Dept. of CSE
Varendra University

Greedy Algorithms: Introduction

Course Instructor:

Sumaiya Tasnim

Lecturer, Department of CSE

Varendra University

Acknowledgement

Mosiur Rahman Sweet
Former Lecturer, Department of CSE
Varendra University

Md. Muktar Hossain
Lecturer, Department of CSE
Varendra University


C
D
E
F
G


SCRABBLE
VNRIALO

Greedy Algorithms

- A greedy algorithm always makes the choice that looks best at the moment. That is, it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution. We will explore some **optimization** problems for which greedy algorithms provide optimal solutions.

- The greedy method is quite powerful and works well for a wide range of problems. **It does not always yield optimal solutions**, but for many problems they do.

- Optimization can be of two types – **Maximization and Minimization.**

Greedy Algorithms

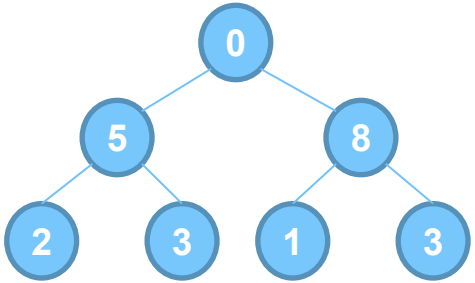
➢ **Advantages**

- They are easier to implement
- They require much less computing resources
- They are much faster to execute
- Greedy algorithms are used to solve optimization problems

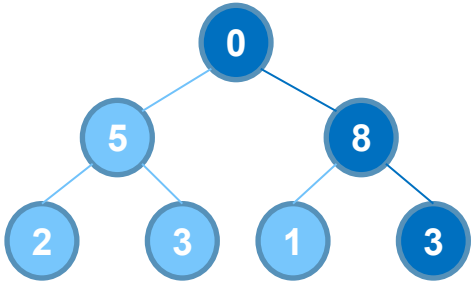
➢ **Disadvantages**

- They don’t always reach the global optimum solution

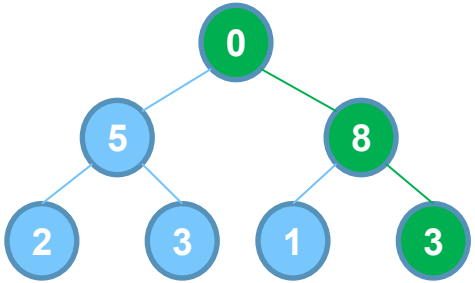
Greedy Choice



Graph

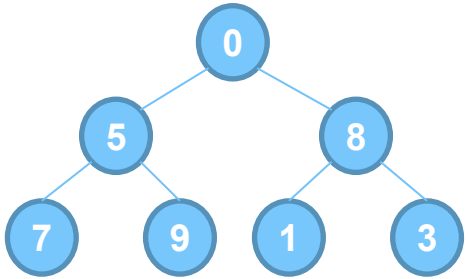


Greedy

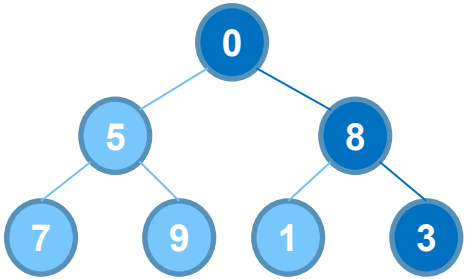


Optimal

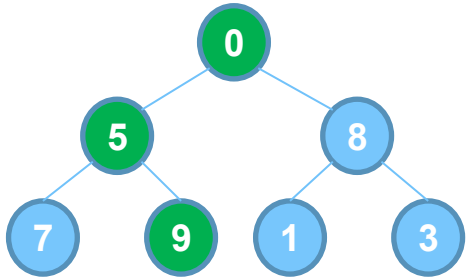
Doesn't always guarantee the best solution…



Graph



Greedy



Optimal

Application of Greedy Design Techniques

01 Fractional Knapsack Problem

02 Activity Selection Problem

03 Job Sequencing Problem

04 Data Compression (Huffman Coding)

05 Minimum Spanning Tree (Kruskal's Algorithm, Prim's Algorithm), Shortest Path (Dijkstra's Algorithm)

Thank You

Greedy Algorithms:
Fractional Knapsack Problem

Dept. of CSE
Varendra University

Course Instructor:

Sumaiya Tasnim

Lecturer, Department of CSE

Varendra University

Acknowledgement

Mosiur Rahman Sweet
Former Lecturer, Department of CSE
Varendra University

Md. Muktar Hossain
Lecturer, Department of CSE
Varendra University

Fractional Knapsack Problem

- You are given a list of products, their values and their weights.
- You have a bag of limited size.

- You need to choose products in such a way that the profit turns out to be **maximum**.

Fractional Knapsack Problem

Product No.	7	6	5	4	3	1	2
Value	120	88	60	36	7	12	12
Weight	12	11	10	9	3	4	6

Bag/Knapsack = 40

Profit = ?

Fractional Knapsack Problem

Product No.	7	6	5	4	3	1	2
Value	120	88	60	36	7	12	12
Weight	12	11	10	9	3	4	6
$\frac{Value}{Weight}$	10	8	6	4	3	3	2
After sorting by $\frac{Value}{Weight}$ in descending order							

Bag= 7-7 = 0

Profit: 268 + (4*7) = 296

Fractional Knapsack Problem

- While knapsack is not full.
  - Choose item i with maximum $v[i]/w[i]$.
  - If the item fits into the knapsack, take all of it.
  - Otherwise take so much as to fill the knapsack.
- Return total value and amounts taken.

- It is to be noted that the list needs to be sorted by $\boldsymbol{Value/Weight}$ in descending order before applying fractional knapsack.

Time Complexity: O(nlogn)

Pseudocode:

```
Function FractionalKnapsack(W, weights[], values[], n)
1. For i = 0 to n-1:
        a. Calculate ratio[i] = values[i] / weights[i]
2. Sort ratio[] in descending order.
3. Initialize totalValue = 0, currentWeight = 0.
4. For each item i in sorted ratio[]:
        a. If currentWeight + weights[i] <= W:
                i. Add values[i] to totalValue.
                ii. Update currentWeight.
        b. Else:
        i. Take fraction of item: totalValue += (W - currentWeight) * ratio[i]
        ii. Break loop.
5. Return totalValue.
End Function
```

Time Complexity

- **Sorting the items**: O(nlogn)
- **Iterating through items**: O(n)
- **Overall Complexity**: O(nlogn)

An Example

- A knapsack is available with a maximum weight capacity of 100 kg.
- There are 6 items, each with a given weight and value.
- The goal is to maximize the total value in the knapsack without exceeding its weight capacity.

Item	Weight (kg)	Value ($)
1	15	120
2	35	300
3	25	200
4	50	500
5	10	90
6	30	400

For i = 0 to n-1:
  Calculate ratio[i] = values[i] / weights[i]

Item	1	2	3	4	5	6
Weight (kg)	15	35	25	50	10	30
Value ($)	120	300	200	500	90	400
v_i/w_i	8.00	8.57	8.00	10.00	9.00	13.33

Sort ratio[] in descending order.

Item	6	4	5	2	1	3
Weight (kg)	30	50	10	35	15	25
Value ($)	400	500	90	300	120	200
v_i/w_i	13.33	10.00	9.00	8.57	8.00	8.00

Initialize totalValue = 0, currentWeight = 0

totalValue = 0
currentWeight = 0

Item	6	4	5	2	1	3
Weight (kg)	30	50	10	35	15	25
Value ($)	400	500	90	300	120	200
v_i/w_i	13.33	10.00	9.00	8.57	8.00	8.00

For each item i in sorted ratio[]:

a. If currentWeight + weights[i] <= W:

i. Add values[i] to totalValue.

ii. Update currentWeight.

b. Else:

i. Take fraction of item:

totalValue += (W - currentWeight) * ratio[i]

ii. Break loop.

W = 100
totalValue = 0
currentWeight = 0

Item	6	4	5	2	1	3
Weight (kg)	30	50	10	35	15	25
Value ($)	400	500	90	300	120	200
v_i/w_i	13.33	10.00	9.00	8.57	8.00	8.00

For each item i in sorted ratio[]:

a. If currentWeight + weights[i] <= W:

i. Add values[i] to totalValue.

ii. Update currentWeight.

b. Else:

i. Take fraction of item:

totalValue += (W - currentWeight) * ratio[i]

ii. Break loop.

W = 100
totalValue = 0
currentWeight = 0

Item	6	4	5	2	1	3
Weight (kg)	30	50	10	35	15	25
Value ($)	400	500	90	300	120	200
v_i/w_i	13.33	10.00	9.00	8.57	8.00	8.00

For each item i in sorted ratio[]:

a. If currentWeight + weights[i] <= W:

i. Add values[i] to totalValue.

ii. Update currentWeight.

b. Else:

i. Take fraction of item:

totalValue += (W - currentWeight) * ratio[i]

ii. Break loop.

W = 100
totalValue = 400
currentWeight = 30

Item	6	4	5	2	1	3
Weight (kg)	30	50	10	35	15	25
Value ($)	400	500	90	300	120	200
v_i/w_i	13.33	10.00	9.00	8.57	8.00	8.00

For each item i in sorted ratio[]:

a. If currentWeight + weights[i] <= W:

i. Add values[i] to totalValue.

ii. Update currentWeight.

b. Else:

i. Take fraction of item:

totalValue += (W - currentWeight) * ratio[i]

ii. Break loop.

W = 100
totalValue = 400
currentWeight = 30

Item	6	4	5	2	1	3
Weight (kg)	30	50	10	35	15	25
Value ($)	400	500	90	300	120	200
v_i/w_i	13.33	10.00	9.00	8.57	8.00	8.00

For each item i in sorted ratio[]:

a. If currentWeight + weights[i] <= W:

i. Add values[i] to totalValue.

ii. Update currentWeight.

b. Else:

i. Take fraction of item:

totalValue += (W - currentWeight) * ratio[i]

ii. Break loop.

W = 100
totalValue = 900
currentWeight = 80

Item	6	4	5	2	1	3
Weight (kg)	30	50	10	35	15	25
Value ($)	400	500	90	300	120	200
v_i/w_i	13.33	10.00	9.00	8.57	8.00	8.00

For each item i in sorted ratio[]:

a. If currentWeight + weights[i] <= W:

i. Add values[i] to totalValue.

ii. Update currentWeight.

b. Else:

i. Take fraction of item:

totalValue += (W - currentWeight) * ratio[i]

ii. Break loop.

W = 100
totalValue = 900
currentWeight = 80

Item	6	4	5	2	1	3
Weight (kg)	30	50	10	35	15	25
Value ($)	400	500	90	300	120	200
v_i/w_i	13.33	10.00	9.00	8.57	8.00	8.00

For each item i in sorted ratio[]:

a. If currentWeight + weights[i] <= W:

i. Add values[i] to totalValue.

ii. Update currentWeight.

b. Else:

i. Take fraction of item:

totalValue += (W - currentWeight) * ratio[i]

ii. Break loop.

W = 100
totalValue = 990
currentWeight = 90

Item	6	4	5	2	1	3
Weight (kg)	30	50	10	35	15	25
Value ($)	400	500	90	300	120	200
v_i/w_i	13.33	10.00	9.00	8.57	8.00	8.00

For each item i in sorted ratio[]:

a. If currentWeight + weights[i] <= W:

i. Add values[i] to totalValue.

ii. Update currentWeight.

b. Else:

i. Take fraction of item:

totalValue += (W - currentWeight) * ratio[i]

ii. Break loop.

W = 100
totalValue = 990
currentWeight = 90

Item	6	4	5	2	1	3
Weight (kg)	30	50	10	35	15	25
Value ($)	400	500	90	300	120	200
v_i/w_i	13.33	10.00	9.00	8.57	8.00	8.00

For each item i in sorted ratio[]:

a. If currentWeight + weights[i] <= W:

i. Add values[i] to totalValue.

ii. Update currentWeight.

b. Else:

i. Take fraction of item:

totalValue += (W - currentWeight) * ratio[i]

ii. Break loop.

W = 100
totalValue = 1075.7
currentWeight = 90

Item	6	4	5	2	1	3
Weight (kg)	30	50	10	35	15	25
Value ($)	400	500	90	300	120	200
v_i/w_i	13.33	10.00	9.00	8.57	8.00	8.00

Return totalValue.

totalValue = 1075.7

W = 100
totalValue = 1075.7
currentWeight = 90

Applications

- **Resource Allocation** – Optimizing project selection, investment planning.
- **Cargo Loading & Logistics** – Maximizing value while staying within weight limits in transport.
- **Manufacturing** – Reducing waste in material cutting (wood, textiles, metals).
- **CPU Scheduling & Memory Management** – Efficient job scheduling in operating systems.
- **Financial Portfolio Optimization** – Selecting the best assets under a budget.
- **Network Bandwidth Allocation** – Distributing bandwidth efficiently in telecommunications.
- **Cryptography** – Used in encryption algorithms (e.g., Merkle–Hellman Knapsack Cryptosystem).
- **Marketing & Ads** – Selecting cost-effective ads under a budget constraint.
- **Robotics & AI** – Optimizing object carrying and path planning.
- **Space Exploration** – Selecting scientific instruments for space missions with weight limits.

Components of Greedy

- **Candidate set:** A solution that is created from the set is known as a candidate set.

- **Selection function:** This function is used to choose the candidate or subset which can be added in the solution.

- **Feasibility function:** A function that is used to determine whether the candidate or subset can be used to contribute to the solution or not.

- **Objective function:** A function is used to assign the value to the solution or the partial solution.

- **Solution function:** This function is used to estimate whether the complete function has been reached or not.

An Example

Problem Statement (Fractional Knapsack) Given ***n*** items, each with a weight ***w[i]*** and a value ***v[i]***, and a knapsack that can carry a maximum weight ***W***, determine the maximum value that can be obtained by putting items (or fractions of them) into the knapsack.

Candidate set: The set of all items *{(w[i], v[i])}* that can be put into the knapsack.
Selection function: Select the item with the highest value-to-weight ratio *(v[i] / w[i])*.

Feasibility function: Before adding an item, check if adding the whole item exceeds the weight capacity W:

- If yes, take only a fraction of it.
- If no, add the whole item.

Objective function:

- Keep track of the total value accumulated in the knapsack.
- Stop when the knapsack is full (or all items are considered).

Solution function: The problem can be broken down into smaller subproblems:

- If we remove the most valuable item, the remaining problem is the same as before but with a smaller capacity.
- The optimal solution to the subproblem contributes to the optimal solution of the main problem.

Thank You

Greedy Algorithms:
Activity Selection Problem

Dept. of CSE
Varendra University

Course Instructor:

Sumaiya Tasnim

Lecturer, Department of CSE

Varendra University

Acknowledgement

Mosiur Rahman Sweet
Former Lecturer, Department of CSE
Varendra University

Md. Muktar Hossain
Lecturer, Department of CSE
Varendra University

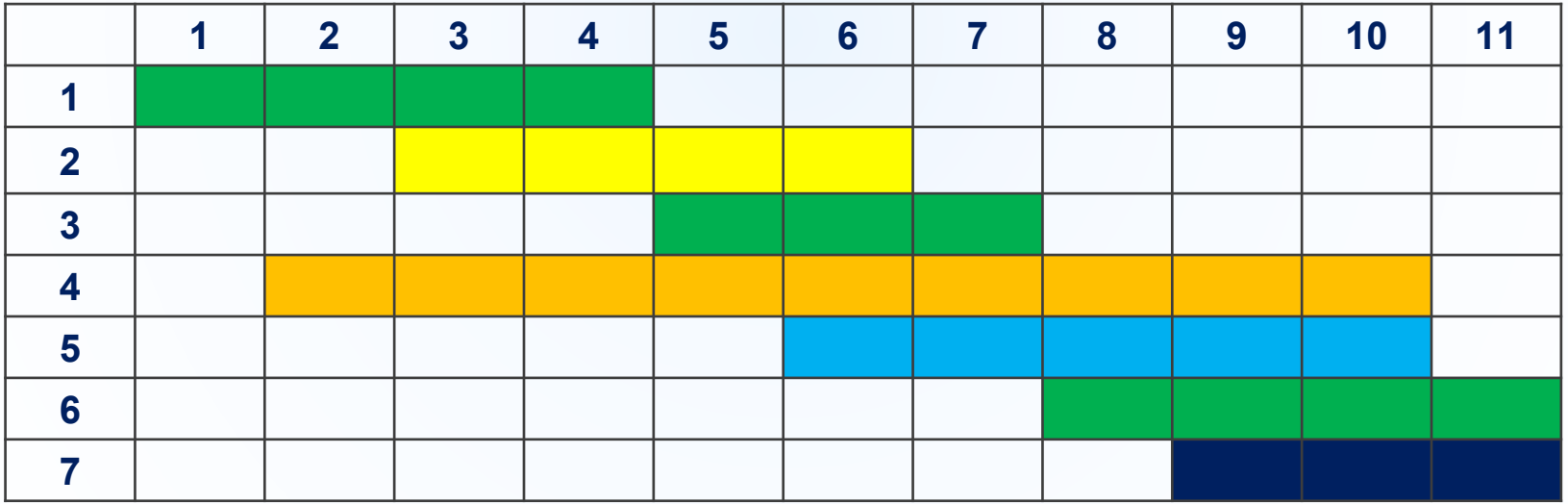
Activity Selection Problem

- It is a problem of scheduling several activities that require exclusive use of a common resource, with the goal of selecting a maximum size set of **mutually compatible activities.**

Activity	1	2	3	4	5	6	7
Start	1	3	5	2	6	8	9
Finish	4	6	7	9	10	11	11
After sorting by finish time in ascending order							

Activity Selection Problem

Activity	1		3		6	
Start	1		5		8	
Finish	4		7		11	



Activity Selection Algorithm

- Sort the activities by their finishing times (in ascending order).
- Select the first activity (which finishes the earliest).
- Iterate through the remaining activities:
  - If the start time of the current activity is greater than or equal to the finish time of the last selected activity, select it.
- Continue until all activities are checked.

Problem Statement

Given ***n*** activities with their ***start*** and ***finish*** times, the goal is to select the ***maximum number of activities*** that can be performed by a single person, assuming that the person can work on only one activity at a time.

Input

- An integer **N** representing the number of activities.
- An array **start[ ]** of size **N** where **start[i]** is the start time of the **i-th** activity.
- An array **finish[ ]** of size **N** where **finish[i]** is the finish time of the **i-th** activity.

Objective

- Select the maximum number of activities that can be performed without any overlapping.

Constraints

- $1 \leq N \leq 10^5$ (large input sizes possible)
- $1 \leq start[i], finish[i] \leq 10^9$

Pseudocode:

```
ACTIVITY_SELECTION(start[], finish[], N)
    Sort activities based on finish times
    Select the first activity and print it
    lastSelected = 0

    FOR i = 1 to N-1 DO
        IF start[i] >= finish[lastSelected] THEN
            Select activity i and print it
            lastSelected = i
        ENDIF
    END FOR
END
```

Time Complexity

- **Sorting the activities**: O(nlogn)
- **Iterating through activities**: O(n)
- **Overall Complexity**: O(nlogn)

An Example

A person is given a set of activities, each with a specific start and finish time.

The goal is to select the maximum number of non-overlapping activities that can be performed by a single person.

Activity	Start Time	Finish Time
1	1	3
2	2	5
3	4	6
4	6	8
5	5	7
6	8	9

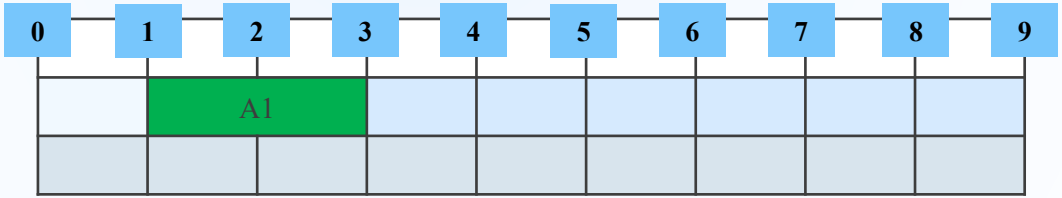
Sort activities based on finish times

Activity	1	2	3	5	4	6
Start Time	1	2	2	5	6	8
Finish Time	3	5	6	7	8	9

Select the first activity and print it

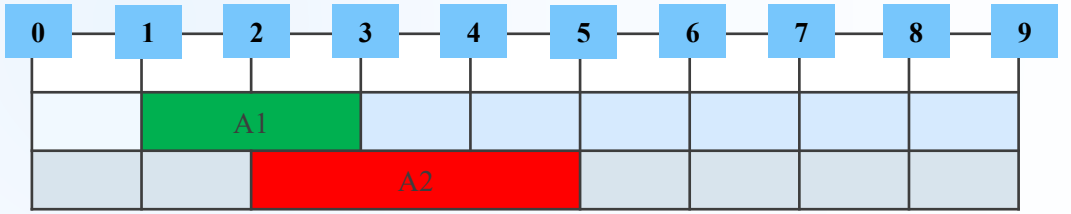
lastSelected = 0

Activity	1	2	3	5	4	6
Start Time	1	2	2	5	6	8
Finish Time	3	5	6	7	8	9



```
FOR i = 1 to N-1 DO
    IF start[i] >= finish[lastSelected] THEN
        Select activity i and print it
        lastSelected = i
    ENDIF
END FOR
```

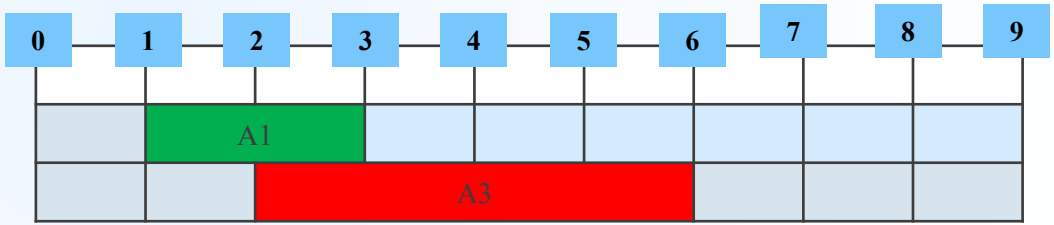
Activity	1	2	3	5	4	6
Start Time	1	2	2	5	6	8
Finish Time	3	5	6	7	8	9



Activity	1	2	3	5	4	6
Start Time	1	2	2	5	6	8
Finish Time	3	5	6	7	8	9

```
FOR i = 1 to N-1 DO
    IF start[i] >= finish[lastSelected] THEN
       Select activity i and print it
       lastSelected = i
    ENDIF
  END FOR
```

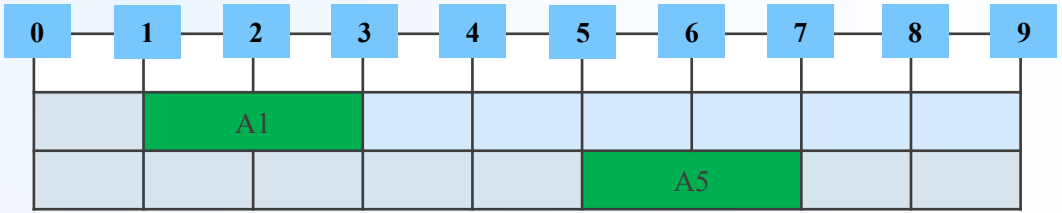
Activity	1	2	3	5	4	6
Start Time	1	2	2	5	6	8
Finish Time	3	5	6	7	8	9



Activity	1	2	3	5	4	6
Start Time	1	2	2	5	6	8
Finish Time	3	5	6	7	8	9

```
FOR i = 1 to N-1 DO
    IF start[i] >= finish[lastSelected] THEN
        Select activity i and print it
        lastSelected = i
    ENDIF
END FOR
```

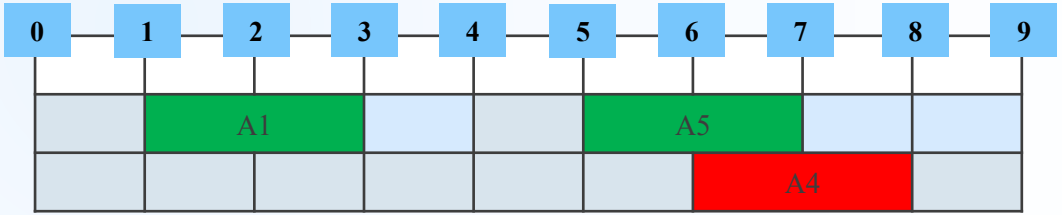
Activity	1	2	3	5	4	6
Start Time	1	2	2	5	6	8
Finish Time	3	5	6	7	8	9



Activity	1	2	3	5	4	6
Start Time	1	2	2	5	6	8
Finish Time	3	5	6	7	8	9

Activity	1	2	3	5	4	6
Start Time	1	2	2	5	6	8
Finish Time	3	5	6	7	8	9

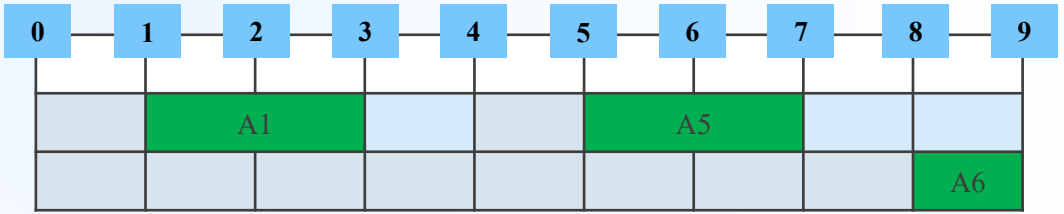
```
FOR i = 1 to N-1 DO
    IF start[i] >= finish[lastSelected] THEN
        Select activity i and print it
        lastSelected = i
    ENDIF
END FOR
```



Activity	1	2	3	5	4	6
Start Time	1	2	2	5	6	8
Finish Time	3	5	6	7	8	9

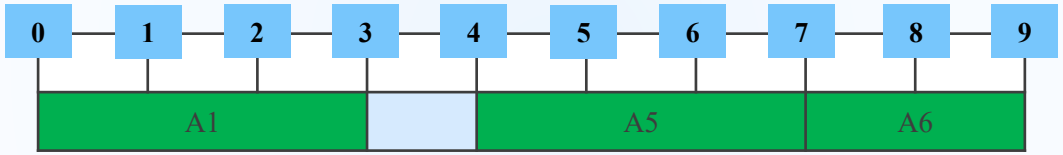
```
FOR i = 1 to N-1 DO
    IF start[i] >= finish[lastSelected] THEN
        Select activity i and print it
        lastSelected = i
    ENDIF
END FOR
```

Activity	1	2	3	5	4	6
Start Time	1	2	2	5	6	8
Finish Time	3	5	6	7	8	9



Activity	1	2	3	5	4	6
Start Time	1	2	2	5	6	8
Finish Time	3	5	6	7	8	9

Example Cont’d



Applications

- **Job scheduling:** Scheduling jobs or tasks where each task has a deadline.
- **Conference room scheduling:** Allocating meeting rooms based on time slots.
- **CPU scheduling:** Selecting processes that can run in a non-overlapping manner.
- **Event planning:** Scheduling events with time constraints.

Thank You

Huffman Coding

MD. MUKTAR HOSSAIN

LECTURER

DEPT. OF CSE

VARENDRA UNIVERSITY

Data Compression

Data compression is the process of reducing the size of data to save storage space or transmission time.

Types of Data Compression

- Lossless Compression
  - Retains the original data exactly after decompression.
  - Used for text files, program files, and medical imaging.
  - Examples: Huffman Coding, Run-Length Encoding
- Lossy Compression
  - Some data is lost, but the reduction in size is significant.
  - Used for images, audio, and videos where perfect accuracy isn't needed.
  - Examples: JPEG, MP3, MPEG, H.264, H.265

Character Encoding Compression

Encoding characters is a key part of lossless compression algorithms, where data is compressed without losing any information, and the original data can be perfectly reconstructed during decompression.

Types of Character Encoding

- Fixed-Length Encoding
  - Each character or symbol is represented using a fixed number of bits.
- Variable-Length Encoding
  - Different characters or symbols are represented using a varying number of bits.
  - More frequent symbols get shorter codes, while less frequent ones get longer codes.

Fixed-Length Encoding

M = “Sleeplessness stresses restless senses endlessly”

s	e	l	sp	n	t	r	p	d	y
17	12	5	4	3	2	2	1	1	1

Total number of characters = 48

ASCII

- Uses 7-bit codes to represent characters (A-Z, a-z, 0-9, symbols).
- Required bits to encode **M** = 48*7 = 336 bits

Minimal Fixed Code

- Since there is 10 different character, we can use 4 bits for each character to encode message M.
- Required bits to encode **M** = 48*4 = 192 bits

Variable-Length Encoding

M = “Sleeplessness stresses restless senses endlessly”

s	e	l	sp	n	t	r	p	d	y
17	12	5	4	3	2	2	1	1	1

Total number of characters = 48

Huffman Coding can be used for variable length encoding.

Huffman Coding - Introduction

- Huffman coding is a greedy algorithm used for lossless data compression.
- It assigns variable-length binary codes to input characters. [shorter codes assigned to more frequent characters and longer codes assigned to less frequent ones]
- This minimizes the overall length of the encoded message.
- The core idea is to build a binary tree (Huffman tree) where each leaf node represents a character and its frequency. The tree is then traversed to assign binary codes.

Huffman Coding - Applications

- **File Storage –** Reduces disk space usage.
- **Network Transmission –** Speeds up data transfer.
- **Multimedia Streaming –** Reduces bandwidth requirements.
- **Cloud Storage –** Saves costs for large-scale data storage.
- **Data Backup & Archiving –** Efficiently stores historical data.

Huffman Coding - Algorithm

- Count the frequency of each character in the input.
- Insert all characters and their frequencies into a priority queue (min-heap).
- While there is more than one node in the queue:
  - Remove the two nodes with the lowest frequency.
  - Create a new internal node with these two as children and a frequency equal to their sum.
  - Insert the new node back into the priority queue.
- The last remaining node is the root of the Huffman tree.
- Generate binary codes by traversing the Huffman tree.

Huffman Coding - Example

M = “Sleeplessness stresses restless senses endlessly”

Count the frequency of each character in the input.

s	l	e	p	n	sp	t	r	d	y
17	5	12	1	3	4	2	2	1	1

Insert all characters and their frequencies into a priority queue (min-heap).

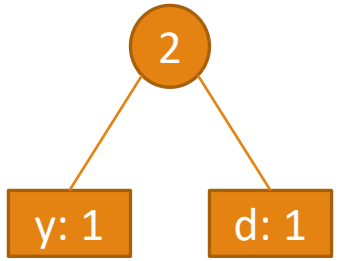
y: 1	d: 1	p: 1	r: 2	t: 2	n: 3	sp: 4	l: 5	e: 12	s: 17

Huffman Coding - Example

y: 1	d: 1	p: 1	r: 2	t: 2	n: 3	sp: 4	l: 5	e: 12	s: 17

- While there is more than one node in the queue:
  - Remove the two nodes with the lowest frequency.
  - Create a new internal node with these two as children and a frequency equal to their sum.
  - Insert the new node back into the priority queue.

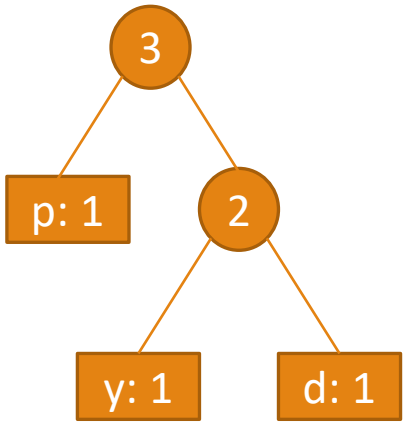
p: 1	2	r: 2	t: 2	n: 3	sp: 4	l: 5	e: 12	s: 17



Huffman Coding - Example

p: 1	2	r: 2	t: 2	n: 3	sp: 4	l: 5	e: 12	s: 17

- While there is more than one node in the queue:
  - Remove the two nodes with the lowest frequency.
  - Create a new internal node with these two as children and a frequency equal to their sum.
  - Insert the new node back into the priority queue.

