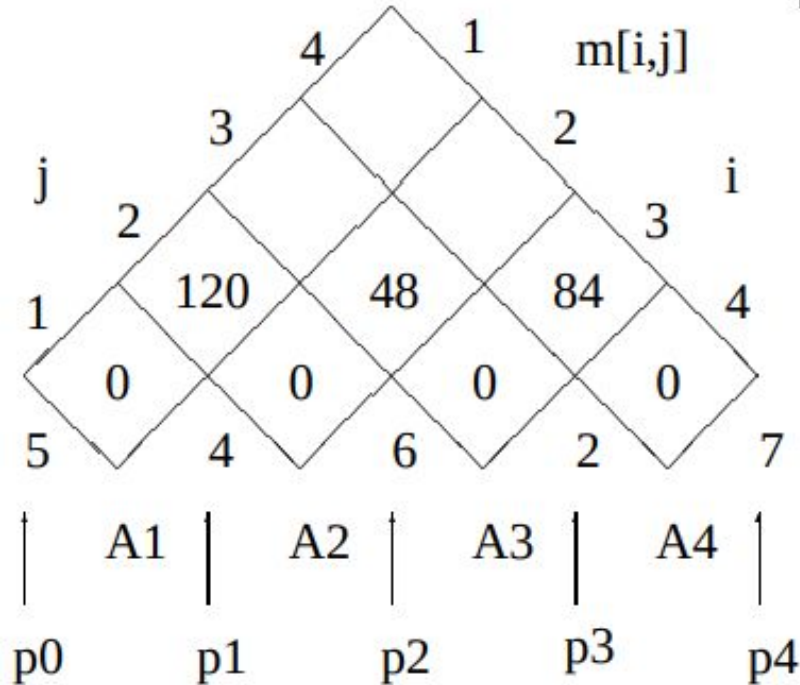
Courtesy : [CSE- Hong Kong University of Science & Technology](https://www.hkust.edu.hk/)

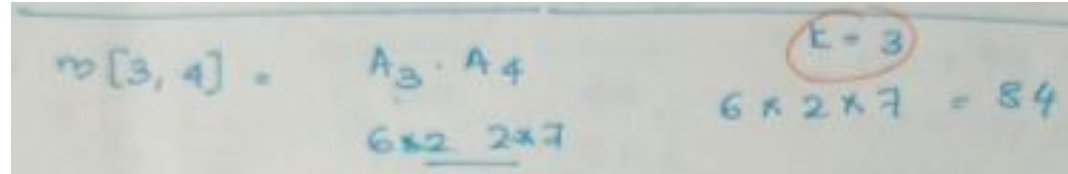
Matrix Chain Multiplication using Dynamic Programming

Step 1c: Computing $m[3,4]$ By definition

$$m[3,4] = \min_{3 \leq k < 4} (m[3,k] + m[k+1,4] + p_2 p_k p_4)$$

$$= m[3,3] + m[4,4] + p_2 p_3 p_4 = 84.$$





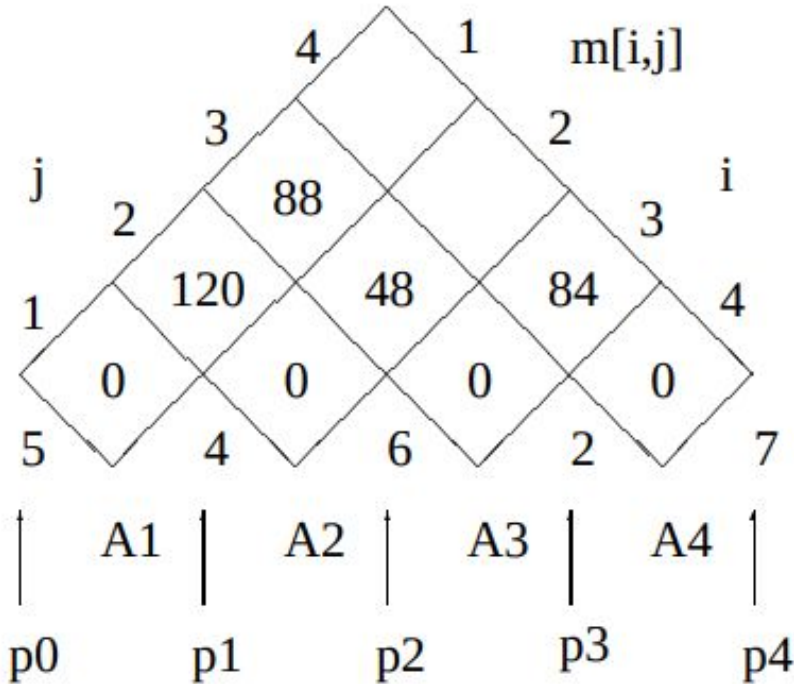
Handwritten calculation of $m[3,4]$ showing the multiplication of matrices A_3 and A_4 with dimensions 6×2 and 2×7 , resulting in 84 . The value $k=3$ is circled.

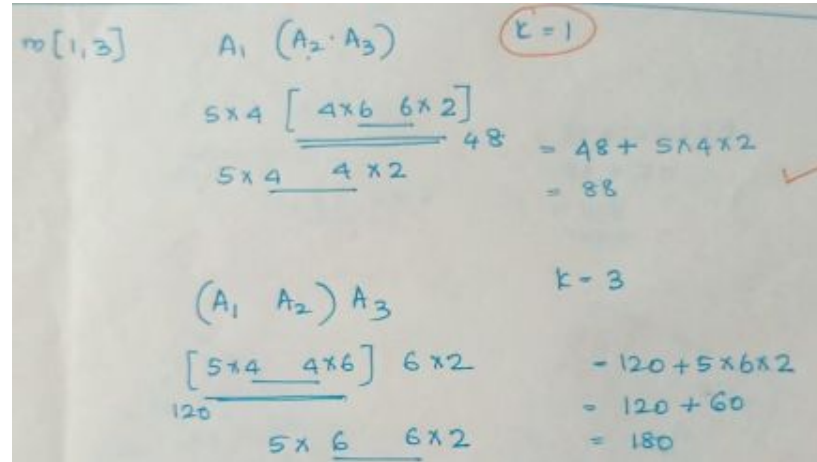
Courtesy : [CSE- Hong Kong University of Science & Technology](https://www.hkust.edu.hk/)

Matrix Chain Multiplication using Dynamic Programming

Step 2a: Computing $m[1,3]$ By definition

$$\begin{aligned}
 m[1, 3] &= \min_{1 \leq k < 3} (m[1, k] + m[k + 1, 3] + p_0 p_k p_3) \\
 &= \min \left\{ \begin{array}{l} m[1, 1] + m[2, 3] + p_0 p_1 p_3 \\ m[1, 2] + m[3, 3] + p_0 p_2 p_3 \end{array} \right\} \\
 &= 88.
 \end{aligned}$$



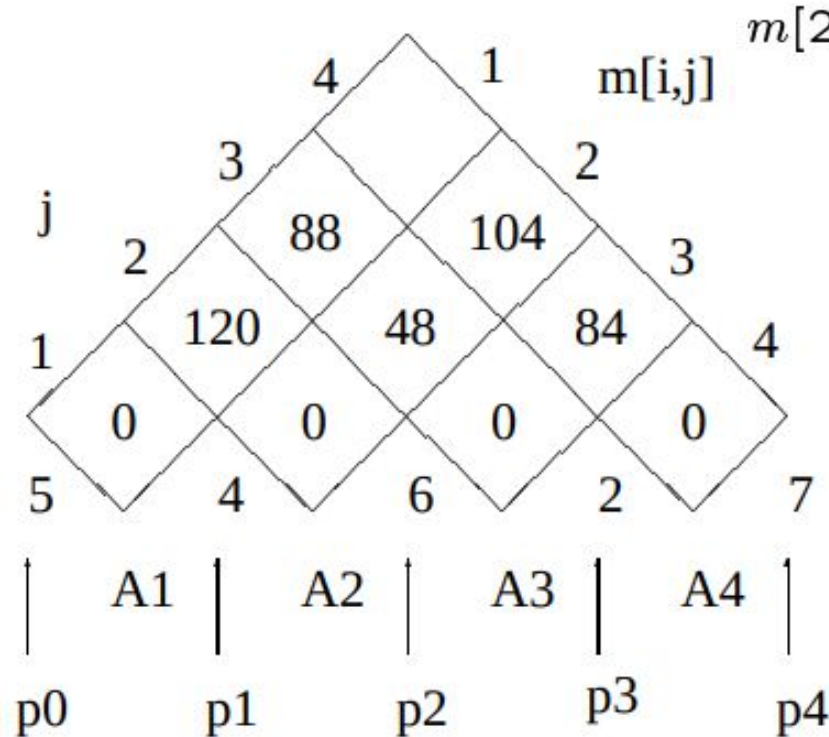


$m[1,3]$ $A_1 (A_2 A_3)$ $k=1$
 $5 \times 4 \left[\frac{4 \times 6 \quad 6 \times 2}{4 \times 2} \right] 48 = 48 + 5 \times 4 \times 2 = 88$
 $(A_1 A_2) A_3$ $k=3$
 $\left[\frac{5 \times 4 \quad 4 \times 6}{120} \right] 6 \times 2 = 120 + 5 \times 6 \times 2 = 120 + 60 = 180$

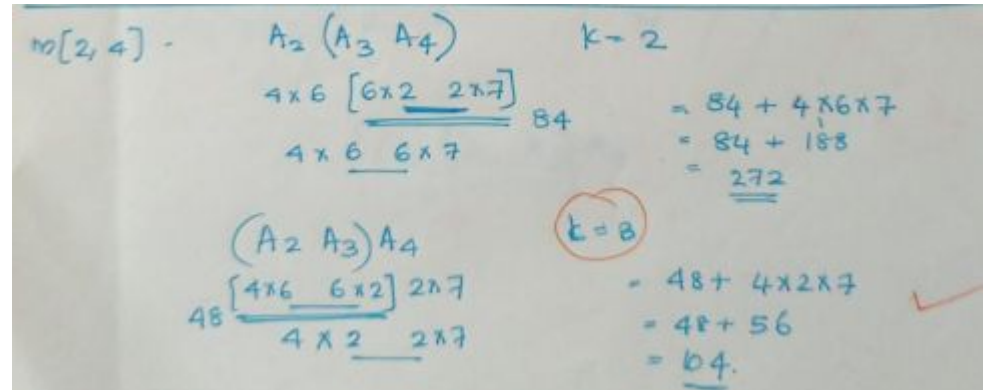
Courtesy : [CSE- Hong Kong University of Science & Technology](https://www.cse.hkust.edu.hk/)

