Mohsin Iban Hossain

AIUB, AI MID Term Notes

ARTIFICIAL INTELLIGENCE AND EXPERT SYSTEM

Table of Contents

INTRODUCTION TO ARTIFICIAL INTELLIGENCE (AI)

What is Artificial Intelligence?

⇨ Artificial Intelligence (AI) is the field of computer science that focuses on creating systems capable of performing tasks that typically require human intelligence. These include learning, reasoning, problem-solving, understanding language, perception, and decision-making.

Approaches to AI

Approach	Description
Acting Humanly	Machines that behave like humans (Turing Test approach).
Thinking Humanly	Machines that try to replicate human thought processes (Cognitive modeling).
Thinking Rationally	Machines that think logically and make rational decisions.
Acting Rationally	Machines that act to achieve the best outcome (Rational agent approach).

Turing Test

Proposed by: Alan Turing in 1950
Purpose: To determine if a machine can exhibit intelligent behavior equivalent to, or indistinguishable from, that of a human.

Test Setup:

- An interrogator communicates with a human and a machine via a text interface.
- If the interrogator can't reliably tell which is which, the machine is said to have passed the Turing Test.

Limitations:

- Focuses only on behavior, not internal reasoning.
- May allow non-intelligent tricks to fool the interrogator.

Foundations of AI

Discipline	Contribution to AI
Philosophy	Logic, reasoning, ethics, mind-body problem.
Mathematics	Algorithms, probability, optimization, formal models.
Psychology	Cognitive processes, learning models, decision-making.
Computer Science	Programming, complexity, data structures, software engineering.
Linguistics	Natural language understanding and generation.
Biology	Neural networks, genetic algorithms, behavior modeling.

Applications of AI

- **Expert Systems:** Decision support in medicine, engineering.
- **Natural Language Processing:** Chatbots, voice assistants.
- **Robotics:** Autonomous vehicles, drones.
- **Computer Vision:** Face recognition, object detection.
- **Machine Learning:** Spam filtering, fraud detection.
- **Games:** AI opponents in chess, Go, etc.

INTELLIGENT AGENTS & PROBLEM SOLVING

What is an Agent?

⇨ An **agent** is anything that perceives its environment through **sensors** and acts upon that environment through **actuators**.

Agent = **Perception → Action**

Agent and Environment

- **Environment:** The world or context in which the agent operates.
- **Sensors:** Devices that collect data from the environment (e.g., camera, temperature sensor).
- **Actuators:** Components that perform actions in the environment (e.g., motors, speakers).

Rational Agent

⇨ A **rational agent** is one that **does the right thing** — it chooses the action that maximizes performance, based on:

- Its **percept sequence**
- **Knowledge** of the environment
- **Possible actions**
- **Performance measure**

PEAS Description

⇨ To define a task environment, use the **PEAS** framework:

Component	Description
Performance	Criteria for success
Environment	What the agent senses
Actuators	What the agent uses to act
Sensors	What the agent uses to perceive the world

Example: Self-driving Car

Component	Description
Performance	Safety, speed, law compliance
Environment	Roads, other cars, pedestrians
Actuators	Steering, throttle, brake
Sensors	Camera, GPS, radar

Problem1: Describe the PEAS framework with an example of a Candy Jar Management Agent.

⇨ **PEAS** stands for **Performance measure, Environment, Actuators, and Sensors**. It is a framework used to define the task environment of an intelligent agent.

Let's apply the PEAS framework to a **Candy Jar Management Agent** – a system that automatically monitors and refills candies in a jar.

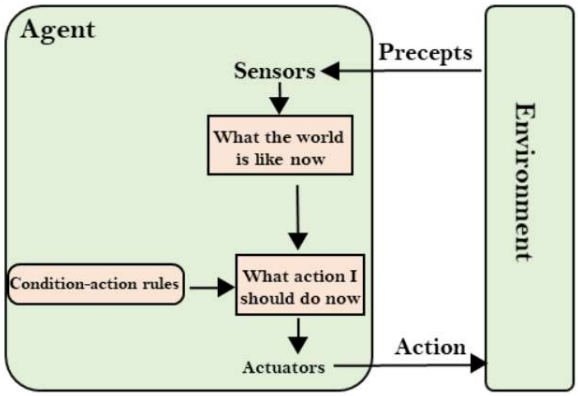
Example: Candy Jar Management Agent

Component	Description
Performance	Accuracy, Speed, Safety, Efficiency
Environment	Conveyor, Sorting Table, Jars, Lighting
Actuators	Robotic Arm, Conveyor Motor, Jar Mechanism
Sensors	Camera, Weight Sensor, Position Sensor, Proximity Sensor

Types of Agents

Type	Description
Simple Reflex Agent	Acts only on current percept; ignores history.
Model-Based Reflex Agent	Maintains internal state for decision-making.
Goal-Based Agent	Considers future consequences to achieve goals.
Utility-Based Agent	Uses a utility function to maximize overall happiness or benefit.
Learning Agent	Improves performance over time based on past experiences.

Simple Reflex Agent

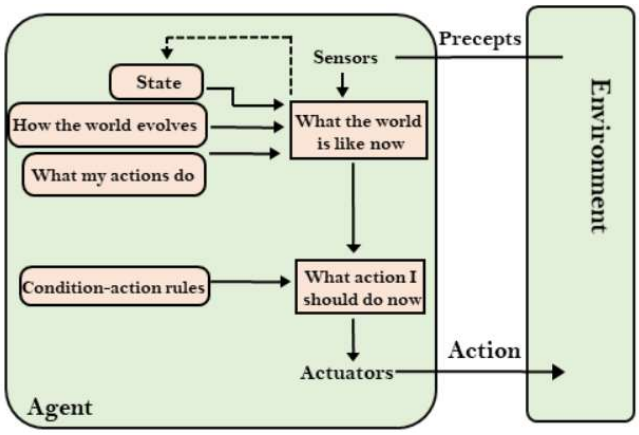


⇨ A simple reflex agent comprises the following parts:

- **Agent:** The agent is the one who performs actions on the environment.
- **Environment:** The environment includes the surroundings of the agent.
- **Actuators:** Actuators are devices that convert energy into motion.
- **Sensors:** Sensors are the things that sense the environment. They are devices that measure physical property.

```
function REFLEX-VACUUM-AGENT( [location,status]) returns an action
    if status = Dirty then return Suck
    else if location = A then return Right
    else if location = B then return Left
```

Model Based Agent



⇨ The model-based reflex agent operates in four stages:

- **Sense:** It perceives the current state of the world with its sensors.
- **Model:** It constructs an internal model of the world from what it sees.
- **Act:** The agent carries out the action that it has chosen.
- **Reason:** It uses its model of the world to decide how to act based on a set of predefined rules or heuristics.

Example: A vacuum cleaner that uses sensors to detect dirt and obstacles and moves and cleans based on a model.

```
function MODEL-BASED-REFLEX-AGENT(percept) returns an action
  persistent: state, the agent's current conception of the world state
              model, a description of how the next state depends on current state and action
              rules, a set of condition–action rules
              action, the most recent action, initially none

  state ← UPDATE-STATE(state, action, percept, model)
  rule ← RULE-MATCH(state, rules)
  action ← rule.ACTION
  return action
```

Figure 2.12 A model-based reflex agent. It keeps track of the current state of the world, using an internal model. It then chooses an action in the same way as the reflex agent.

Goal Based Agent

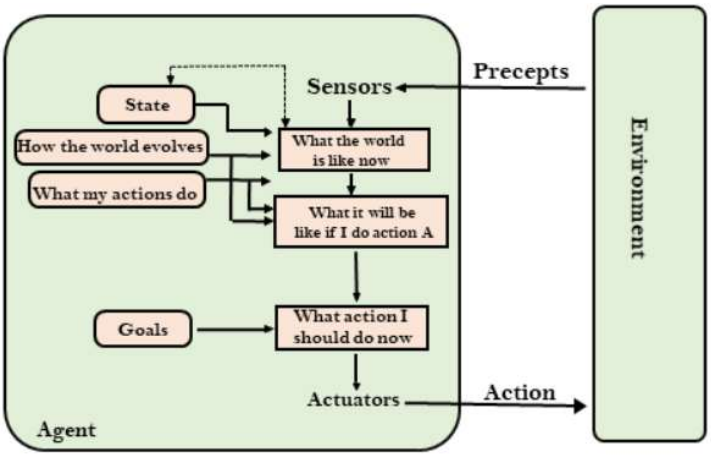


Figure 2.13 A model-based, goal-based agent. It keeps track of the world state as well as a set of goals it is trying to achieve, and chooses an action that will (eventually) lead to the achievement of its goals.

Utility Based Agent

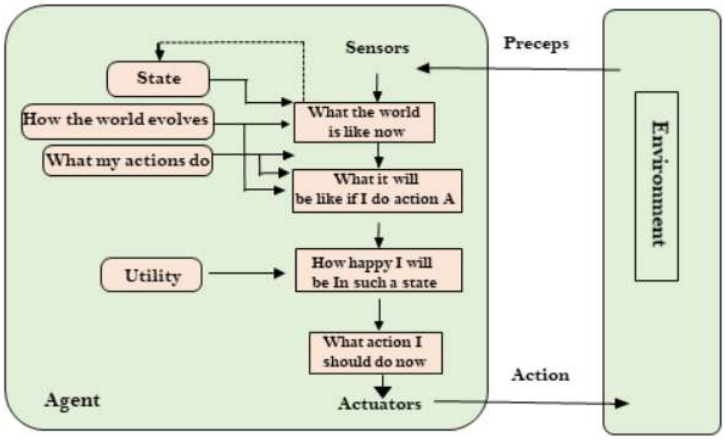
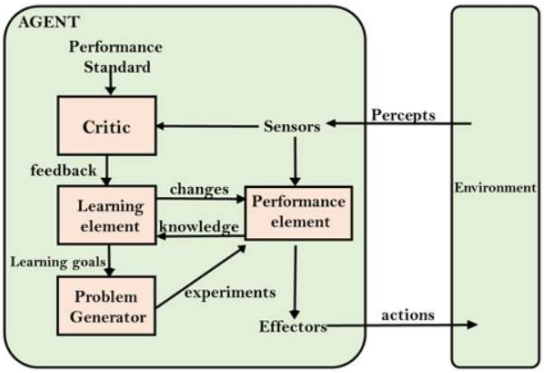


Figure 2.14 A model-based, utility-based agent. It uses a model of the world, along with a utility function that measures its preferences among states of the world. Then it chooses the action that leads to the best expected utility, where expected utility is computed by averaging over all possible outcome states, weighted by the probability of the outcome.