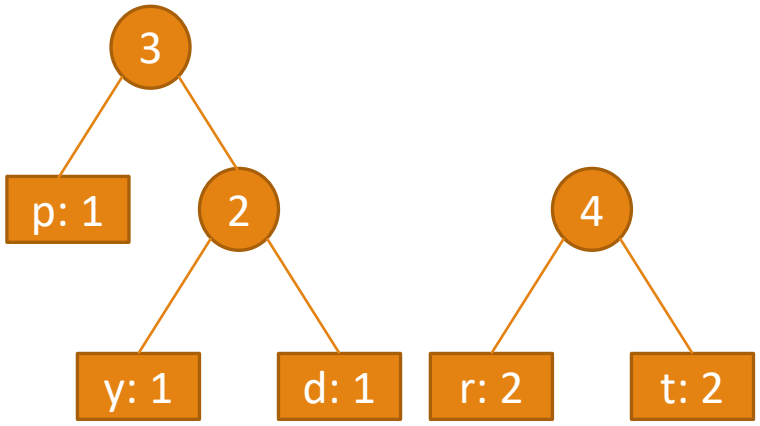


r: 2	t: 2	3	n: 3	sp: 4	l: 5	e: 12	s: 17

Huffman Coding - Example

r: 2	t: 2	3	n: 3	sp: 4	l: 5	e: 12	s: 17

- While there is more than one node in the queue:
  - Remove the two nodes with the lowest frequency.
  - Create a new internal node with these two as children and a frequency equal to their sum.
  - Insert the new node back into the priority queue.

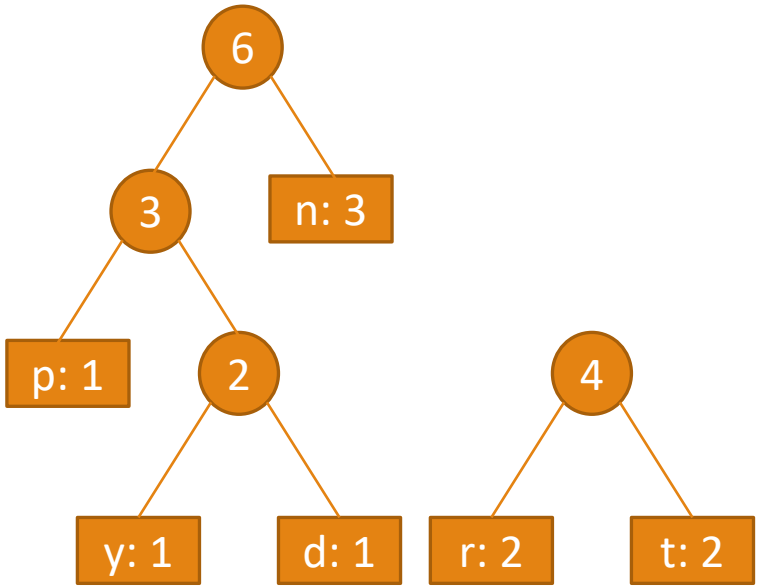


3	n: 3	4	sp: 4	l: 5	e: 12	s: 17

Huffman Coding - Example

3	n: 3	4	sp: 4	l: 5	e: 12	s: 17

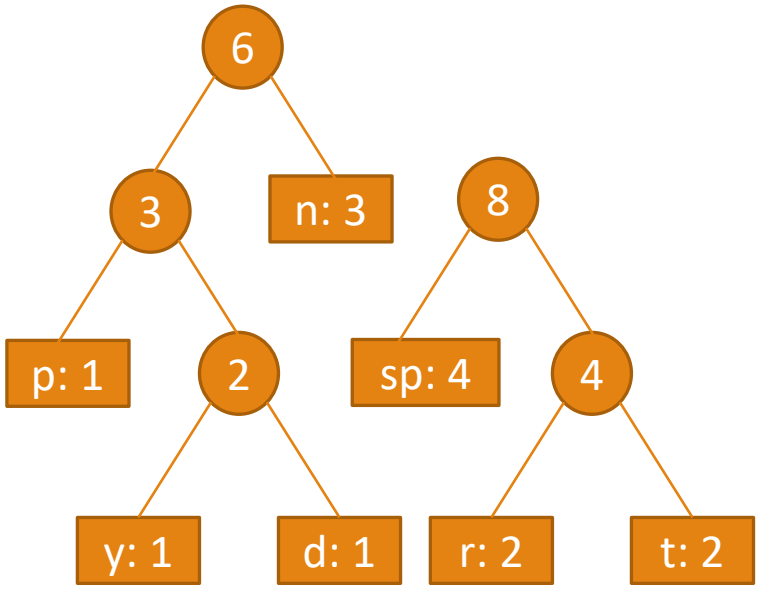
- While there is more than one node in the queue:
  - Remove the two nodes with the lowest frequency.
  - Create a new internal node with these two as children and a frequency equal to their sum.
  - Insert the new node back into the priority queue.



4	sp: 4	l:5	6	e: 12	s: 17

Huffman Coding - Example

4	sp: 4	l:5	6	e: 12	s: 17



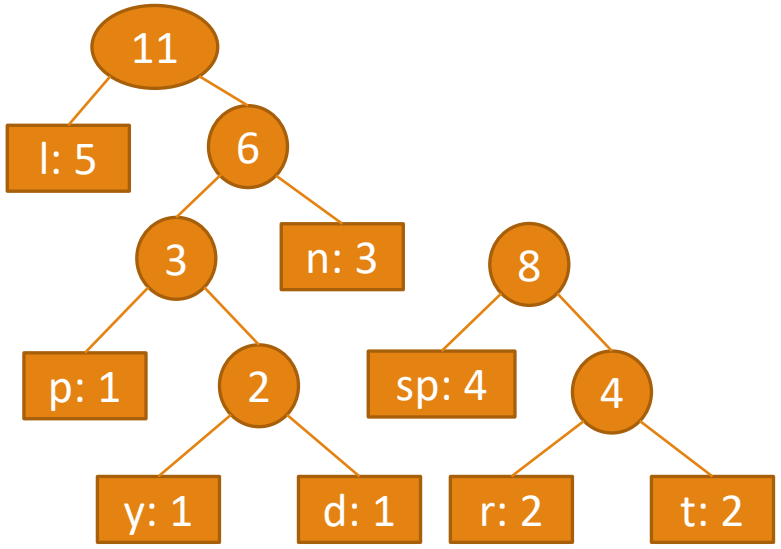
- While there is more than one node in the queue:
  - Remove the two nodes with the lowest frequency.
  - Create a new internal node with these two as children and a frequency equal to their sum.
  - Insert the new node back into the priority queue.

l: 5	6	8	e: 12	s: 17

Huffman Coding - Example

l: 5	6	8	e: 12	s: 17

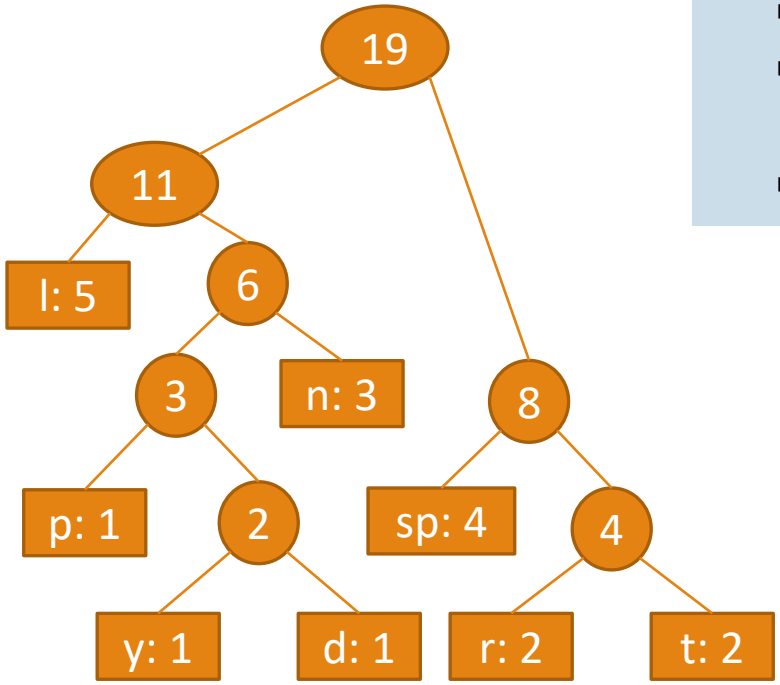
- While there is more than one node in the queue:
  - Remove the two nodes with the lowest frequency.
  - Create a new internal node with these two as children and a frequency equal to their sum.
  - Insert the new node back into the priority queue.



8	11	e: 12	s: 17

Huffman Coding - Example

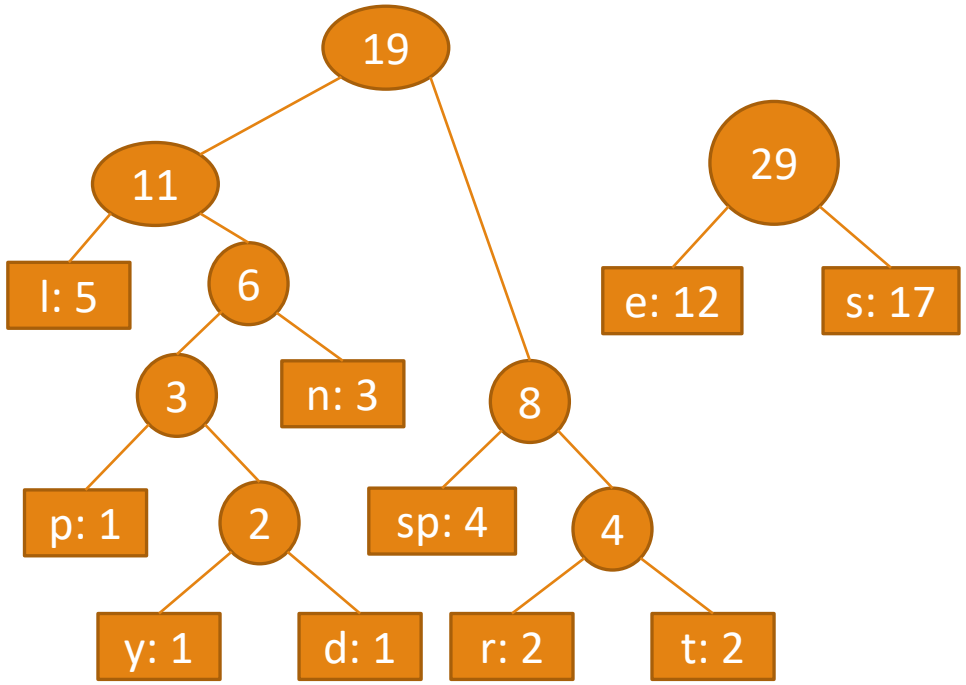
8	11	e: 12	s: 17



- While there is more than one node in the queue:
  - Remove the two nodes with the lowest frequency.
  - Create a new internal node with these two as children and a frequency equal to their sum.
  - Insert the new node back into the priority queue.

e: 12	s: 17	19

Huffman Coding - Example

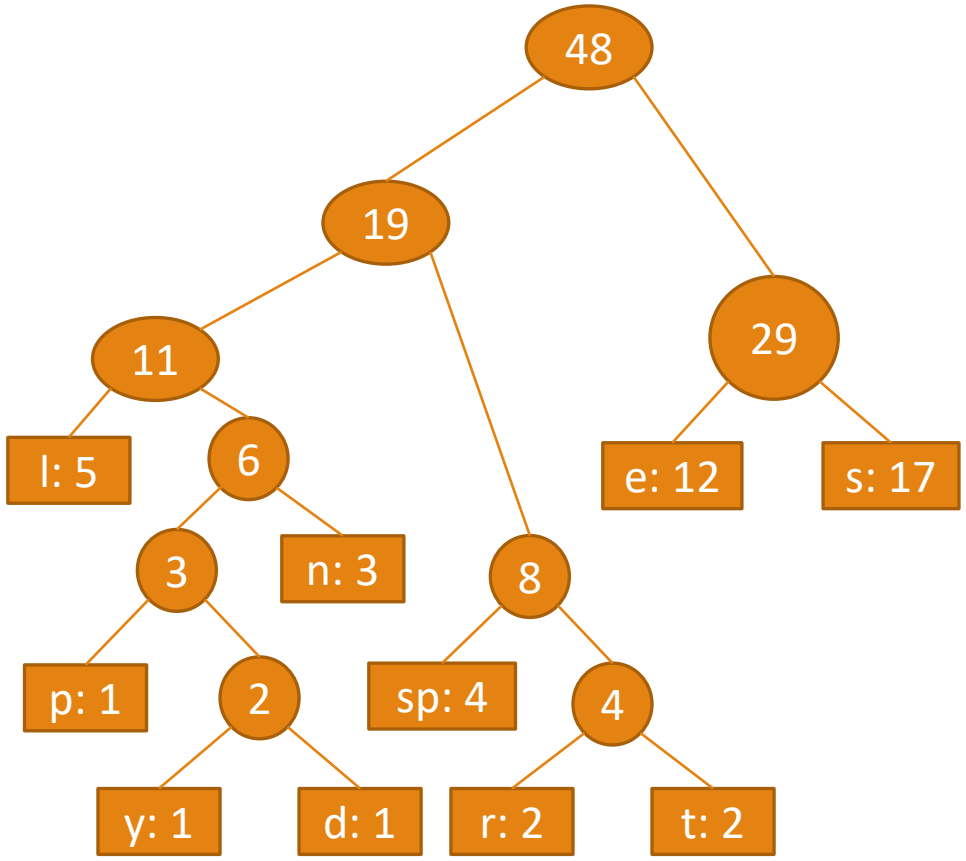


e: 12	s: 17	19

- While there is more than one node in the queue:
  - Remove the two nodes with the lowest frequency.
  - Create a new internal node with these two as children and a frequency equal to their sum.
  - Insert the new node back into the priority queue.

19	29

Huffman Coding - Example



19	29

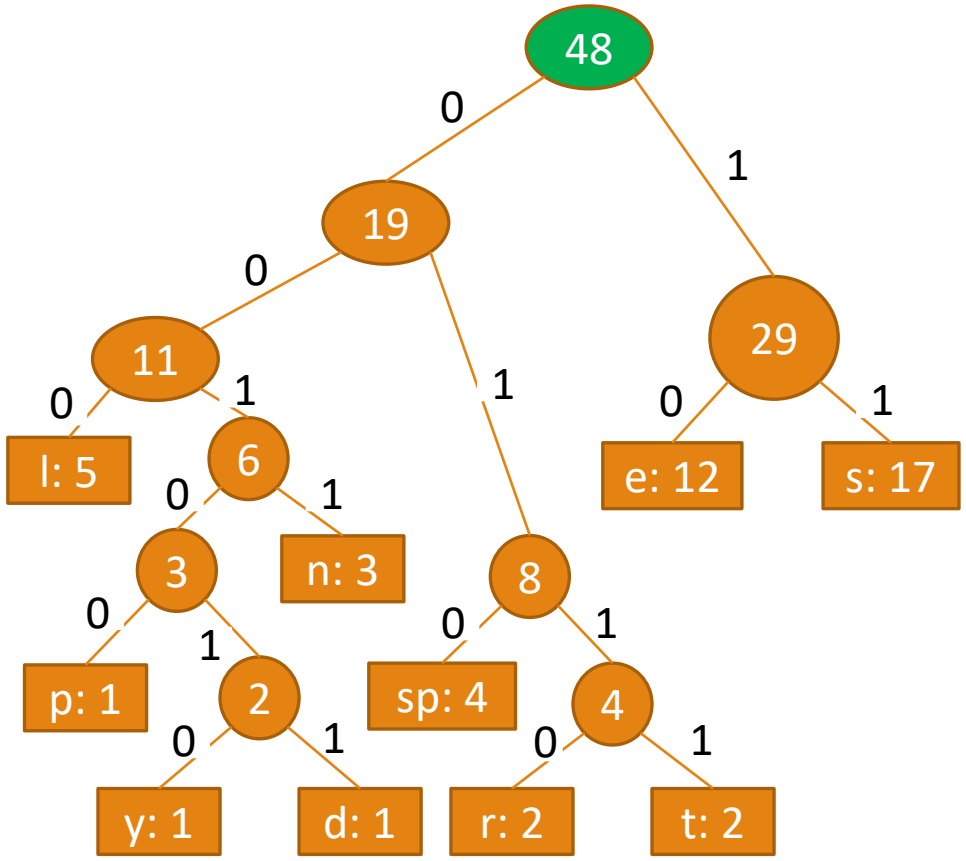
- While there is more than one node in the queue:
  - Remove the two nodes with the lowest frequency.
  - Create a new internal node with these two as children and a frequency equal to their sum.
  - Insert the new node back into the priority queue.

48

The last remaining node is the root of the Huffman tree.

48

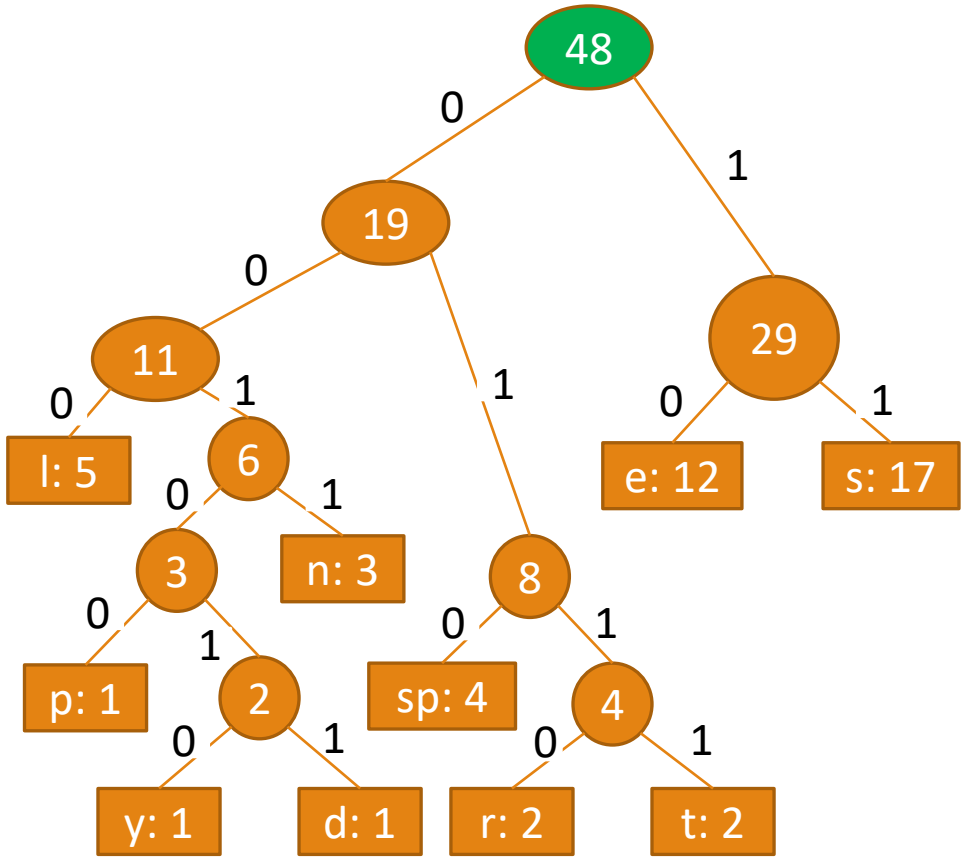
Huffman Coding - Example



Generate binary codes by traversing the Huffman tree.

Character	Codes
s	11
l	000
e	10
p	00100
n	0011
sp	010
t	0111
r	0110
d	001011
y	001010

Huffman Coding - Example



Generate binary codes by traversing the Huffman tree.

Char	Codes	Freq	Required Bits
s	11	17	2*17 = 34
l	000	5	3*5 = 15
e	10	12	2*12 = 24
p	00100	1	5*1 = 5
n	0011	3	4*3 = 12
sp	010	4	3*4 = 12
t	0111	2	4*2 = 8
r	0110	2	4*2 = 8
d	001011	1	6*1 = 6
y	001010	1	6*1 = 6

Required Total Bits = **130** bits

Average Bits per Character

$$= \frac{130}{48} bits/char$$

$$= \mathbf{2.708}\ bits/char$$

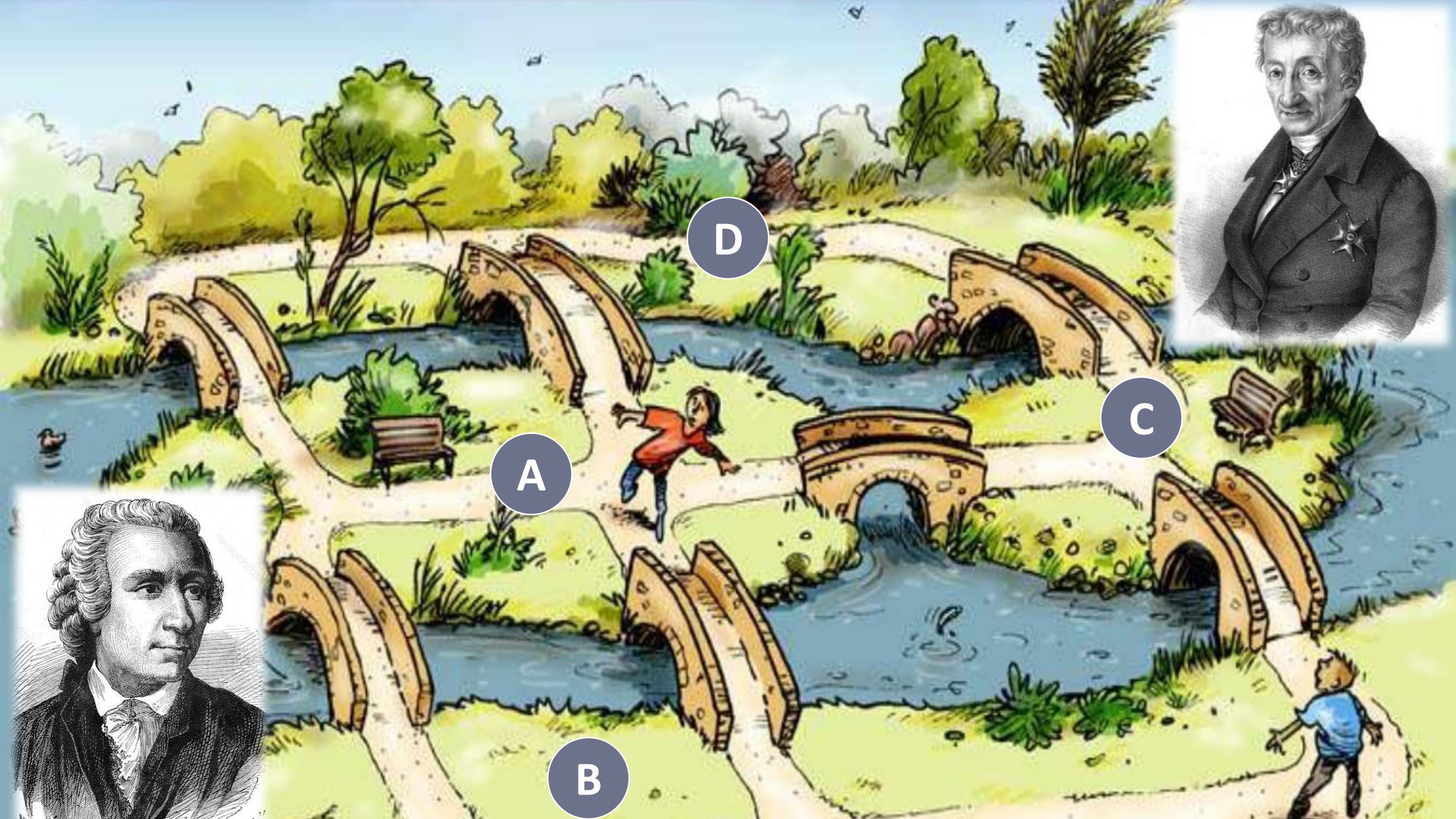
Home Task:

Analysis the Time Complexity of Huffman Coding.

THANK YOU

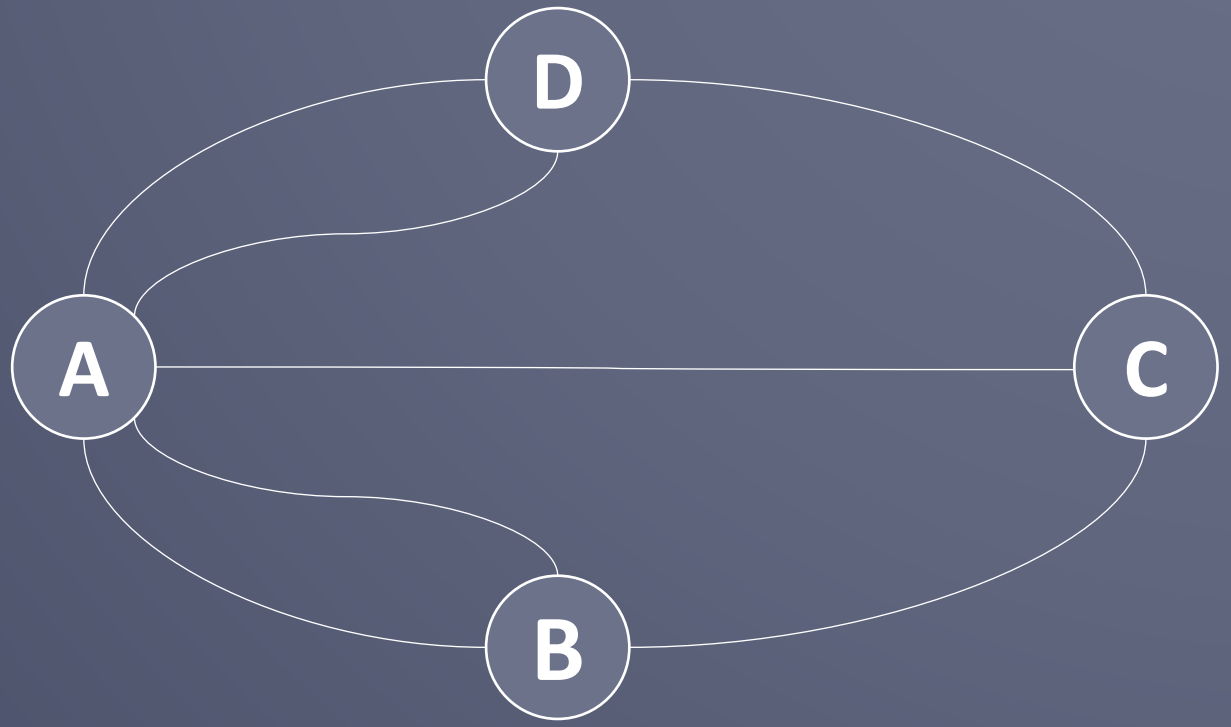
GRAPH THEORY

Mosiur Rahman Sweet
Lecturer (Former)
Department of Computer Science and Engineering
Varendra University


D
A
C
B

Bridge Of Konigsberg

DEGREE OF A : 5
DEGREE OF B: 3
DEGREE OF C: 3
DEGREE OF D: 3



Euler Path and Circuit

- A graph has an Euler path if and only if there are at most two vertices with odd degree.

- A graph has an Euler circuit if and only if the degree of every vertex is even.

The Geometry of Position

The Geometry of Position aka Graph Theory is the study of graphs, which are mathematical structures used to model pairwise relations between objects.

A graph is a collection of **vertices** (also called nodes or points) which are connected by **edges** (also called links or lines) to represent complex problems in a simpler manner.

Adjacent Node

Any two nodes connected by an edge or any two edges connected by a node are said to be adjacent.

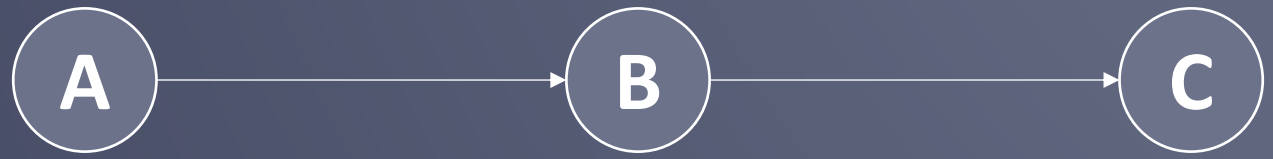
In this graph, A & C both are the adjacent node of B.

B is the adjacent node of A

Also, B is the adjacent node of C

Directed & Undirected Graph

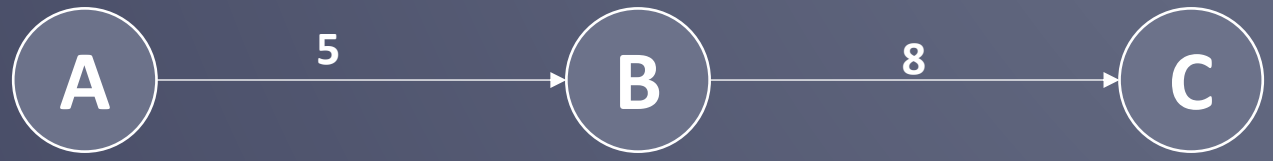
Directed graph are the ones with unidirectional edges.



Undirected graph are the ones with bidirectional edges.

Weighted & Unweighted Graph

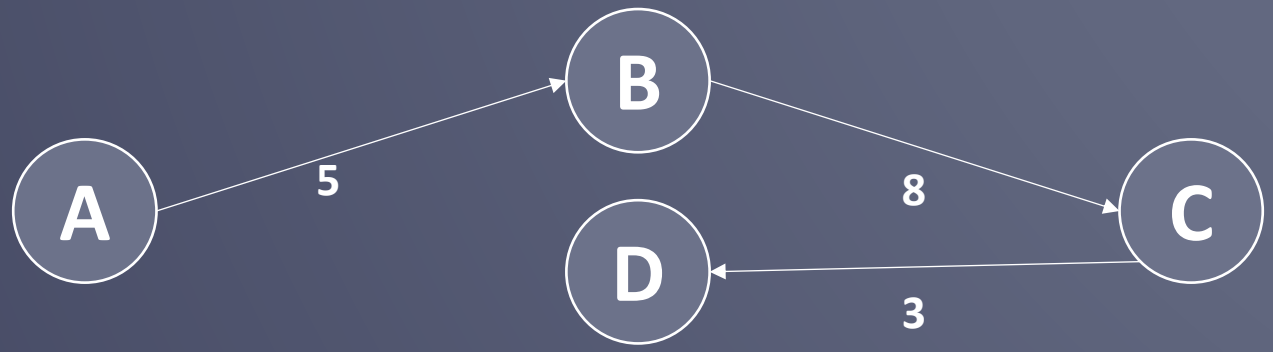
If the edges has a weight or cost it is called weighted graph.



If the edges has no weight or cost it is called unweighted graph.

Path

Path can be defined as the list of edges by which we can visit one node to another.