Matrix Chain Multiplication using Dynamic Programming

Step 2b: Computing $m[2,4]$ By definition



$$\begin{aligned}
 m[2, 4] &= \min_{2 \leq k < 4} (m[2, k] + m[k + 1, 4] + p_1 p_k p_4) \\
 &= \min \left\{ \begin{array}{l} m[2, 2] + m[3, 4] + p_1 p_2 p_4 \\ m[2, 3] + m[4, 4] + p_1 p_3 p_4 \end{array} \right\} \\
 &= 104.
 \end{aligned}$$



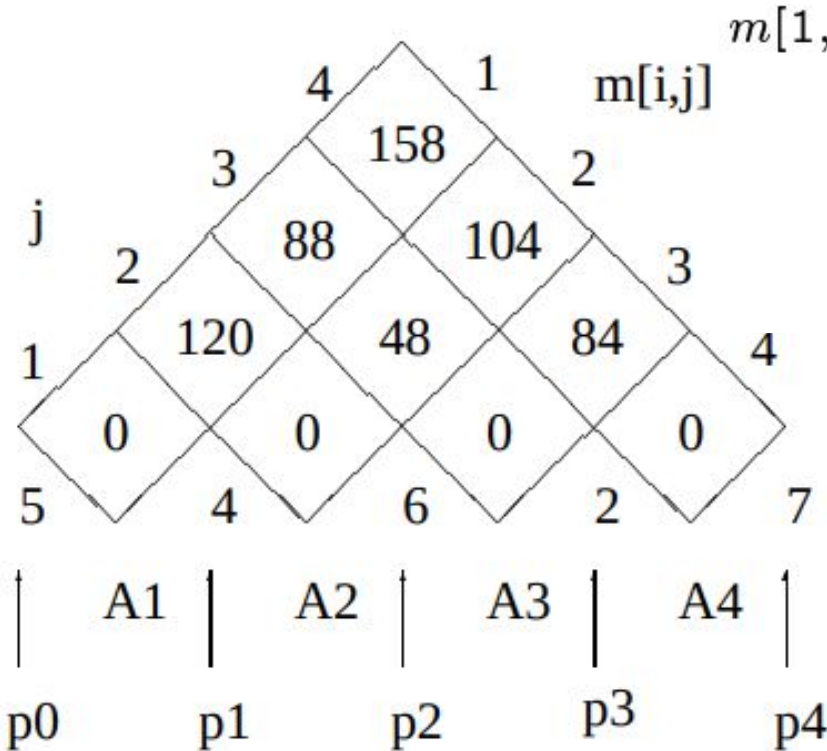
$m[2, 4] = A_2 (A_3 A_4) \quad k=2$
 $4 \times 6 \left[\begin{array}{c} 6 \times 2 \quad 2 \times 7 \\ \hline 6 \times 7 \end{array} \right] 84$
 $4 \times 6 \quad 6 \times 7$
 $= 84 + 4 \times 6 \times 7$
 $= 84 + 168$
 $= 272$

$(A_2 A_3) A_4 \quad k=3$
 $48 \left[\begin{array}{c} 4 \times 6 \quad 6 \times 2 \\ \hline 4 \times 2 \quad 2 \times 7 \end{array} \right] 2 \times 7$
 $4 \times 2 \quad 2 \times 7$
 $= 48 + 4 \times 2 \times 7$
 $= 48 + 56$
 $= 104$

Courtesy : [CSE- Hong Kong University of Science & Technology](https://www.hkust.edu.hk/)

Matrix Chain Multiplication using Dynamic Programming

Step 3: Computing $m[1,4]$ By definition

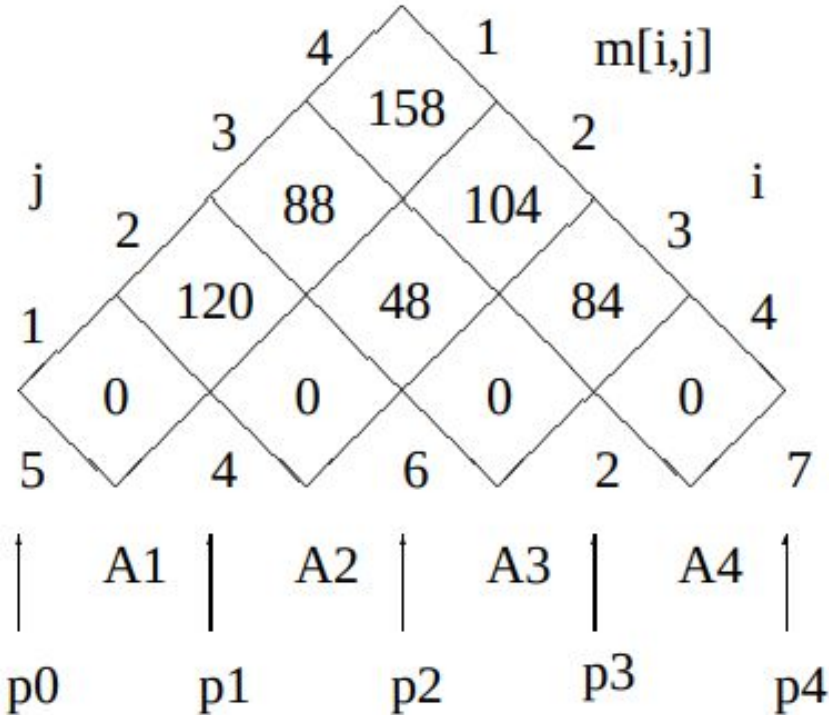


$$\begin{aligned}
 m[1, 4] &= \min_{1 \leq k < 4} (m[1, k] + m[k + 1, 4] + p_0 p_k p_4) \\
 &= \min \left\{ \begin{array}{l} m[1, 1] + m[2, 4] + p_0 p_1 p_4 \\ m[1, 2] + m[3, 4] + p_0 p_2 p_4 \\ m[1, 3] + m[4, 4] + p_0 p_3 p_4 \end{array} \right\} \\
 &= 158.
 \end{aligned}$$

Courtesy : [CSE- Hong Kong University of Science & Technology](https://www.cse.hkust.hk/)

Matrix Chain Multiplication using Dynamic Programming

Step 3: Computing $m[1,3]$ By definition



$$m[1,4] = A_1 (A_2 A_3) A_4 \quad k=1$$

$$5 \times 4 \quad 4 \times 7 \quad \rightarrow 104 + 5 \times 4 \times 7$$

$$= 104 + 140 = 244$$

$$(A_1 A_2) (A_3 A_4)$$

$$120 \quad 84$$

$$5 \times 6 \quad 6 \times 7$$

$$= 120 + 84 + 5 \times 6 \times 7$$

$$= 120 + 84 + 210$$

$$= 414$$

$$A_1 (A_2 A_3) A_4$$

$$88 \quad 5 \times 2 \quad 2 \times 7$$

$$= 88 + 5 \times 2 \times 7$$

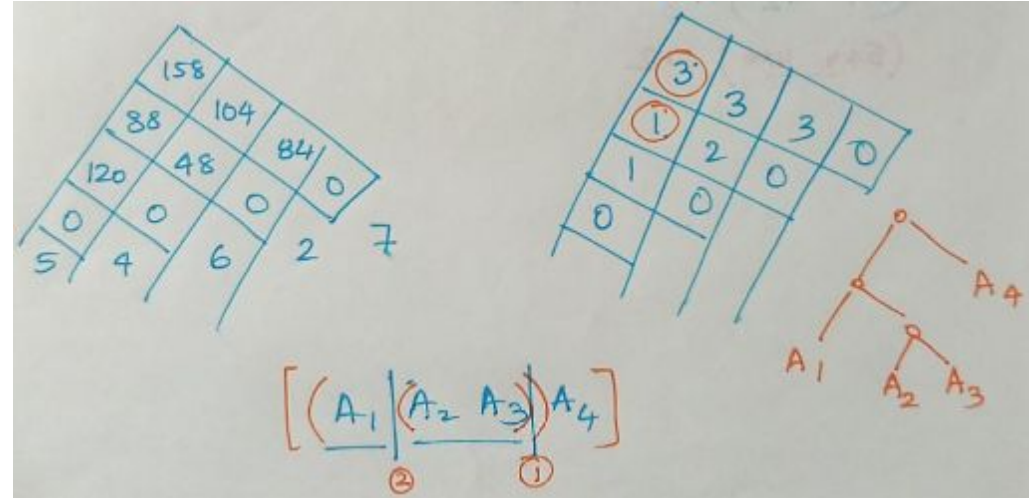
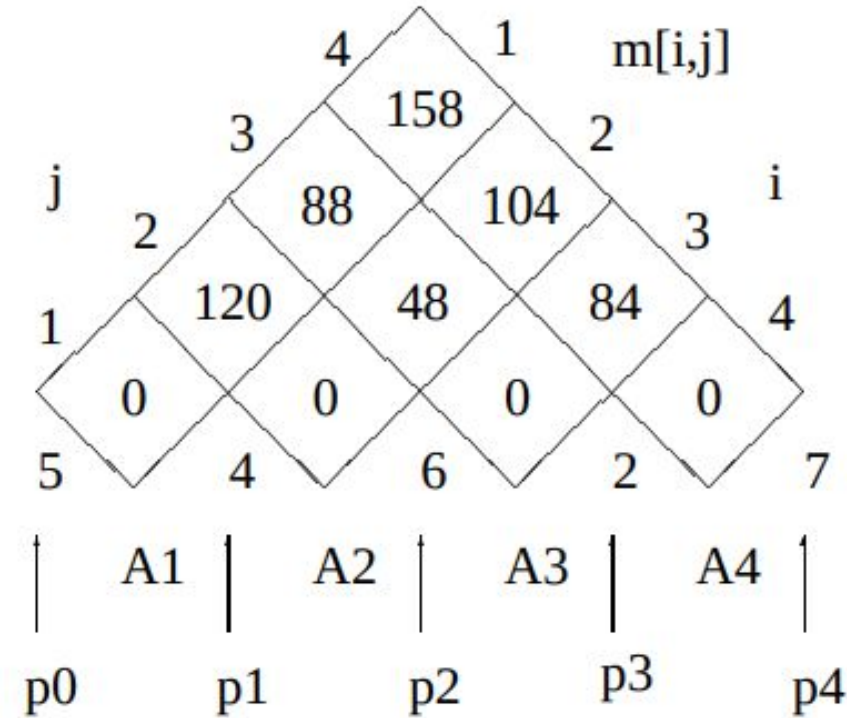
$$= 88 + 70$$

$$= 158$$

Courtesy : [CSE- Hong Kong University of Science & Technology](https://www.cse.hkust.edu.hk/)

Matrix Chain Multiplication using Dynamic Programming

Step 3: Computing $m[1,3]$ By definition



Courtesy : [CSE- Hong Kong University of Science & Technology](https://www.hkust.edu.hk/)

Matrix Chain Multiplication using Dynamic Programming

Step 4: Construct an optimal solution from computed information – extract the actual sequence.