Learning Based Agent



HOW COMPONENT OF AGENT PROGRAMS WORK

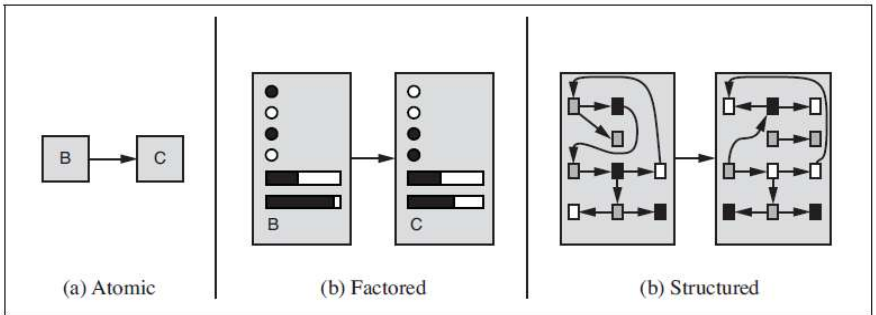


Figure 2.16 Three ways to represent states and the transitions between them. (a) Atomic representation: a state (such as B or C) is a black box with no internal structure; (b) Factored representation: a state consists of a vector of attribute values; values can be Boolean, real-valued, or one of a fixed set of symbols. (c) Structured representation: a state includes objects, each of which may have attributes of its own as well as relationships to other objects.

Problem-Solving Agent

⇨ A **problem-solving agent** decides what to do by searching through **possible sequences of actions** that lead to the goal.

Steps in Problem Formulation:

1. **Initial state**
2. **Actions**
3. **Transition model**
4. **Goal test**
5. **Path cost**

Together these define a **search problem**.

Example: Vacuum Cleaner World

- **States:** Location and status (clean/dirty)
- **Actions:** Move Left, Move Right, Suck
- **Goal:** Clean all locations
- **Performance:** Number of clean squares, energy used

UNINFORMED & INFORMED SEARCH TECHNIQUES

Problem Solving by Searching

⇨ A **search algorithm** explores a **state space** to find a **path from the initial state to the goal state**.

State Space = All possible configurations
Search Tree = Path from initial to goal through actions
Search Node = Contains a state, path cost, and parent

Uninformed Search (Blind Search)

⇨ Uninformed search strategies **do not use any domain-specific knowledge**. They rely purely on the structure of the problem.

Algorithm	Description
Breadth-First Search (BFS)	Explores nodes level by level. Guaranteed to find the shortest path in an unweighted graph.
Depth-First Search (DFS)	Explores as deep as possible before backtracking. Can go into infinite loops without limit.
Uniform Cost Search (UCS)	Expands the node with the lowest path cost (like Dijkstra's). Optimal and complete.
Depth-Limited Search (DLS)	DFS with a depth limit. Avoids infinite loops.
Iterative Deepening Search (IDS)	Repeated DLS with increasing depth. Combines space efficiency of DFS and completeness of BFS.
Bidirectional Search	Simultaneously searches forward from the start and backward from the goal. Very fast when applicable.

Informed Search (Heuristic Search)

⇨ Informed search algorithms use **heuristic functions (h(n))** to guide the search. These functions estimate the cost from node **n** to the goal.

◇ Best-First Search

- Chooses the node with the **lowest heuristic value (h(n))**
- Not guaranteed to be optimal

◇ A* Search

- Uses **f(n) = g(n) + h(n)**
  where:
  - g(n): Cost to reach node n
  - h(n): Estimated cost from n to goal
- **Complete and Optimal**, if h(n) is **admissible** (never overestimates) and **consistent** (monotonic).

Example Heuristics

- **Manhattan Distance:** For grid-based problems.
- **Euclidean Distance:** For map or spatial problems.
- **Number of misplaced tiles:** For puzzle-solving.

Comparison of Search Strategies

Algorithm	Time	Space	Optimal?	Complete?
BFS	Exponential	Exponential	Yes	Yes
DFS	$O(b^m)$	O(m)	No	No
UCS	$O(b^d)$	$O(b^d)$	Yes	Yes
DLS	$O(b^l)$	O(l)	No	Yes (if $l \geq d$)
IDS	$O(b^d)$	O(d)	Yes	Yes
A*	Depends on h(n)	Depends	Yes (if h is admissible)	Yes

b = branching factor
d = depth of the shallowest solution
m = maximum depth
l = limit

BREADTH-FIRST SEARCH (BFS)

Breadth-First Search (BFS)

⇨ **Breadth-First Search (BFS)** is an uninformed search algorithm that explores the **shallowest** (closest to the root) **nodes first**, level by level.

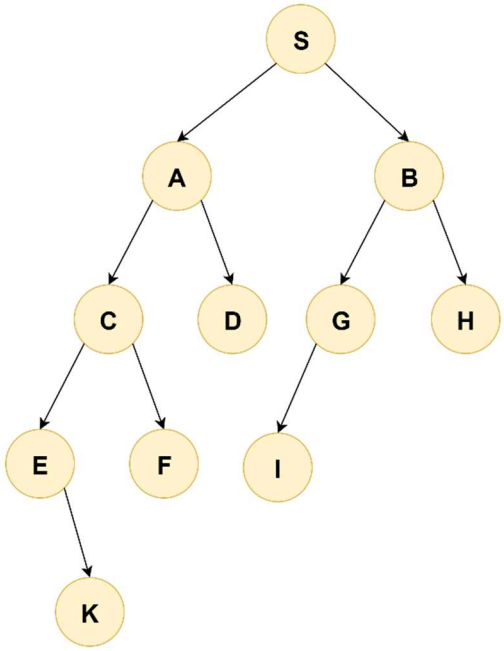
Note: It uses a **queue (FIFO)** data structure to keep track of the next nodes to visit.

Algorithm Steps

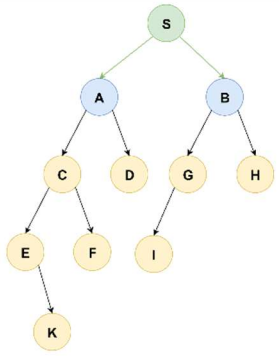
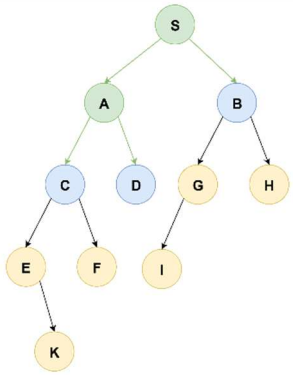
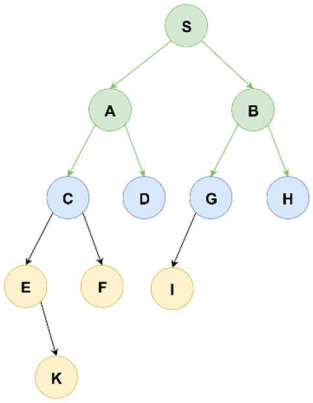
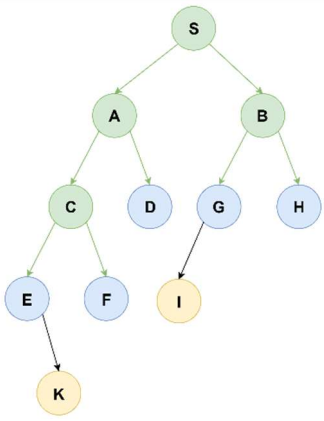
1. **Initialize** the queue with the **initial state**.
2. **Repeat** until the queue is empty:
   - Remove the **front node**.
   - If it is the **goal**, return the solution.
   - Otherwise, expand the node (generate its children).
   - **Add the children to the end of the queue**.

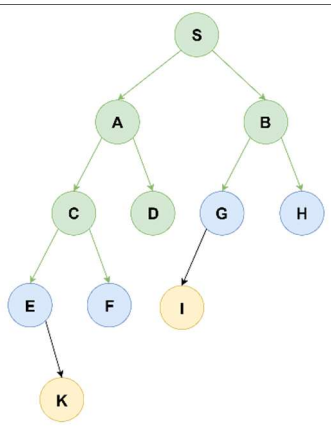
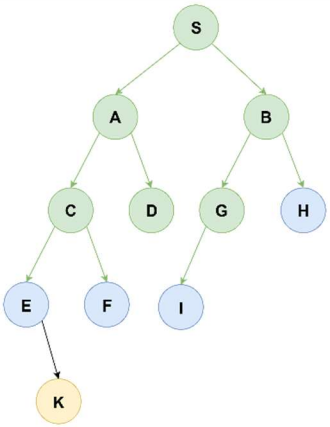
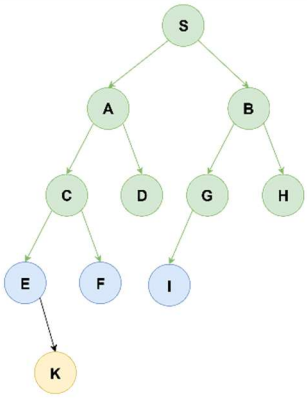
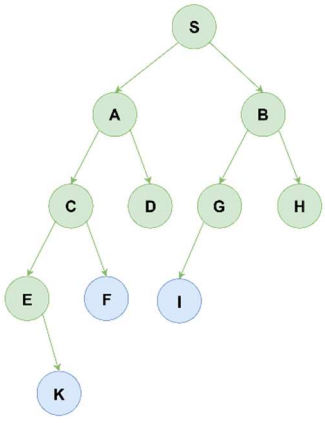
Problem1:

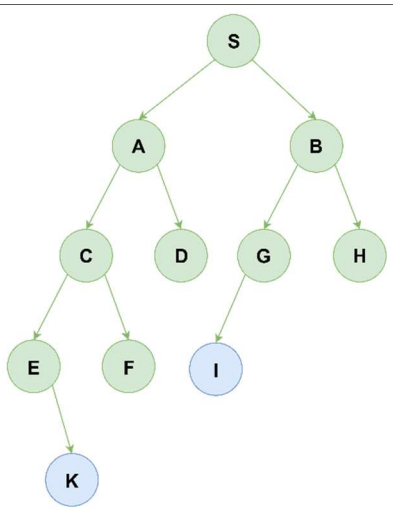
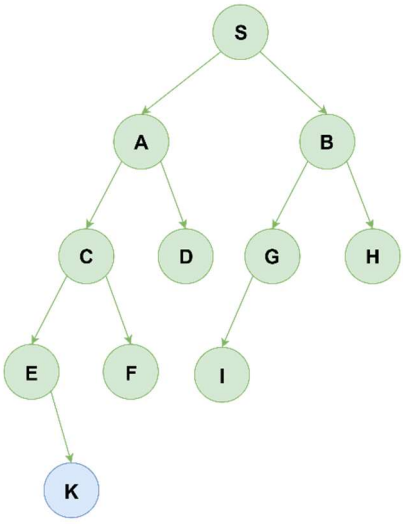
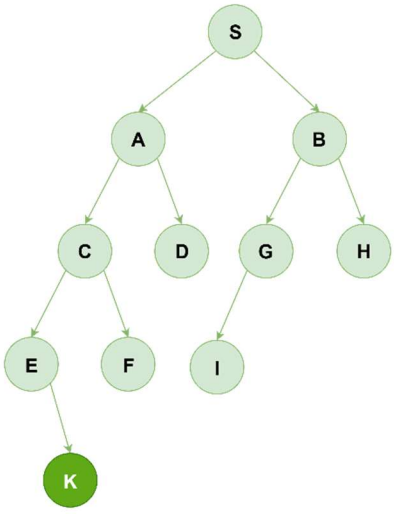
≫From the following graph to solve the problem where **S is the start node** and **K is the goal node**. Note that the **edges in this graph are directed**. For this problem Find the Goal node using Breadth-First Search (BFS).



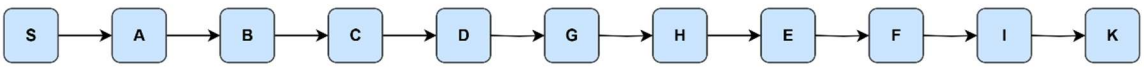
Solutions:

Vertex Visited	Vertex in Queue	Graph
S	A B	
S A	B C D	
S A B	C D G H	
S A B C	D G H E F	

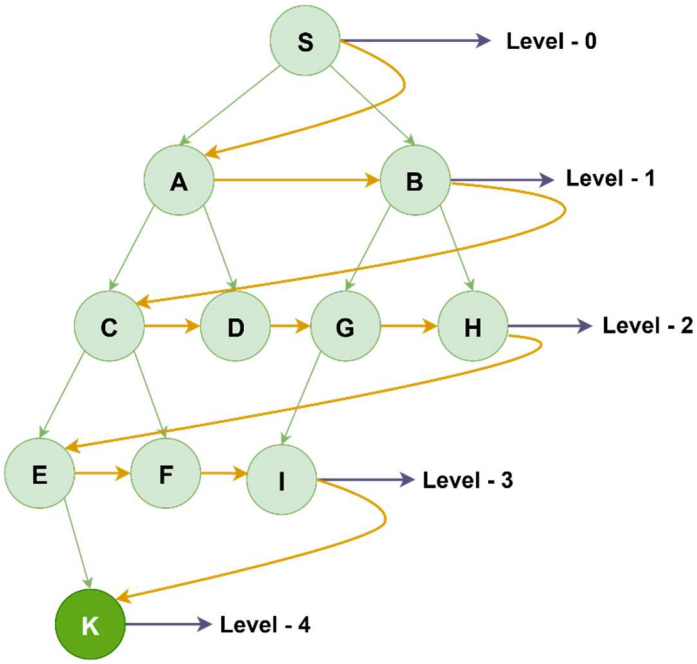
S A B C D	G H E F	
S A B C D G	H E F I	
S A B C D G H	E F I	
S A B C D G H E	F I K	

S A B C D G H E F	I K	
S A B C D G H E F I	K	
S A B C D G H E F I K		

⇨ **Now the path is:**



How its work

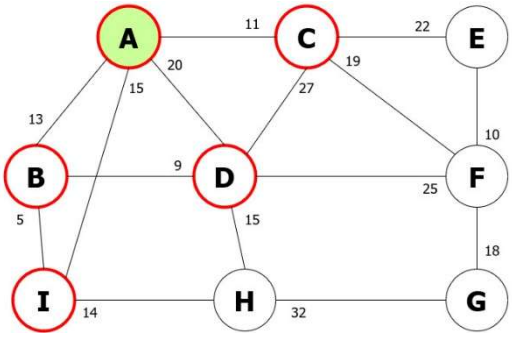
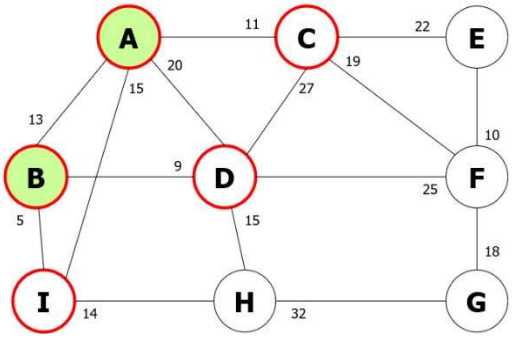
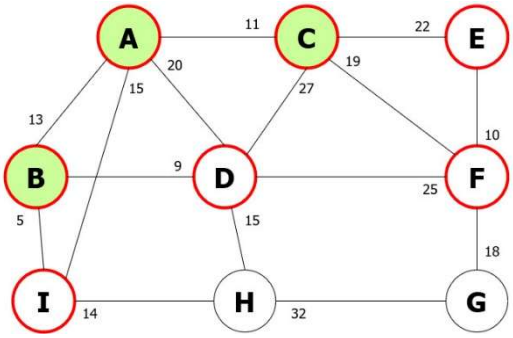


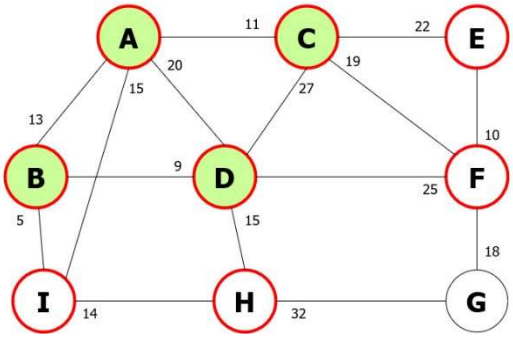
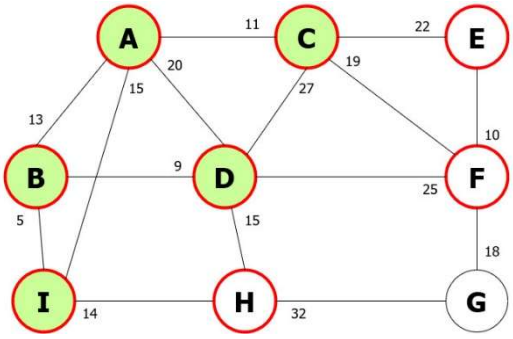
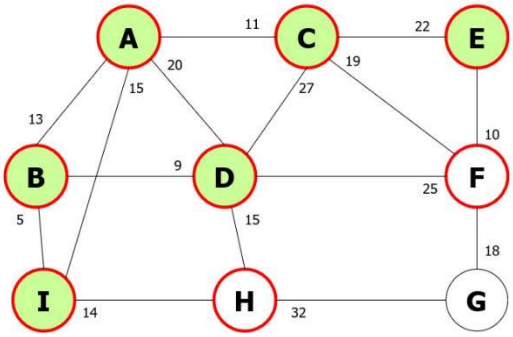
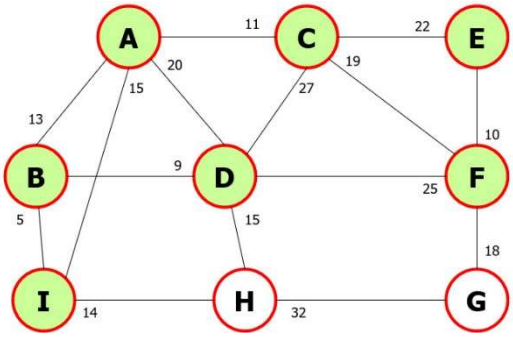
Properties

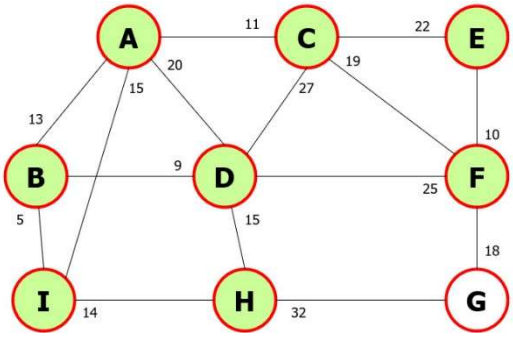
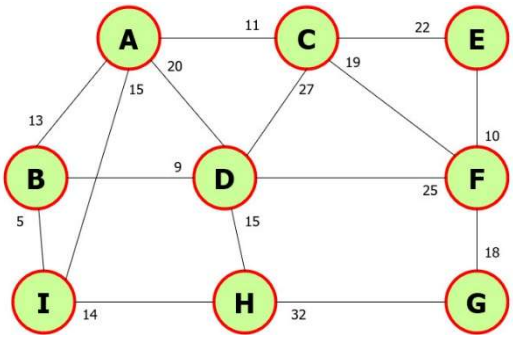
Property	Description
Completeness	Yes, if branching factor is finite.
Optimality	Yes, if all step costs are equal.
Time Complexity	$O(b^d)$, where b = branching factor, d = depth of solution.
Space Complexity	$O(b^d)$ – because it stores all nodes in memory.