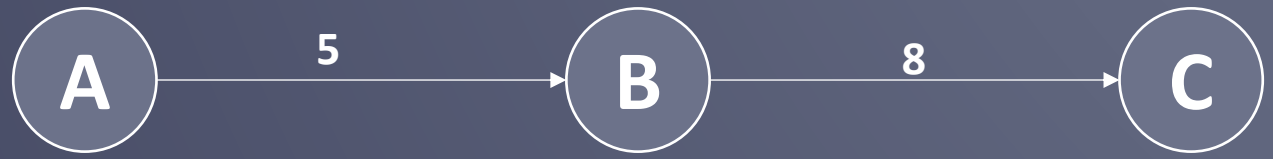


In this graph, To visit D from A. The path will be:

A -> B -> C -> D

Degree

It is the number of vertices adjacent to a vertex V. In directed graph, the number of head ends adjacent to a vertex is called the indegree of the vertex and the number of tail ends adjacent to a vertex is its outdegree



In undirected graph, indegree = outdegree

Tree

A tree is an undirected graph in which any two vertices are connected by exactly one path.

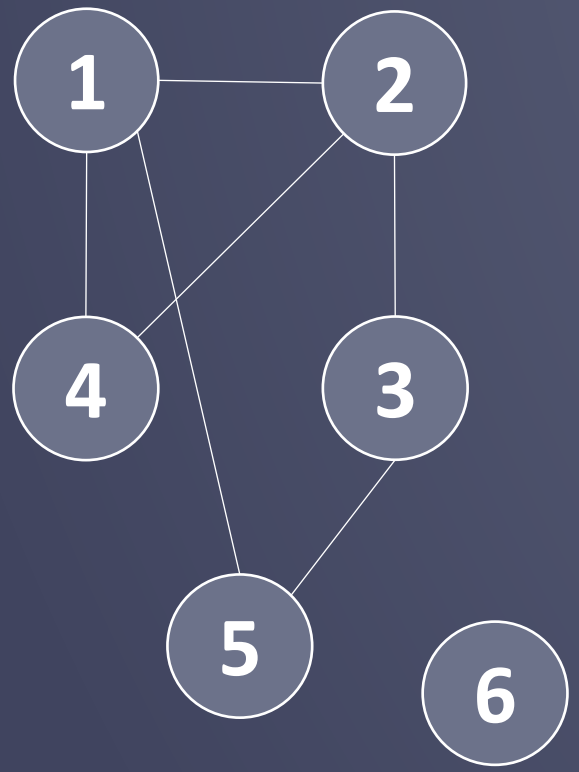
In Tree, for **n-node there will be exactly (n-1) edges**. We can also create a forest combining two or more trees.

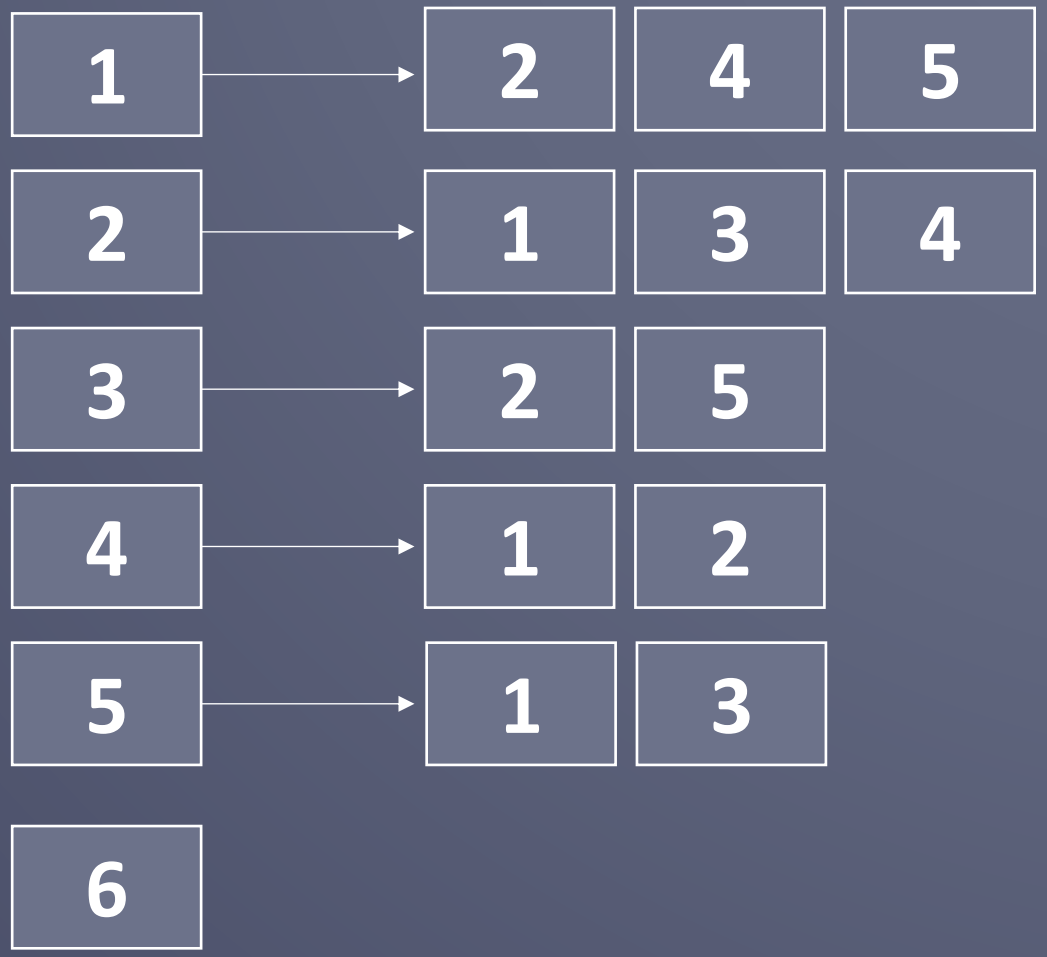
Graph Representation

In code , we can represent graph with

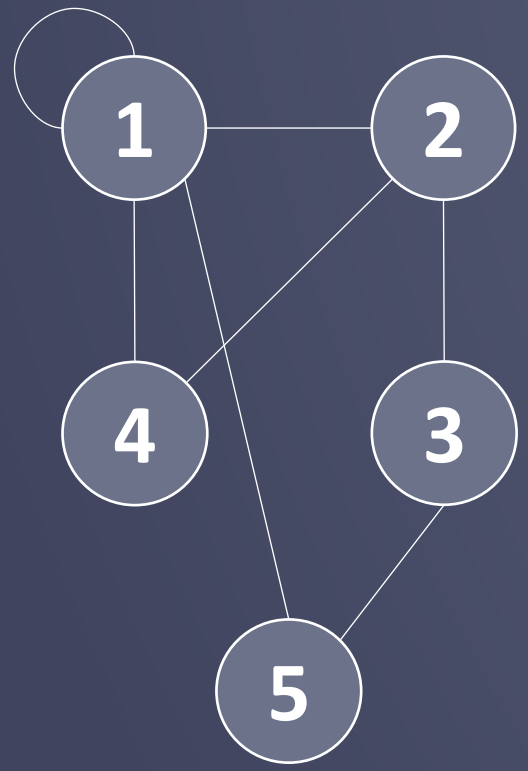
- Adjacency Matrix (2D Array)
- Adjacency List (STL Vector/List)

Adjacency List



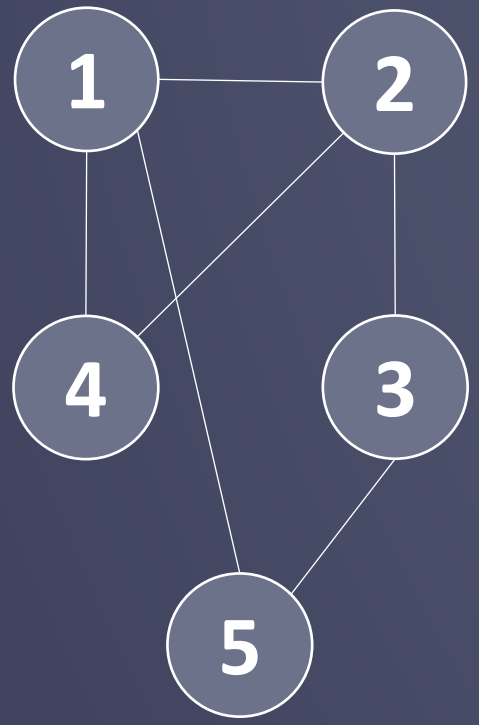


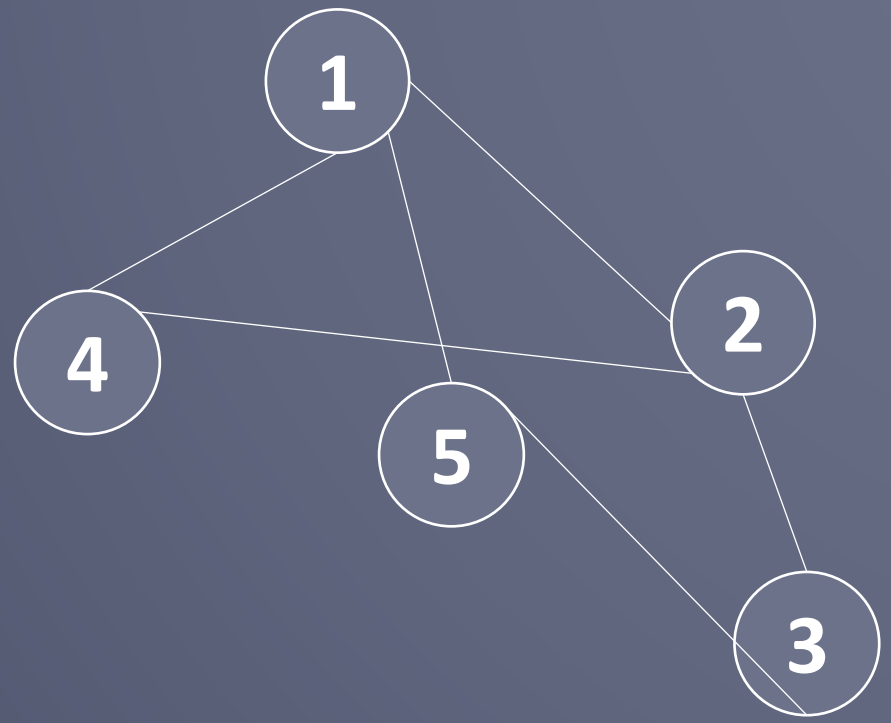
Adjacency Matrix



	1	2	3	4	5
1	1	1	0	1	1
2	1	0	1	1	0
3	0	1	0	0	1
4	1	1	0	0	0
5	1	0	1	0	0

Level





Minimum Spanning Tree

MD. MUKTAR HOSSAIN

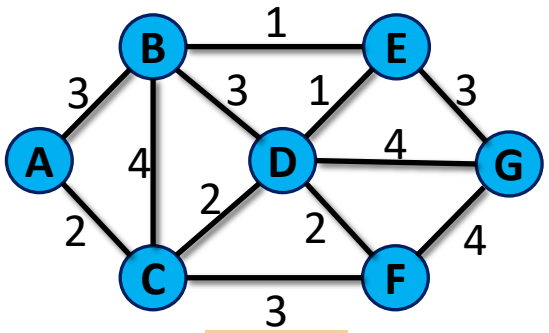
LECTURER

DEPT. OF CSE

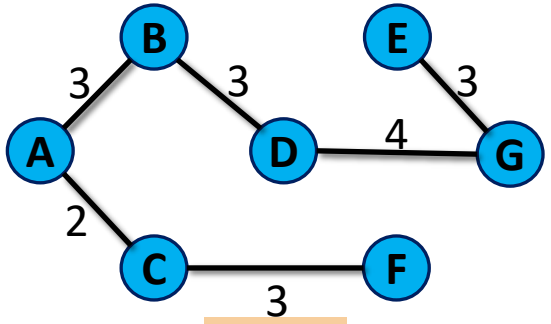
VARENDRA UNIVERSITY

Spanning Tree

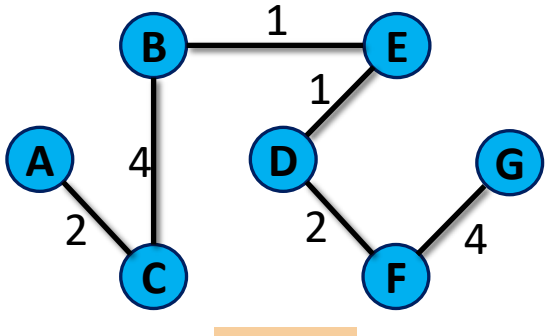
A ***spanning tree*** of a graph is a subgraph that includes all the vertices of the original graph and is also a tree (a ***connected acyclic graph***).



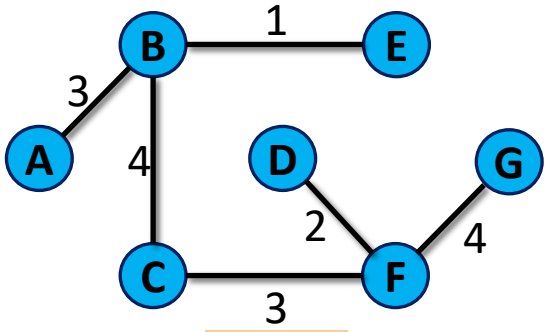
Graph



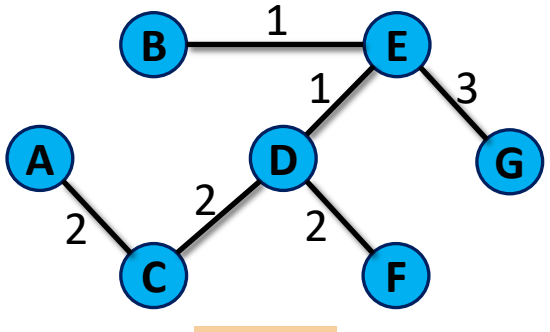
ST-1



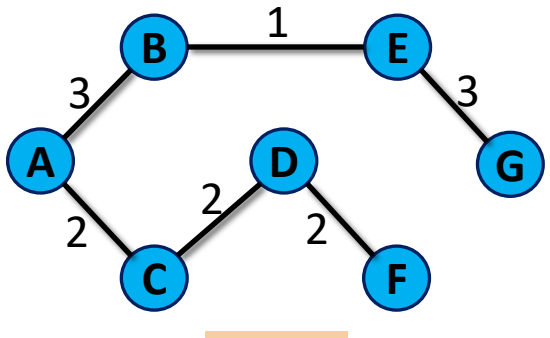
ST-2



ST-3



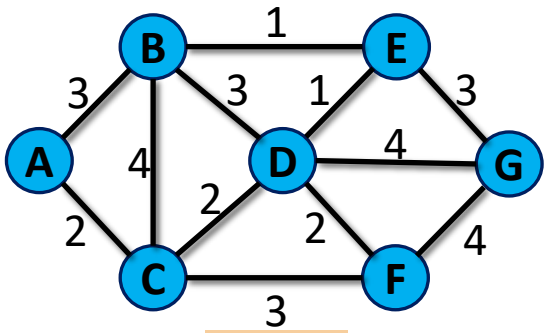
ST-4



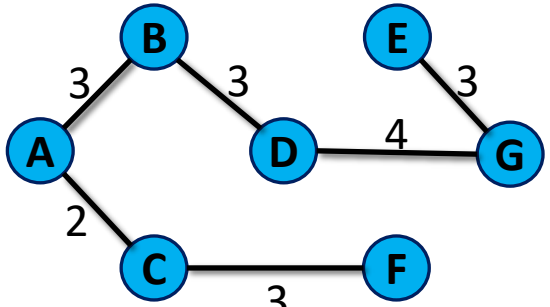
ST-5

Minimum Spanning Tree

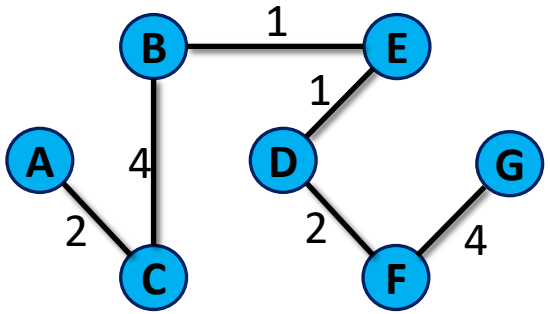
A ***minimum spanning tree (MST)*** is defined as a spanning tree that has the ***minimum weight among all the possible spanning trees.***



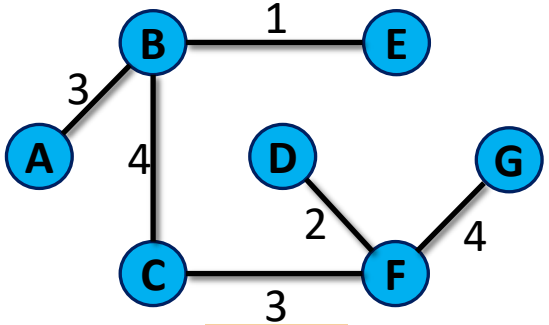
Graph



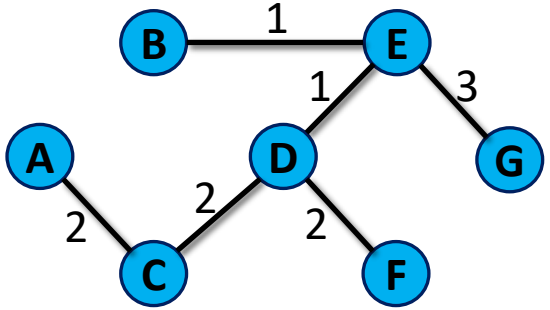
ST-1 | 18



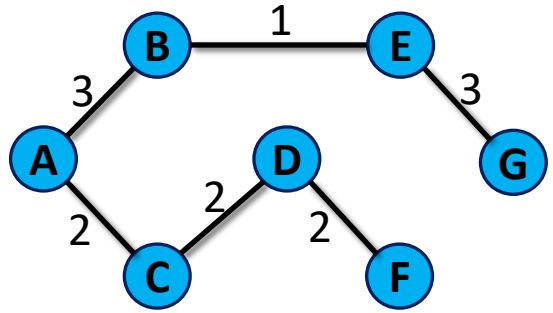
ST-2 | 14



ST-3 | 17



ST-4 | 11



ST-5 | 13

MST Applications

- **Computer Networks**
  - ✓ Connect computers/routers with minimum cabling cost.
  - ✓ Avoids cycles, ensures full connectivity.
- **Telecommunication Networks**
  - ✓ Design cost-effective layouts for telephone lines and internet infrastructure.
- **Road and Railway Systems**
  - ✓ Connect cities or regions with minimal construction cost.
  - ✓ Ideal for planning roadways, pipelines, and railway tracks.
- **Electrical Grid Design**
  - ✓ Distribute power with minimum wiring.
  - ✓ Reduces construction and energy transmission costs.
- **Game Map Design**
  - ✓ Connect rooms/areas with minimal paths in procedurally generated maps.
- **Traveling Salesman Problem (TSP) Approximation**
  - ✓ MST is used in heuristics for finding near-optimal TSP tours.
- **Wireless Sensor Networks**
  - ✓ Minimizes energy usage by reducing total communication path cost.
- **Clustering in Machine Learning**
  - ✓ Used in hierarchical (single-linkage) clustering.
  - ✓ Groups similar data points by cutting longest edges in the MST.

Minimum Spanning Tree

Requirements

- The graph must be connected and weighted (edges have weights).
- There can be more than one MST if multiple trees have the same total weight.

Key Properties

- Total edges = V – 1
- Optimizes cost, such as minimizing the length of wire needed to connect all computers in a network.

Algorithms to find Minimum Spanning Tree

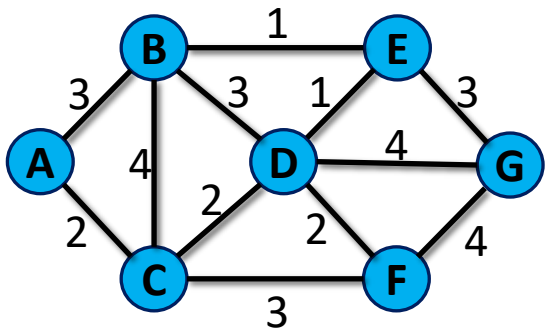
- Kruskal's Minimum Spanning Tree Algorithm
- Prim's Minimum Spanning Tree Algorithm

Kruskal's MST Algorithm

Algorithm

1. Sort all the edges in a ascending or non-decreasing order of their weight.
2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far.
   i. If the cycle is not formed, include this edge.
   ii. Else, discard it.
3. Repeat step **2** until there are (V-1) edges in the spanning tree.

Kruskal's MST Algorithm

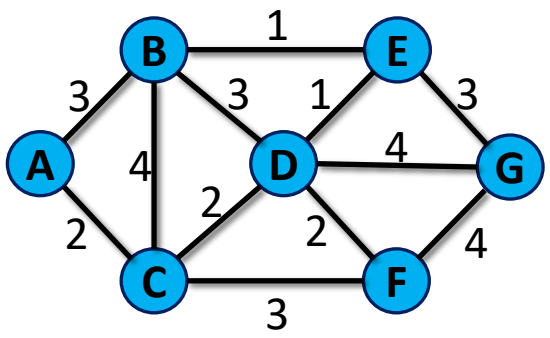


Edges	Cost
AB	3
AC	2
BC	4
BE	1
BD	3
CD	2
CF	3
DE	1
DF	2
DG	4
EG	3
FG	4

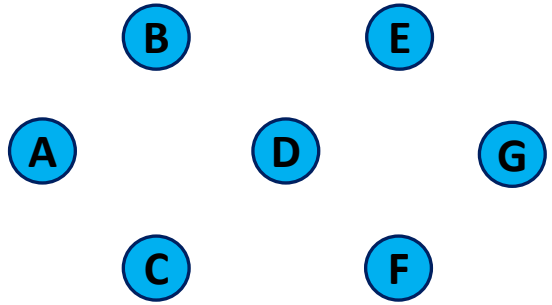
Sort by Cost

Edges	Cost
BE	1
DE	1
AC	2
CD	2
DF	2
AB	3
BD	3
CF	3
EG	3
BC	4
DG	4
FG	4

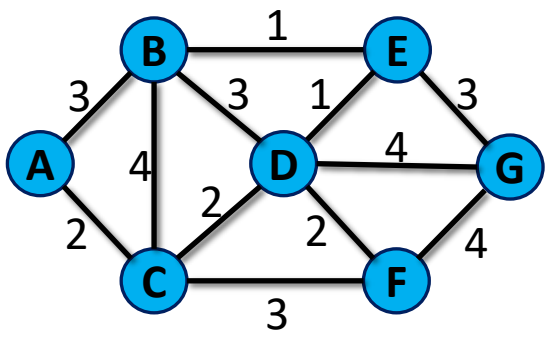
Kruskal's MST Algorithm



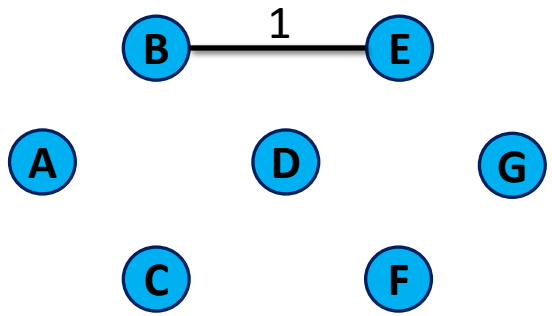
Edges	Cost
BE	1
DE	1
AC	2
CD	2
DF	2
AB	3
BD	3
CF	3
EG	3
BC	4
DG	4
FG	4



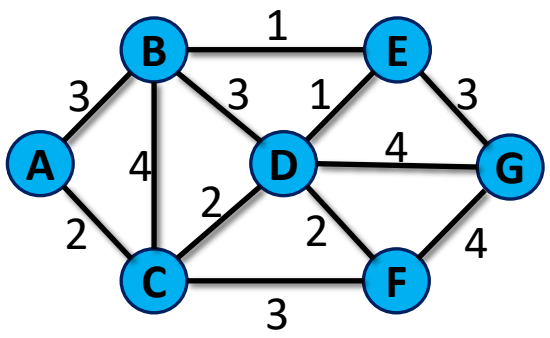
Kruskal's MST Algorithm



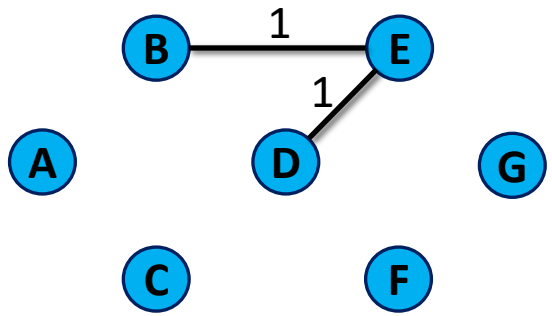
Edges	Cost
BE	1
DE	1
AC	2
CD	2
DF	2
AB	3
BD	3
CF	3
EG	3
BC	4
DG	4
FG	4



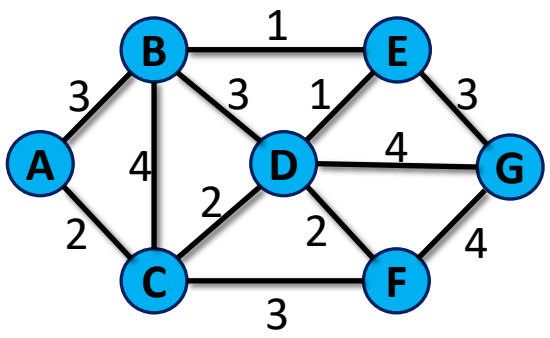
Kruskal's MST Algorithm



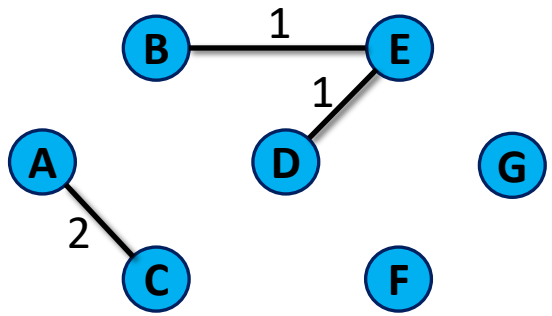
Edges	Cost
BE	1
DE	1
AC	2
CD	2
DF	2
AB	3
BD	3
CF	3
EG	3
BC	4
DG	4
FG	4



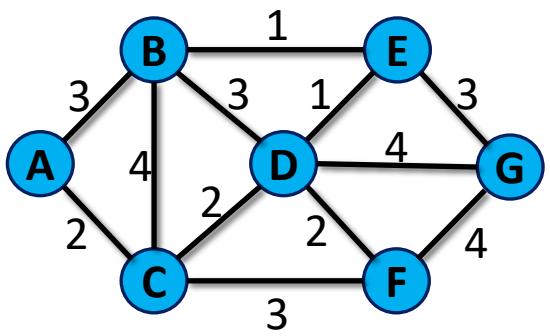
Kruskal's MST Algorithm



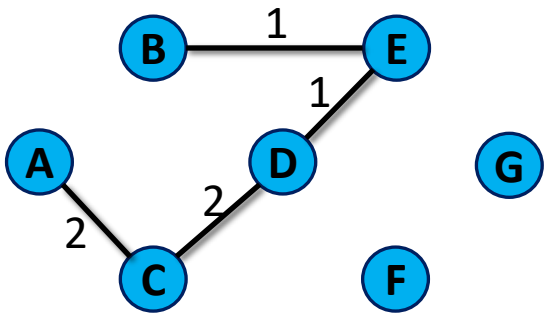
Edges	Cost
BE	1
DE	1
AC	2
CD	2
DF	2
AB	3
BD	3
CF	3
EG	3
BC	4
DG	4
FG	4



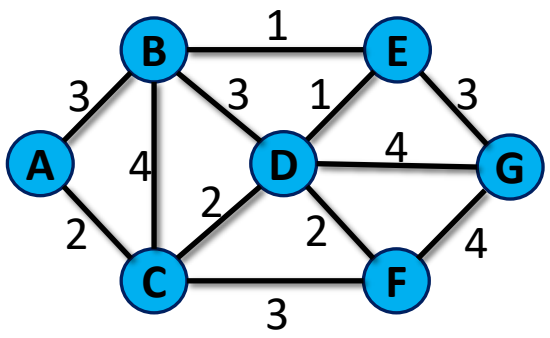
Kruskal's MST Algorithm



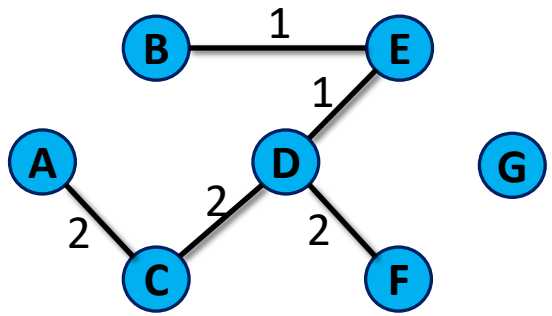
Edges	Cost
BE	1
DE	1
AC	2
CD	2
DF	2
AB	3
BD	3
CF	3
EG	3
BC	4
DG	4
FG	4



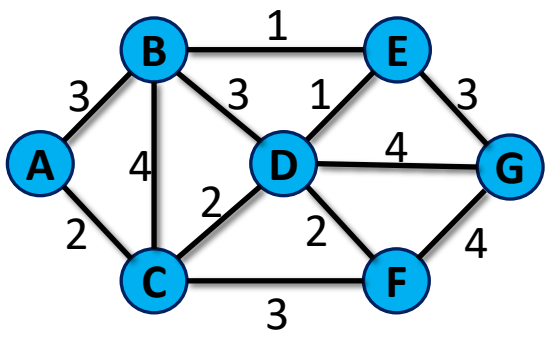
Kruskal's MST Algorithm



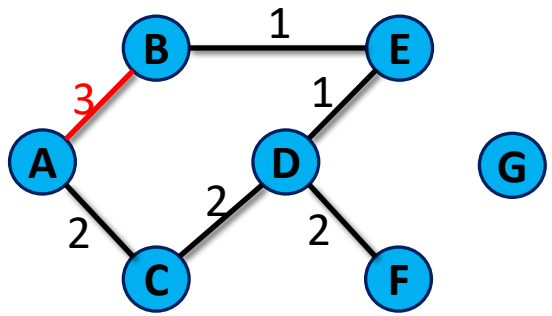
Edges	Cost
BE	1
DE	1
AC	2
CD	2
DF	2
AB	3
BD	3
CF	3
EG	3
BC	4
DG	4
FG	4



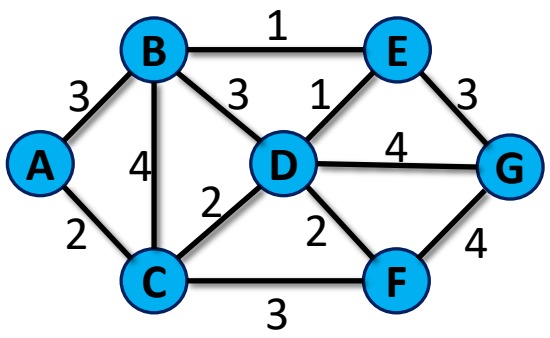
Kruskal's MST Algorithm



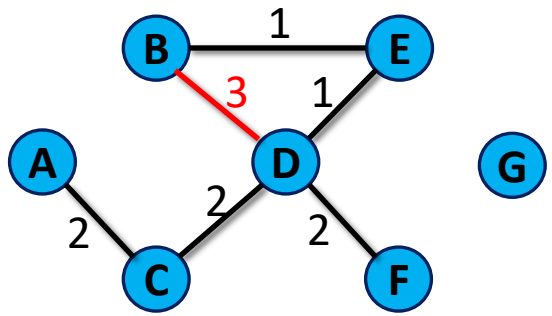
Edges	Cost
BE	1
DE	1
AC	2
CD	2
DF	2
AB	3
BD	3
CF	3
EG	3
BC	4
DG	4
FG	4