Maintain an array $s[1..n, 1..n]$, where $s[i,j]$ denotes k for the optimal splitting in computing $A_{i..j} = A_{i..k} A_{k+1..j}$. The array $s[1..n, 1..n]$ can be used recursively to recover the multiplication sequence.

How to Recover the Multiplication Sequence?

$$\begin{aligned}
 s[1, n] & \quad (A_1 \cdots A_{s[1,n]}) (A_{s[1,n]+1} \cdots A_n) \\
 s[1, s[1, n]] & \quad (A_1 \cdots A_{s[1, s[1, n]]}) (A_{s[1, s[1, n]]+1} \cdots A_{s[1, n]}) \\
 s[s[1, n] + 1, n] & \quad (A_{s[1, n]+1} \cdots A_{s[s[1, n]+1, n]}) \times \\
 & \quad (A_{s[s[1, n]+1, n]+1} \cdots A_n) \\
 \vdots & \quad \vdots
 \end{aligned}$$

Courtesy : [CSE- Hong Kong University of Science & Technology](https://www.cse.hkust.edu.hk/)

Matrix Chain Multiplication using Dynamic Programming

Let's consider **4 matrices** A_1 to A_4 with dimensions:

- $A_1 : 10 \times 20$
- $A_2 : 20 \times 30$
- $A_3 : 30 \times 40$
- $A_4 : 40 \times 30$

So, the dimension array is:

 $p = [10, 20, 30, 40, 30]$

Matrix Chain Multiplication using Dynamic Programming

Let's consider **4 matrices** A_1 to A_4 with dimensions:

- $A_1 : 10 \times 20$
- $A_2 : 20 \times 30$
- $A_3 : 30 \times 40$
- $A_4 : 40 \times 30$

1. **Initialize:** Create a table $m[n][n]$ where $n = 4$. Initialize all $m[i][i] = 0$ since multiplying one matrix needs 0 multiplications.

2. **Fill in increasing chain length** $l = 2$ to n .

Step-by-Step Table Filling:

So, the dimension array is:

▶ $p = [10, 20, 30, 40, 30]$

Chain Length	Subproblem (i, j)	Possible k values	Cost Calculation	Minimum Cost
$l=2$	(1,2)	$k=1$	$10 \times 20 \times 30 = 6000$	$m[1][2]=6000$
	(2,3)	$k=2$	$20 \times 30 \times 40 = 24000$	$m[2][3]=24000$
	(3,4)	$k=3$	$30 \times 40 \times 30 = 36000$	$m[3][4]=36000$

Matrix Chain Multiplication using Dynamic Programming

Let's consider **4 matrices** A_1 to A_4 with dimensions:

- $A_1 : 10 \times 20$
- $A_2 : 20 \times 30$
- $A_3 : 30 \times 40$
- $A_4 : 40 \times 30$

So, the dimension array is:

▶ $p = [10, 20, 30, 40, 30]$

Chain Length $l = 3$

- $(1, 3)$:
 - Split at $k=1$: $m[1][1] + m[2][3] + 10 \times 20 \times 40 = 0 + 24000 + 8000 = 32000$
 - Split at $k=2$: $m[1][2] + m[3][3] + 10 \times 30 \times 40 = 6000 + 0 + 12000 = 18000$
 - ☒ Min: 18000
- $(2, 4)$:
 - Split at $k=2$: $m[2][2] + m[3][4] + 20 \times 30 \times 30 = 0 + 36000 + 18000 = 54000$
 - Split at $k=3$: $m[2][3] + m[4][4] + 20 \times 40 \times 30 = 24000 + 0 + 24000 = 48000$
 - ☒ Min: 48000

Matrix Chain Multiplication using Dynamic Programming


Let's consider **4 matrices** A_1 to A_4 with dimensions:

- $A_1 : 10 \times 20$
- $A_2 : 20 \times 30$
- $A_3 : 30 \times 40$
- $A_4 : 40 \times 30$

So, the dimension array is:

▶ $p = [10, 20, 30, 40, 30]$

Chain Length $l = 4$ (full chain)

- $(1, 4) :$
 - $k=1: 0 + 48000 + 10 \times 20 \times 30 = 48000 + 6000 = 54000$
 - $k=2: 6000 + 36000 + 10 \times 30 \times 30 = 42000 + 9000 = 51000$
 - $k=3: 18000 + 0 + 10 \times 40 \times 30 = 18000 + 12000 = 30000$
 -  Min: 30000

Final Result:

- Minimum number of scalar multiplications: **30000**
- Optimal parenthesization: $((A_1 \times (A_2 \times A_3)) \times A_4)$

Topics to be covered

- General Method
- Multistage graphs
- Single source shortest path: Bellman Ford Algorithm
- All pair shortest path: Floyd Warshall Algorithm
- Matrix Chain Multiplication
- Longest common subsequence
- Optimal Binary Search Trees
- 0/1 knapsack Problem

Longest Common Subsequence using DP method

Problem Statement :

- Given two sequences (strings), find the **Longest Common Subsequence (LCS)** that appears in both strings in the same relative order but not necessarily contiguous.

Subsequence : string generated from the original string by deleting 0 or more characters, without changing the relative order of the remaining characters.

Ex :

String 1	String 2	LCS	len(LCS)
"ACDBE"	"ABCDE"	"ACDE"	4
"ABC"	"ACD"	"AC"	2
"AGGTAB"	"GXTXAYB"	"GTAB"	4
"ABC",	"CBA"	"A", "B" and "C"	1

Longest Common Subsequence using DP method

Step 1 : Define the DP Table, $dp[n][m]$ of order $(n+1) * (m+1)$
where $n = \text{len}(X)$ and $m = \text{len}(Y)$

Step 2 : Define a Recurrence Relation

- If $X[i-1] == Y[j-1]$, then this character is part of LCS:
 $dp[i][j] = 1 + dp[i-1][j-1]$
- Else
 $dp[i][j] = \max(dp[i-1][j], dp[i][j-1])$

Step 3 : LCS Construction

- Start from **$dp[m][n]$** (bottom-right corner of the table).
- If $X[i-1] == Y[j-1]$,
add $X[i]$ to the result.
- Else,
move to the direction where **$dp[i][j]$** has the maximum value.

Special Case : If either string is empty, the LCS is 0:

$$dp[i][0] = 0, dp[0][j] = 0$$

		A	Y	Z	X	
		0	1	2	3	4
A X Y T	0	0	0	0	0	0
	1	0				
	2	0				
	3	0				
	4	0				

Longest Common Subsequence using DP method

Step 1 : Define the DP Table, $dp[n][m]$ of order $(n+1) * (m+1)$
where $n = \text{len}(X)$ and $m = \text{len}(Y)$

Step 2 : Define a Recurrence Relation

- If $X[i-1] == Y[j-1]$, then this character is part of LCS:
 $dp[i][j] = 1 + dp[i-1][j-1]$
- Else
 $dp[i][j] = \max(dp[i-1][j], dp[i][j-1])$

Step 3 : LCS Construction

- Start from **$dp[m][n]$** (bottom-right corner of the table).
- If $X[i-1] == Y[j-1]$,
add $X[i]$ to the result.
- Else,
move to the direction where **$dp[i][j]$** has the maximum value.

Special Case : If either string is empty, the LCS is 0:

$$dp[i][0] = 0, dp[0][j] = 0$$

cs:

		A	Y	Z	X	
		0	1	2	3	4
A X Y T	0	0	0	0	0	0
	1	0	1	1	1	1
	2	0				
	3	0				
	4	0				

Longest Common Subsequence using DP method

Step 1 : Define the DP Table, $dp[n][m]$ of order $(n+1) * (m+1)$
where $n = \text{len}(X)$ and $m = \text{len}(Y)$

Step 2 : Define a Recurrence Relation

- If $X[i-1] == Y[j-1]$, then this character is part of LCS:
 $dp[i][j] = 1 + dp[i-1][j-1]$
- Else
 $dp[i][j] = \max(dp[i-1][j], dp[i][j-1])$

Step 3 : LCS Construction

- Start from **$dp[m][n]$** (bottom-right corner of the table).
- If $X[i-1] == Y[j-1]$,
add $X[i]$ to the result.
- Else,
move to the direction where **$dp[i][j]$** has the maximum value.

Special Case : If either string is empty, the LCS is 0:

$$dp[i][0] = 0, dp[0][j] = 0$$

		A	Y	Z	X
	0	1	2	3	4
A	0	0	0	0	0
X	1	0	1	1	1
Y	2	0	1	1	2
Z	3	0			
T	4	0			

Longest Common Subsequence using DP method

Step 1 : Define the DP Table, $dp[n][m]$ of order $(n+1) * (m+1)$
where $n = \text{len}(X)$ and $m = \text{len}(Y)$

Step 2 : Define a Recurrence Relation

- If $X[i-1] == Y[j-1]$, then this character is part of LCS:
 $dp[i][j] = 1 + dp[i-1][j-1]$
- Else
 $dp[i][j] = \max(dp[i-1][j], dp[i][j-1])$

Step 3 : LCS Construction

- Start from **$dp[m][n]$** (bottom-right corner of the table).
- If $X[i-1] == Y[j-1]$,
add $X[i]$ to the result.
- Else,
move to the direction where **$dp[i][j]$** has the maximum value.