Problem2:

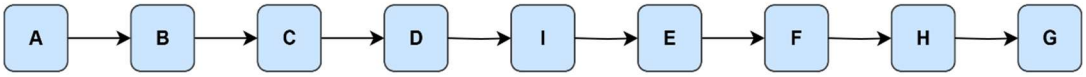
≫ Simulate Breadth-First Search (BFS) on the graph shown in below, **starting from the node A** and **aiming to reach the goal node G**. (push nodes into the frontier in alphabetic order)

Vertex Visited	Vertex in Queue	Graph
A	B C D I	
A B	C D I	
A B C	D I E F	

A B C D	I E F H	
A B C D I	E F H	
A B C D I E	F H	
A B C D I E F	H G	

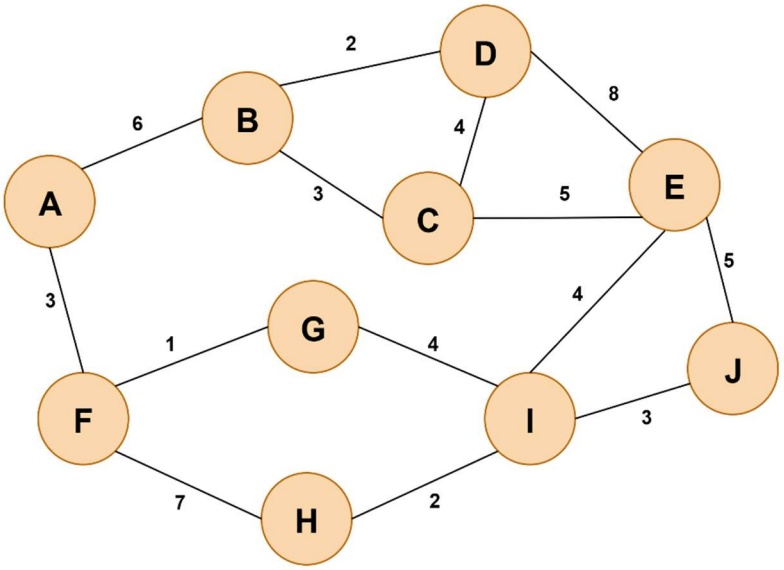
A B C D I E F H	G	
A B C D I E F H G		

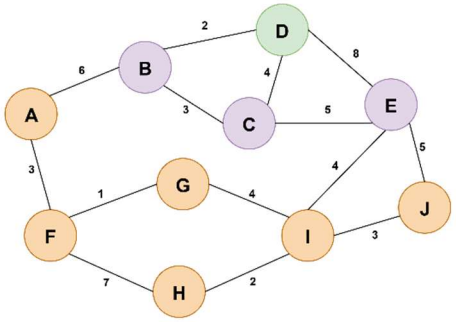
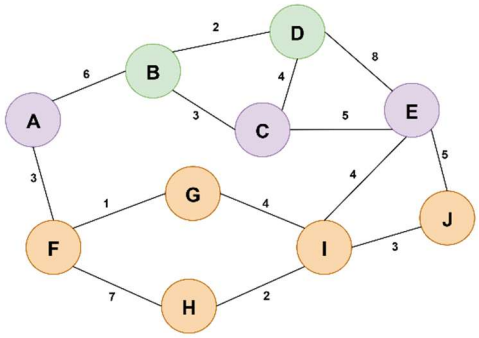
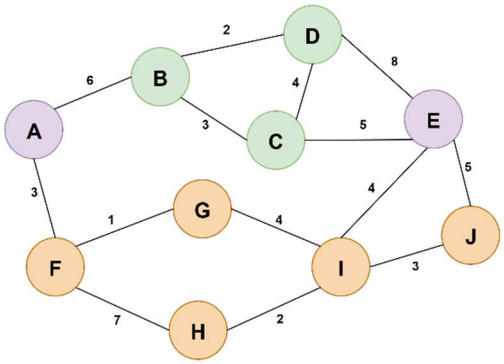
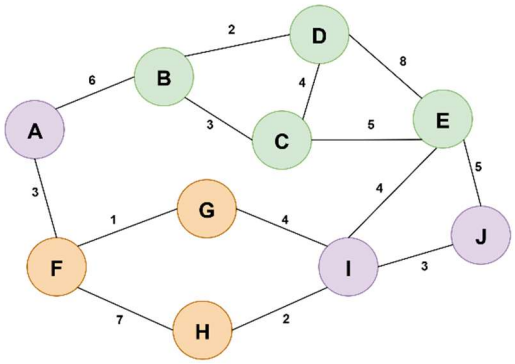
⇨ **Now the path is:**

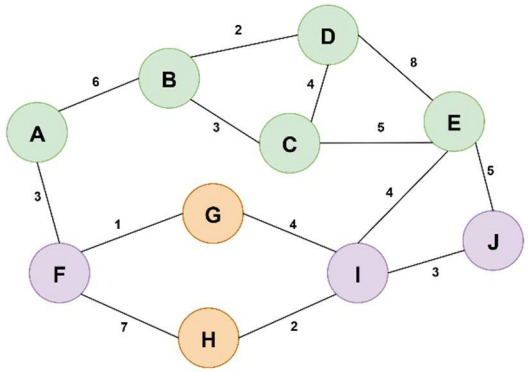
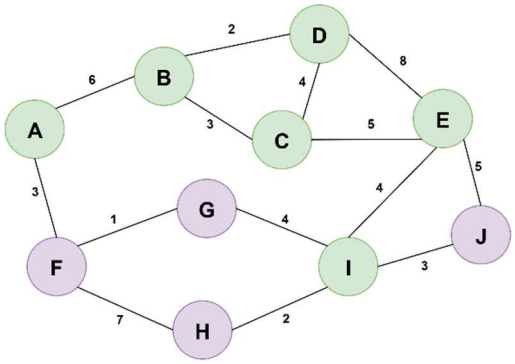
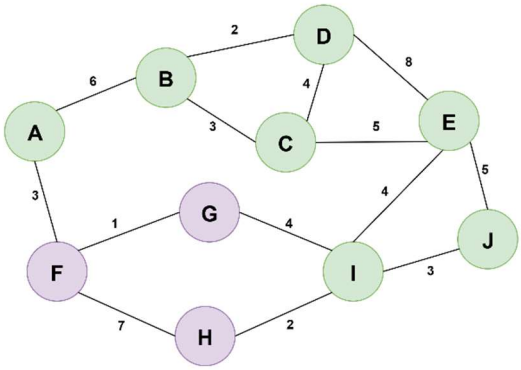
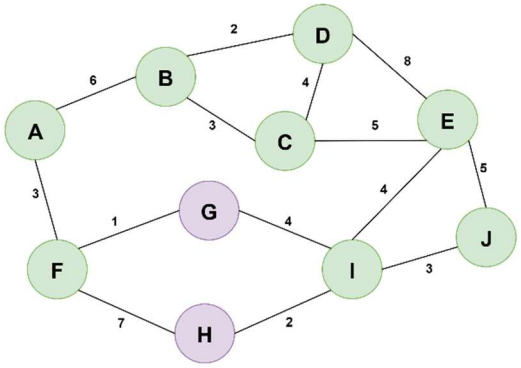


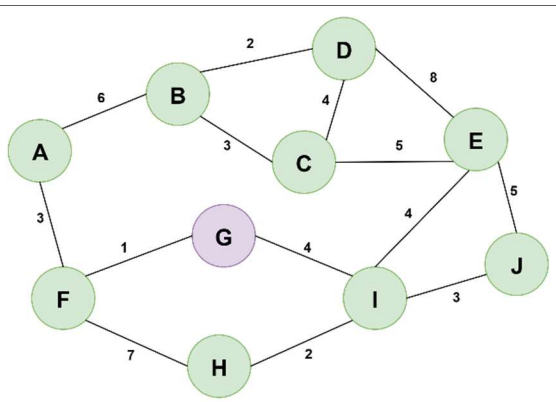
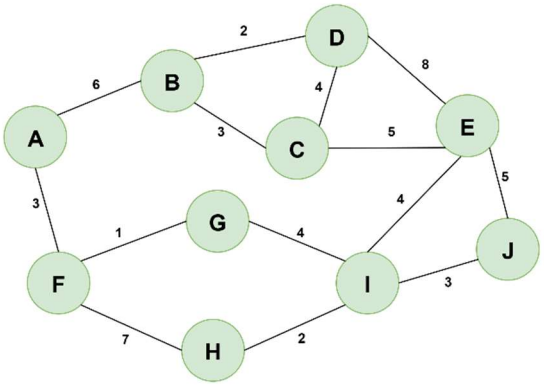
☌ Problem3:

≫From the following graph to solve the problem where **destination node G Stating from node D**. Note that the **edges in this graph are undirected**. For this problem Find the Goal node using Breadth-First Search (BFS).

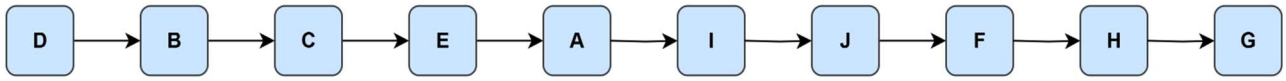


Vertex Visited	Vertex in Queue	Graph
D	B C E	
D B	C E A	
D B C	E A	
D B C E	A I J	

D B C E A	I J F	
D B C E A I	J F H G	
D B C E A I J	F H G	
D B C E A I J F	H G	

D B C E A I J F H	G	
D B C E A I J F H G		

⇨ **Now the path is:**



Properties

Property	Description
Completeness	Yes, if branching factor is finite.
Optimality	Yes, if all step costs are equal.
Time Complexity	$O(b^d)$, where b = branching factor, d = depth of solution.
Space Complexity	$O(b^d)$ – because it stores all nodes in memory.

DEPTH-FIRST SEARCH (DFS)

Depth-First Search (DFS)

⇨ **Depth-First Search (DFS)** explores **as deep as possible** along a branch before **backtracking**.

Note: It uses a **stack (LIFO)** data structure or recursion.