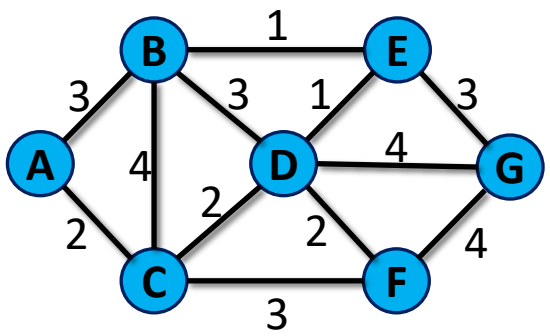
Kruskal's MST Algorithm



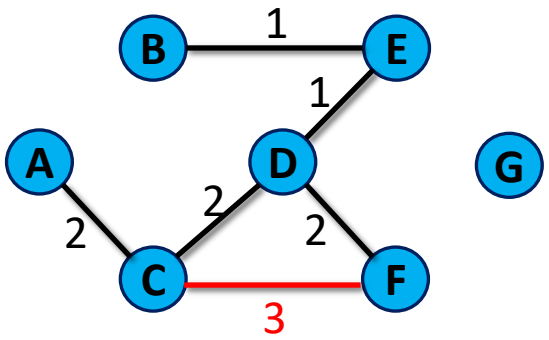
Edges	Cost
BE	1
DE	1
AC	2
CD	2
DF	2
AB	3
BD	3
CF	3
EG	3
BC	4
DG	4
FG	4



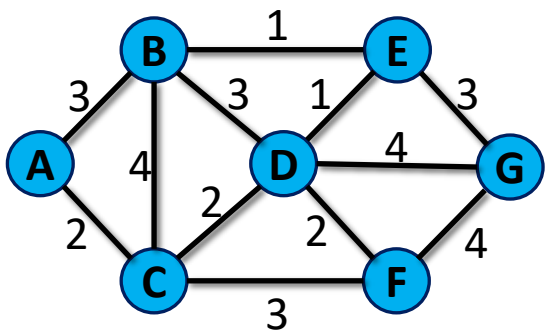
Kruskal's MST Algorithm



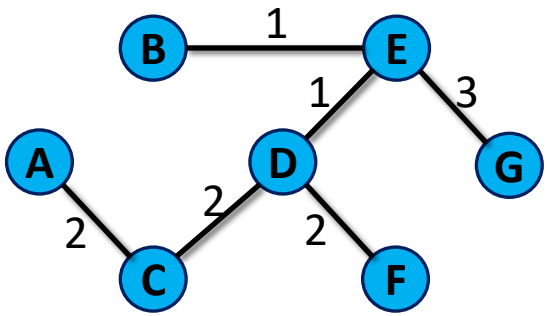
Edges	Cost
BE	1
DE	1
AC	2
CD	2
DF	2
AB	3
BD	3
CF	3
EG	3
BC	4
DG	4
FG	4



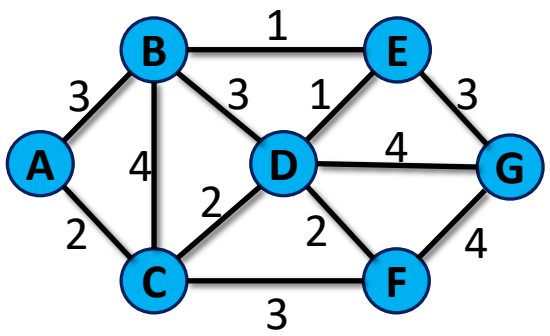
Kruskal's MST Algorithm



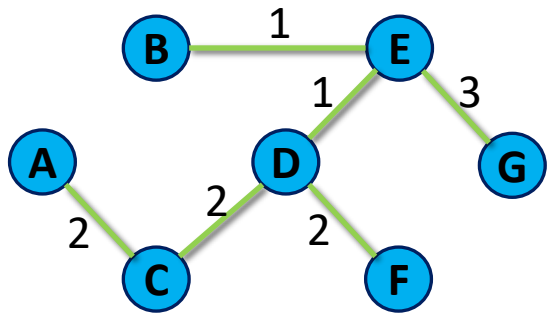
Edges	Cost
BE	1
DE	1
AC	2
CD	2
DF	2
AB	3
BD	3
CF	3
EG	3
BC	4
DG	4
FG	4



Kruskal's MST Algorithm



Edges	Cost
BE	1
DE	1
AC	2
CD	2
DF	2
AB	3
BD	3
CF	3
EG	3
BC	4
DG	4
FG	4



Total Cost = 1+1+2+2+2+3 = 11

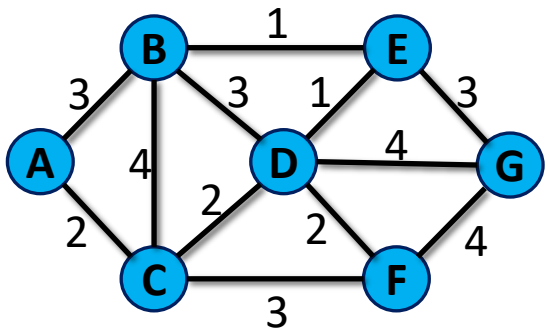
Prim’s MST Algorithm

Algorithm

1. Start with any vertex; add it to the MST.
2. While MST doesn't include all vertices:
    a. Pick the smallest edge connecting MST to a new vertex.
    b. Add that edge and vertex to the MST.
3. Repeat until all vertices are included.

Prim's MST Algorithm

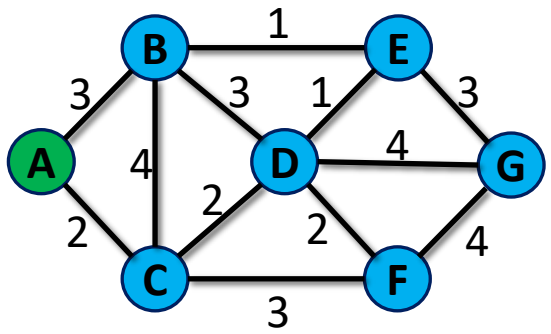
1. Start With Vertex A



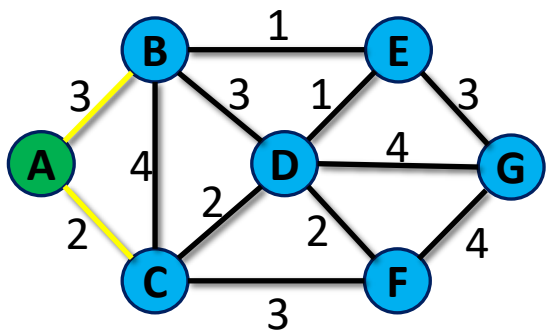
Prim's MST Algorithm

1. Start With Vertex A

2. Explore the connected edges to MST



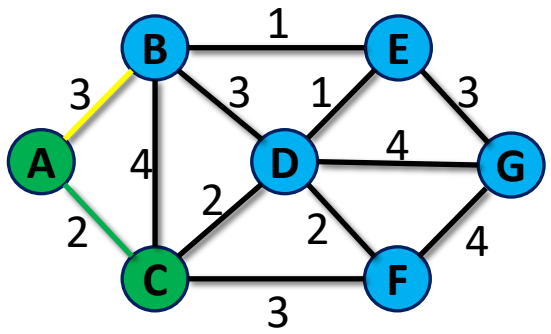
Prim's MST Algorithm



1. Start With Vertex A
2. Explore the connected edges to MST
3. Select minimum edge then add that vertex and edges to MST

AC - 2	AB - 3						

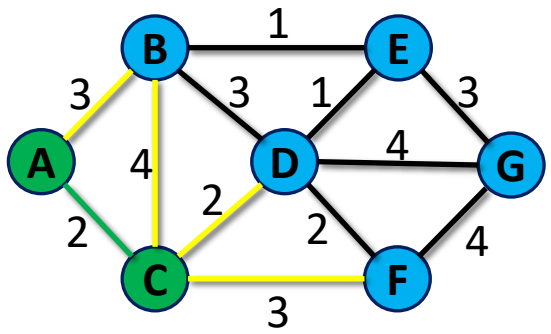
Prim's MST Algorithm



1. Start With Vertex A
2. Explore the connected edges to MST
3. Select minimum edge then add that vertex and edges to MST
4. Repeat 2 and 3 until all vertices are included in MST.

AB - 3							

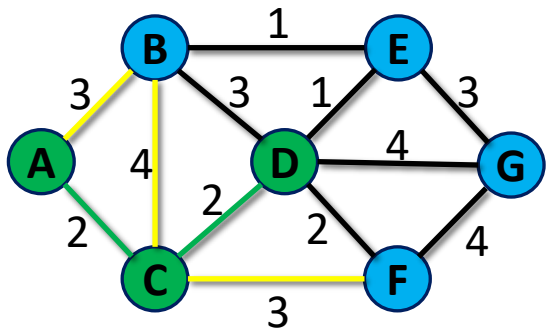
Prim's MST Algorithm



1. Start With Vertex A
2. Explore the connected edges to MST
3. Select minimum edge then add that vertex and edges to MST
4. Repeat 2 and 3 until all vertices are included in MST.

CD - 2	AB - 3	CF - 3	CB - 4				

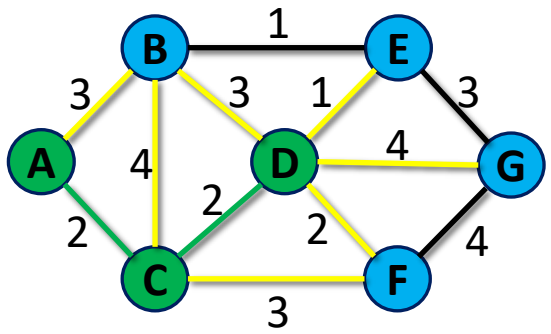
Prim's MST Algorithm



1. Start With Vertex A
2. Explore the connected edges to MST
3. Select minimum edge then add that vertex and edges to MST
4. Repeat 2 and 3 until all vertices are included in MST.

AB - 3	CF - 3	CB - 4					

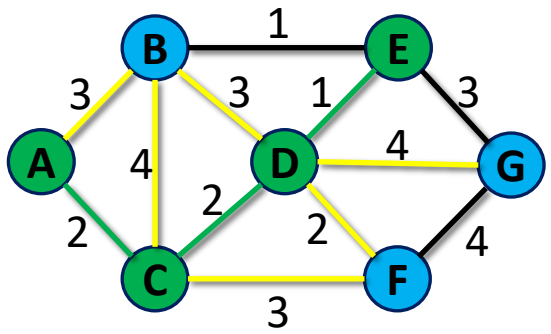
Prim's MST Algorithm



1. Start With Vertex A
2. Explore the connected edges to MST
3. Select minimum edge then add that vertex and edges to MST
4. Repeat 2 and 3 until all vertices are included in MST.

DE - 1	DF - 2	AB - 3	CF - 3	DB - 3	CB - 4	DG - 4	

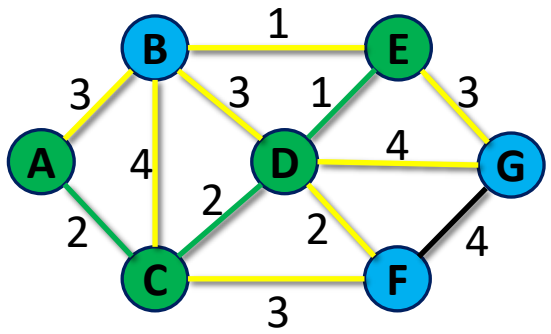
Prim's MST Algorithm



1. Start With Vertex A
2. Explore the connected edges to MST
3. Select minimum edge then add that vertex and edges to MST
4. Repeat 2 and 3 until all vertices are included in MST.

DF - 2	AB - 3	CF - 3	DB - 3	CB - 4	DG - 4		

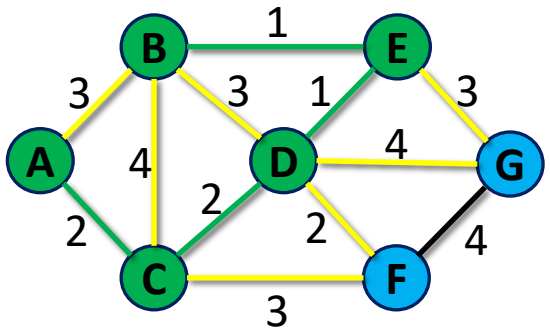
Prim's MST Algorithm



1. Start With Vertex A
2. Explore the connected edges to MST
3. Select minimum edge then add that vertex and edges to MST
4. Repeat 2 and 3 until all vertices are included in MST.

EB - 1	DF - 2	AB - 3	CF - 3	DB - 3	EG - 3	CB - 4	DG - 4

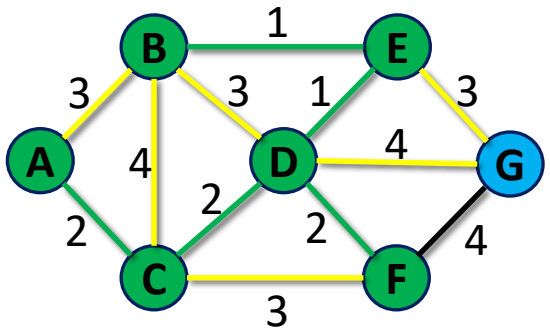
Prim's MST Algorithm



1. Start With Vertex A
2. Explore the connected edges to MST
3. Select minimum edge then add that vertex and edges to MST
4. Repeat 2 and 3 until all vertices are included in MST.

DF - 2	AB - 3	CF - 3	DB - 3	EG - 3	CB - 4	DG - 4	

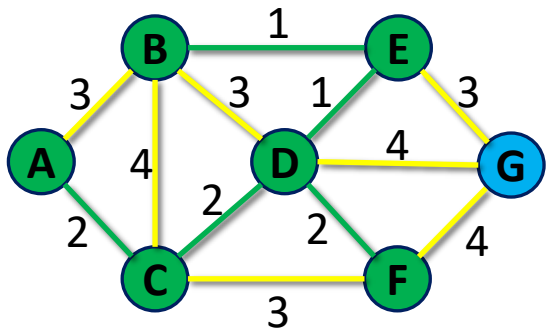
Prim's MST Algorithm



1. Start With Vertex A
2. Explore the connected edges to MST
3. Select minimum edge then add that vertex and edges to MST
4. Repeat 2 and 3 until all vertices are included in MST.

AB - 3	CF - 3	DB - 3	EG - 3	CB - 4	DG - 4		

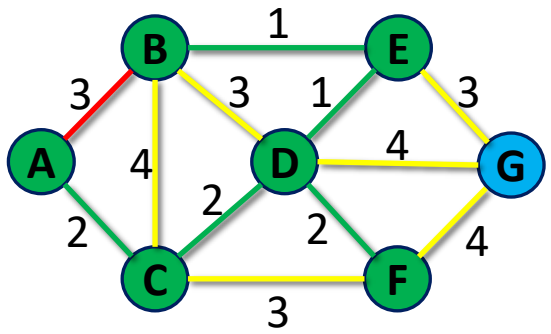
Prim's MST Algorithm



1. Start With Vertex A
2. Explore the connected edges to MST
3. Select minimum edge then add that vertex and edges to MST
4. Repeat 2 and 3 until all vertices are included in MST.

AB - 3	CF - 3	DB - 3	EG - 3	CB - 4	DG - 4	FG - 4	

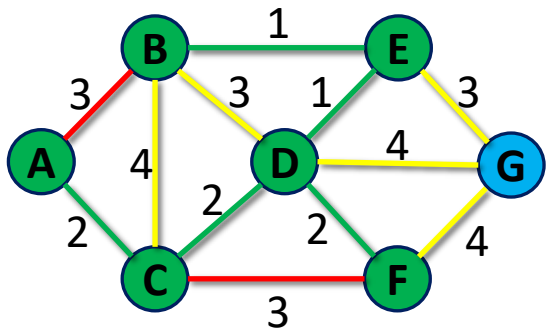
Prim's MST Algorithm



1. Start With Vertex A
2. Explore the connected edges to MST
3. Select minimum edge then add that vertex and edges to MST
4. Repeat 2 and 3 until all vertices are included in MST.

CF - 3	DB - 3	EG - 3	CB - 4	DG - 4	FG - 4		

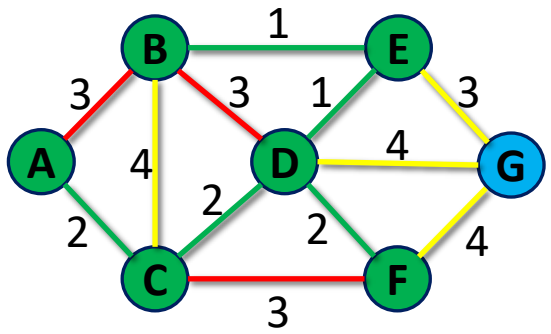
Prim's MST Algorithm



1. Start With Vertex A
2. Explore the connected edges to MST
3. Select minimum edge then add that vertex and edges to MST
4. Repeat 2 and 3 until all vertices are included in MST.

DB - 3	EG - 3	CB - 4	DG - 4	FG - 4			

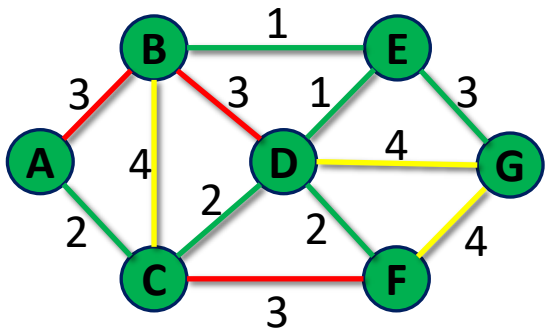
Prim's MST Algorithm



1. Start With Vertex A
2. Explore the connected edges to MST
3. Select minimum edge then add that vertex and edges to MST
4. Repeat 2 and 3 until all vertices are included in MST.

EG - 3	CB - 4	DG - 4	FG - 4				

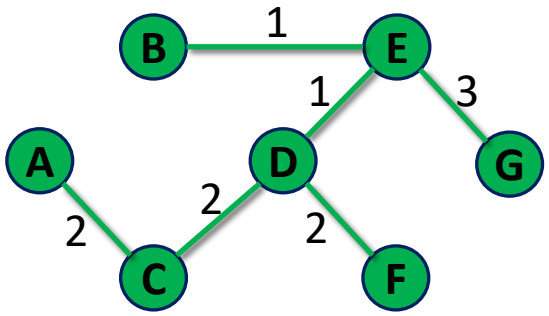
Prim's MST Algorithm



1. Start With Vertex A
2. Explore the connected edges to MST
3. Select minimum edge then add that vertex and edges to MST
4. Repeat 2 and 3 until all vertices are included in MST.

CB - 4	DG - 4	FG - 4					

Prim's MST Algorithm



So, the cost of the MST is = 2+2+2+1+1+3 = **11**

THANK YOU

Single-Source Shortest Path Algorithm

MD. MUKTAR HOSSAIN

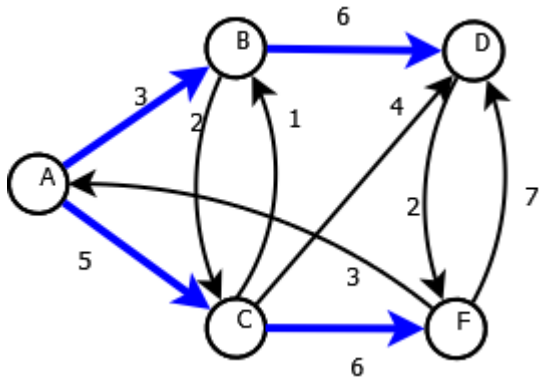
LECTURER

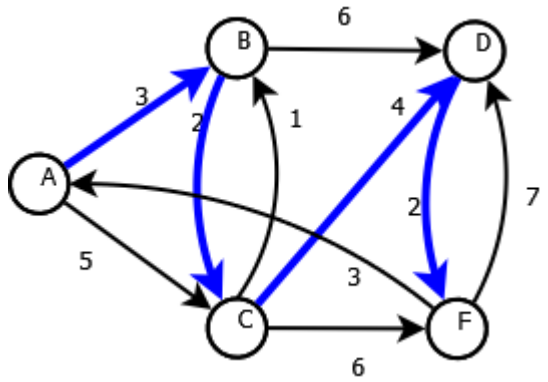
DEPT. OF CSE

VARENDRA UNIVERSITY

Single-Source Shortest Path Problem

Single-Source Shortest Path Problem - The problem of finding shortest paths from a source vertex ***v*** to all other vertices in the graph.

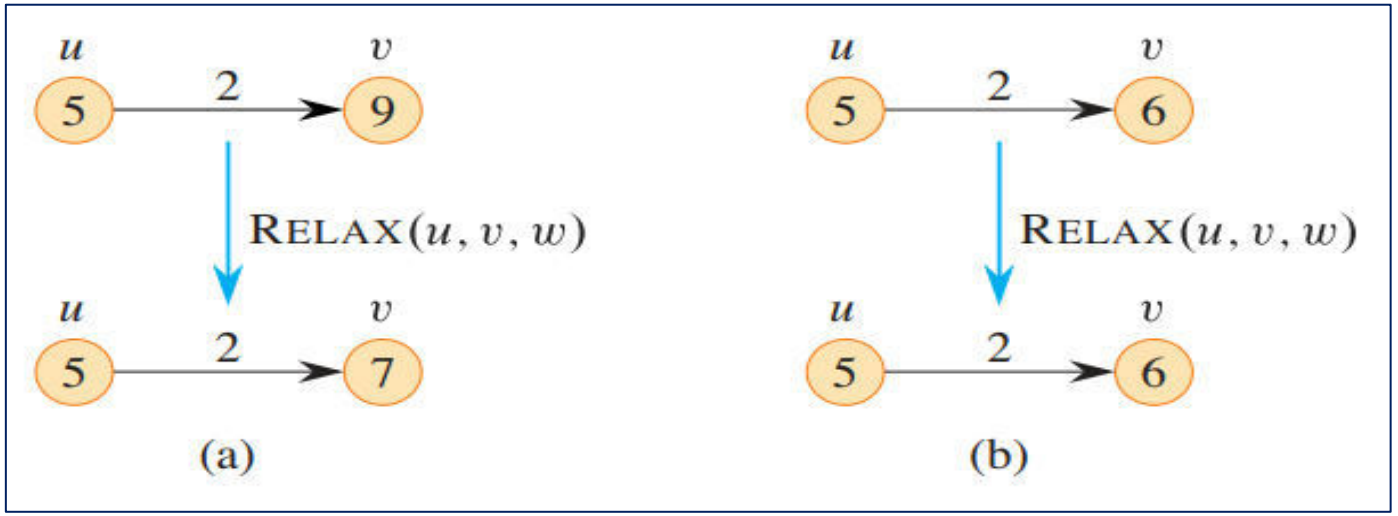




Relaxation

For an edge ***(u,v)*** with weight ***w***, relaxation checks if the known shortest path to ***v*** can be improved by going through ***u***.

```
RELAX(u, v, w)
1  if v.d > u.d + w(u, v)
2        v.d = u.d + w(u, v)
3        v.π = u
```



Introduction

Dijkstra's algorithm - is a solution to the single-source shortest path problem in graph theory.

Works on both directed and undirected graphs. However, all edges must have nonnegative weights.

Input: Weighted graph G={E,V} and source vertex v∈V, such that all edge weights are nonnegative

Output: Lengths of shortest paths (or the shortest paths themselves) from a given source vertex v∈V to all other vertices

Applications of Dijkstra's Algorithm

1.Network Routing Protocols:
- ✓ Used in **OSPF (Open Shortest Path First)** and **IS-IS** routing algorithms to compute the shortest path tree for routing packets.

2.Mapping and GPS Systems:
- ✓ Calculating the quickest route between two locations using road network data.

3.Game Development:
- ✓ Pathfinding for characters or AI agents (though A* is often preferred for efficiency in certain cases).

4.Telecommunications:
- ✓ Optimizing data flow through networks by finding the least-cost path.

5.Compiler Design:
- ✓ Used in **data-flow analysis** and **code optimization** to calculate the shortest distance in control flow graphs.

6.Robot Motion Planning:
- ✓ In robotics, Dijkstra helps robots find the shortest path in a known environment.

7.Dependency Resolution:
- ✓ Used in systems like **package managers** to resolve dependency chains efficiently.

Dijkstra's Algorithm

1. Initialize distances from the source to all nodes as infinity, except the source (0).
2. Use a priority queue (min-heap) to repeatedly select the node with the smallest tentative distance.
3. Update the distances to neighboring nodes if a shorter path is found.
4. Repeat until all nodes are visited or the destination is reached.

```
function Dijkstra(Graph, source):
create a priority queue Q
    for each vertex v in Graph:
        dist[v] := infinity
        prev[v] := undefined
        add v to Q
    dist[source] := 0
```

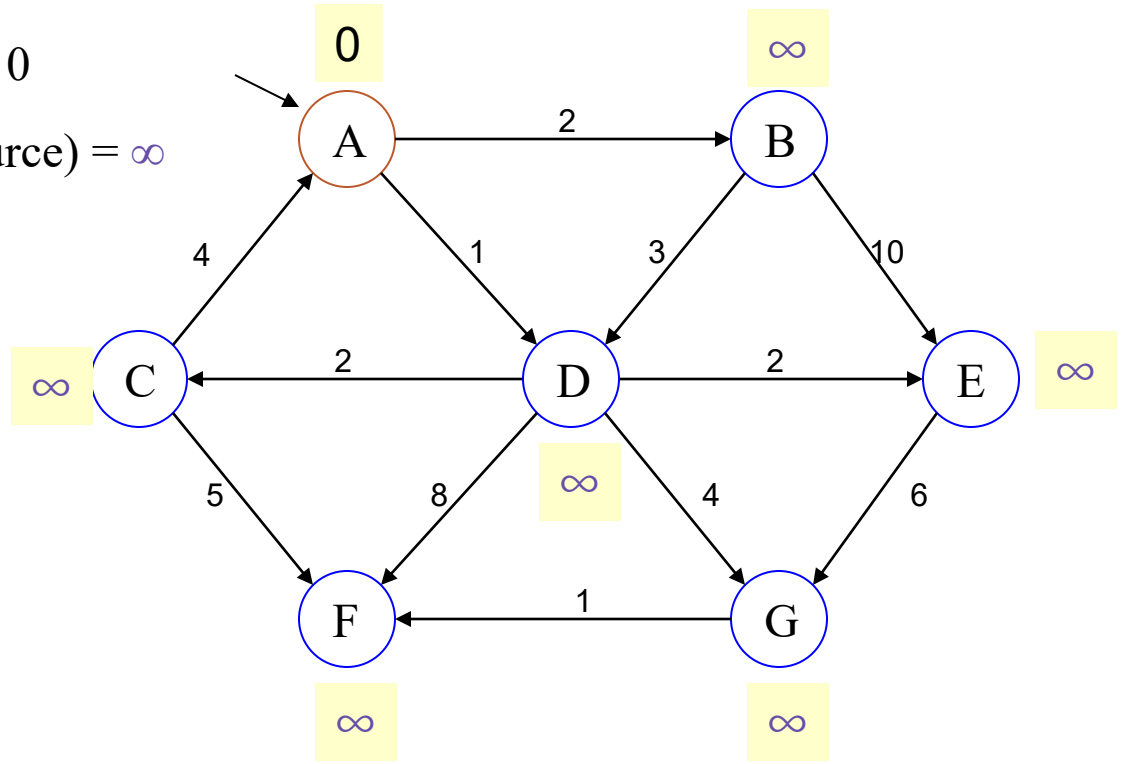
```
while Q is not empty:
     u := vertex in Q with smallest dist[u]
     remove u from Q
     for each neighbor v of u:
         alt := dist[u] + length(u, v)
         if alt < dist[v]:
             dist[v] := alt
             prev[v] := u

    return dist, prev
```

Example: Initialization

Distance(source) = 0

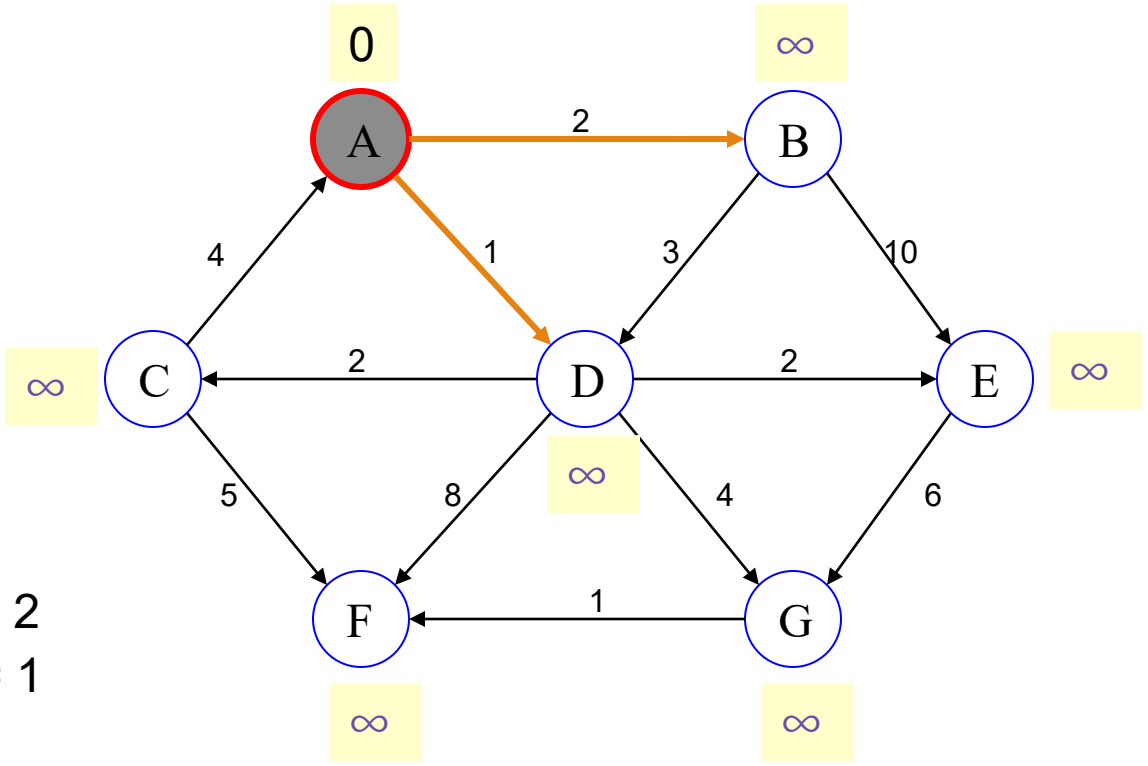
Distance (all vertices but source) = ∞



Node	Distance	Parent
A	0	
B	∞	
C	∞	
D	∞	
E	∞	
F	∞	
G	∞	

Pick vertex in List with minimum distance.