Special Case : If either string is empty, the LCS is 0:

$$dp[i][0] = 0, dp[0][j] = 0$$

CS:

		A	Y	Z	X	
		0	1	2	3	4
A X Y T	0	0	0	0	0	0
	1	0	1	1	1	1
	2	0	1	1	1	2
	3	0	1	2	2	2
	4	0				

Longest Common Subsequence using DP method

Step 1 : Define the DP Table, $dp[n][m]$ of order $(n+1) * (m+1)$
where $n = \text{len}(X)$ and $m = \text{len}(Y)$

Step 2 : Define a Recurrence Relation

- If $X[i-1] == Y[j-1]$, then this character is part of LCS:
 $dp[i][j] = 1 + dp[i-1][j-1]$
- Else
 $dp[i][j] = \max(dp[i-1][j], dp[i][j-1])$

Step 3 : LCS Construction

- Start from **$dp[m][n]$** (bottom-right corner of the table).
- If $X[i-1] == Y[j-1]$,
add $X[i]$ to the result.
- Else,
move to the direction where **$dp[i][j]$** has the maximum value.

Special Case : If either string is empty, the LCS is 0:

$$dp[i][0] = 0, dp[0][j] = 0$$

CS:

		A	Y	Z	X	
		0	1	2	3	4
A X Y T	0	0	0	0	0	0
	1	0	1	1	1	1
	2	0	1	1	1	2
	3	0	1	2	2	2
	4	0	1	2	2	2

Longest Common Subsequence using DP method

Step 1 : Define the DP Table, $dp[n][m]$ of order $(n+1) * (m+1)$
where $n = \text{len}(X)$ and $m = \text{len}(Y)$

Step 2 : Define a Recurrence Relation

- If $X[i-1] == Y[j-1]$, then this character is part of LCS:
 $dp[i][j] = 1 + dp[i-1][j-1]$
- Else
 $dp[i][j] = \max(dp[i-1][j], dp[i][j-1])$

Step 3 : LCS Construction

- Start from $dp[m][n]$ (bottom-right corner of the table).
- If $X[i-1] == Y[j-1]$,
add $X[i]$ to the result.
- Else,
move to the direction where $dp[i][j]$ has the maximum value.

Special Case : If either string is empty, the LCS is 0:
 $dp[i][0] = 0, dp[0][j] = 0$

		A	Y	Z	X
	0	1	2	3	4
0	0	0	0	0	0
1	0	1	1	1	1
2	0	1	1	1	2
3	0	1	2	2	2
4	0	1	2	2	2

Diagram illustrating the DP table for the Longest Common Subsequence (LCS) problem. The table shows the values of $dp[i][j]$ for strings $X = \text{ATX}$ and $Y = \text{AYZ}$. The table is a 5x5 grid. The first row and column are initialized to 0. The values are calculated based on the recurrence relation. The bottom-right cell (4,4) contains the value 2, which is circled in red. A blue line traces the path from the bottom-right cell (4,4) back to the top-left cell (1,1), indicating the construction of the LCS. The characters 'A' and 'Y' are circled in red in the header row, and 'A' and 'Y' are circled in red in the first column, corresponding to the characters in the LCS.

LCS : AY

Longest Common Subsequence using DP method

String 1	String 2	LCS	len(LCS)
"ACDBE"	"ABCDE"	"ACDE"	4
"ABC"	"ACD"	"AC"	2
"AGGTAB"	"GXTXAYB"	"GTAB"	4
"ABC",	"CBA"	"A", "B" and "C"	1

Longest Common Subsequence using DP method

X \ Y		A	B	C	D	E
	0	0	0	0	0	0
A	0	1	1	1	1	1
C	0	1	1	2	2	2
D	0	1	1	2	3	3
B	0	1	2	2	3	3
E	0	1	2	2	3	4

LCS : **ACDE**

Longest Common Subsequence using DP method - Complexity

Time Complexity

The algorithm iterates through a 2D table of size $m \times n$, where:

- m is the length of X .
- n is the length of Y .

Thus, Time Complexity = $O(mn)$.

Space Complexity

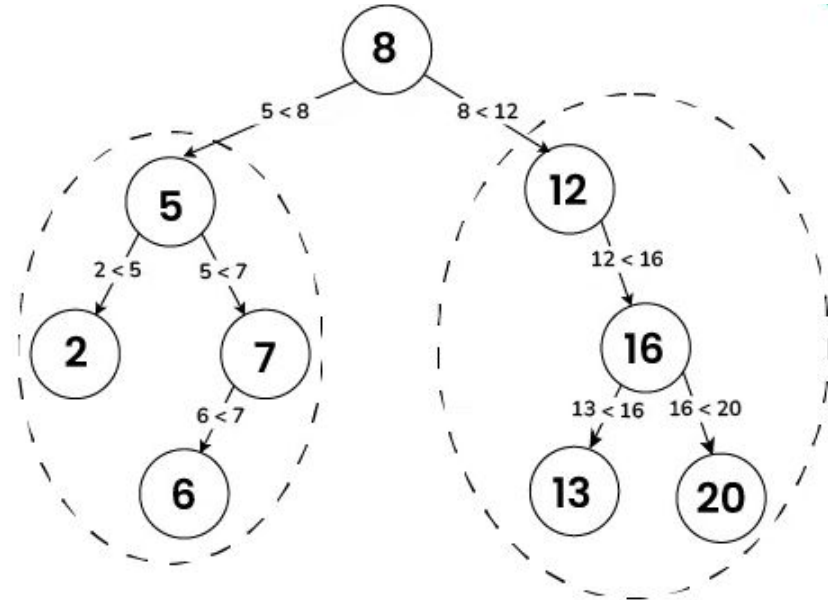
- With Table (2D DP Approach): $O(mn)$ for storing the table.
- With Space Optimization (1D Array Approach): $O(n)$, reducing the memory footprint.

Topics to be covered

- General Method
- Multistage graphs
- Single source shortest path: Bellman Ford Algorithm
- All pair shortest path: Floyd Warshall Algorithm
- Matrix Chain Multiplication
- Longest common subsequence
- **Optimal Binary Search Trees**
- 0/1 knapsack Problem

Optimal Binary Search Tree

- Optimization problem in **Dynamic Programming**
- **Objective** : Construct a **binary search tree (BST)** with the minimum possible search cost.
- **Binary Search Tree**
 - Special Binary Tree such that,
 - Left subtree values < Root
 - Right subtree values \geq Root
 - Search Time : $O(\log n)$



Left subtree contains
all elements less than 8

Right subtree contains all
elements greater than 8

Why do we need Optimal Binary Search Tree

- **Efficiency:** It reduces the average search time.
- **Cost-effective:** Less time spent searching means more time for snacks!
- **Dynamic:** It adapts to the frequency of access for different elements.
- **Structured:** It maintains a balanced structure, preventing skewed trees.
- **Memory Management:** It optimizes space usage.
- **Real-world Applications:** Used in databases and file systems.
- **Improved Performance:** Faster insertions and deletions.
- **Predictable Behavior:** Better worst-case performance.
- **Enhanced User Experience:** Quicker responses in applications.
- **Algorithmic Elegance:** It's just cool!

Courtesy : [OBST - Heycoach](#)

Applications of Optimal Binary Search Tree

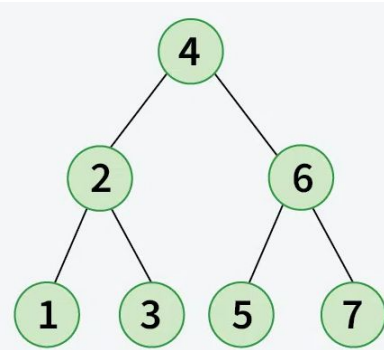
- **Databases:** OBSTs are used in indexing to speed up data retrieval.
- **Compilers:** They help in syntax tree generation for programming languages.
- **File Systems:** OBSTs optimize file storage and retrieval.
- **Network Routing:** They assist in efficient data packet routing.
- **Search Engines:** OBSTs improve search query performance.
- **Data Compression:** Used in Huffman coding for efficient data storage.
- **Artificial Intelligence:** OBSTs are used in decision-making algorithms.
- **Game Development:** They help in optimizing game state management.
- **Statistics:** OBSTs are used in frequency distribution analysis.
- **Machine Learning:** They assist in decision tree algorithms.

Courtesy : [OBST - Heycoach](#)

Construct a Balanced BST from a Sorted Array

1. Set The middle element of the array as root.
2. Recursively do the same for the left half and right half.
 - a. Get the middle of the left half and make it the left child of the root created in step 1.
 - b. Get the middle of the right half and make it the right child of the root created in step 1.
3. Print the preorder of the tree.

$arr[] = \{1, 2, 3, 4, 5, 6, 7\}$



Courtesy : [Infibnet](https://www.infibnet.org/)

Optimal Binary Search Tree Problem

Given a set of sorted keys $K = \{k_1, k_2, \dots, k_n\}$ and their corresponding search probabilities $P = \{p_1, p_2, \dots, p_n\}$, construct a binary search tree (BST) such that the expected search cost is minimized.

Search Cost Definition

The cost of searching for a key in a BST is determined by:

- The depth of the node (root has depth 1, its children have depth 2, etc.).
- The probability of searching for each key.