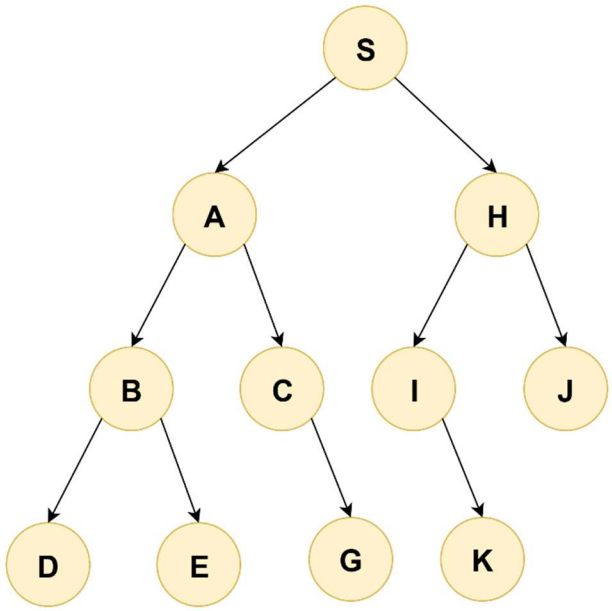
Algorithm Steps

1. Start from the **initial node**.
2. Push the node onto a **stack**.
3. While the stack is not empty:

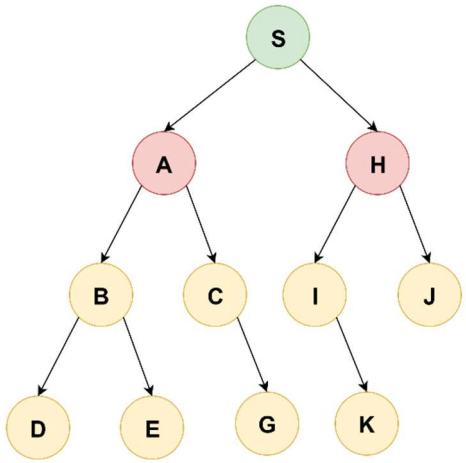
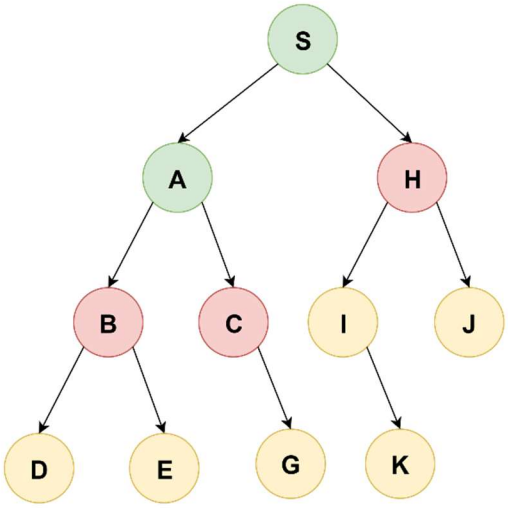
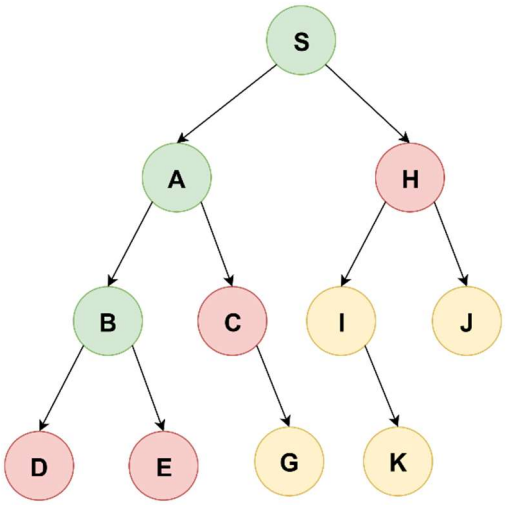
    - Pop the top node.
    - If it's the goal, return success.
    - Else, expand it and **push all its unvisited children** onto the stack.

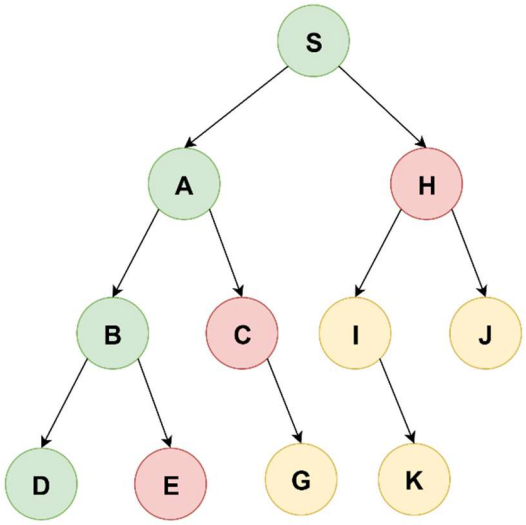
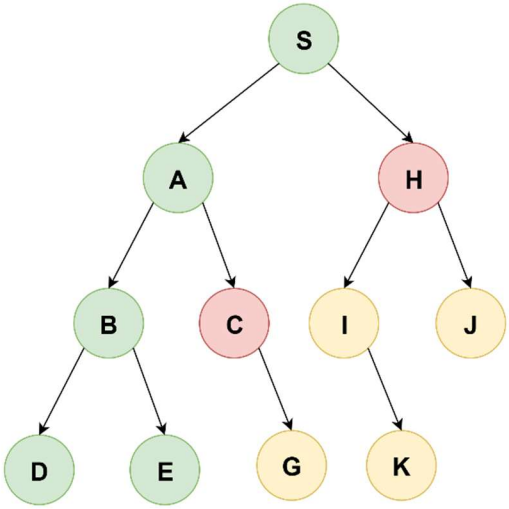
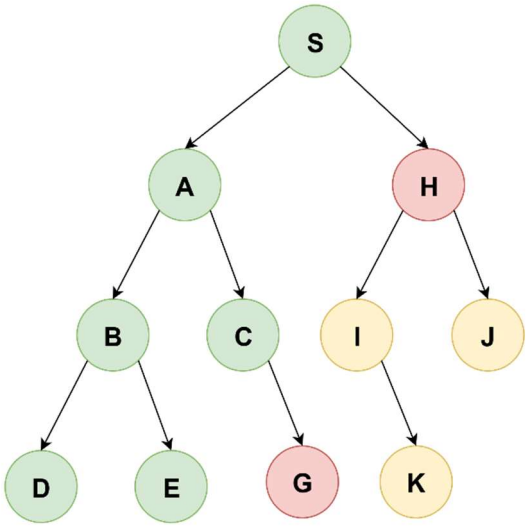
Problem1:

≫From the following graph to solve the problem where **S is the start node** and **G is the goal node**. Note that the **edges in this graph are directed**. For this problem Find the Goal node using Depth-First Search (DFS).



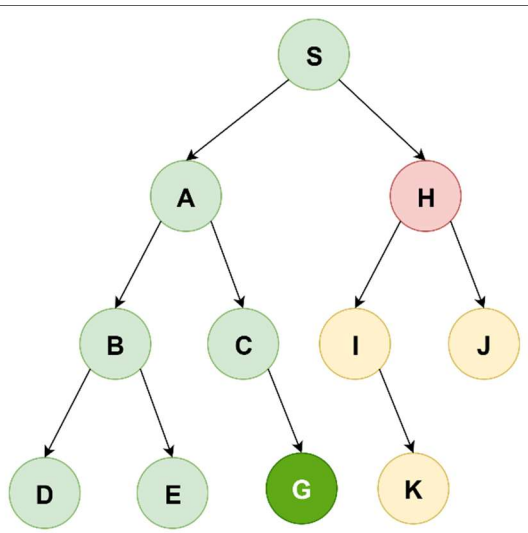
⇨ Solutions:

Vertex Visited	Vertex in Stack	Graph
S	A H	
S A	B C H	
S A B	D E C H	

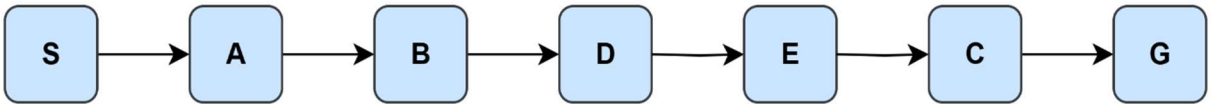
S A B D	E C H	
S A B D E	C H	
S A B D E C	G H	

S	A	B	D	E	C	G

H



⇨ **Now the path is:**

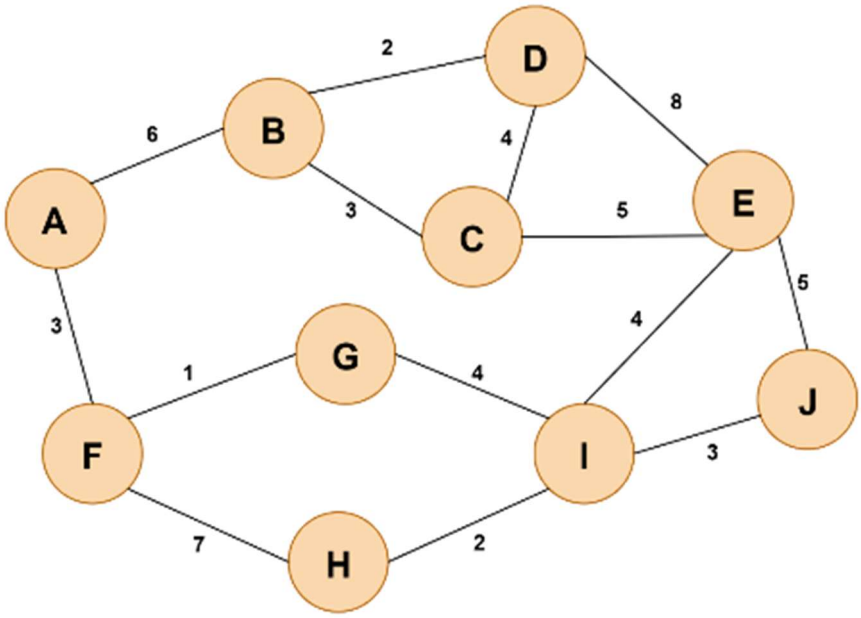


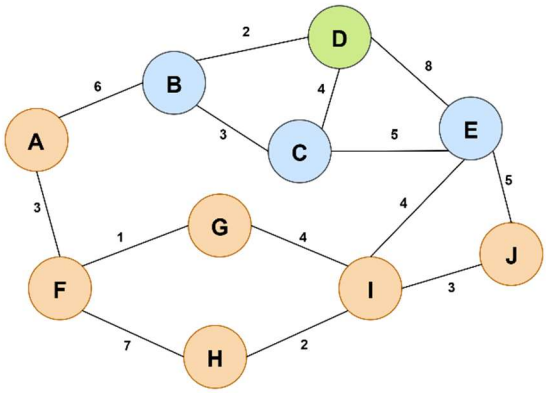
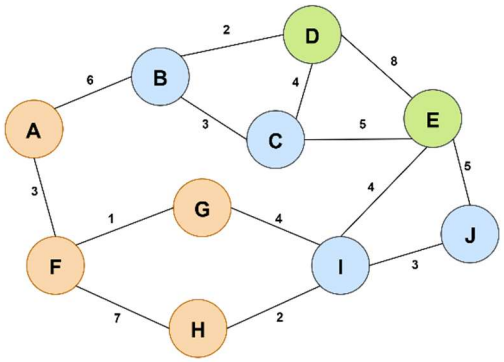
Properties of DFS