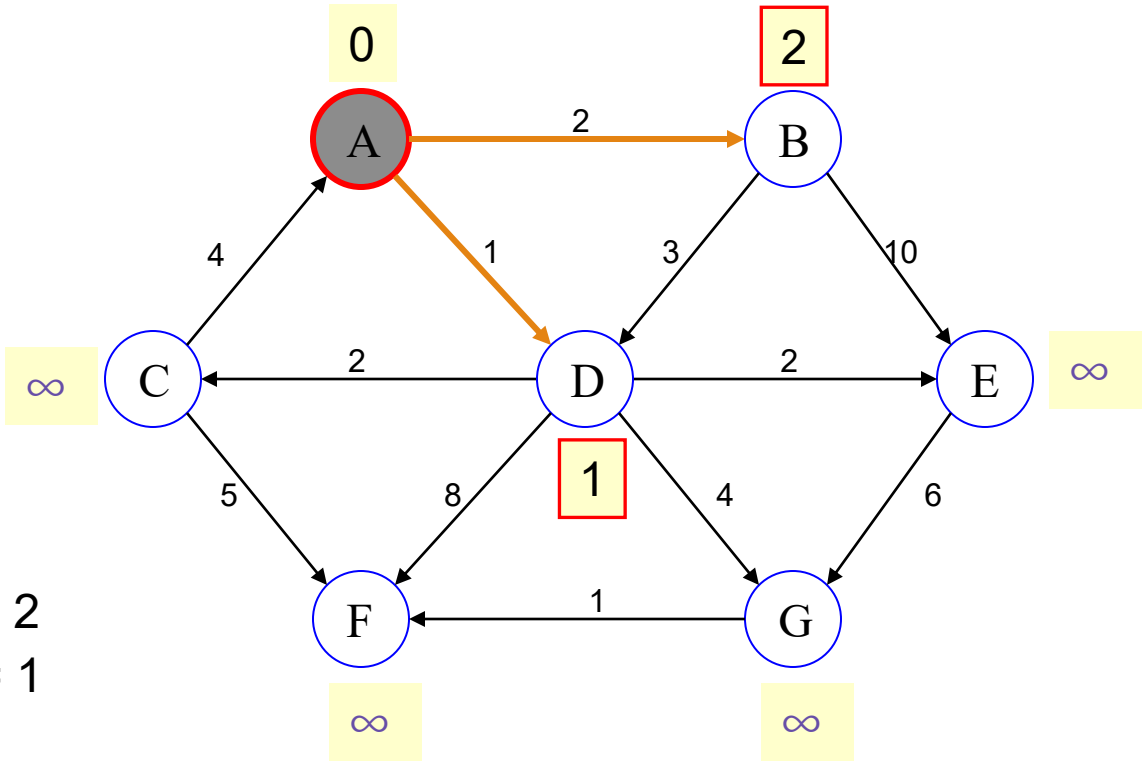
Example: Update neighbors' distance



Distance(B) = 2
Distance(D) = 1

Node	Distance	Parent
A	0	
B	∞	
C	∞	
D	∞	
E	∞	
F	∞	
G	∞	

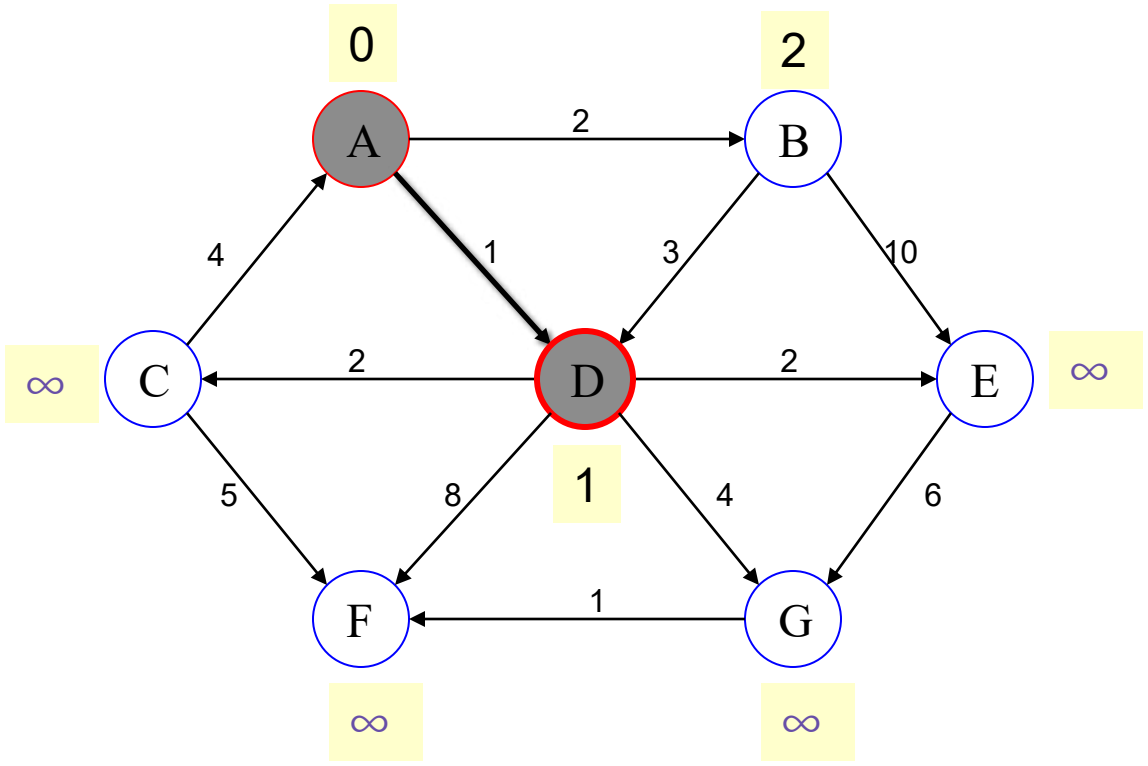
Example: Update neighbors' distance



Distance(B) = 2
Distance(D) = 1

Node	Distance	Parent
A	0	
B	2	A
C	∞	
D	1	A
E	∞	
F	∞	
G	∞	

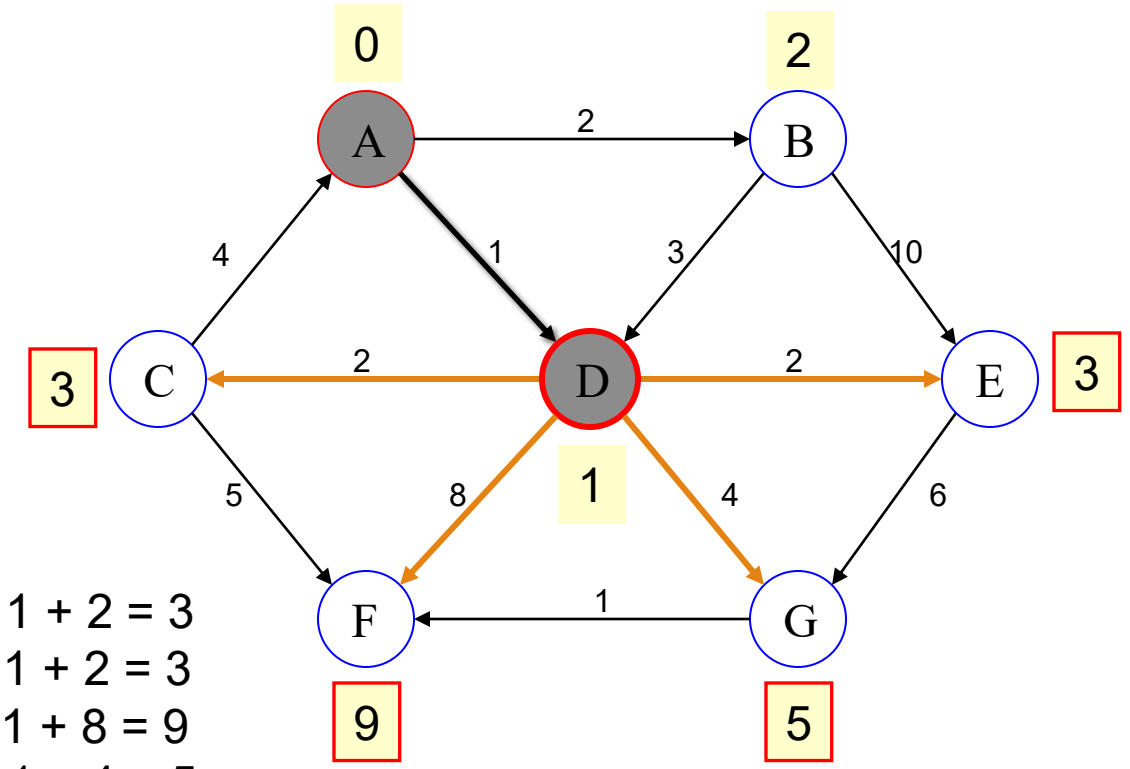
Example: Pick vertex with minimum distance



Node	Distance	Parent
A	0	
B	2	A
C	∞	
D	1	A
E	∞	
F	∞	
G	∞	

Pick vertex in List with minimum distance, i.e., D

Example: Update neighbors

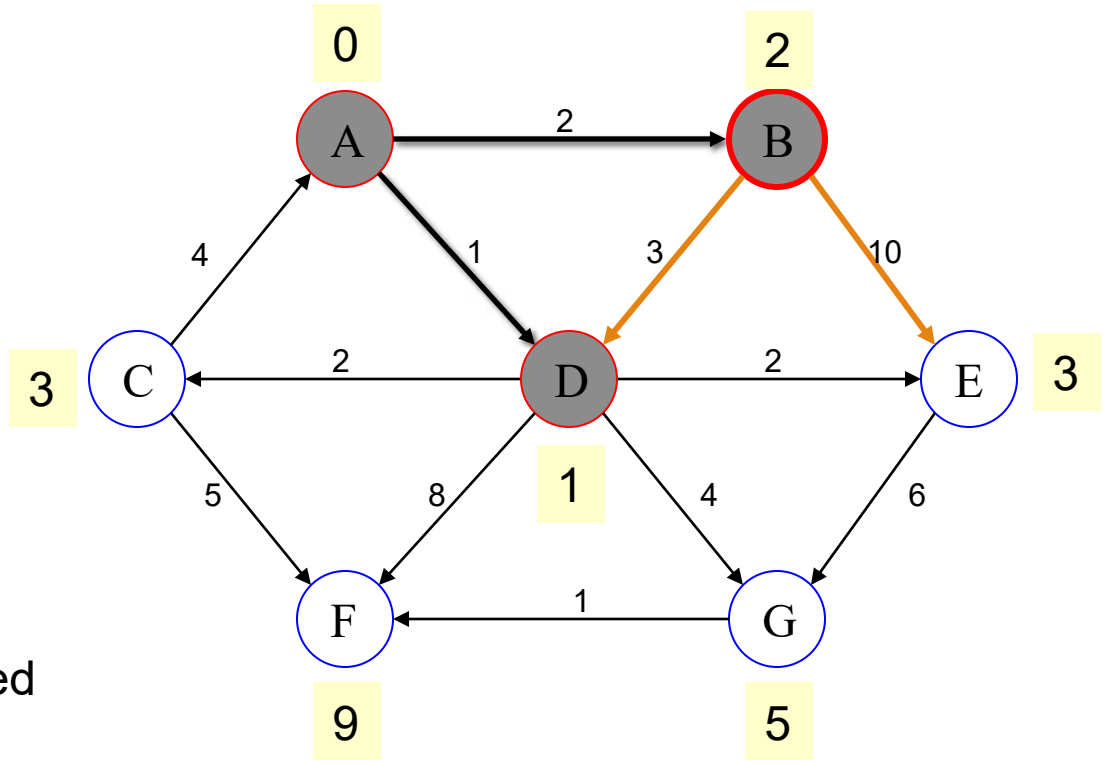


Distance(C) = 1 + 2 = 3
Distance(E) = 1 + 2 = 3
Distance(F) = 1 + 8 = 9
Distance(G) = 1 + 4 = 5

Node	Distance	Parent
A	0	
B	2	A
C	3	D
D	1	A
E	3	D
F	9	D
G	5	D

Example: Continued...

Pick vertex in List with minimum distance (B) and update neighbors

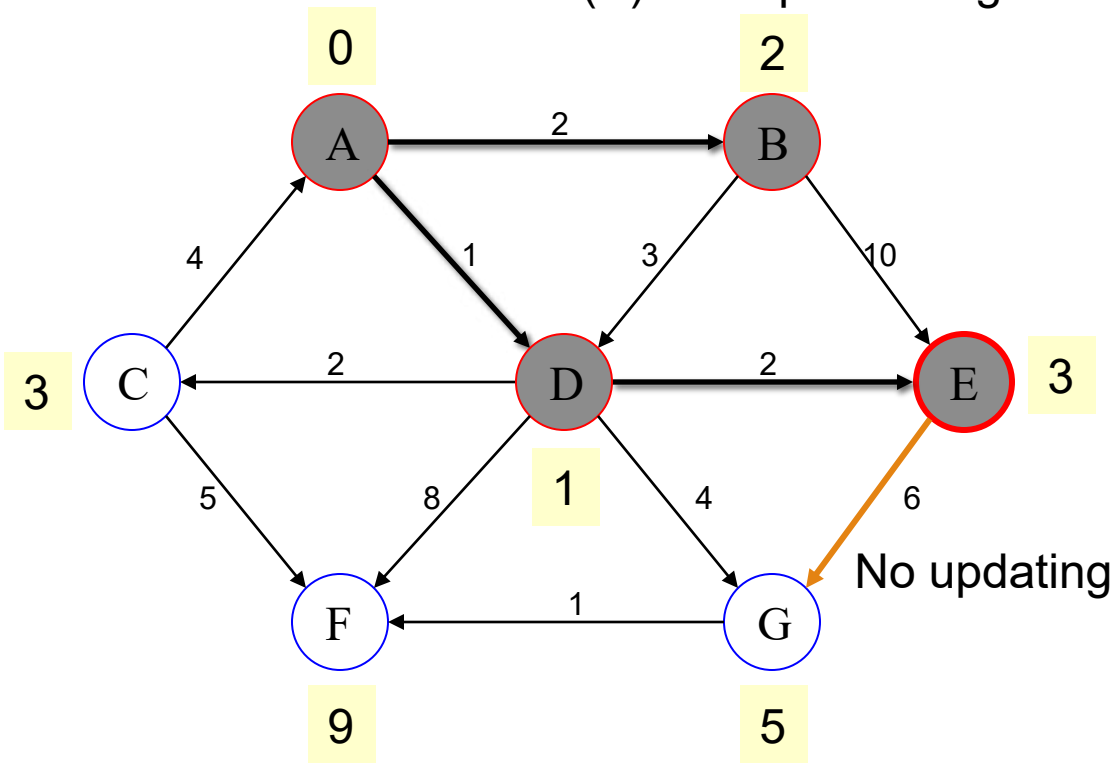


Node	Distance	Parent
A	0	
B	2	A
C	3	D
D	1	A
E	3	D
F	9	D
G	5	D

Note : distance(D) not updated since D is already known and distance(E) not updated since it is larger than previously computed

Example: Continued...

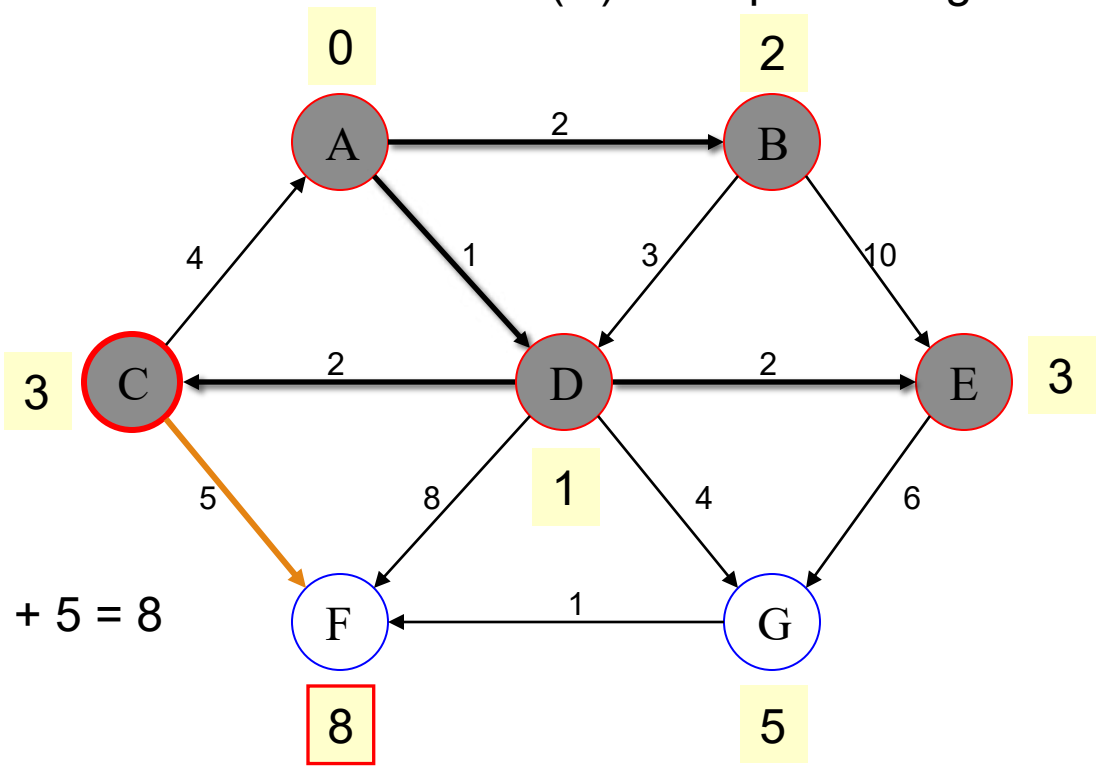
Pick vertex List with minimum distance (E) and update neighbors



Node	Distance	Parent
A	0	
B	2	A
C	3	D
D	1	A
E	3	D
F	9	D
G	5	D

Example: Continued...

Pick vertex List with minimum distance (C) and update neighbors

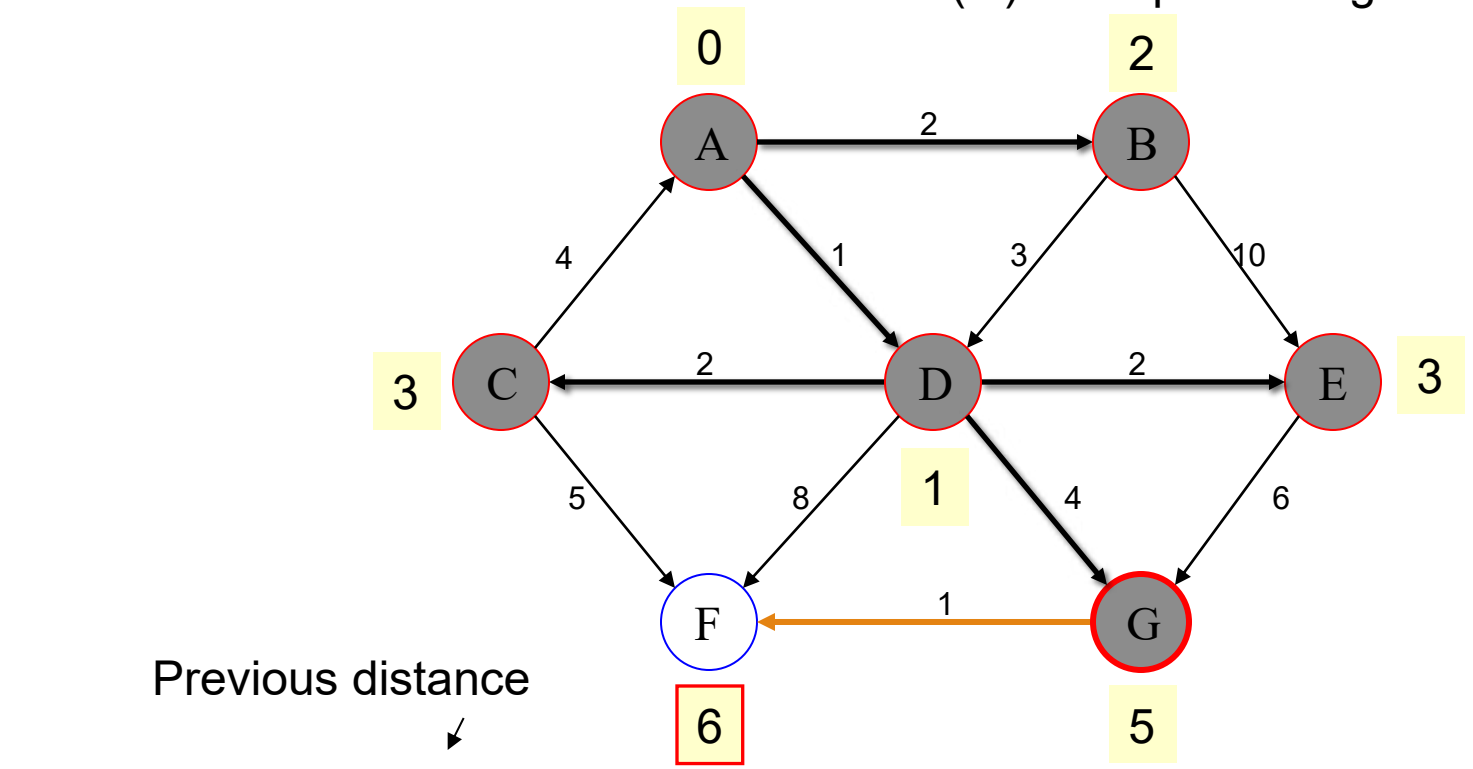


Distance(F) = 3 + 5 = 8

Node	Distance	Parent
A	0	
B	2	A
C	3	D
D	1	A
E	3	D
F	8	C
G	5	D

Example: Continued...

Pick vertex List with minimum distance (G) and update neighbors

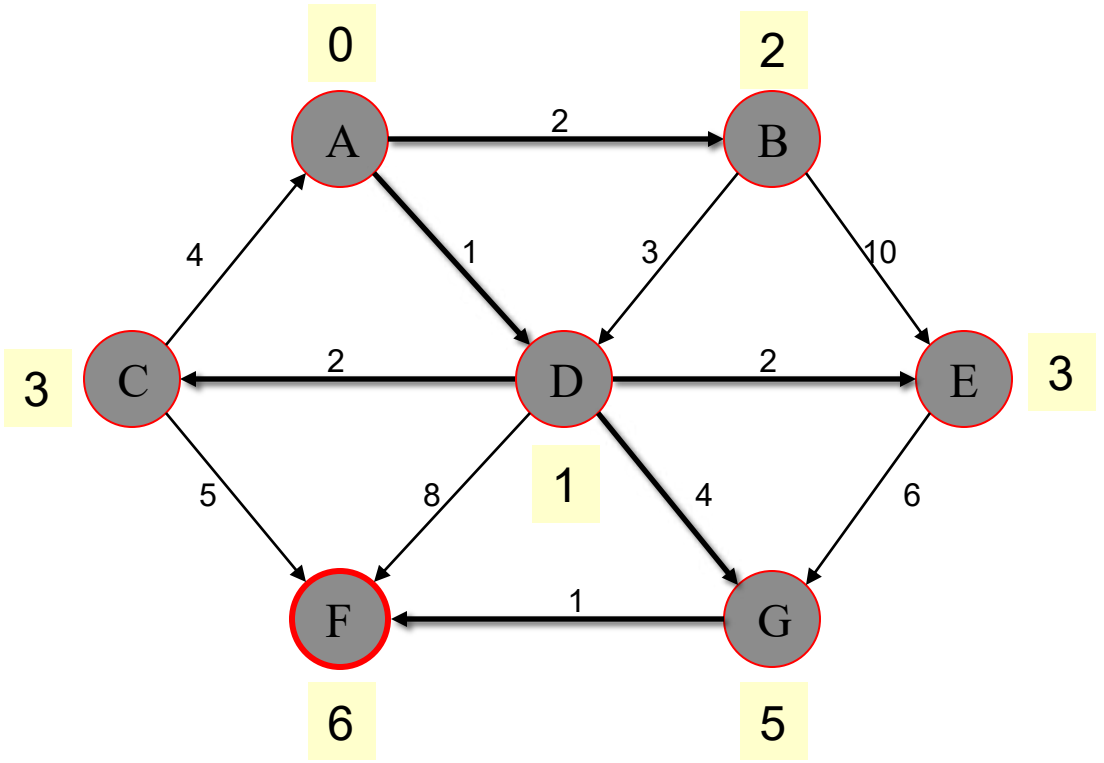


Previous distance

Distance(F) = min (8, 5+1) = 6

Node	Distance	Parent
A	0	
B	2	A
C	3	D
D	1	A
E	3	D
F	6	G
G	5	D

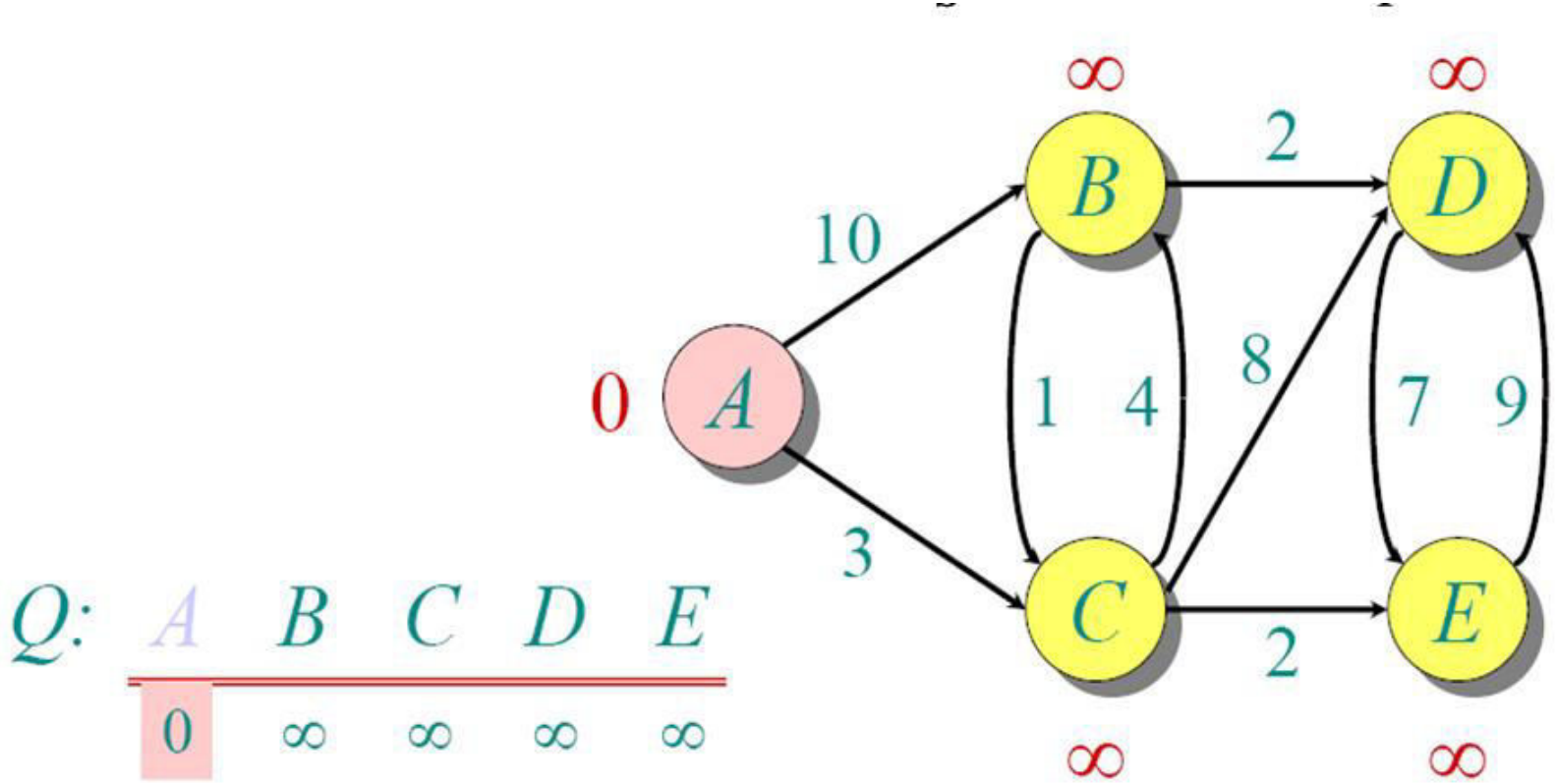
Example (end)



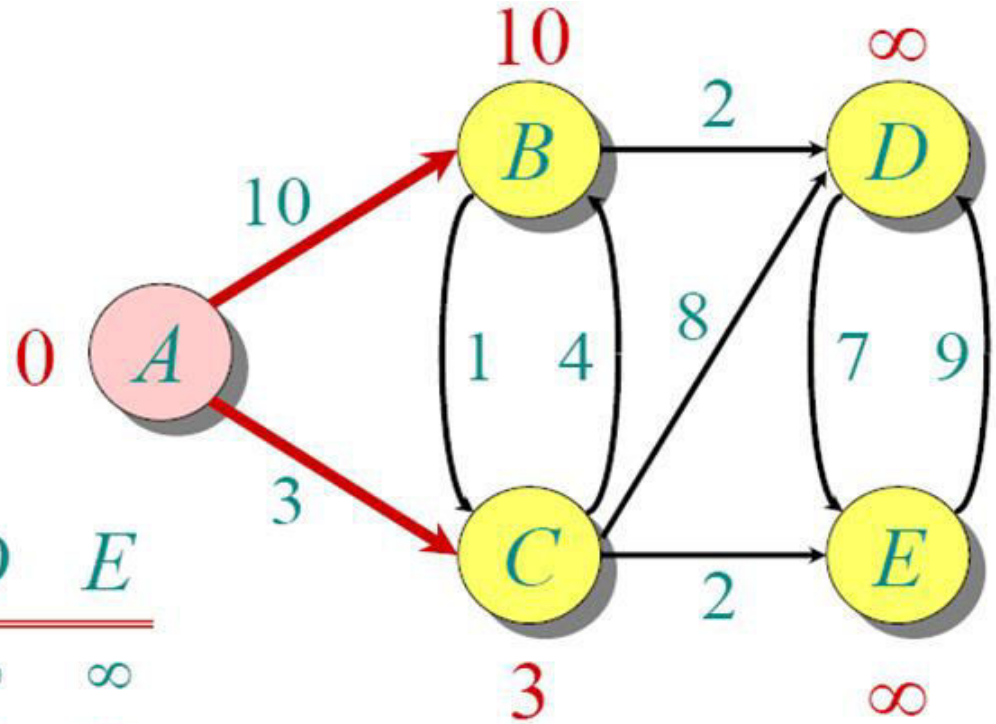
Node	Distance	Parent
A	0	
B	2	A
C	3	D
D	1	A
E	3	D
F	6	G
G	5	D

Pick vertex not in S with lowest cost (F) and update neighbors

Another Example



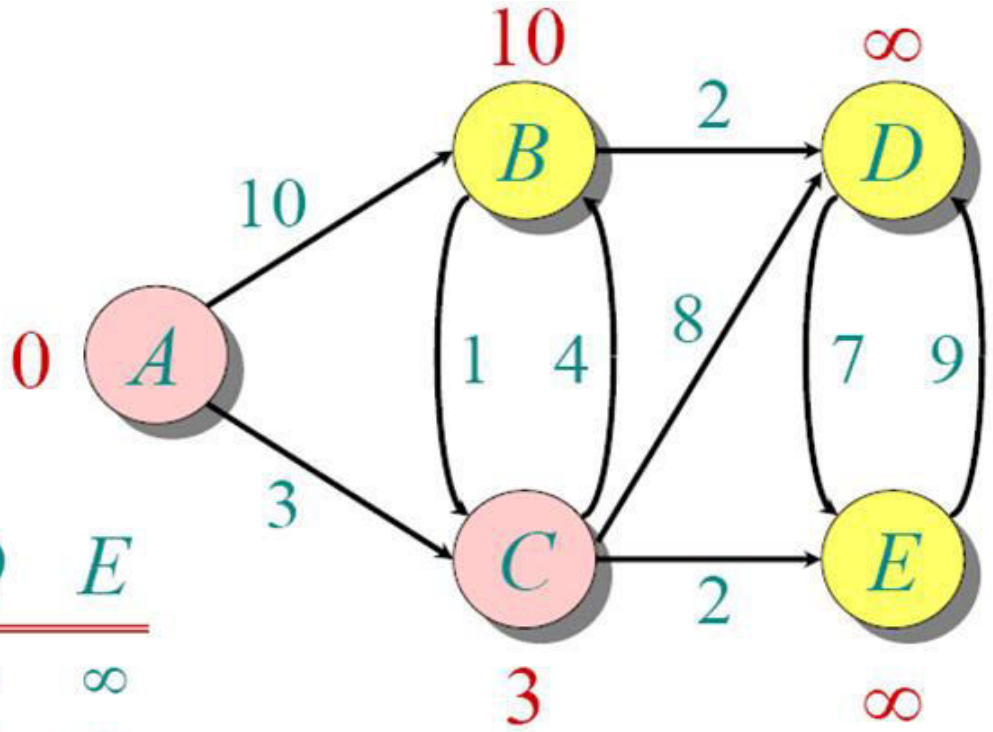
Example: Continued...