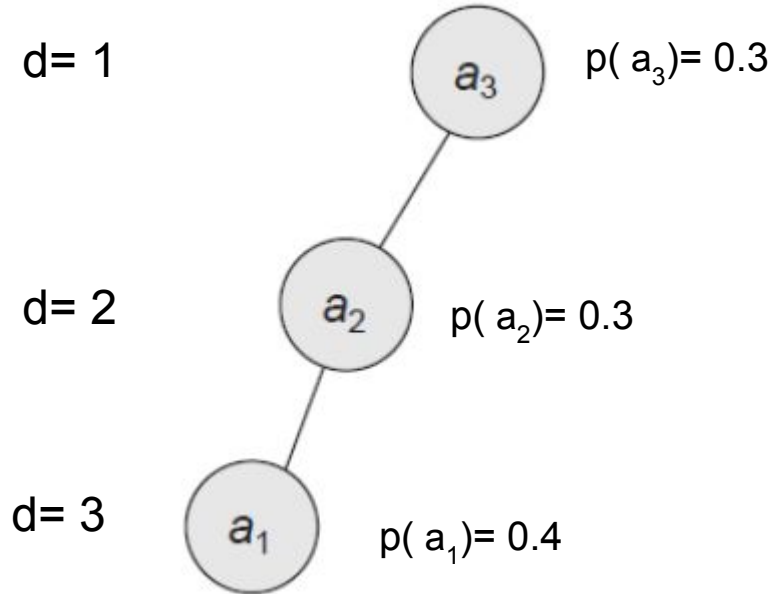
The goal is to construct a BST that minimizes the **expected search cost**, given by:

$$C(T) = \sum_{i=1}^n (p_i \times \text{depth}(k_i))$$

Courtesy : [Infibnet](https://www.infibnet.org/)

Optimal Binary Search Tree Problem

Ex: Find the cost of the tree where the items probability is given as follows: $a_1 = 0.4$, $a_2 = 0.3$, $a_3 = 0.3$



$$C(T) = \sum_{i=1}^n (p_i \times \text{depth}(k_i))$$

$$\text{Cost of BST} = 3(0.4) + 2(0.3) + 1(0.3) = 2.1$$

Courtesy : [Inflibnet](https://www.inflibnet.org/)

Optimal Binary Search Tree Problem

Ex: Construct optimal binary search tree for the three items: $a_1 = 0.4$, $a_2 = 0.3$, $a_3 = 0.3$

Calculate the search cost based on the cost function :
$$C(T) = \sum_{i=1}^n (p_i \times \text{depth}(k_i))$$

Note : Using Brute Force Method

- # Trees = Catalan sequence

$$c(n) = \binom{2n}{n} \frac{1}{n+1} \text{ for } n > 0, \quad c(0) = 1$$

- When $n = 3$, **Five search trees** are possible

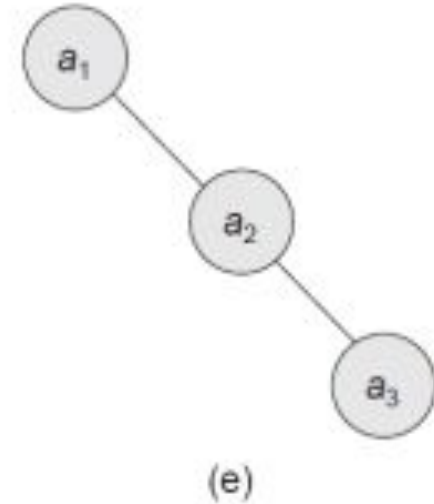
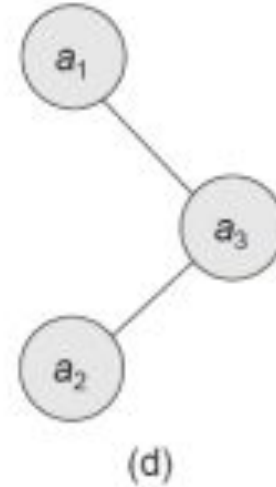
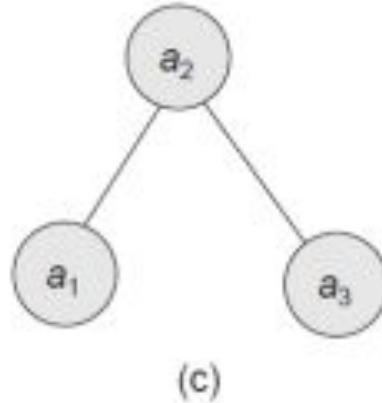
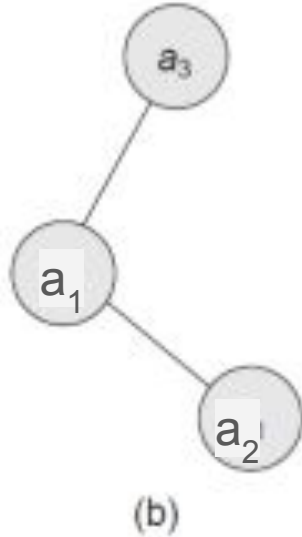
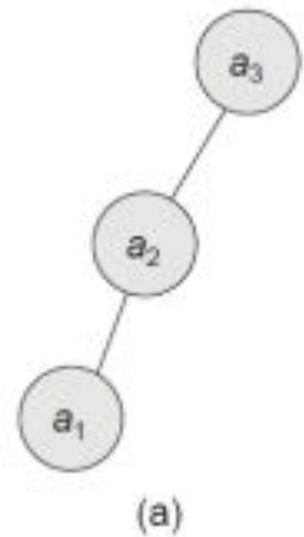
$$C_3 = \frac{1}{3+1} \binom{6}{3} = 5$$

Courtesy : [Inflibnet](https://www.inflibnet.org/)

Optimal Binary Search Tree Problem

Ex: Construct optimal binary search tree for the three items: $a_1 = 0.3$, $a_2 = 0.2$, $a_3 = 0.5$

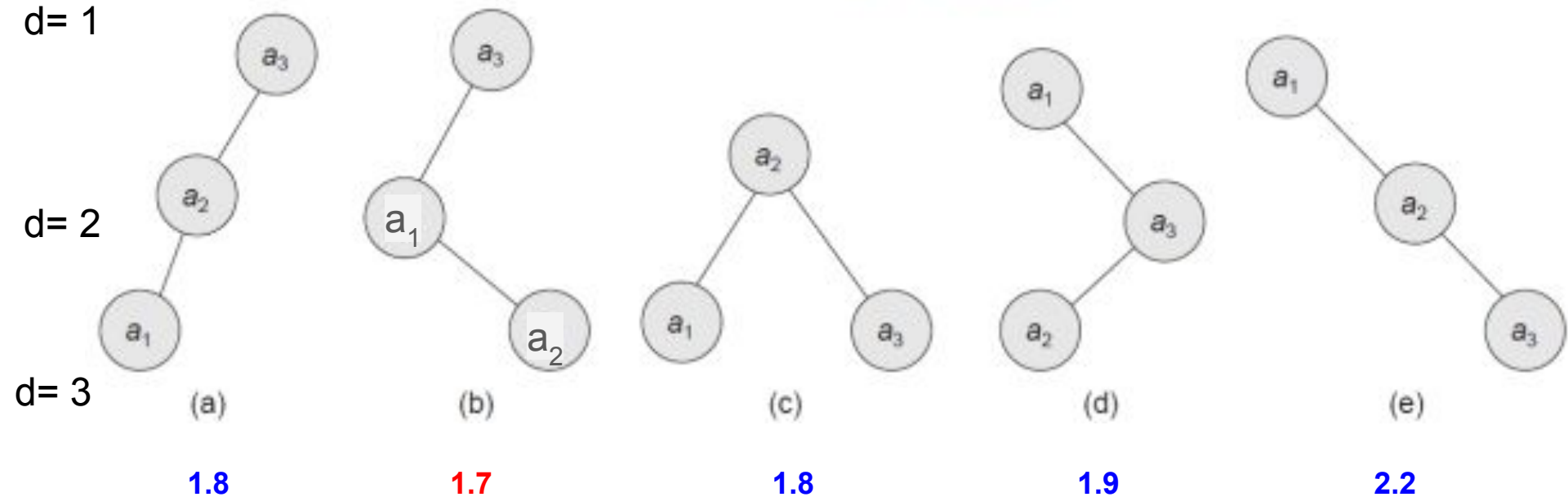
Calculate the search cost based on the cost function :
$$C(T) = \sum_{i=1}^n (p_i \times \text{depth}(k_i))$$



Courtesy : [Infibnet](https://www.infibnet.in/)

Optimal Binary Search Tree Problem

Ex: Construct optimal binary search tree for the three items: $a_1 = 0.3$, $a_2 = 0.2$, $a_3 = 0.5$



1.8

1.7

1.8

1.9

2.2

Courtesy : [Infibnet](https://www.infibnet.com/)

Optimal Binary Search Tree Algorithm using DP Method

Step 1: Define the DP Table

$\text{cost}[i][j]$: minimum cost of the optimal BST for keys k_i, k_{i+1}, \dots, k_j

Step 2 : Base Case For a single key k_i , the cost is simply its probability:

Step 3: Recurrence Relation

Cost of an optimal BST rooted at k_r (where r is a root between i and j) is:

$$\text{cost}[i][j] = \min_{r=i}^j (\text{cost}[i][r-1] + \text{cost}[r+1][j] + \text{sum}(i, j))$$

where,

$\text{cost}[i][r-1]$ is the cost of the left subtree.

$\text{cost}[r+1][j]$ is the cost of the right subtree.

$\text{sum}(i, j)$ is the sum of probabilities from p_i to p_j , calculated as:

$$\text{sum}(i, j) = \sum_{k=i}^j p_k$$

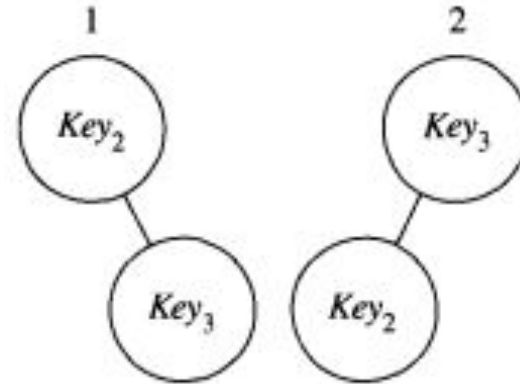
Courtesy : [Infibnet](https://www.infibnet.org/)

Optimal Binary Search Tree Problem

Constructed Table for building Optimal BST

	0	1					j	n
1	0	p_1						*
		0	p_2					
i							$C[i,j]$	
								p_n
n+1								0

To compute $C[2,3]$ of two items, say key 2 and key 3, two possible trees are constructed



Two possible ways of BST for key 2 and key 3

Optimal Binary Search Tree Problem

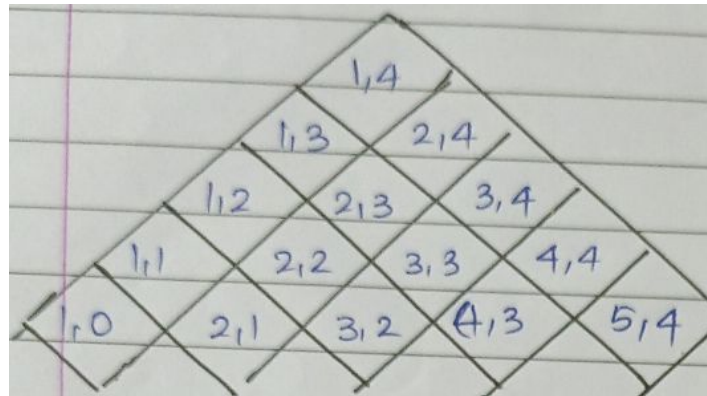
Given four items A (Danny), B(Ian), C (Radha) , and D (zee) with probability $\frac{2}{7}$ $\frac{1}{7}$ $\frac{3}{7}$ $\frac{1}{7}$. Apply dynamic programming approach and construct optimal binary search trees?