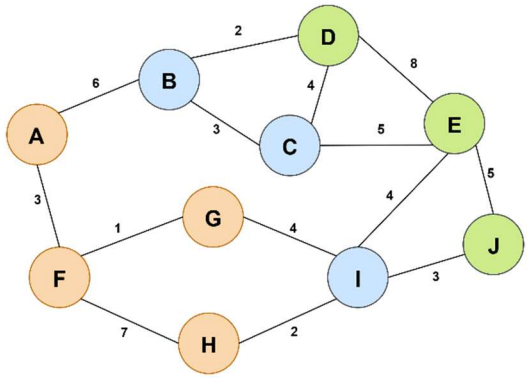
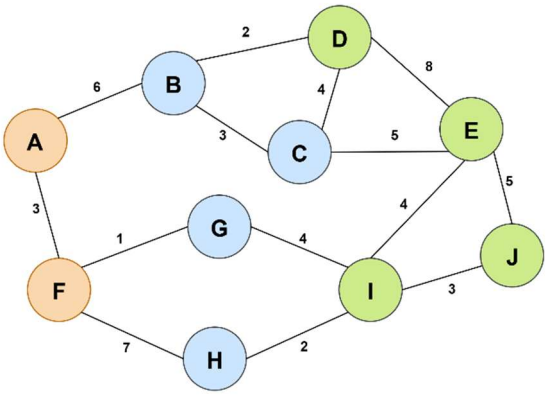
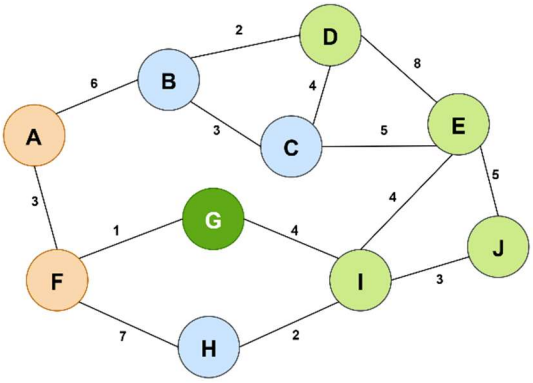
Property	Value
Complete	No (fails in infinite-depth spaces)
Optimal	No (may return suboptimal path)
Time	$O(b^m)$, where b = branching factor, m = max depth
Space	O(m) for recursive DFS; $O(b^m)$ for iterative
Best for	Problems with **deep solutions** or **low memory**

Problem2:

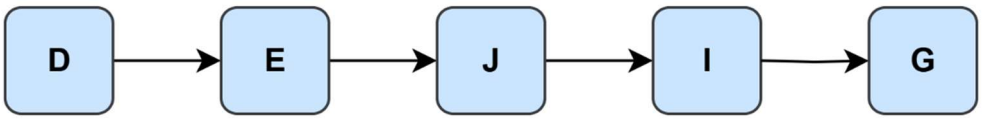
≫From the following graph to solve the problem where **D is the start node** and **G is the goal node**. Note that the **edges in this graph are undirected**. For this problem Find the Goal node using Depth-First Search (DFS).



Vertex Visited	Vertex in Stack	Graph
D	E C B	
D E	J I C B	

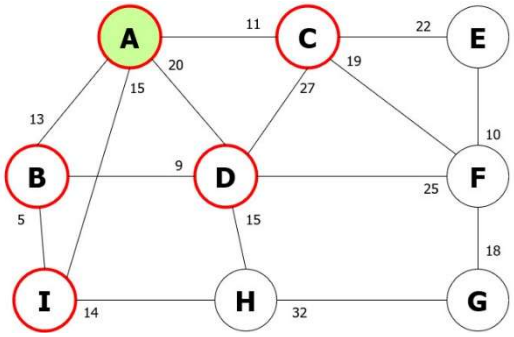
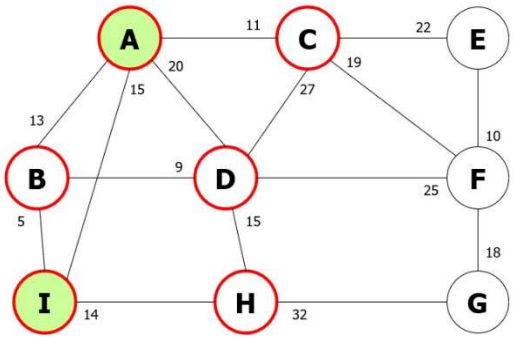
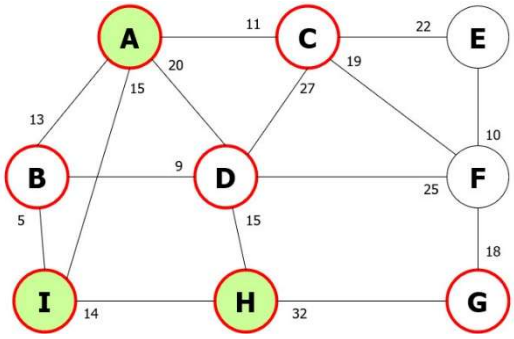
D E J	I C B	
D E J I	G H C B	
D E J I G	F H C B	

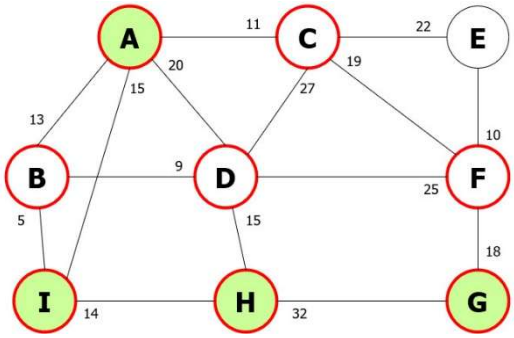
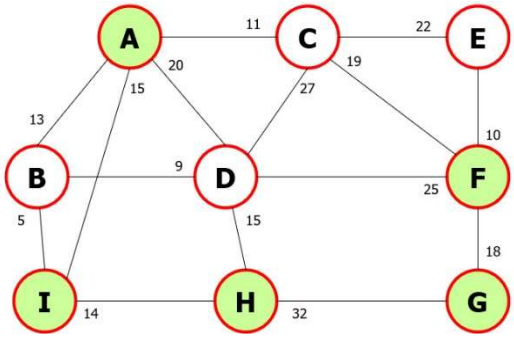
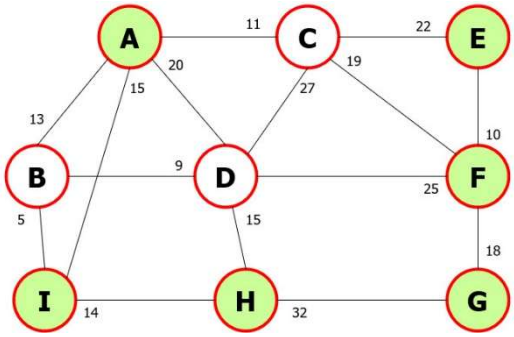
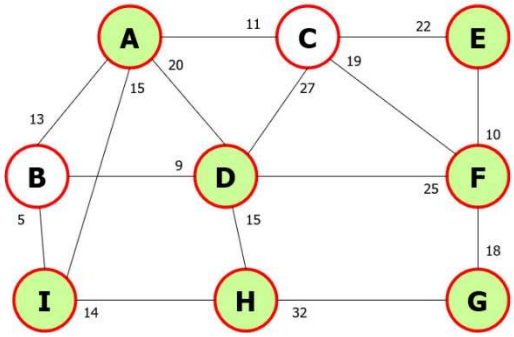
⇨ **Now the path is:**

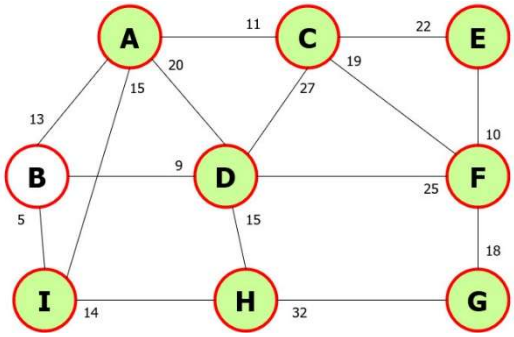
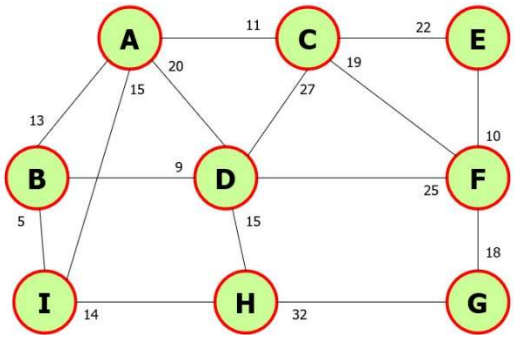


Problem3:

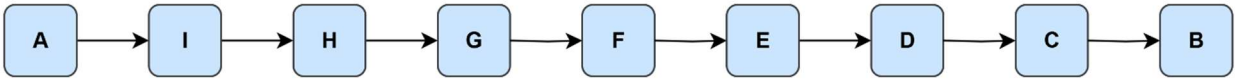
≫Simulate Depth-First Search (DFS) on the graph shown in below, **starting from the node A** and **aiming to reach the goal node B**. (push nodes into the frontier in alphabetic order)

Vertex Visited	Vertex in Stack	Graph
A	I D C B	
A I	H D C B	
A I H	G D C B	

A I H G	F D C B	
A I H G F	E D C B	
A I H G F E	D C B	
A I H G F E D	C B	

A I H G F E D C	B	
A I H G F E D C B		

⇨ **Now the path is:**



UNIFORM COST SEARCH (UCS)

Uniform Cost Search (UCS)

⇨ **Uniform Cost Search (UCS)** is an **uninformed** search algorithm that expands the node with the **lowest total path cost** so far.