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞

S: { A }

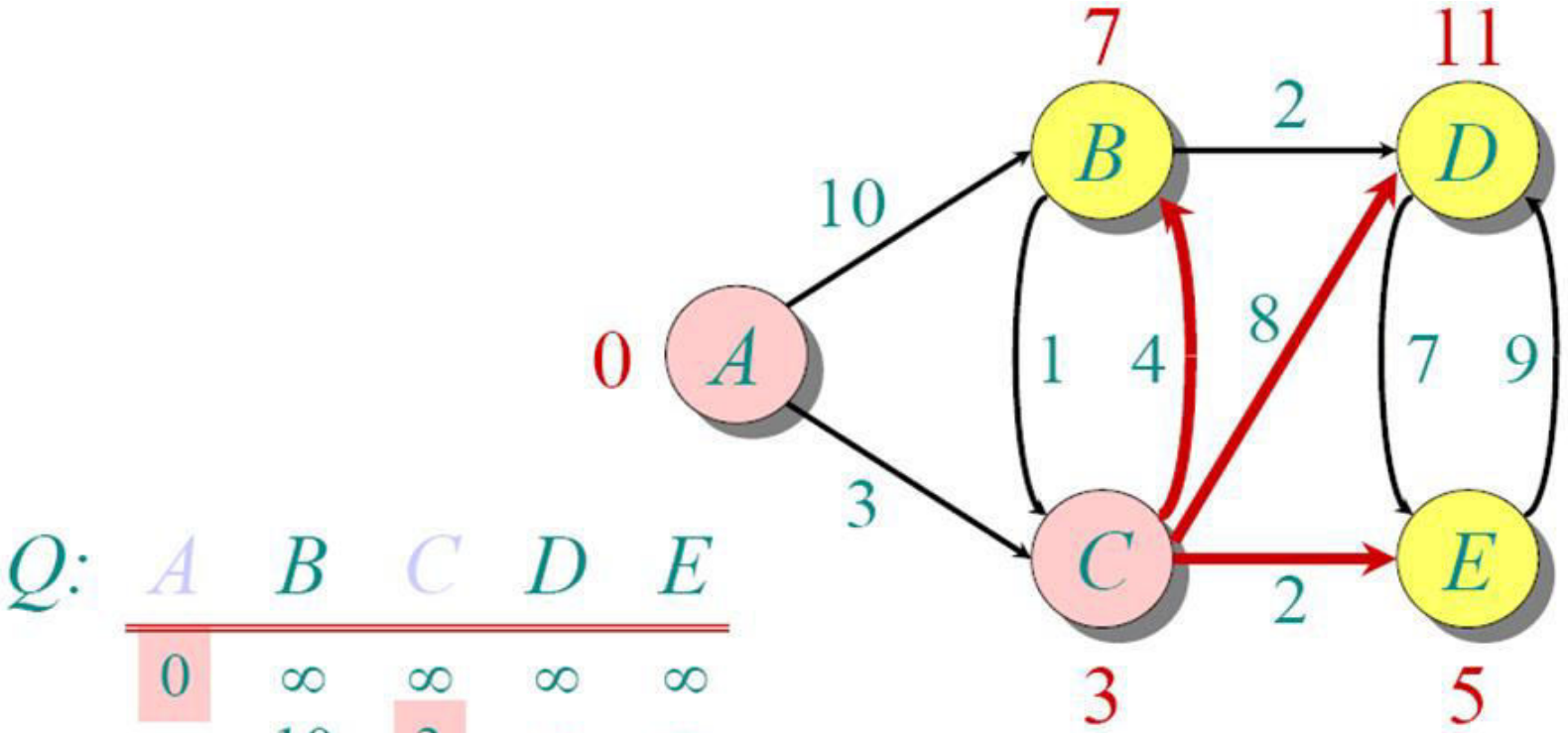
Example: Continued....



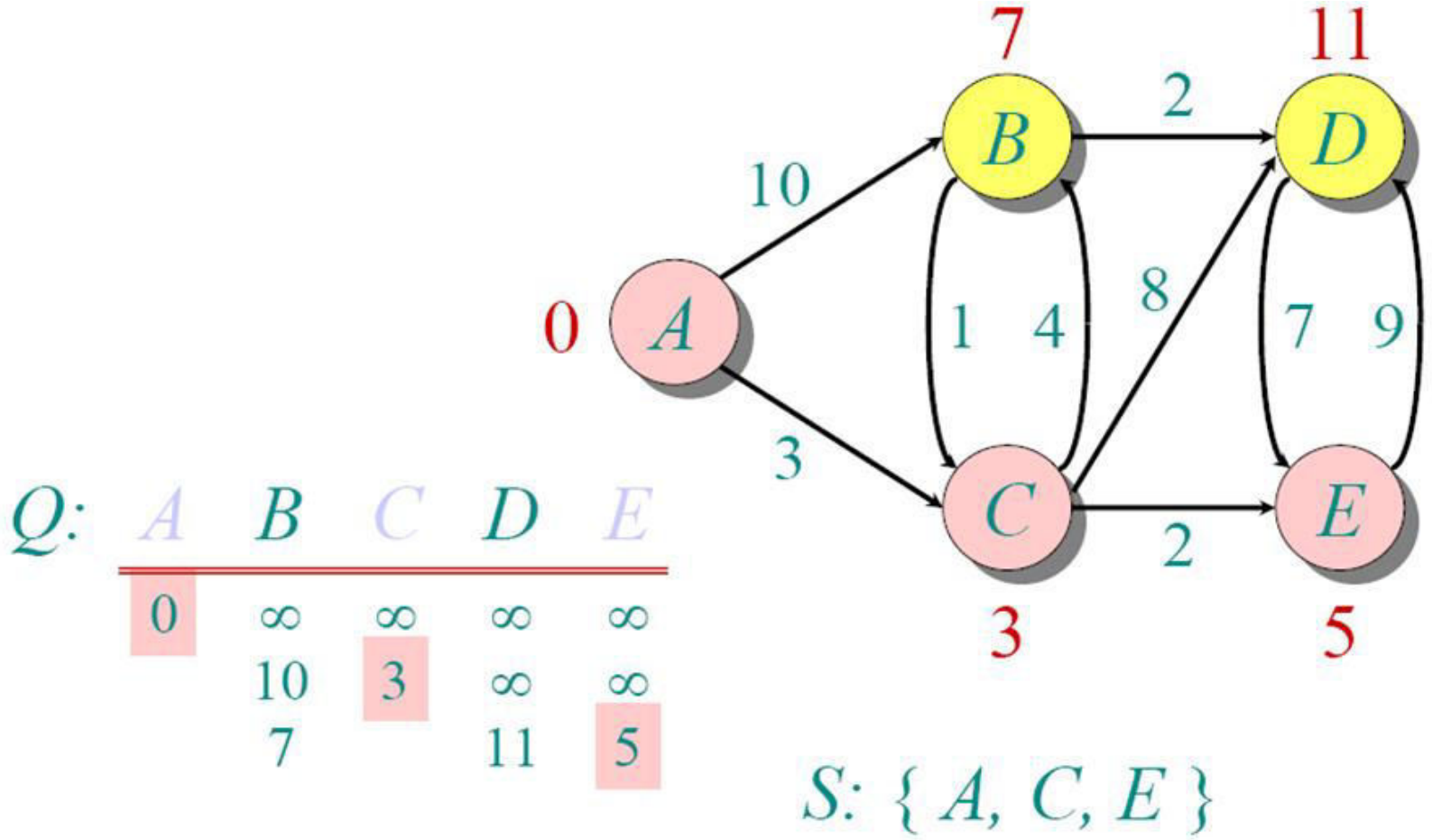
Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞

S: { A, C }

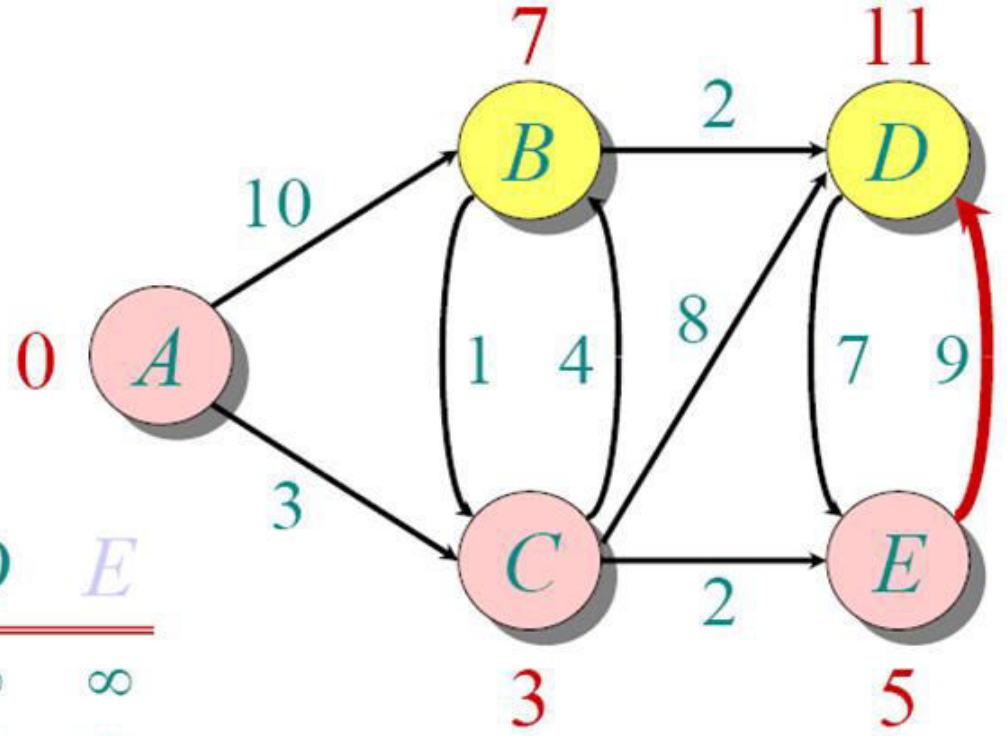
Example: Continued...



Example: Continued....



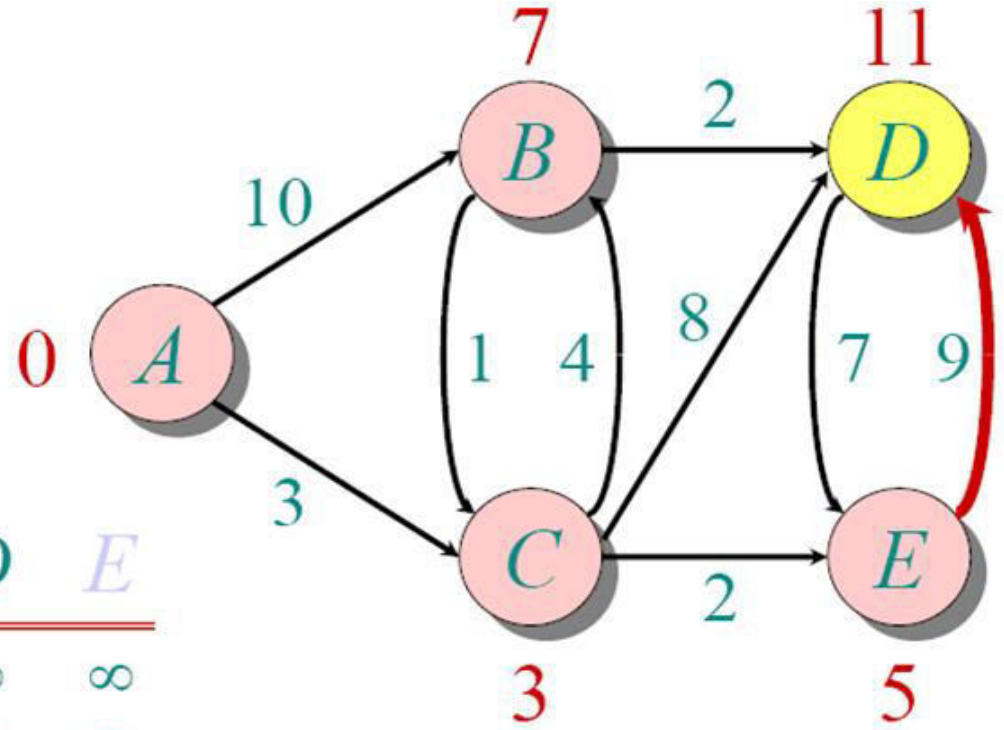
Example: Continued...



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

$S: \{ A, C, E \}$

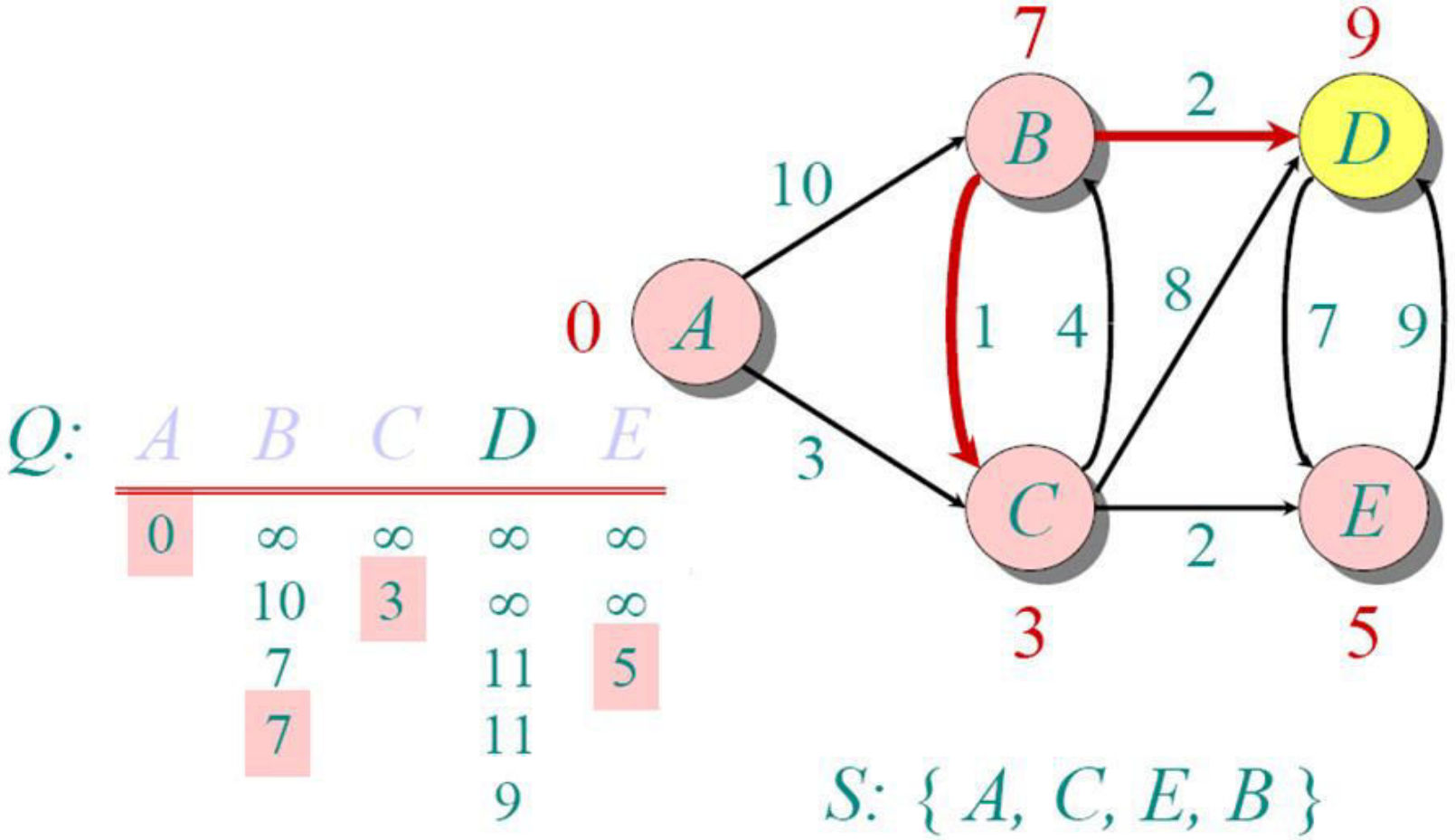
Example: Continued....



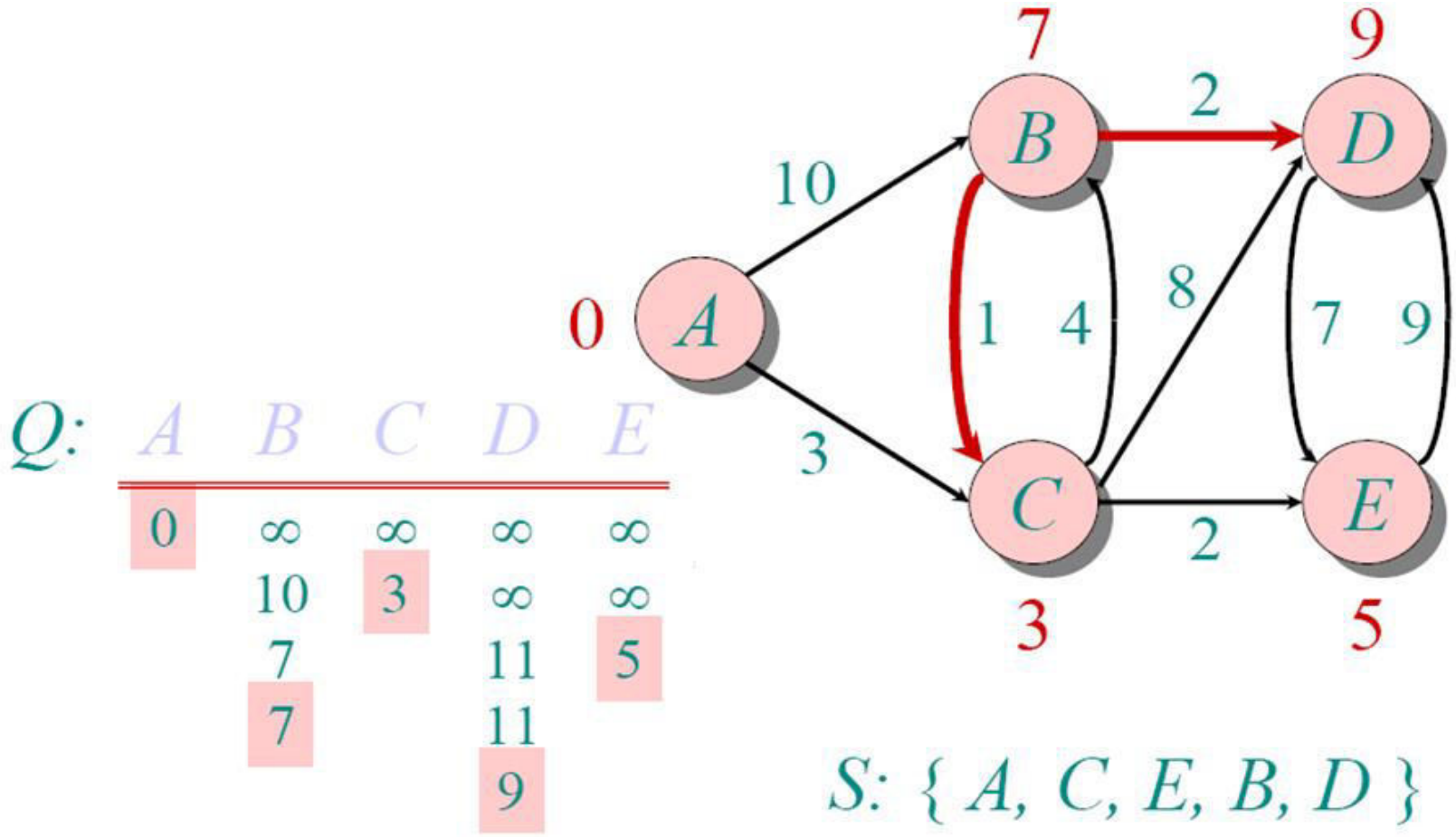
Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

S: { A, C, E, B }

Example: Continued....



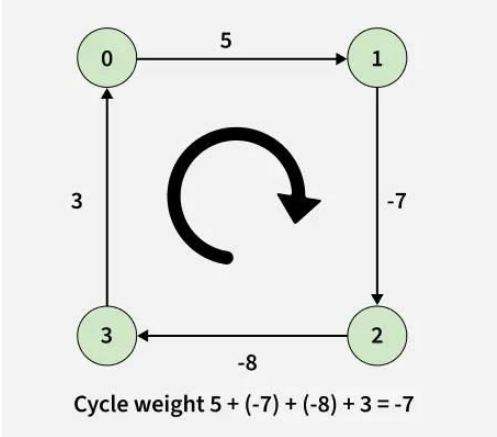
Example: Continued...



What if Negative Edge/Cycle??

Dijkstra is not suitable when the graph consists of negative edges. The reason is, it doesn't revisit those nodes which have already been marked as visited. If a shorter path exists through a longer route with negative edges, Dijkstra's algorithm will fail to handle it.

A negative weight cycle is a cycle in a graph, whose sum of edge weights is negative. If you traverse the cycle, the total weight accumulated would be less than zero.



In the presence of negative weight cycle in the graph, the shortest path doesn't exist because with each traversal of the cycle shortest path keeps decreasing.

Bellman–Ford Algorithm

Bellman-Ford *is a* ***single source*** *shortest path algorithm. It effectively works in the cases of negative edges and is able to detect negative cycles as well. It works on the principle of* ***relaxation of the edge***

```
function BellmanFord(G, source):
    for each vertex v in G:
        distance[v] = ∞
    distance[source] = 0

    for i from 1 to |V| - 1:
        for each edge (u, v) with weight w in G:
            if distance[u] + w < distance[v]:
                distance[v] = distance[u] + w

    for each edge (u, v) with weight w in G:
        if distance[u] + w < distance[v]:
            report "Negative-weight cycle detected"
            return
    return distance
```

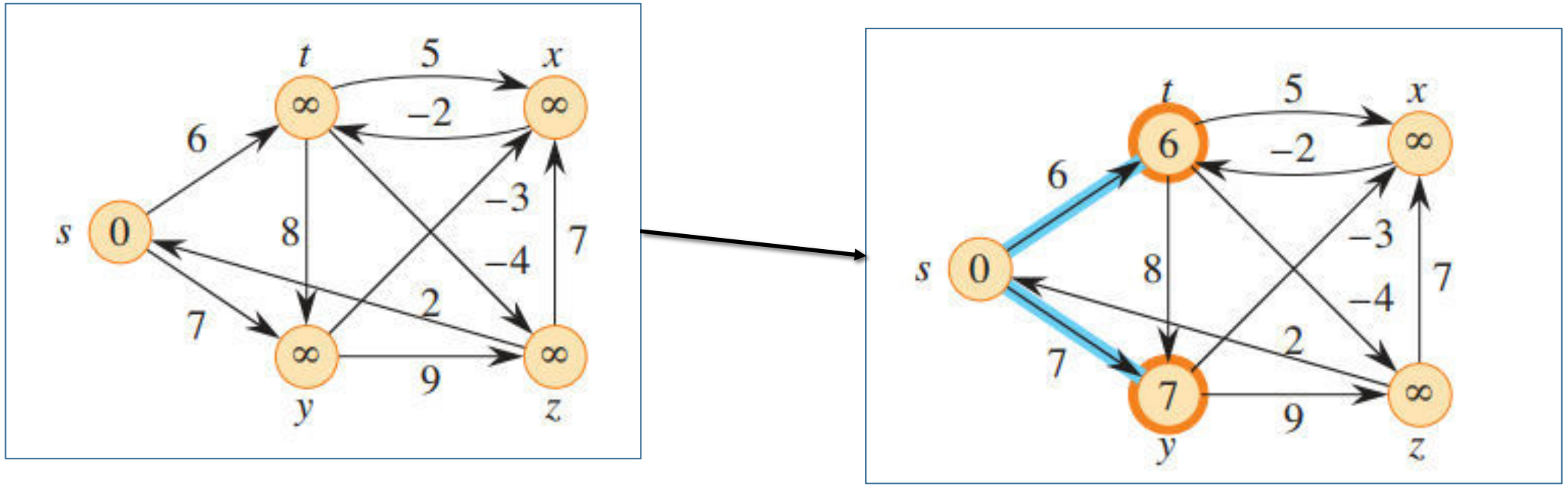
Set the distance to all other nodes as ∞ (infinity).

Set the distance to the source node as 0.

Relax all edges (V - 1) times

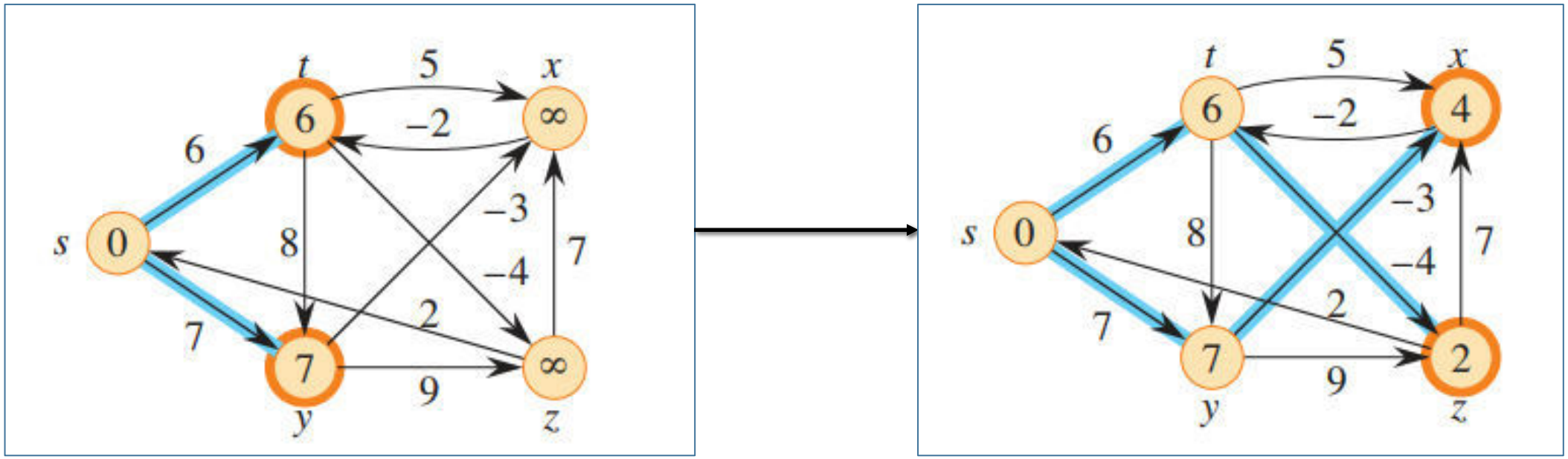
Check for negative-weight cycles (optional but important)