	0	1	2	3	4
1	0	$\frac{2}{7}$			
2		0	$\frac{1}{7}$		
3			0	$\frac{3}{7}$	
4				0	$\frac{1}{7}$
5					0

Step - 2 : Base Case

a. Fill all the diagonal elements with 0

b. $\text{cost}[i][i] = p_i$

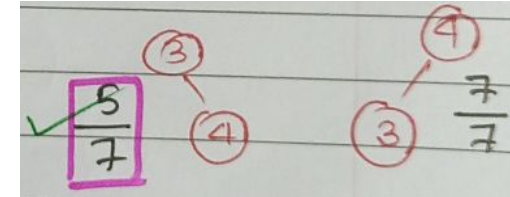
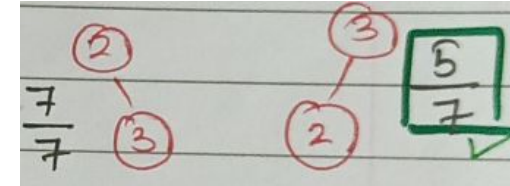
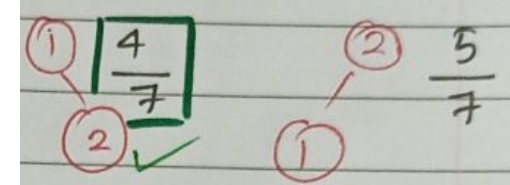
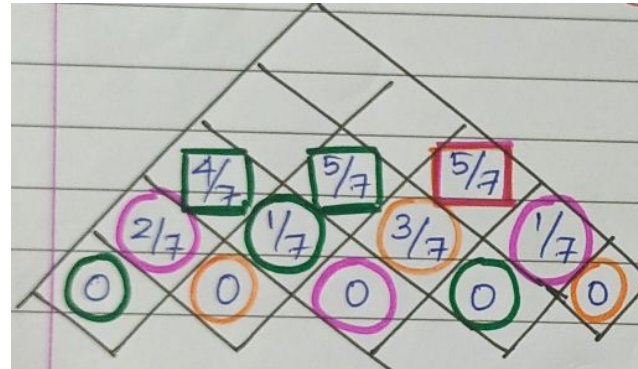
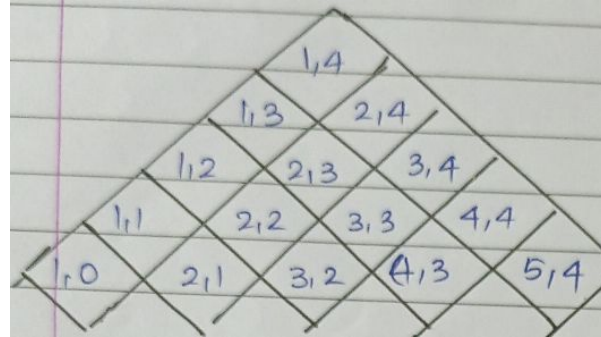


Courtesy : [Radford University](https://www.radford.ac.uk/)

Optimal Binary Search Tree Problem

Given four items A (Danny), B(Ian), C (Radha) , and D (zee) with probability $\frac{2}{7}$ $\frac{1}{7}$ $\frac{3}{7}$ $\frac{1}{7}$. Apply dynamic programming approach and construct optimal binary search trees?

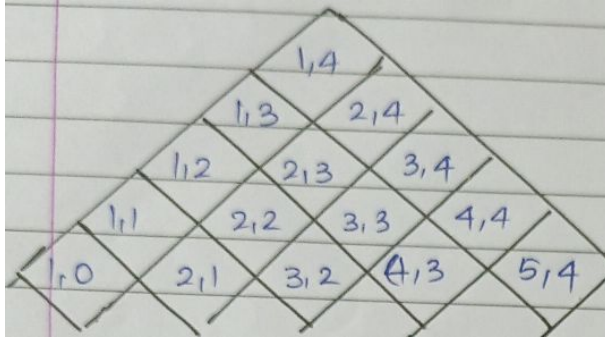
	0	1	2	3	4
1	0	$\frac{2}{7}$ (1)	$\frac{4}{7}$ (1)		
2		0	$\frac{1}{7}$ (1)	$\frac{5}{7}$ (3)	
3			0	$\frac{3}{7}$ (3)	$\frac{5}{7}$ (3)
4				0	$\frac{1}{7}$ (4)
5					0



Courtesy : [Radford University](https://www.radford.ac.uk/)

Optimal Binary Search Tree Problem

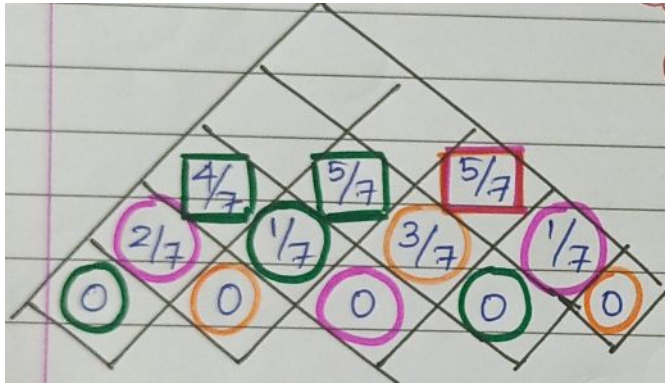
Given four items A (Danny), B(Ian), C (Radha) , and D (zee) with probability $\frac{2}{7}$ $\frac{1}{7}$ $\frac{3}{7}$ $\frac{1}{7}$. Apply dynamic programming approach and construct optimal binary search trees?



$$C[1,2] = \min \left\{ \begin{array}{l} C[1,0] + C[2,2] + p_1 + p_2, \text{ when } k = 1 \\ C[1,1] + C[3,2] + p_1 + p_2, \text{ when } k = 2 \end{array} \right\}$$

$$C[2,3] = \min \left\{ \begin{array}{l} C[2,1] + C[3,3] + p_2 + p_3, \text{ when } k = 2 \\ C[2,2] + C[4,3] + p_2 + p_3, \text{ when } k = 3 \end{array} \right\}$$

$$C[3,4] = \min \left\{ \begin{array}{l} C[3,2] + C[4,4] + p_3 + p_4, \text{ when } k = 3 \\ C[3,3] + C[5,4] + p_3 + p_4, \text{ when } k = 4 \end{array} \right\}$$

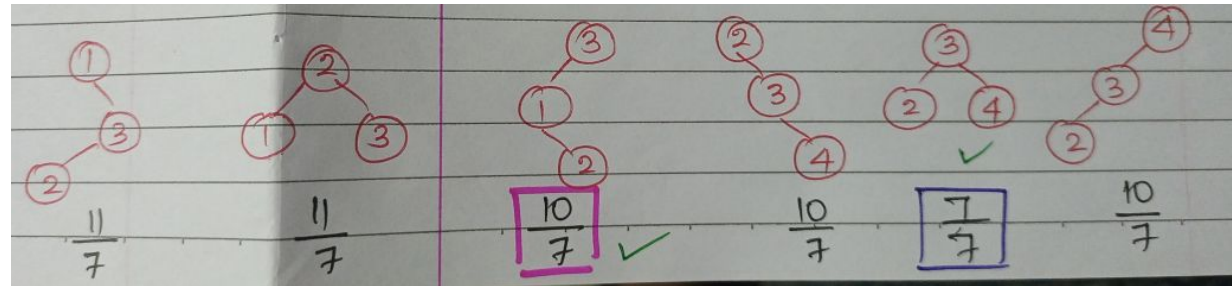
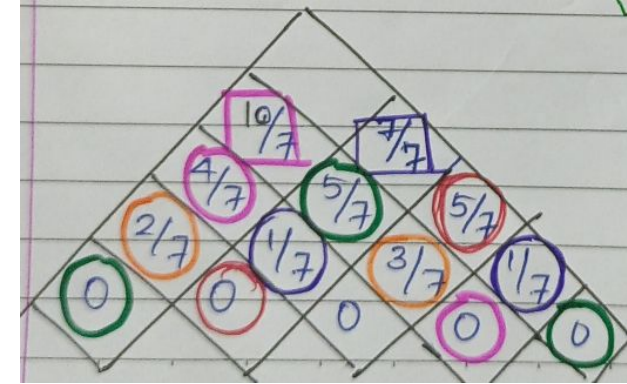
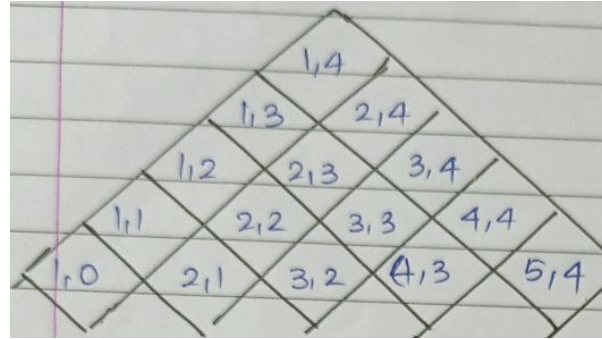


Courtesy : [Radford University](https://www.radford.ac.uk/)

Optimal Binary Search Tree Problem

Given four items A (Danny), B(Ian), C (Radha) , and D (zee) with probability $\frac{2}{7}$ $\frac{1}{7}$ $\frac{3}{7}$ $\frac{1}{7}$. Apply dynamic programming approach and construct optimal binary search trees?