Note: UCS is a generalization of BFS that works on **weighted graphs** (with **different edge costs**).

⇨ **Key Concepts:**

- It uses a **priority queue (min-heap)** to always expand the **cheapest path**.
- Nodes are prioritized by `g(n)`, the **cost from the start node to node n**.
- UCS ignores the number of steps or depth and focuses only on cost.

Algorithm Steps

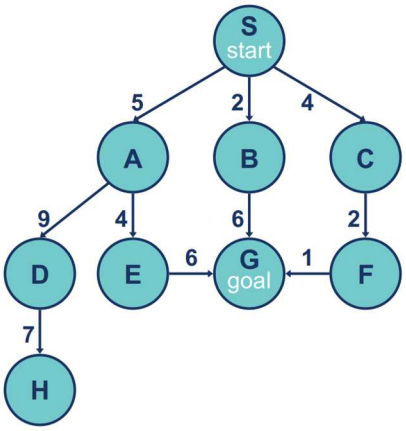
1. Initialize a **priority queue** with the initial node (cost = 0).
2. Loop until the queue is empty:
   - Remove the node with the **lowest path cost**.
   - If it's the **goal**, return the path.
   - Else, **expand** the node:
     - For each child, calculate path cost `g(child)`
     - If child is unvisited or has a lower cost now, **update the queue**.

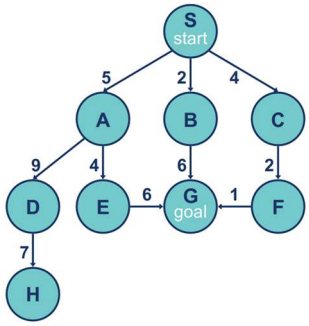
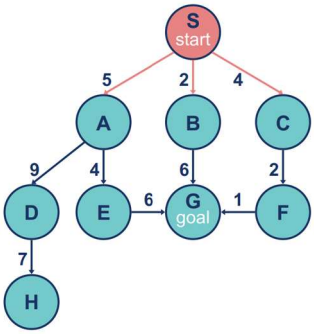
UCS vs BFS

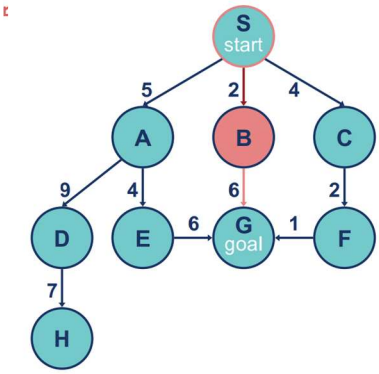
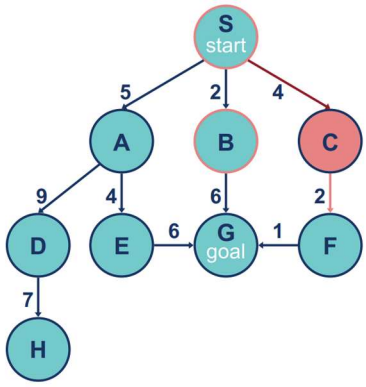
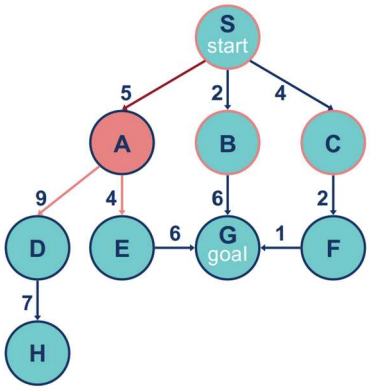
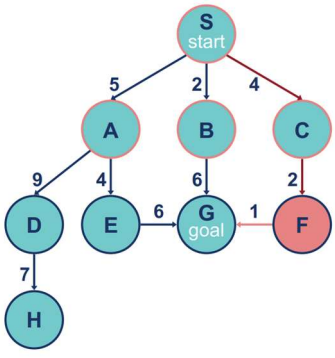
Feature	UCS	BFS
Edge Cost	Can handle **different costs**	Assumes **equal cost**
Priority	Based on **path cost** (`g(n)`)	Based on **depth/level**
Data Structure	**Priority Queue**	**FIFO Queue**
Optimality	**Optimal** (guaranteed)	If all costs are equal
Completeness	**Complete**	Complete

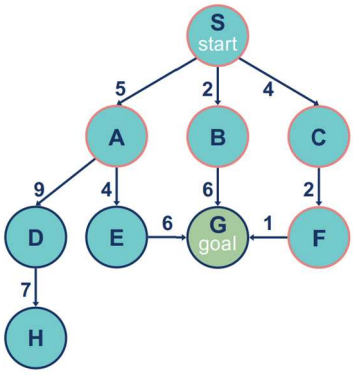
Problem1:

≫From the following graph to solve the problem where you need to reach the **destination node G & starting from node S.** Apply **Uniform Cost Search (UCS).** Show the steps of searching for the best path.

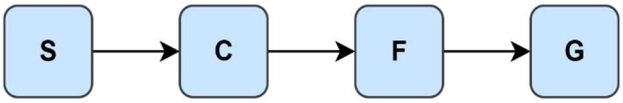


Frontier List	Expand List	Graph Explored
{(S,0)}	S	
{(S-B,2)} {(S-C,4)} {(S-A,5)}	B	

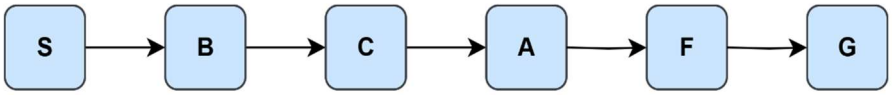
{(S-C,4)} {(S-A,5)} {(S-B-G,8)}	C	
{(S-A,5)} {(S-B-G,8)} {(S-C-F,6)}	A	
{(S-B-G,8)} {(S-C-F,6)} {(S-A-E,9)} {(S-A-D,14)}	F	
{(S-B-G,8)} {(S-A-E,9)} {(S-A-D,14)} {(S-C-F-G,7)}	G goal	

{(S-B-G,8)}	{(S-A-E,9)}	No Expend	
{(S-A-D,14)}	{(S-C-F-G,7)}		

⇨ **Actual path is:**



⇨ **Traversed path:**



⇨ **Actual Cost:**

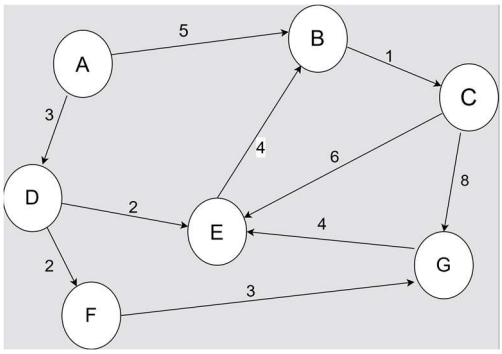
Cost: 7

Properties of UCS

Property	Value
Complete	Yes (if step costs $\geq \varepsilon > 0$)
Optimal	Yes (finds the least-cost path)
Time	**$O(b^{(1 + C^*/\varepsilon)})$,** where C^* is optimal cost, ε is smallest edge cost
Space	Can be large (stores all expanded paths)

Problem2:

≫From the following graph to solve the problem where you need to reach the **destination node G & starting from node A.** Apply **Uniform Cost Search (UCS).** Show the steps of searching for the best path.



Step	Frontier List	Expand List	Explored List
1	{(A,0)}	A	NULL
2	{(A-**D**, 3), (A-B, 5)}	D	{A}
3	{(A-**B**, 5), (A-D-E, 5), (A-D-F, 5)}	B	{A, D}
4	{(A-D-**E**, 5), (A-D-F, 5), (A-B-C, 6)}	E	{A, D, B}
5	{(A-D-**F**, 5), (A-B-C, 6), ~~(A-D-E-B, 9)~~} *here **B** is **already explored**	F	{A, D, B, E}
6	{(A-B-**C**, 6), (A-D-F-G,8)}	C	{A, D, B, E, F}
7	{(A-D-F-**G**,8),~~(A-B-C-E,12)~~, (A-B-C-G, 14)} *here **E** is **already explored**	G	{A, D, B, E, F, C}
8	{(A-D-F-G,8)}	NULL	{A, D, B, E, F, C, **G**} **# GOAL Found!**

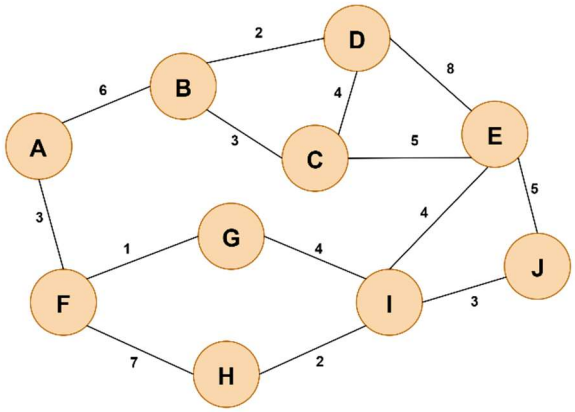
⇨ **Actual path is:** A → D → F → G

⇨ **Traversed path:** A → D → B → E → F → C → G

⇨ **Actual Cost:** **Cost: 8**

Problem3:

≫From the following graph to solve the problem where you need to reach the **destination node G & starting from node D.** Apply **Uniform Cost Search (UCS).** Show the steps of searching for the best path.