Bellman–Ford Algorithm - Example



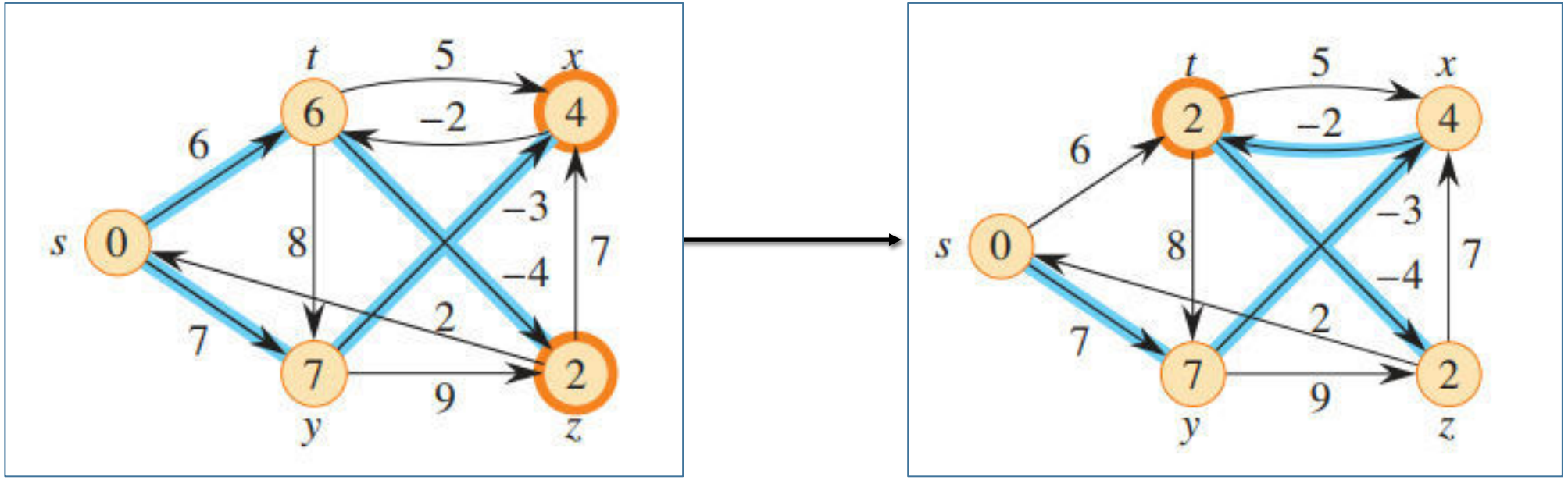
First Relaxation

Example Continued…



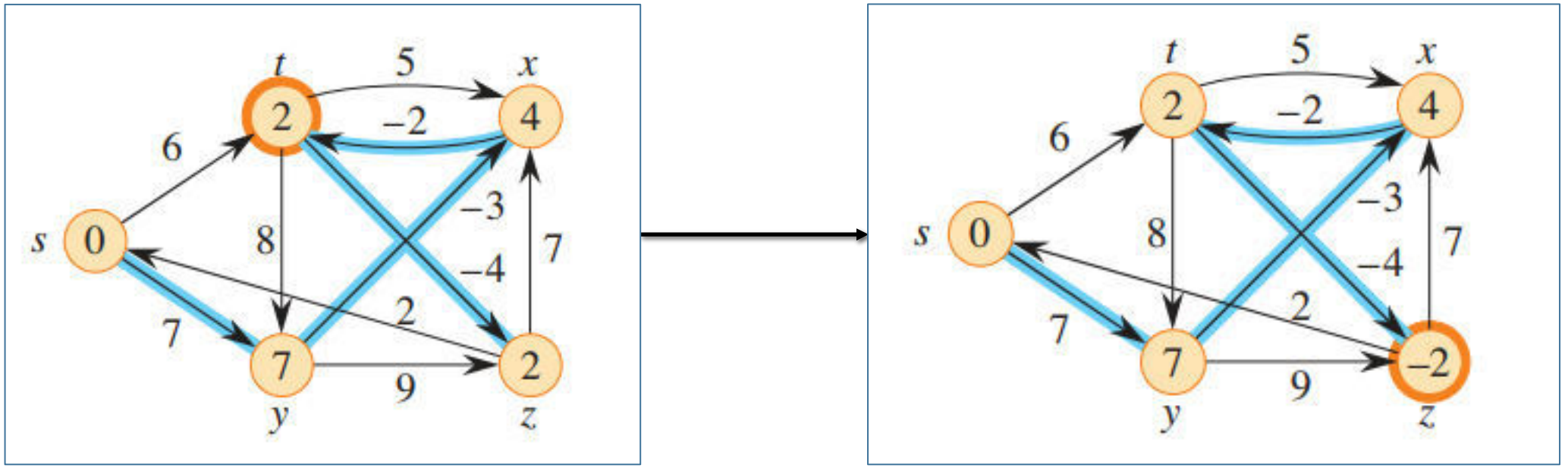
Second Relaxation

Example Continued…



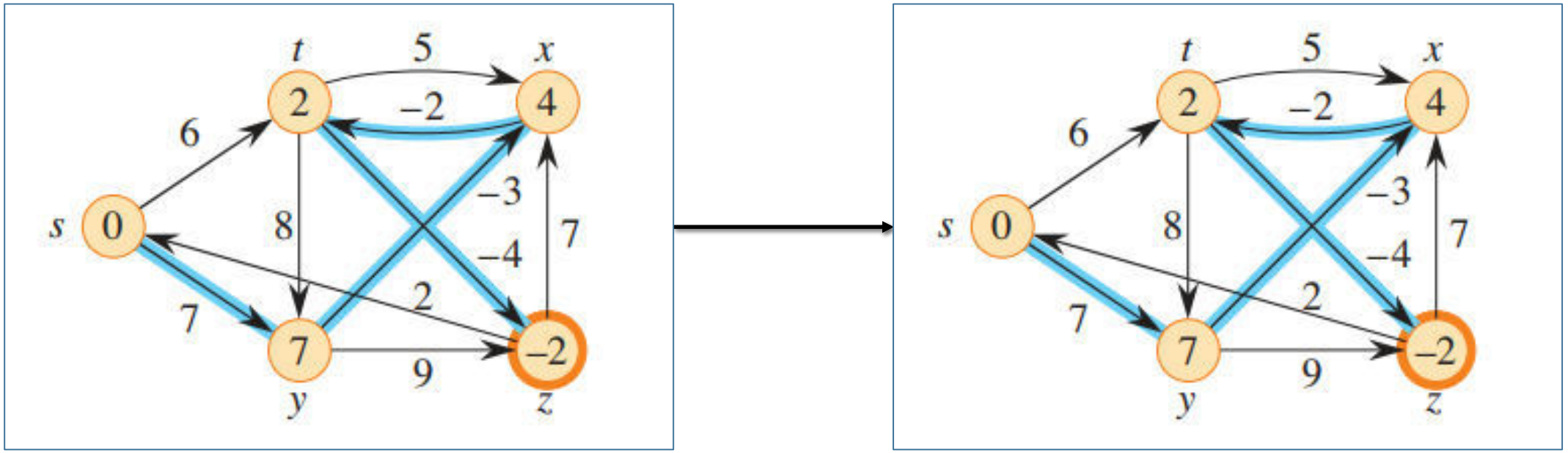
Third Relaxation

Example Continued…



Fourth Relaxation

Example Continued…

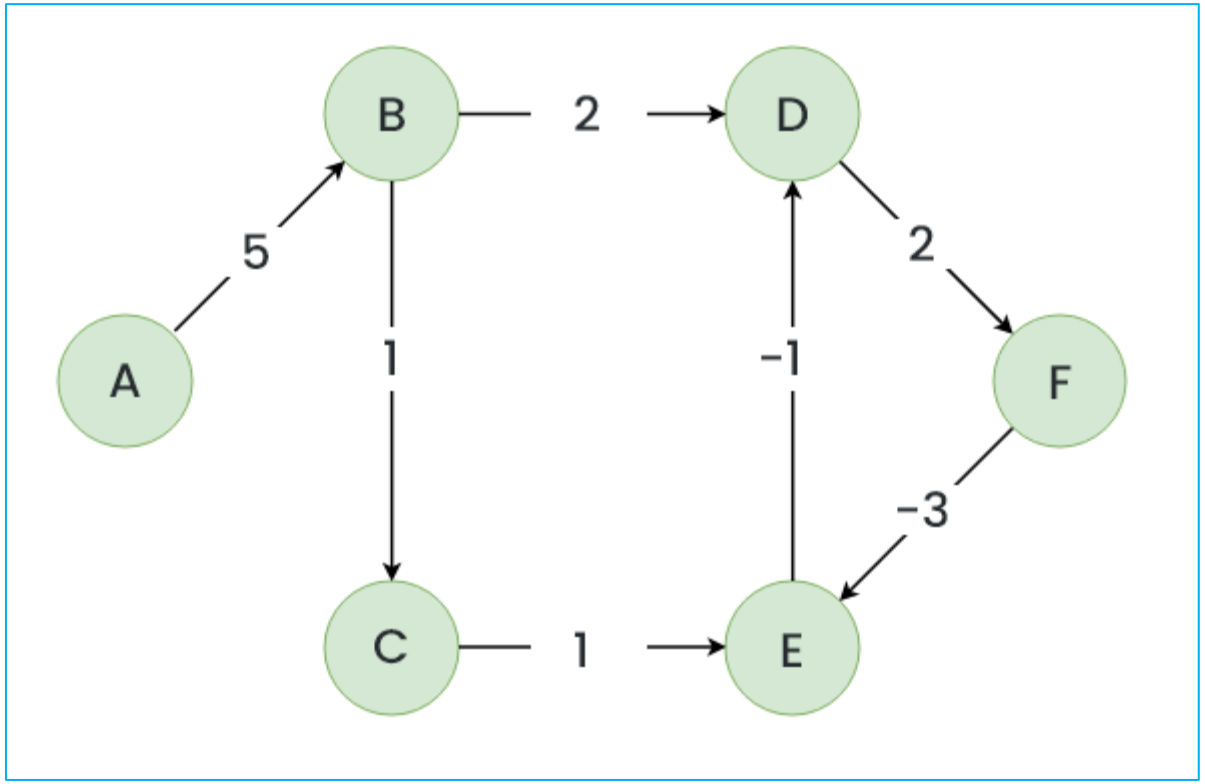


Fifth Relaxation

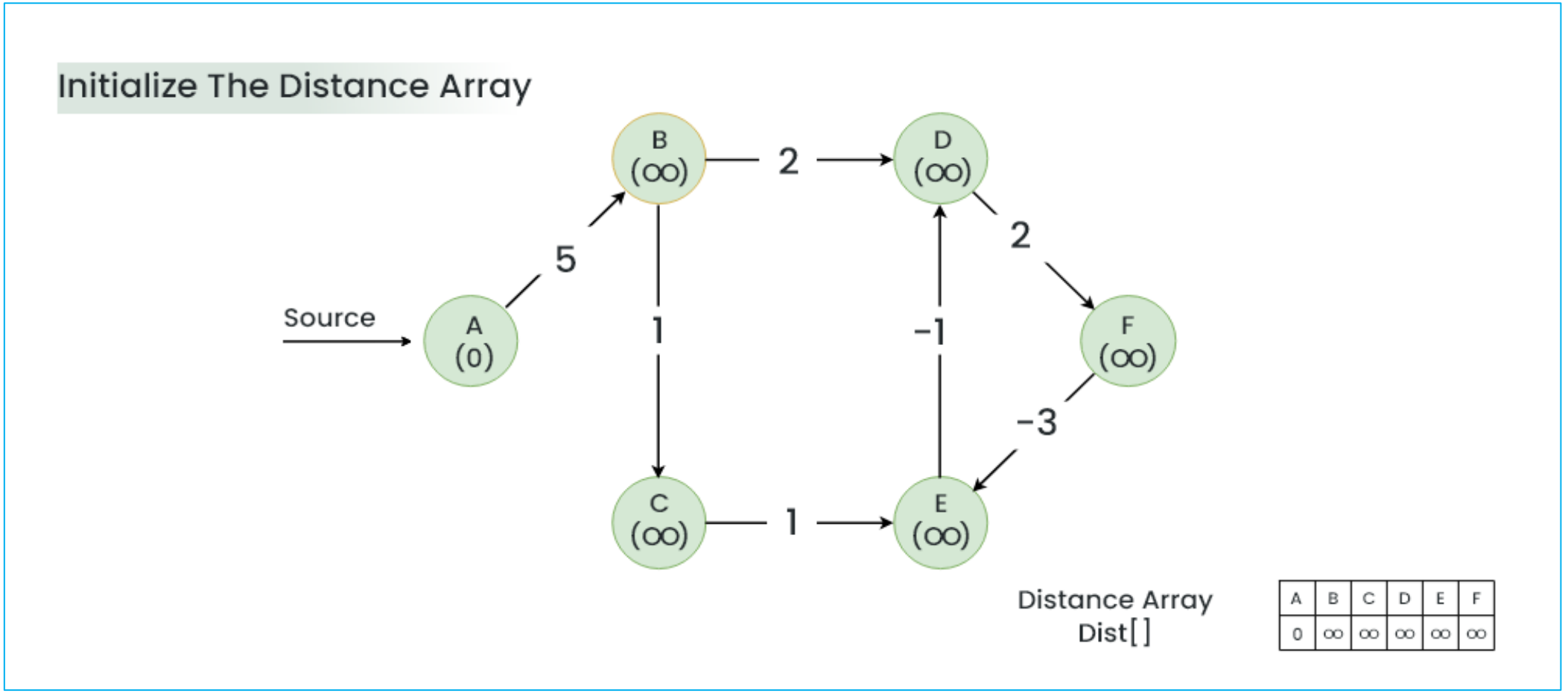
No updates found. Hence, no negative edge cycle present in the graph.

Another Example

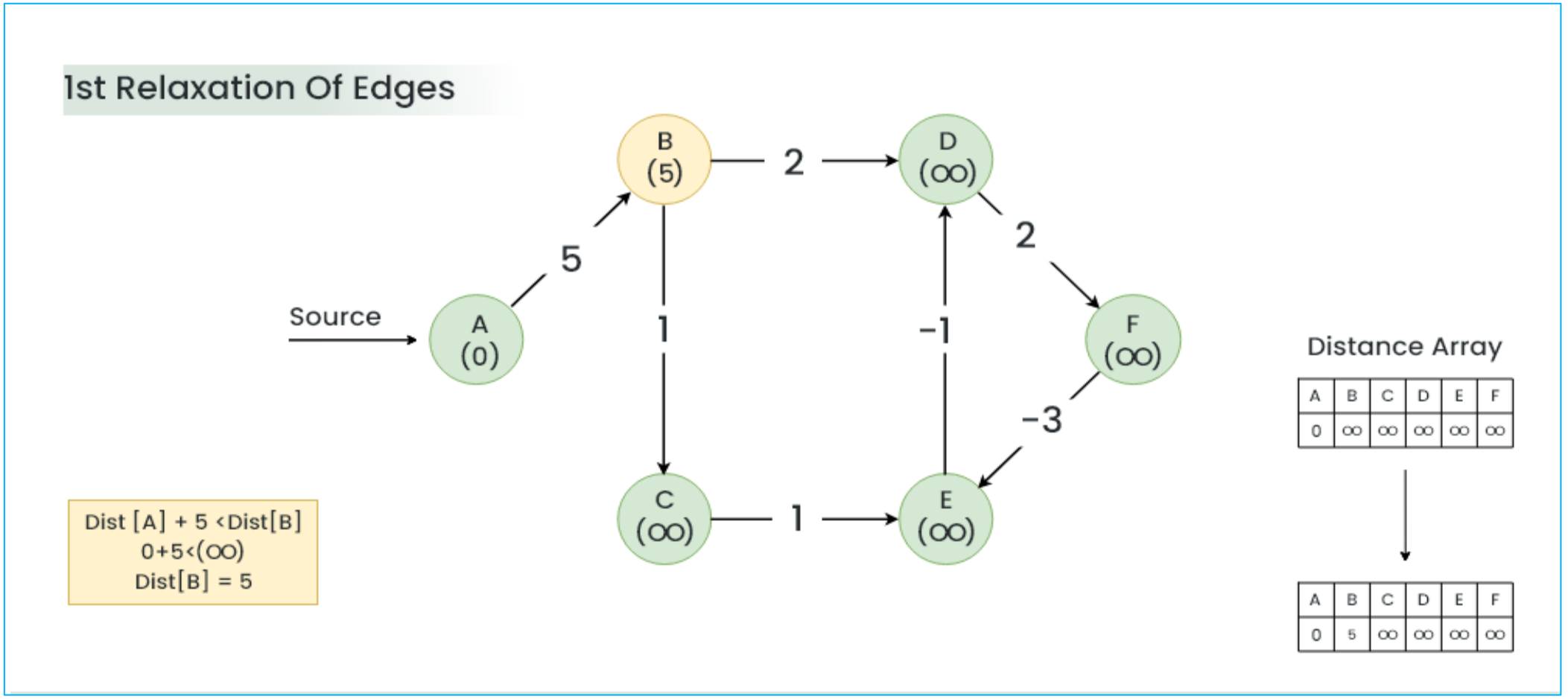
Let's suppose we have a graph which is given below and we want to find whether there exists a negative cycle or not using Bellman-Ford.



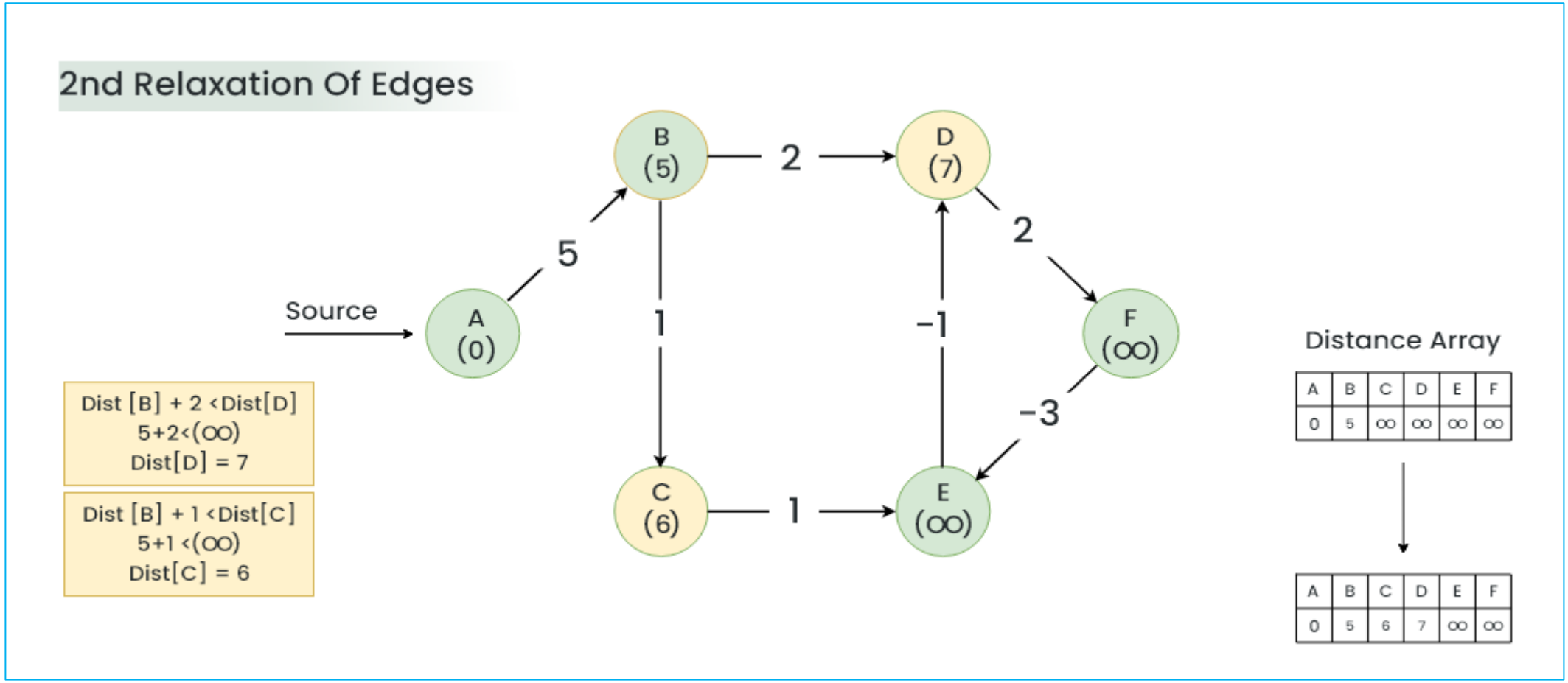
Example Continued…



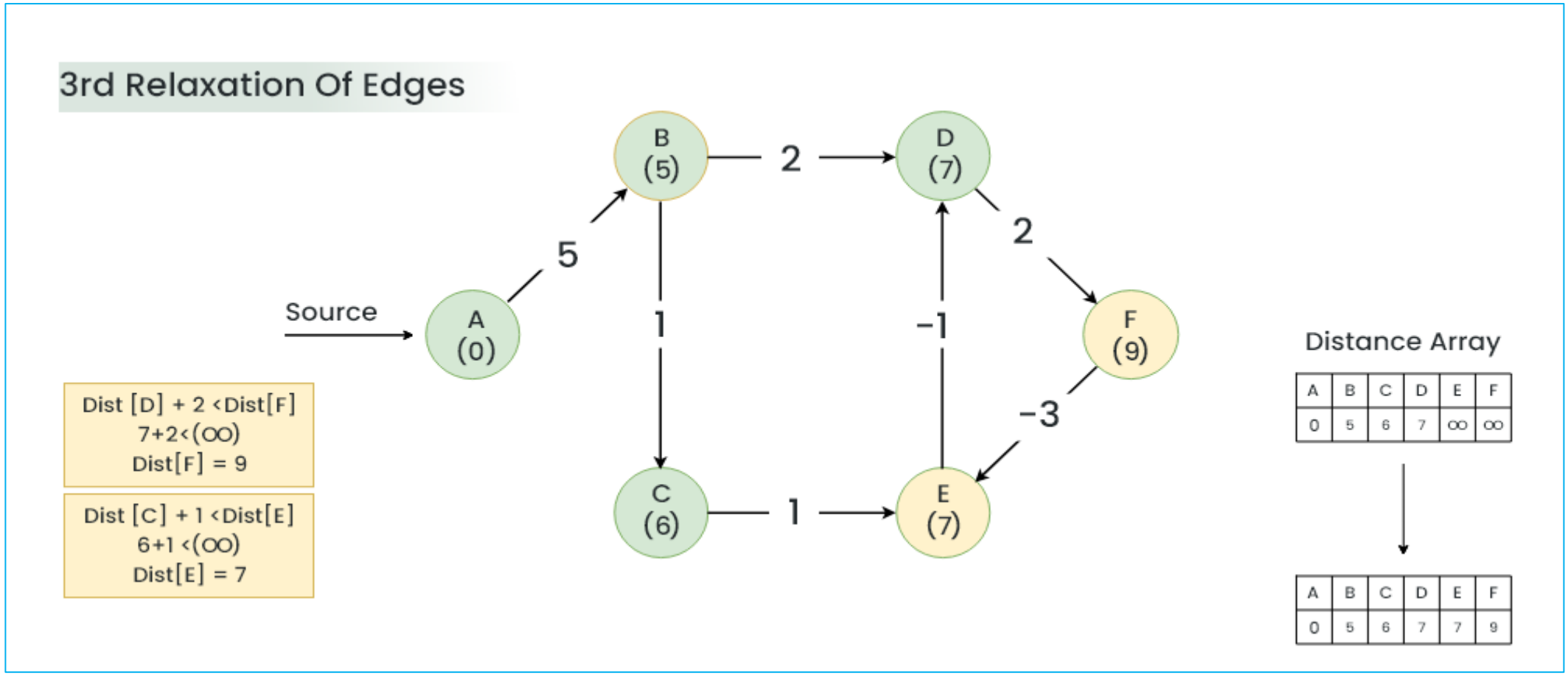
Example Continued…



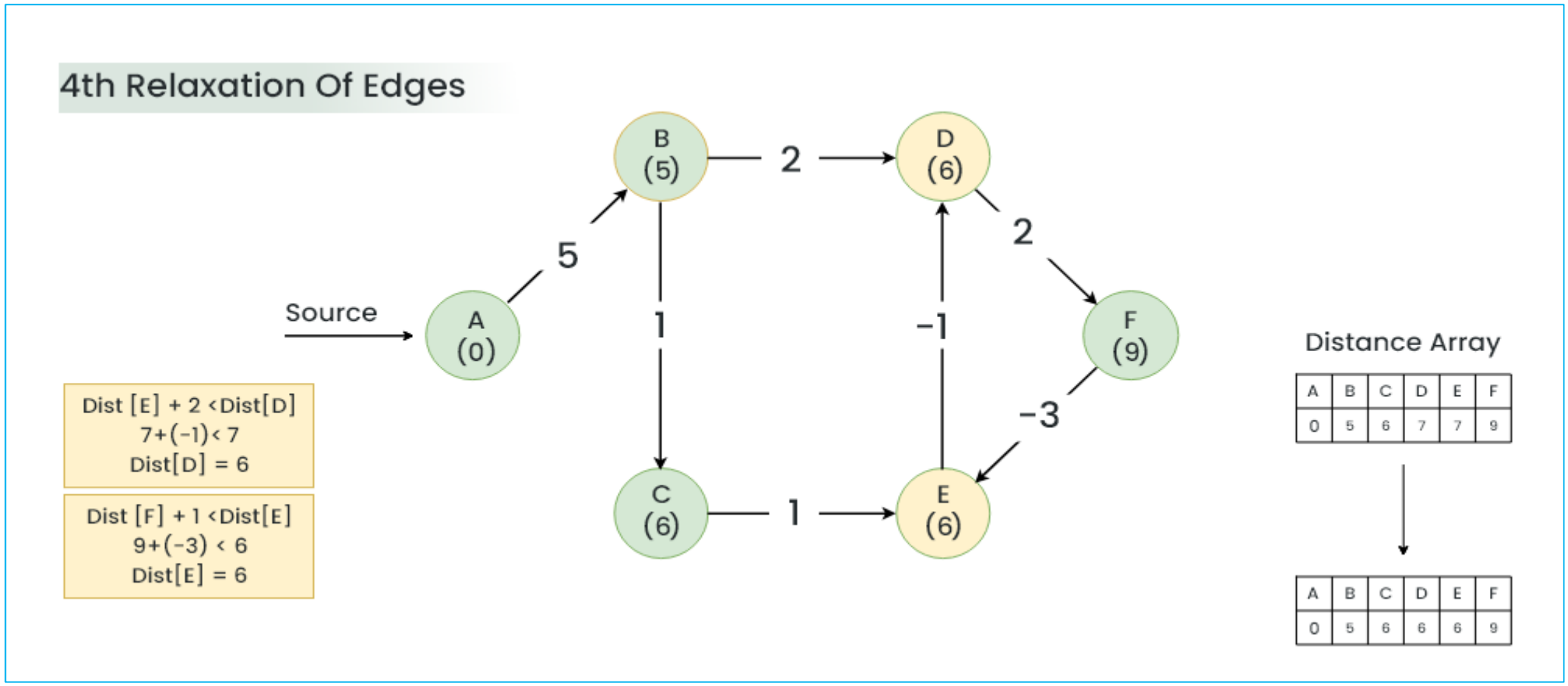
Example Continued…



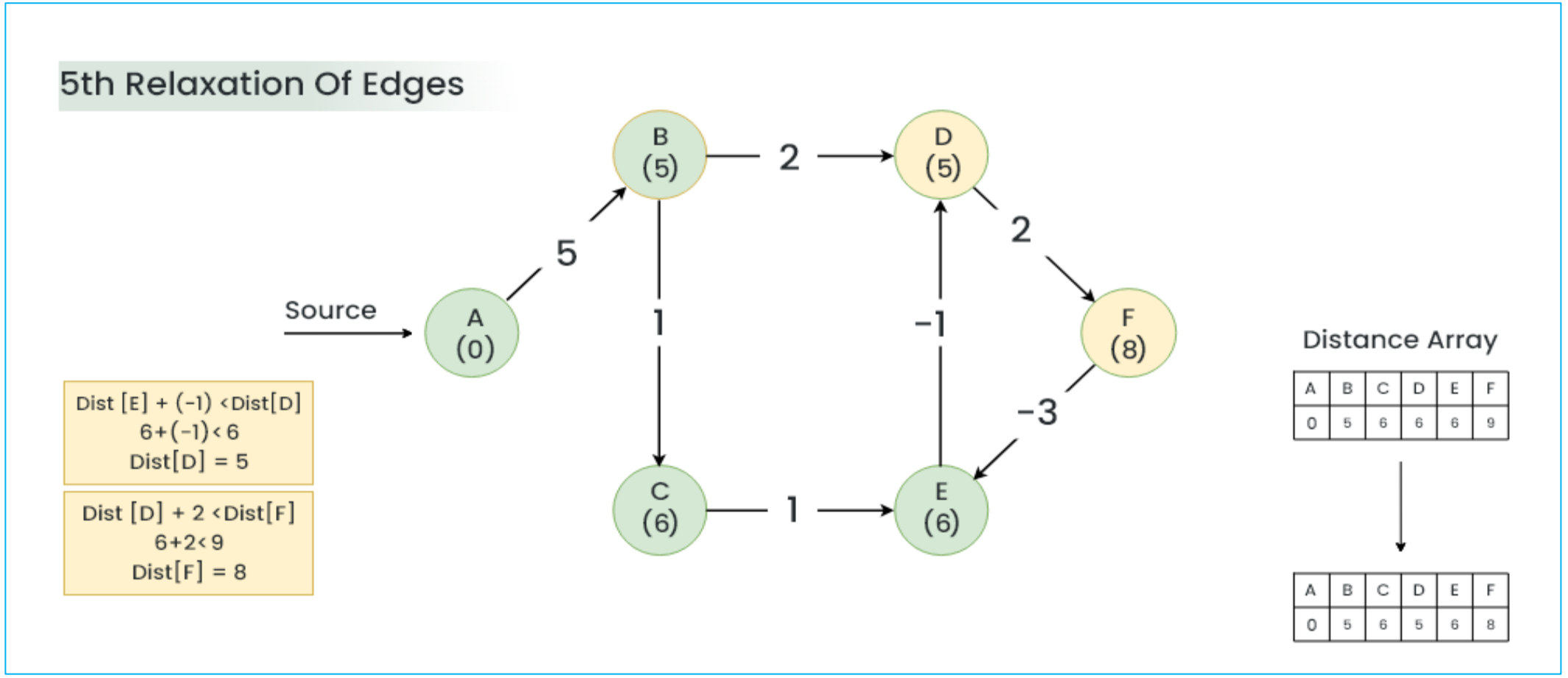
Example Continued…



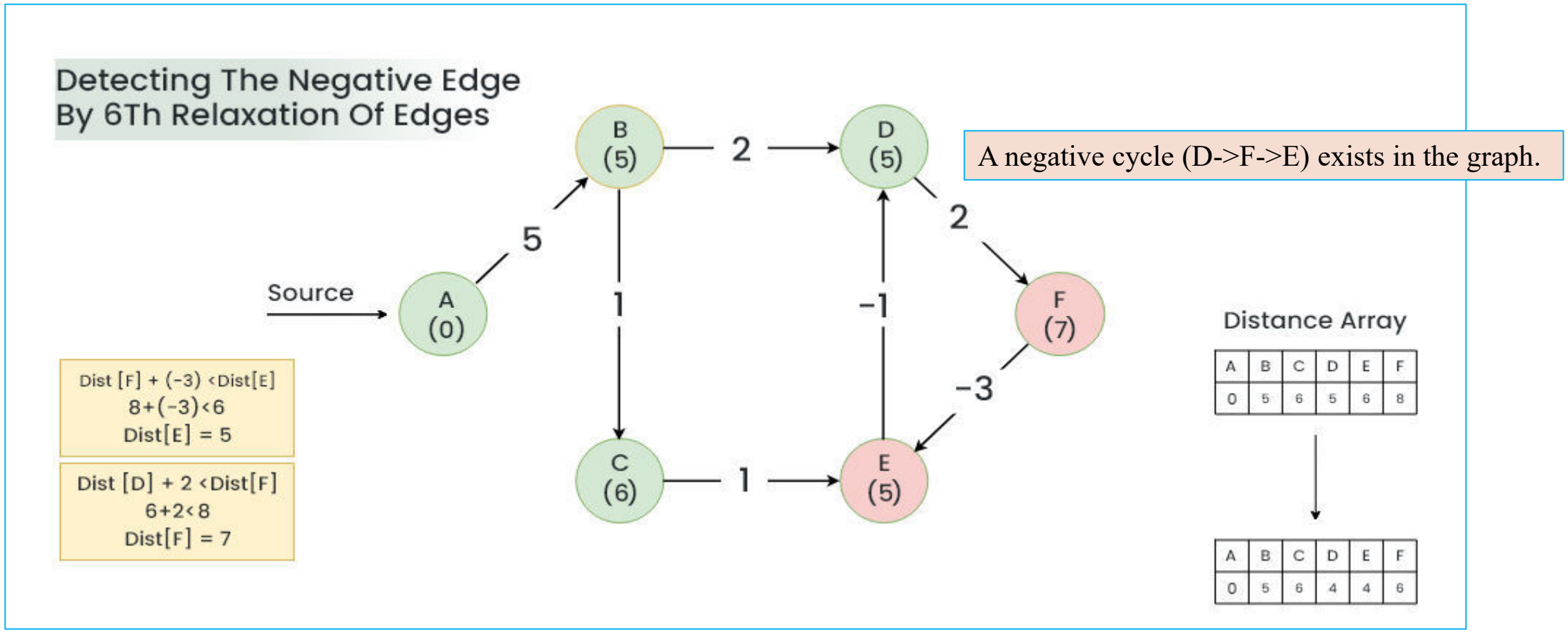
Example Continued…



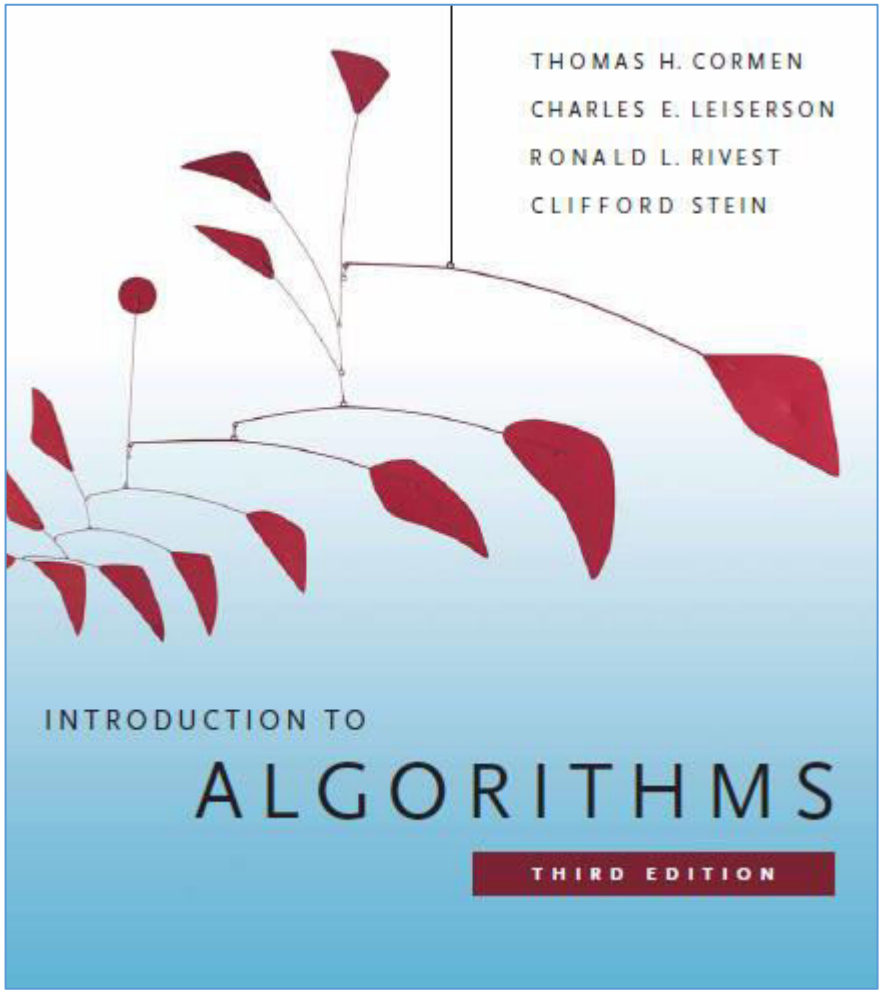
Example Continued…



Example Continued…



THANK YOU



Dynamic Programming

- LCS

Dynamic Programming (DP)

- Dynamic Programming is a strategy for designing algorithm.
  - » A meta-technique, not an algorithm (like divide & conquer)
  - » “Programming” in this context refers to a tabular method
  - » not to writing computer code.