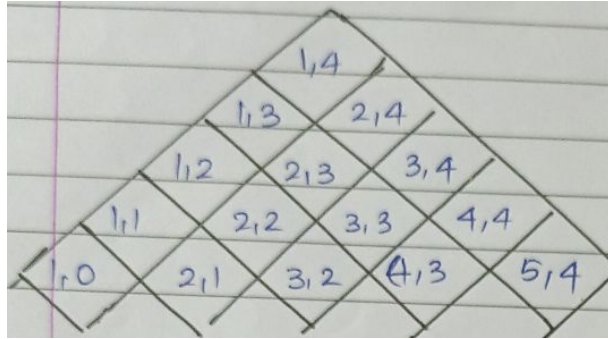
	0	1	2	3	4
1	0	$\frac{2}{7}$ (1)	$\frac{4}{7}$ (1)	$\frac{10}{7}$ (3)	
2		0	$\frac{1}{7}$ (1)	$\frac{5}{7}$ (3)	$\frac{7}{7}$ (3)
3			0	$\frac{3}{7}$ (3)	$\frac{5}{7}$ (3)
4				0	$\frac{1}{7}$ (4)
5					0



Courtesy : [Radford University](https://www.radford.ac.uk/)

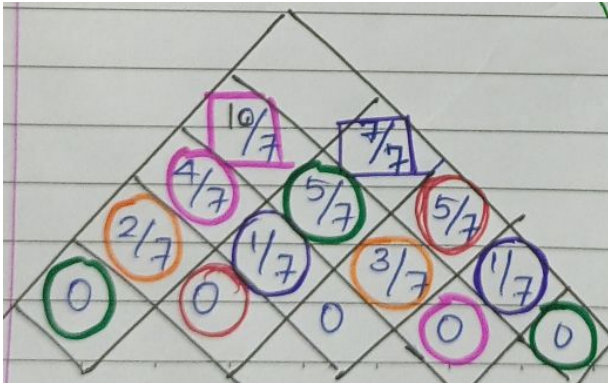
Optimal Binary Search Tree Problem

Given four items A (Danny), B(Ian), C (Radha) , and D (zee) with probability $\frac{2}{7}$ $\frac{1}{7}$ $\frac{3}{7}$ $\frac{1}{7}$. Apply dynamic programming approach and construct optimal binary search trees?



$$C[1,3] = \min \left\{ \begin{array}{l} C[1,0] + C[2,3] + p_1 + p_2 + p_3, \text{ when } k = 1 \\ C[1,1] + C[3,3] + p_1 + p_2 + p_3, \text{ when } k = 2 \\ C[1,2] + C[4,3] + p_1 + p_2 + p_3, \text{ when } k = 3 \end{array} \right\}$$

$$C[1,3] = \min \{ 11/7, 11/7, 10/7 \} = 10/7$$



$$C[2,4] = \min \left\{ \begin{array}{l} C[2,1] + C[3,4] + p_2 + p_3 + p_4, \text{ when } k = 2 \\ C[2,2] + C[4,4] + p_2 + p_3 + p_4, \text{ when } k = 3 \\ C[2,3] + C[5,4] + p_2 + p_3 + p_4, \text{ when } k = 4 \end{array} \right\}$$

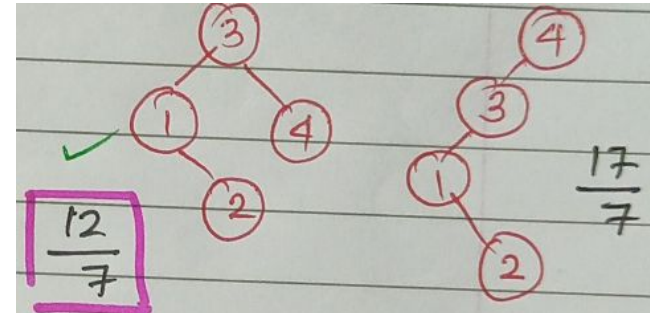
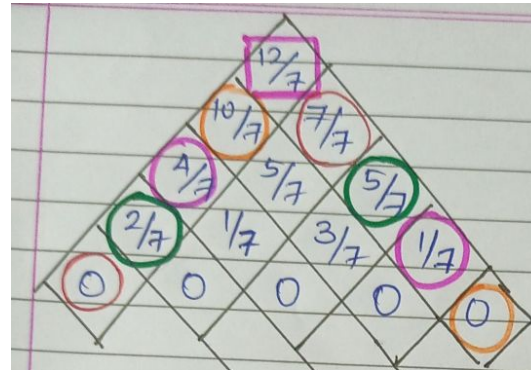
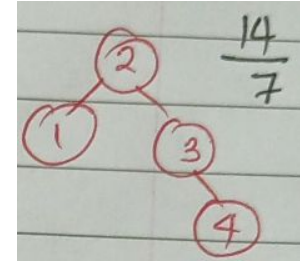
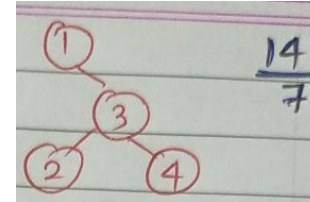
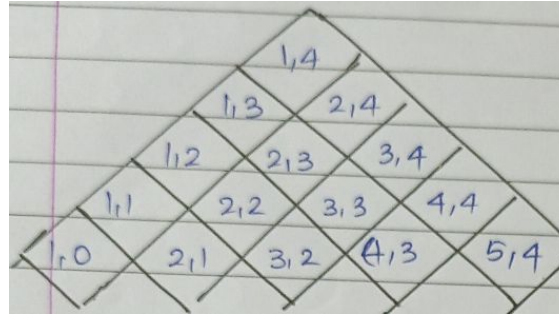
$$C[2,4] = \min \{ 10/7, 7/7, 10/7 \} = 7/7$$

Courtesy : [Radford University](https://www.radford.ac.uk/)

Optimal Binary Search Tree Problem

Given four items A (Danny), B(Ian), C (Radha) , and D (zee) with probability $\frac{2}{7}$ $\frac{1}{7}$ $\frac{3}{7}$ $\frac{1}{7}$. Apply dynamic programming approach and construct optimal binary search trees?

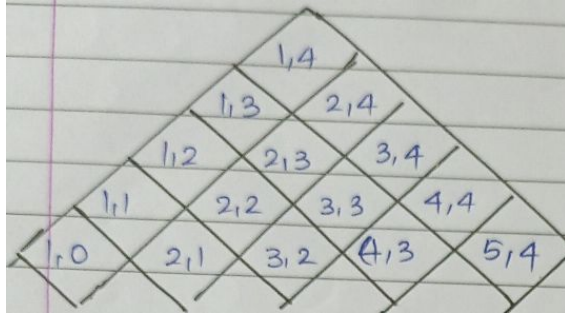
	0	1	2	3	4
1	0	$\frac{2}{7}$ (1)	$\frac{4}{7}$ (1)	$\frac{10}{7}$ (3)	$\frac{12}{7}$ (3)
2		0	$\frac{1}{7}$ (1)	$\frac{5}{7}$ (3)	$\frac{7}{7}$ (3)
3			0	$\frac{3}{7}$ (3)	$\frac{5}{7}$ (3)
4				0	$\frac{1}{7}$ (4)
5					0



Courtesy : [Radford University](https://www.radford.ac.uk/)

Optimal Binary Search Tree Problem

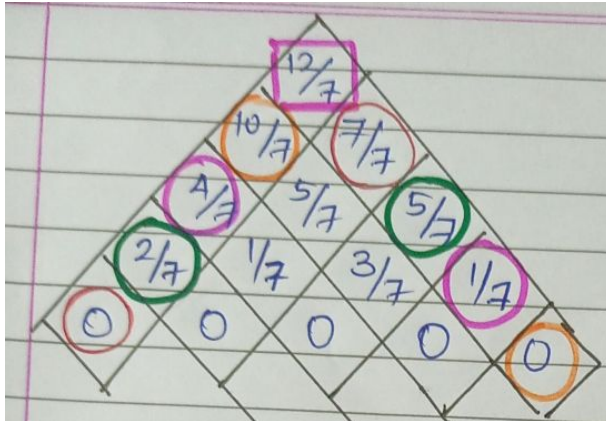
Given four items A (Danny), B(Ian), C (Radha) , and D (zee) with probability $\frac{2}{7}$ $\frac{1}{7}$ $\frac{3}{7}$ $\frac{1}{7}$. Apply dynamic programming approach and construct optimal binary search trees?



$$C[1,4] = \min \left\{ \begin{array}{l} C[1,0] + C[2,4] + p_1 + p_2 + p_3 + p_4, \text{ when } k = 1 \\ C[1,1] + C[3,4] + p_1 + p_2 + p_3 + p_4, \text{ when } k = 2 \\ C[1,2] + C[4,4] + p_1 + p_2 + p_3 + p_4, \text{ when } k = 3 \\ C[1,3] + C[5,4] + p_1 + p_2 + p_3 + p_4, \text{ when } k = 4 \end{array} \right\}$$

$$= \min \{14/7, 14/7, 12/7, 17/7\}$$

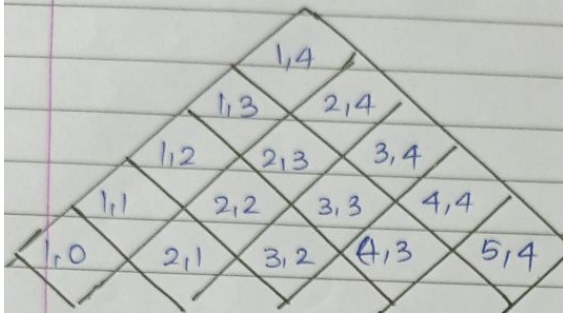
$$= 12/7$$



Courtesy : [Radford University](https://www.radford.ac.uk/)

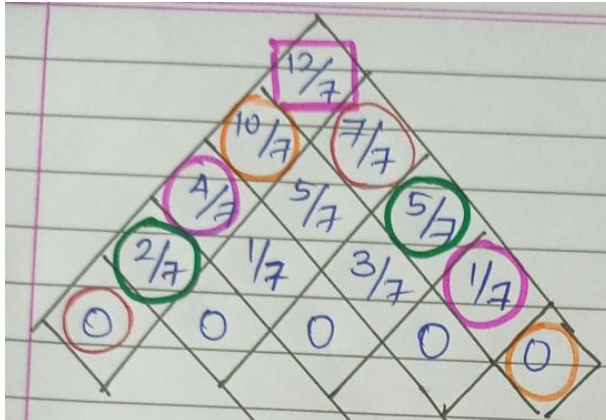
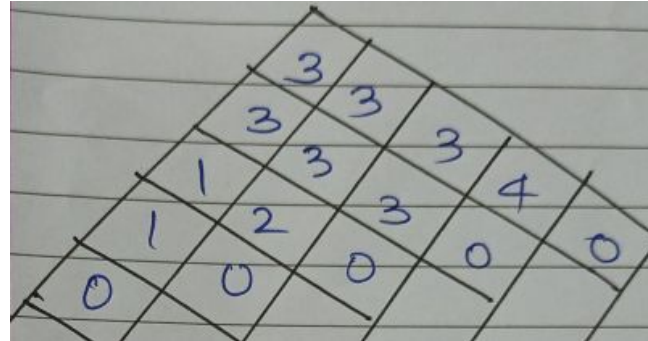
Optimal Binary Search Tree Problem

Given four items A (Danny), B(Ian), C (Radha) , and D (zee) with probability $\frac{2}{7}$ $\frac{1}{7}$ $\frac{3}{7}$ $\frac{1}{7}$. Apply dynamic programming approach and construct optimal binary search trees?