Basic Principle

- Used when problem breaks down into recurring small subproblems
- Used when the solution can be recursively described in terms of solutions to subproblems (optimal substructure).
- Algorithm finds solutions to subproblems and stores them in memory for later use.

Dynamic Programming vs Divide and Conquer

Dynamic programming

Dynamic programming solves problems by combining the solutions to sub-problems.

Dynamic Programming vs Divide and Conquer

Are they same?

Dynamic Programming vs Divide and Conquer

- No!
- Divide-and-conquer algorithms partition the problem into **disjoint subproblems**, solve the subproblems recursively, and then combine their solutions to solve the original problem.
- In contrast, dynamic programming applies when the **subproblems overlap**, that is, when subproblems share sub-subproblems.

Dynamic Programming vs Divide and Conquer

Overlapping Subproblems

- In this context, a divide-and-conquer algorithm does more work than necessary, repeatedly solving the common sub-subproblems.
- A dynamic-programming algorithm solves each sub-subproblem **just once** and then saves its answer in a table.
- Avoids the work of re-computing the answer every time it solves each sub-subproblem.

Dynamic Programming: When Apply?

We typically apply dynamic programming to **optimization problems**.

- Such problems can have many possible solutions.
- Each solution has a value, and we wish to find a solution with the optimal (minimum or maximum) value.
- We call such a solution an optimal solution to the problem, as opposed to the optimal solution, since there may be several solutions that achieve the optimal value.

Using Dynamic Programming for Optimal Solution

- There are two approach to develop a dynamic programming solution:

  - **Top-Down (Memoization):** In this **recursive** approach, we write the procedure recursively in a natural manner, but modified to save the result of each subproblem (usually in an array). The procedure now first checks to see whether it has previously solved this subproblem. If so, it returns the saved value, saving further computation at this level; if not, the procedure computes the value in the usual manner. Here we start from the main problem and then break it down.

  - **Bottom-Up (Tabulation):** In this **iterative** approach, we start from the base case and build up the solution. When solving a particular subproblem, we have already solved all of the smaller subproblems its solution depends upon, and we have saved their solutions. We solve each subproblem only once, and when we first see it, we have already solved all of its prerequisite subproblems.

Dynamic Programing

Every DP problem can be solved in two ways

- Iteratively
- Recursively

Longest Common Subsequence

LCS Problem Statement: Given two sequences, find the length of longest subsequence present in both of them.

A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous.

Longest Common Subsequence

- What are the Subsequences of “ABC”
- Total way = 2^n
- Common Subsequence between two string and LCS

Examples:

LCS for input Sequences “ABCDGH” and “AEDFHR” is “ADH” of length 3.

LCS for input Sequences “AGGTAB” and “GXTXAYB” is “GTAB” of length 4.

- Brut force approach (if n > 25)

LCS : “HELLOM” and “HMLLD”

- Start with first element

HELLOM

HMLLD

- If they are same: include them into LCS

LCS : “HELLOM” and “HMLLD”

- Recursively solve the rest string

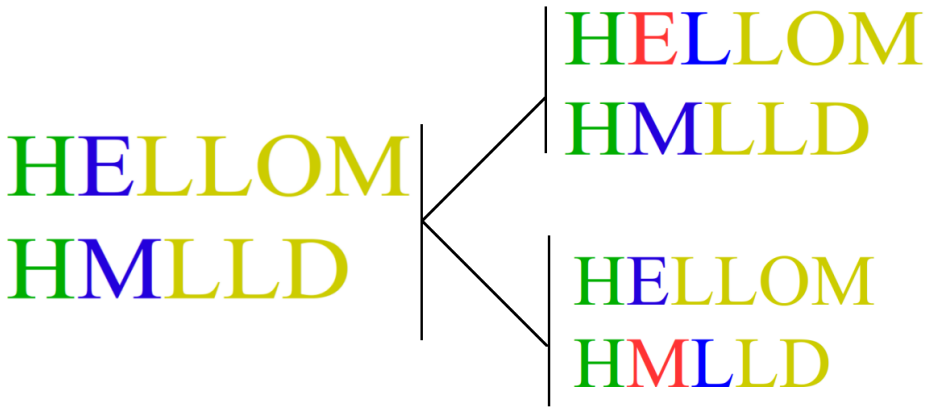
HELLOM

HMLLD

- We have sorter strings “ELLOM” and “MLLD”. Solve it recursively.
- Here E and M are not same. So, we have to drop one.

LCS : “HELLOM” and “HMLLD”

- Once we have to drop E and next we drop M.



- LCS(“ELLOM”, “MLLD”) ->Max[LCS(“LLOM”, “MLLD”) and LCS(“ELLOM”, “LLD”)]

Longest Common Subsequence

Theorem 15.1 (Optimal substructure of an LCS)
Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be sequences, and let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ be any LCS of X and Y.

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y.
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}.

Longest Common Subsequence: Base Case

- Let, x[m] represented by x[i] where i is 1 to m.
- Let, y[n] represented by y[j] where j is 1 to n.
- C[i, j] represents LCS.
- If i or j is 0, then, one of the sequences has length 0, and so the LCS has length 0.

Longest Common Subsequence: Base Case

- Let, x[m] represented by x[i] where i is 1 to m.
- Let, y[n] represented by y[j] where j is 1 to n.
- C[i, j] represents LCS.

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0\,, \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j\,, \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j\,. \end{cases}$$

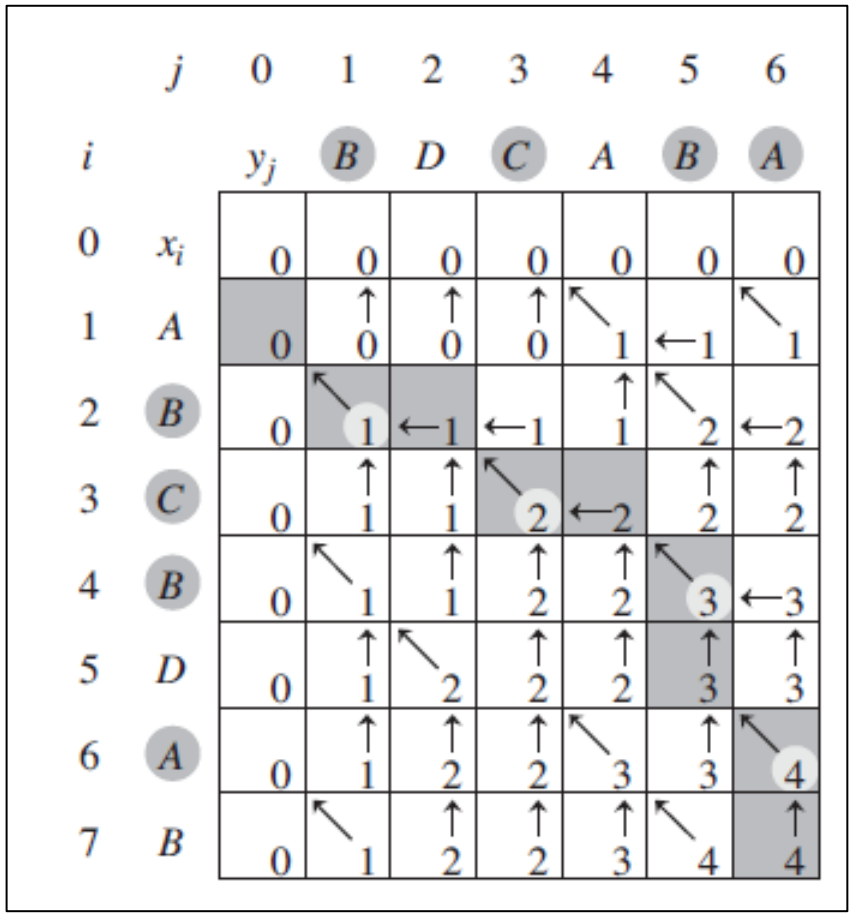
Longest Commc

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i-1,j-1]+1 & \text{if } i,j > 0 \text{ and } x_i = y_j, \\ \max(c[i,j-1], c[i-1,j]) & \text{if } i,j > 0 \text{ and } x_i \neq y_j. \end{cases}$$

i		j	0	1	2	3	4	5	6
			y_j	B	D	C	A	B	A
0	x_i		0	0	0	0	0	0	0
1	A		0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B		0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C		0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B		0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D		0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A		0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B		0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

```
LCS-LENGTH(X, Y)
 1  m = X.length
 2  n = Y.length
 3  let b[1..m, 1..n] and c[0..m, 0..n] be new tables
 4  for i = 1 to m
 5      c[i, 0] = 0
 6  for j = 0 to n
 7      c[0, j] = 0
 8  for i = 1 to m
 9      for j = 1 to n
10          if x_i == y_j
11              c[i, j] = c[i − 1, j − 1] + 1
12              b[i, j] = "↖"
13          elseif c[i − 1, j] ≥ c[i, j − 1]
14              c[i, j] = c[i − 1, j]
15              b[i, j] = "↑"
16          else c[i, j] = c[i, j − 1]
17              b[i, j] = "←"
18  return c and b
```

Longest Common Subsequence



i		j	0	1	2	3	4	5	6
		y_j		B	D	C	A	B	A
0	x_i		0	0	0	0	0	0	0
1	A		0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B		0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C		0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B		0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D		0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A		0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B		0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

```
PRINT-LCS(b, X, i, j)
1  if i == 0 or j == 0
2       return
3  if b[i, j] == "↖"
4       PRINT-LCS(b, X, i − 1, j − 1)
5       print x_i
6  elseif b[i, j] == "↑"
7       PRINT-LCS(b, X, i − 1, j)
8  else PRINT-LCS(b, X, i, j − 1)
```

Longest Common Subsequence : Recursviely

```cpp
#define MAXC 1000
char A[MAXC],B[MAXC];
int lenA,lenB;
int dp[MAXC][MAXC];
bool visited[MAXC][MAXC];
int calcLCS(int i,int j)
{
    if(A[i]=='\0' or B[j]=='\0') return 0;
    if(visited[i][j])return dp[i][j];

    int ans=0;
    if(A[i]==B[j]) ans=1+calcLCS(i+1,j+1);
    else
    {
        int val1=calcLCS(i+1,j);
        int val2=calcLCS(i,j+1);
        ans=max(val1,val2);
    }
    visited[i][j]=1;
    dp[i][j]=ans;
    return dp[i][j];
}
int main() {
    scanf("%s%s",A,B);
    lenA=strlen(A);
    lenB=strlen(B);
    printf("%d\n",calcLCS(0,0));
    return 0;

}
```

```cpp
string ans;
void printLCS(int i,int j)
{
    if(A[i]=='\0' or B[j]=='\0'){
        cout<<ans<<endl;
        return;
    }
    if(A[i]==B[j]){
        ans+=A[i];
        printLCS(i+1,j+1);
    }
    else
    {
        if(dp[i+1][j]>dp[i][j+1]) printLCS(i+1,j);
        else printLCS(i,j+1);
    }
}
```

Longest Common Subsequence :

```
int main()
{   int m=0;
    int n=0;
    cout<<"The length of 1st sequence:";
    cin>>m;
    cout<<"The first sequence:";
    char s1[m];
    for(int i=0;i<m;i++)
    {
        cin>>s1[i];
    }
    cout<<"The length of 2nd sequence:";
    cin>>n;
    cout<<"The second sequence:";
    char s2[n];
    for(int j=0;j<n;j++)
    {
        cin>>s2[j];
    }
    lcsAlgo(s1,s2,m,n);
}
```

```
#include<bits/stdc++.h>
using namespace std;
void lcsAlgo(char *s1,char *s2,int m,int n)
{
    int lcsTable[m+1][n+1];
    for(int i=0;i<=m;i++)
    {for(int j=0;j<=n;j++)
        {if(i==0 || j==0)
            {
                lcsTable[i][j]=0;
            }
            else if(s1[i-1]==s2[j-1])
            {
                lcsTable[i][j]=lcsTable[i-1][j-1]+1;
            }
            else
            {
                lcsTable[i][j]=max(lcsTable[i-1][j],lcsTable[i][j-1]);
            } } }
int lcs_length=lcsTable[m][n];
```

Longest Common Subsequence :

```
#include<bits/stdc++.h>
using namespace std;
void lcsAlgo(char *s1,char *s2,int m,int n)
{
   int lcsTable[m+1][n+1];
   for(int i=0;i<=m;i++)
   {for(int j=0;j<=n;j++)
      {if(i==0 || j==0)
         {
            lcsTable[i][j]=0;
         }
         else if(s1[i-1]==s2[j-1])
         {
            lcsTable[i][j]=lcsTable[i-1][j-1]+1;
         }
         else
         {
            lcsTable[i][j]=max(lcsTable[i-1][j],lcsTable[i][j-1]);
         } } }
int lcs_length=lcsTable[m][n];
char lcs[lcs_length];
   lcs[lcs_length]='\0';
   int i=m,j=n;
   while(i>0 && j>0)
   {  if(s1[i-1]==s2[j-1])
      {
         lcs[lcs_length-1]=s1[i-1];
         i--;
         j--;
         lcs_length--;
      }
      else if(lcsTable[i-1][j]>lcsTable[i][j-1])
         i--;
      else
         j--; }
   lcs_length=lcsTable[m][n];
   cout<<"The first sequence:";
   cout<<s1<<endl;
   cout<<"The second sequence:";
   cout<<s2<<endl;
   cout<<"The LCS:"<<lcs<<endl;
   cout<<"LCS length:"<<lcs_length;}
```

Lecture – 17 (problem solving)

Problem-01: Be Positive

Input:

First line: t (number of test cases)
For each test case:
First line: n ($1 \leq n \leq 8$)
Second line: n integers, each -1, 0, or 1

Output:
For each test case: minimum operations to make product > 0

Algorithm:

- Count number of -1's (let's call it neg)
- Count number of 0's
- The product is positive if:
  - Even number of negative numbers → positive
  - Odd number of negatives → negative (unless we fix it)
- If there are zeros, we must turn at least one into 1 (costs 1 op per zero → 1)
- If odd number of -1's and no zeros → turn one -1 into 0 (cost 1), then 0 into 1 (cost 1) → total 2
- Otherwise: just turn one -1 into 1 (cost 2: -1 → 0 → 1)

Code:

```cpp
#include <iostream>
using namespace std;

int main() {
    int t;
    cin >> t;
    while (t--) {
        int n,e;
        cin >> n;
        int result = 0, temp = 0;
        for (int i = 0; i < n; i++)
        {
            cin >> e;
            if (e == 0) result++;
            else if (e == -1) temp++;
        }
        if (temp % 2 == 1)
        {
            result= result+2;
        }
        cout << result << "\n";
    }
    return 0;
}

```

Problem-02: Sublime Sequence

Input:
First line: t ($1 \le t \le 100$) - the number of test cases.
Each test case: two integers x and n ($1 \le x, n \le 10$)

Output:
Sum of sequence: x, -x, x, -x, ..., (n terms starting with x)

Algorithm:

- Sequence: x, -x, x, -x, ..., length n
- Count how many -x
- Count how many +x
- If n odd → sum = x
- If n even → sum = 0

Code:

```cpp
#include <iostream>
using namespace std;

int main() {
    int t;
    cin >> t;
    while (t--) {
        int x, n;
        cin >> x >> n;
        if (n % 2 == 0) cout << 0 << endl;
        else cout << x << endl;
    }
    return 0;
}
```

Problem-03: Maple and Multiplication

Input:
First line: t (number of test cases)
Each test case: two positive integers a and b ($1 \le a, b \le 10^9$)

Output:
Minimum operations to make a == b by multiplying a or b by any x > 1

Algorithm:

- If:
  a == b → already equal → **0 operations**
  Else:
  - Let big = max(a,b)
  - Let small = min(a,b)

- If:
  big % small == 0 → multiply once → **1 operation**

  Else:
  need two multiplications → **2 operations**

Code:

```cpp
#include <iostream>
using namespace std;

int main() {
    int t;
    cin >> t;
    while (t--) {
        int a, b;
        cin >> a >> b;
        if (a == b) cout << 0 << endl;
        else if (a % b == 0 || b % a == 0) cout << 1 << endl;
        else cout << 2 << endl;
    }
    return 0;
}
```

Problem-04: Binary Imbalance

Input:
First line: t - the number of test cases.
Each test case:
n (length of string)
string s of '0' and '1'

Output:

"YES" if you can make number of 0s > number of 1s by inserting '0' or '1' between positions, else "NO"

Algorithm:

- Count number of '0' → z
- Count number of '1' → o
- if (z > o) → YES
  else → NO

Code:

```cpp
#include <bits/stdc++.h>
using namespace std;
int main() {
    int t; cin >> t;
    while (t--) {
        int n,temp=0;
        cin >> n;
        char s[n];
        for(int i=0;i<n;i++)
        {
            cin>>s[i];
            if(s[i]=='0')
            {
                temp++;
            }
        }
        if(temp>0) cout<<"Yes";
        else cout<<"No";
    }
}
```

Lecture – 18 (problem solving)

Problem-05: Shortest Increasing Path

Input:
First line: t (1 ≤ t ≤ 10^4) - the number of test cases.
Each test case: two integers x y (1 ≤ x, y ≤ 10^9)

Output:
Minimum steps to reach (x,y) with alternating x/y moves and strictly increasing step length
Or -1 if impossible.

Algorithm:

- If x == y → can reach in 2 steps: right x, up x → but wait, steps must increase
- Actually impossible in most cases except specific ones

Code:

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    int t;
    cin>>t;
    while(t--)
    {
        long x,y;
        cin>>x;
        cin>>y;

        if(x==y || x==y+1 || y==1)
        {
            cout<<-1<<endl;
        }
        else if(x<y)
        {
            cout<<2<<endl;
        }
        else cout<<3;
    }

}
```

Problem-06: Shift Sort

Input:
First line: t (1 ≤ t ≤ 100) - the number of test cases.
Each test case: First line a single integer n(3 ≤ n ≤ 100) – length of the string
Second line a binary string s of length n

Output:
For each test case, output a single integer — the minimum number of operations required to sort the given binary string.

Algorithm:

- Loop through array
- Count how many times: a[i] > a[i+1]
- If broken **0 or 1 times** → sortable by shift
- If broken **more than 1 time** → impossible

Code:

```cpp
#include <iostream>
using namespace std;
int main() {
    int t;
    cin >> t;
    while (t--) {
        int n;
        cin >> n;
        string s;
        cin >> s;
        int z = 0;
        for (int i = 0; i < n; i++) if (s[i] == '0') z++;
        int cnt = 0;
        for (int i = 0; i < z; i++) if (s[i] == '1') cnt++;
        cout << cnt << "\n";
    }
    return 0;
}

```

Problem-07: Only One Digit

Input:

First line: t ($1 \leq t \leq 1000$) - the number of test cases.
Each test case: First line contains one integer x ($1 \leq x \leq 1000$)

Output:

For each test case, output one integer *y* — the minimum non-negative number that satisfies the condition.

Algorithm:

- Sort the number ascending order.
- Track **minimum digit**
- Output that digit only

Code:

```cpp
#include <bits/stdc++.h>
using namespace std;
int main() {
    int t;
    cin >> t;
    while (t--) {

        string s;
        cin>>s;
        sort(s.begin(),s.end());
        cout<<s[0]<<endl;
    }
    return 0;
}


```

Problem-08: Mirror Word

Input:
First line: T (number of test cases)
Next T lines: a string made of b, o, d only

Output:
For each test case, Print the string as it appears in the mirror.

Algorithm:

- Reverse the input string
- Replace each character using mirror rule

Code:

```cpp
#include <iostream>
using namespace std;

int main() {
    int t;
    cout<<"Test Case: ";
    cin>>t;
    while(t--){
    string s;
    cin >> s;

    string ans = "";

    for (int i = s.size() - 1; i >= 0; i--) {

        if (s[i] == 'b') {
            ans = ans + 'd';
        }
        else if (s[i] == 'd') {
            ans = ans + 'b';
        }
        else {
            ans = ans + 'o';
        }
    }

    cout << ans;
} return 0;
}
```

Problem-09: Target Pair (Two Sum)

Input:
First line: T (number of test cases)
For each test case:
First line: n (array size)
Second line: n integers
Third line: target value

Output:
For each test case, print 1or 0

Algorithm:

For each test case:

- Read n, array a[ ], target
- Use a set seen
- For each x in array:
  - if (target - x) is in seen → print 1 and break
  - else → insert x into seen
- If no pair found → print 0

Code:

```cpp
#include <iostream>
using namespace std;

int main() {
int t;
    cout<<"Test Case: ";
    cin>>t;
    while(t--){
    int n;
    cout<<"Array size: ";
    cin >> n;

    int arr[n];
    cout<<"Array elements: ";
    for (int i = 0; i < n; i++) {
        cin >> arr[i];
    }

    int target;
    cout<<"Target: ";
    cin >> target;

    int found = 0;

    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            if (arr[i] + arr[j] == target) {
                found = 1;
            }
        }
    }

    cout << found<<endl;
    }

    return 0;
}
```

Set

- In C++, sets are associative container which stores unique elements in some sorted order.
- By default, it is sorted ascending order of the keys

Creating a set

```
#include <iostream>
#include <set>
using namespace std;

int main() {

    // Creating a set of integers
    set<int> s = {3, 5, 2, 1};

    for (auto x : s)
        cout << x << " ";
    return 0;
}
```

Output

1 2 3 5

Inserting elements

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    set<int> s = {1, 4, 2};

    // Insert elements into set
    s.insert(5);
    s.emplace(3);
    s.insert(5);

    for (auto x: s) cout << x << " ";
    return 0;
}
```

Output

1 2 3 4 5

Accessing elements

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    set<int> s = {1, 4, 2, 3, 5};

    // Accessing first element
    auto it1 = s.begin();

    // Accessing third element
    auto it2 = next(it1, 2);

    cout << *it1 << " " << *it2;
    return 0;
}
```

Output

1 3

Finding elements

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    set<int> s = {1, 4, 2, 3, 5};

    // Finding 3
    auto it = s.find(3);

    if (it != s.end()) cout << *it;
    else cout << "Element not Found!";
    return 0;
}
```

Output

3

Traversing elements

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    set<int> s = {5, 1, 4, 3, 2};

    // Traversing using traditional for loop
    for(auto it = s.begin(); it != s.end(); it++)
        cout << *it << " ";
    return 0;
}
```

Output

1 2 3 4 5

Deleting elements

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    set<int> s = {1, 4, 2, 3, 5};

    // Deleting elements by value
    s.erase(5);

    // Deleting first element by iterator
    s.erase(s.begin());

    for (auto x: s) cout << x << " ";
    return 0;
}
```

Output

2 3 4

#if an element in a given set is even, delete it

```cpp
#include <bits/stdc++.h>
using namespace std;

int main() {
    set<int> s = {1, 2, 3, 4, 5, 6, 7, 8};

    for (auto it = s.begin(); it != s.end(); ) {
        if (*it % 2 == 0)
            it = s.erase(it);   // erase returns next iterator
        else
            it++;
    }

    for (int x : s)
        cout << x << " ";

    return 0;
}
```

Updating elements

- We cannot change the value of elements once they are stored in the set.

10100011

$2^2 \geq 7+2+1 = 9$

$2^4 = 16 \geq 7+4+1 = 12$

$2^0 = P1$

$2^1 = P2$

$2^2 = P4$

$2^3 = P8$

$2^4 = P16$

Set-01

1

Item A, ratio $= \frac{30}{5} = 6$

Item B, ratio $= \frac{50}{10} = 5$

Item C, ratio $= \frac{60}{15} = 4$

Item D, ratio $= \frac{70}{7} = 10$

Item, E, ratio $= \frac{40}{8} = 5$

Ranking item based on ration (Descending order)

Item	D	A	B	E	C
Weight (kg)	7	5	10	8	15
Value($)	70	30	50	40	60
ratio	10	6	5	5	4

2 and (n)

: 20 kg capacity:

Greedy picks in ratio order

~~D full (7 kg, value)~~

Take D full: weight 7 kg, value 70$, remaining capacity 13 kg

Take A full:

weight 5 kg, value 30$, remaining capacity 8 kg

Take ~~B full~~, frac:

weight 8 kg, value 40$, remaining cap 0 kg

Total for 20 kg:

From D : 7 kg, 70$

From A : 5 kg, 30$

From E : 8 kg, 40$

Maximum value = 70 + 30 + 40 = 140

(4)

```
#include <bits/stdc++.h>

using namespace std;

struct Knapsack {
    char name;
    double weight, value, ratio;
};

int main() {
    vector<Knapsack> items = {
        {'A', 5, 30},
        {'B', 10, 50},
        {'C', 15, 60},
        {'D', 7, 70},
        {'E', 8, 40}
    }

double capacity = 20;

for (auto &itm : items) {
    itm.ratio = itm.value / itm.weight;
}
```

```
sort( items.begin(), items.end(), [](const Item
                                     Knapsack
      &a, const Item &b){
          return a.ration > b.ratio;
      });

double remaining = capacity.
double totalValue = 0.0;

cout << fixed << setprecision(2);
cout << "Capacity : " << capacity << " kg\n";
cout << "Take items (possibly fractional) : \n";

for(auto &itm : items){
    if(remaining <= 0)
        break;
    if( itm.weight <= remaining){
        cout << "Take full item " << itm.name
             << " (w=" << itm.weight << ", v=" << itm.
             value << ")\n";
        remaining -= itm.weight;
        totalValue += itm.value;
    }else{
        double frac = remaining / itm.weight;
```

```
        double w = remaining;
        double v = frac * itm.value;
        cout << "Take " << frac * 100 << "% of item "
             << itm.name;
        cout << " (w=" << w << ", v=" << v << ")\n";
        totalValue += v;
        remaining = 0;
    }
}
cout << "Maximum total value = " << totalValue << endl;
return 0;
}
```

Set - 02

(1) Activities : A(1,4), B(3,5), C(0,6), D(5,7)
E(8,9); F(5,9), G(6,10)

Greedy Approach:

→ Pick (A1,4) first (earliest finish)
→ Next that starts ≥ 4. But B(3,5) starts 3<4, so skip B.

Sort by finish time:

Activity	Start Time	Finish time
A	1	4
B		

→ ~~Pick~~ D(5,7) starts $5 \geq 4$ → Pick D.

→ E(8,9) starts $8 \geq 7$ → pick E

→ F(5,9) starts $5 \not\geq 9$ → skip F

→ G(6,10) starts $6 \not\geq 9$ → skip G

∴ Selected Activities : A, D, E (3 activities)

Activity - A: first (earliest finish 4) → Pick A

Activity - B: starts $3 \not\geq$ finish 4 → skip B

Activity - C: starts $0 \not\geq$ finish 4 → skip C

Activity - D: starts $5 \geq 4$ finish 4 → pick D
updated finish 7

Activity - E: starts $8 \geq$ finish 7 → Pick E
updated finish 9

Activity - F: starts $5 \not\geq$ finish 9 → skip F

Activity - G: starts $6 \not\geq$ finish 9 → skip G

picked Activity A, D, E

(2) same as ans-1

(3)
```
#include <bits/stdc++.h>
using namespace std;

struct Activity{
    char name;
    int start, finish;
};
int main(){
  vector<Activity> acts = {
        {'A', 1, 4},
        {'B', 3, 5},
        {'C', 0, 6},
        {'D', 5, 7},
        {'E', 8, 9},
        {'F', 5, 9},
        {'G', 6, 10}
  };
```

```
sort(acts.begin(), acts.end(), [](const Activity &a,
    const Activity &b) {
        return a.finish < b.finish;
    });

cout << "Selected Activities : \n";
int lastfinish = -1;
for(auto &a : acts) {
    if (a.start >= lastfinish) {
        cout << a.name << " (" << a.start << ","
        << a.finish << ")\n";
        lastfinish = a.finish;
    }
}
return 0;
}
```

Set-03

1.

(2)

```
#include <bits/stdc++.h>
using namespace std;

int lcsLength(const string &a, const string &b){
    int m = a.size(), n = b.size();

    vector<vector<int>> dp(n+1, vector<int>(m+1,0));

    for(int i = 1; i <= m; i++){
        for(int j=1; j<=n; j++){
            if(a[i-1] == b[j-1]){
                dp[i][j] = dp[i-1][j-1] + 1;
            }
            else{
                dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
            }
        }
    }
    return dp[m][n]
}

int main(){

}
```

(3)

```
#include <bits/stdc++.h>
using namespace std;

string lcsString(const string &a, const string &b) {
    int m = a.size(), n = b.size();
    vector<vector<int>> dp(n+1, vector<int>
                           (m+1, 0));

    for(int i=1; i<=m; i++) {
        for(int j=1; j<=n; j++) {
            if(dp[i][j] == dp
            if(dp[i-1][j-1] ==
            if( a[i-1] == b[j-1]) {
                dp[i][j] = dp[i-1][j-1]+1;
            }
            else{
                dp[i][j] = max(dp[i-1][j], dp[i][j-1]
            }
        }
    }

    string lcs = "";
    int i=m, j=n;
    while( i>0 && j>0){
```

```
if (a[i-1] == b[j-1]) {
    lcs.push_back(a[i-1]);
    i--, j--;
}
else if (dp[i-1][j] >= dp[i][j-1]) {
    i--;
}
else {
    j--;
}
reverse(lcs.begin(), lcs.end());
return lcs;
}
```

Set - 04

Algorithm:

(1)

1. Input : list of n activities with start[i], fini
2. Sort activities by ascending order of finis time.
3. Select first activity then repeatedly select next activity whose start time ≥ finish tim of last selected.
4. Output : set of selected activities and coun

(2)

```
struct Activity {
    int id;
    int start, finish;
};
int main() {
    int n;
    cout << "Enter number of activities: ";
    cin >> n;
    vector<Activity> acts(n);
    cout << "Enter start and finish times for each
        activity: \n";
```

```
for(int i=0; i<n; i++){
    cout<<"Activity "<<i+1<<" (start, finish): ";
    cin >> acts[i].start >> acts[i].finish;
    acts[i].id = i+1;
}
sort(acts.begin(), acts.end(), [](const Activity
    & &a, const Activity &b){
        return a.finish > b.finish;
    });
vector<Activity> selected;
int lastFinish = -1;

for(auto &a : acts){
    if(a.start >= lastfinish){
        selected.push_back(a);
        lastFinish = a.finish;
    }
}
cout << "\nMaximum number of non- - - "
    <<selected.size() << endl;
cout << "selected activities (id, start, finish)
    - - - ";
```

```
for(auto &a : acts selected){
    cout << "Activity " << a.id << ": "
    << a.start << " " << a.finish << endl;
}
return 0;
}
```