Derive the tree structure

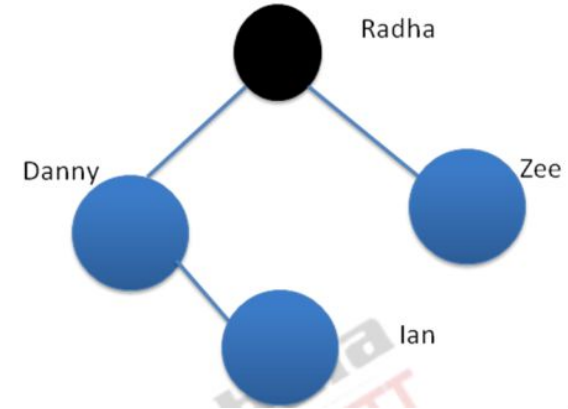
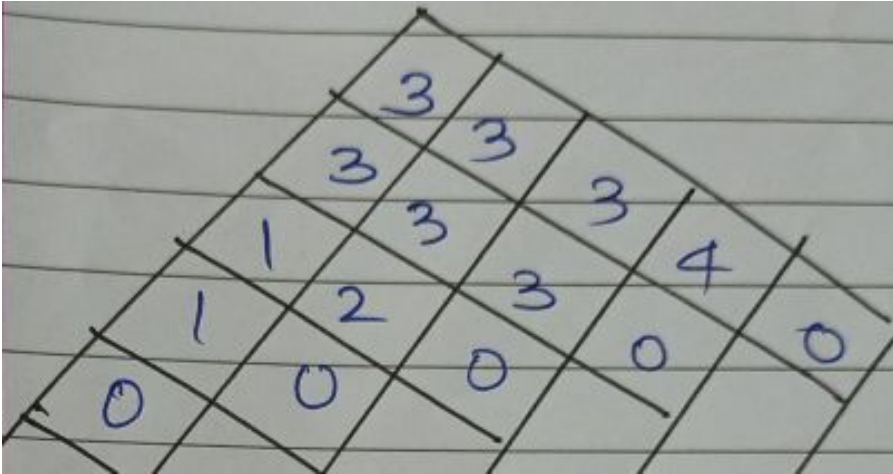
Copy the minimum k in another table



Courtesy : [Radford University](https://www.radford.ac.uk/)

Optimal Binary Search Tree Problem

Given four items A (Danny), B(Ian), C (Radha) , and D (zee) with probability $\frac{2}{7}$ $\frac{1}{7}$ $\frac{3}{7}$ $\frac{1}{7}$. Apply dynamic programming approach and construct optimal binary search trees?



Courtesy : [Radford University](https://www.radford.ac.uk/)

Optimal Binary Search Tree Problem

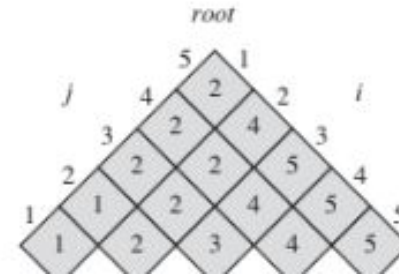
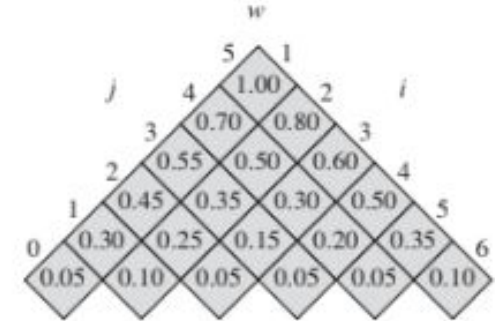
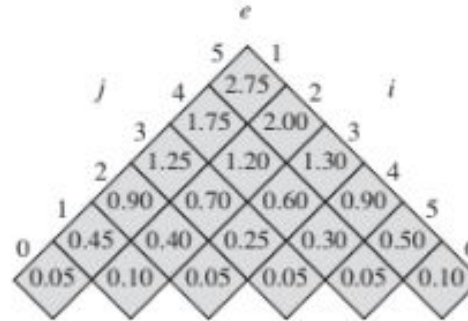
Given the probability table (p_i is the probability of key k_i)

i	1	2	3	4	5
k_i	k_1	k_2	k_3	k_4	k_5
p_i	0.25	0.20	0.05	0.20	0.30

Determine the cost and structure of an optimal binary search tree for a keys with the following probabilities:

i	0	1	2	3	4	5	6	7
p_i		0.04	0.06	0.08	0.02	0.10	0.12	0.14
q_i	0.06	0.06	0.06	0.06	0.05	0.05	0.05	0.05

i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10



<https://medium.com/@kudumalade206039>

Optimal Binary Search Tree Problem - Complexity Analysis

The algorithm involves:

1. Filling an $n \times n$ DP table.
2. Each cell computation requires iterating over all possible root choices.

Thus, the time complexity is:

$$O(n^3)$$

The space complexity is:

$$O(n^2)$$

Courtesy : [Radford University](https://www.radford.ac.uk/)

Topics to be covered

- General Method
- Multistage graphs,
- Single source shortest path: Bellman Ford Algorithm,
- All pair shortest path: Floyd Warshall Algorithm,
- Matrix Chain Multiplication,
- Longest common subsequence,
- Optimal Binary Search Trees,
- 0/1 knapsack Problem

- classic **optimization problem** in which:

- **Problem Statement**

Given n items, each with a **weight** and **value** and a **knapsack** with a maximum weight capacity W .

Goal :

- **Maximize** the total value of items in the knapsack such that,
- The total weight **does not exceed** W .
- An item has to be **included** or **excluded** in its entirety (hence "0/1" Knapsack).

0 / 1 Knapsack Problem

Given the i items with their weights and values. (w_i, v_i) is the weight and value of the i th item

1. **Create a table** $dp[n+1][W+1]$
where $dp[i][w]$: maximum value achievable with i items and a weight limit of w .
2. **Initialize the base cases:**
If $i == 0$ (no items) or $w == 0$ (zero capacity),
 $dp[i][w] = 0$
3. **Fill the table using the recurrence relation.**

$$dp[i][w] = \begin{cases} dp[i-1][w] & \text{if } w_i > w \text{ (item can't be included)} \\ \max(dp[i-1][w], v_i + dp[i-1][w - w_i]) & \text{otherwise} \end{cases}$$

4. **Return** $dp[n][W]$, which contains the optimal solution.

0 / 1 Knapsack Problem

Given the i items with their weights and values. (w_i, v_i) is the weight and value of the i th item

1. Create a table $dp[n+1][W+1]$
where $dp[i][w]$: maximum value achievable with i items and a weight limit of w .
2. Initialize the base cases:
If $i == 0$ (no items) or $w == 0$ (zero capacity),
 $dp[i][w] = 0$
3. Fill the table using the recurrence relation.

$$dp[i][w] = \begin{cases} dp[i-1][w] & \text{if } w_i > w \text{ (item can't be included)} \\ \max(dp[i-1][w], v_i + dp[i-1][w - w_i]) & \text{otherwise} \end{cases}$$

4. Return $dp[n][W]$, which contains the optimal solution.

0 / 1 Knapsack Problem - Example

Number of items: $n = 3$, Weight capacity: $W = 50$

Items (weight, value): (10, 60) , (20, 100), (30, 120)

Items \ W (wt, val)	0	10	20	30	40	50
(10,60)	0	60 (item -1)	60 (item -1)	60 (item -1)	60 (item -1)	60 (item -1)
(20,100)	0	60 (item -1)	100 (item -2)	160 (item -1,2)	160 (item -1,2)	160 (item -1,2)
(30,120)	0	60 (item -1)	100 (item -2)	160 (item -1,2)	180 (Item -1,3)	220 (Item - 2,3)

0 / 1 Knapsack Problem - Example

Number of items: $n = 3$, Weight capacity: $W = 5$

Items (weight, value): (1,1) , (2,7), (3,11)

Items \ W (wt, val)	0	1	2	3	4	5
(1,1)	0	1 (item -1)	1 (item -1)	1 (item -1)	1 (item -1)	1 (item -1)
(2,7)	0	1 (item -1)	7 (item -2)	8 (item -1,2)	8 (item -1,2)	8 (item -1,2)
(3,11)	0	1 (item -1)	7 (item -2)	11 (item -1,2)	12 (Item -1,3)	18 (Item - 2,3)

0 / 1 Knapsack Problem - Example

Number of items: $n = 3$, Weight capacity: $W = 7$

Items (weight, value): (3,4) , (4,5), (7,8)

Items \ W (wt, val)	0	1	2	3	4	5	6	7
(3,4)	0	0	0	4 (item -1)	4 (item -1)	4 (item -1)	4 (item -1)	4 (item -1)
(4,5)	0	0	0	4 (item -1)	5 (item -2)	5 (item -2)	5 (item -2)	9 (item -1,2)
(7,8)	0	0	0	4 (item -1)	5 (item -2)	5 (item -2)	5 (item -2)	9 (Item - 1,2)

0 / 1 Knapsack Problem - Problems

A 0-1 knapsack problem has four items and knapsack capacity 11.
The weight and profit of each item is given in below table.

P_i (Rs.)	50	30	32	27	$W =$ 11
W_i (Kg)	5	6	4	3	



Knapsack
\$ 0
0/10 kg











$$n = 4$$

$$w = 5 \text{ kg}$$

$$(w_1, w_2, w_3, w_4) = (2, 3, 4, 5)$$

$$(b_1, b_2, b_3, b_4) = (3, 4, 5, 6)$$

0 / 1 Knapsack Problem - Problems

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 (\$)															

Maximum Value in Knapsack: \$ 1200

Simulation Courtesy : [w3Schools](https://www.w3schools.com)

0/1 Knapsack Problem using DP method - Complexity

Time Complexity: $O(nW)$

- We fill an $n \times W$ table, where each entry takes $O(1)$ time.

Space Complexity: $O(nW)$

- A dp table of size $(n+1) \times (W+1)$ is used.