Step	Frontier List	Expand List	Explored List
1	{(D,0)}	D	NULL
2	{(D-**B**, 2), (D-C, 4), (D-E, 8)}	B	{D}
3	{**(D-C, 4),** (D-E, 8), (D-**B-C**, 5), (D-**B-A**, 8)}	C	{D, B}
4	{(D-E, 8), ~~(D-**B-C**, 5)~~, (D-**B-A**, 8), ~~(D-C-B,7)~~, (D-C-E,9)}	E	{D, B, C}
5	{(D-**B-A**, 8), ~~(D-C-E,9)~~ (D-E-I, 12), (D-E-J, 13),}	A	{D, B, C, E}
6	{(D-E-I, 12), (D-E-J, 13), (D-**B-A-F**, 11)}	F	{D, B, C, E, A}
7	{(D-E-I, 12), (D-E-J, 13), (D-**B-A-F-G**, 12), (D-**B-A-F-H**, 18)}	G	{D, B, C, E, A, F}
8	(D-**B-A-F-G**, 12)	NULL	{D, B, C, E, A, F, G} **# GOAL Found!**

⇨ **Actual path is:** D → B → A → F → G

⇨ **Traversed path:** D → B → C → E → A → F → G

⇨ **Actual Cost:** **Cost: 12**

DEPTH-LIMITED SEARCH (DLS)

Depth-Limited Search (DLS)

⇨ **Depth-Limited Search (DLS)** is a version of **Depth-First Search (DFS)** where the search is **limited to a predefined depth (ℓ)**.

Note: It helps **avoid infinite loops** in graphs or trees with cycles or infinite depth.

DFS can fail in:

- Infinite trees or graphs
- Cycles

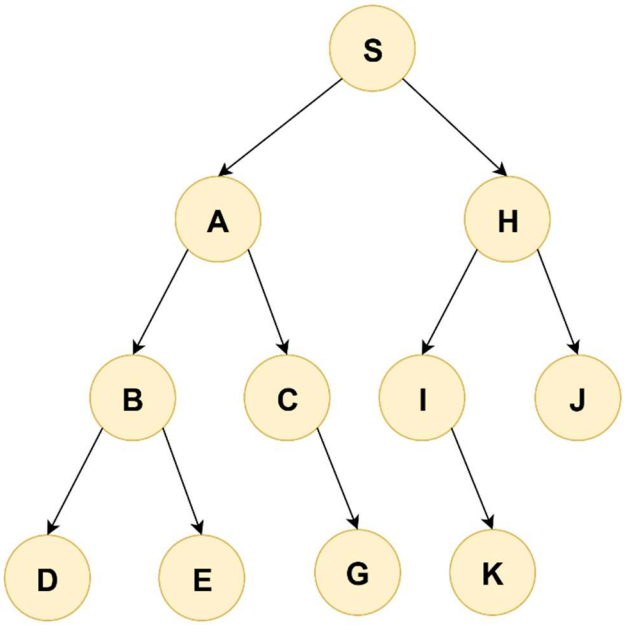
By limiting the depth of exploration, **DLS provides more control** and helps avoid such failures.

Algorithm Steps

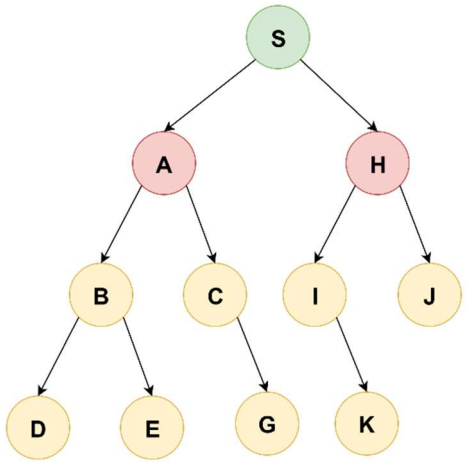
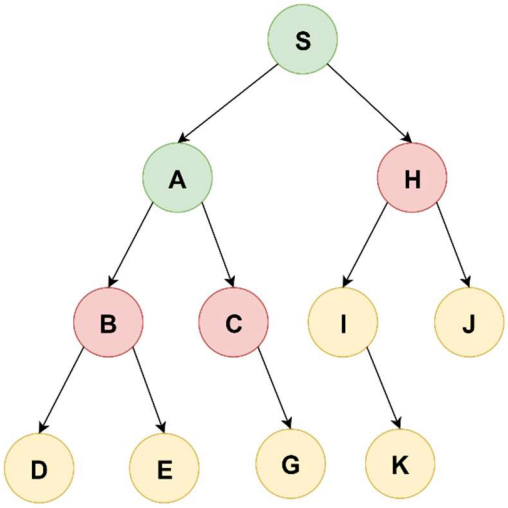
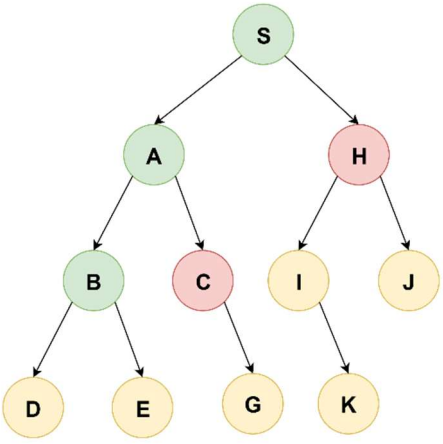
1. Start from the initial node.
2. Traverse the search tree **recursively**, tracking the current depth.
3. If the current depth exceeds the limit ℓ, **cut off** that path.
4. Continue until the goal is found or all paths within the limit are explored.

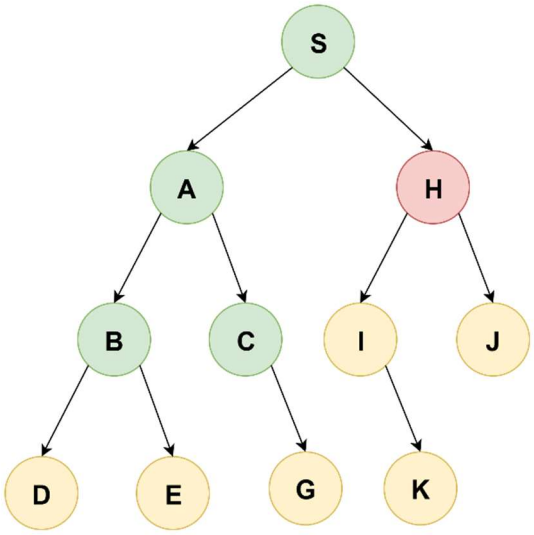
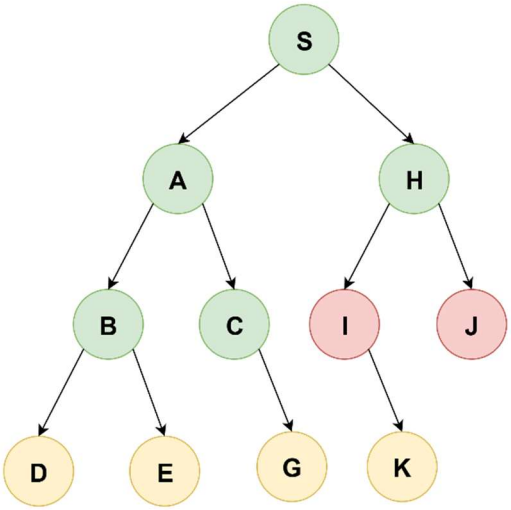
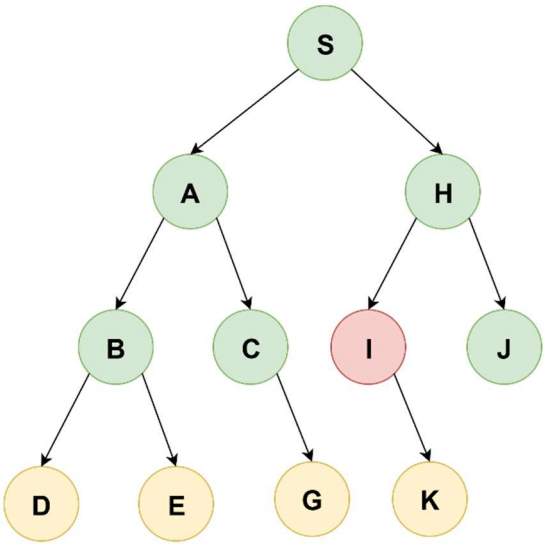
Problem1:

≫From the following graph to solve the problem where **S is the start node,** and **I is the goal node**. Note that the **edges in this graph are directed**. For this problem Goal node using Depth-**Limited** Search (DLS) Where and the depth limit is 2.

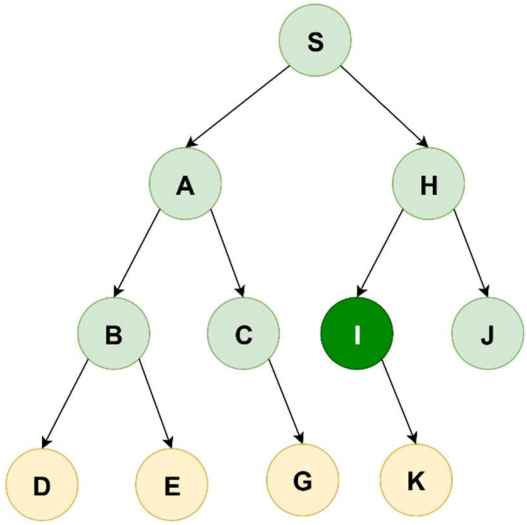


⇨ Solutions:

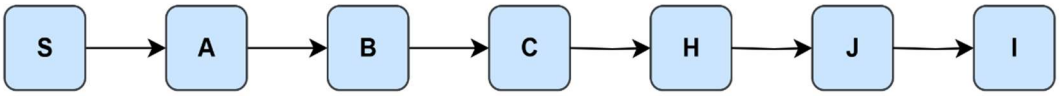
Vertex Visited	Vertex in Stack	Graph
S	A H	
S A	B C H	
S A B	C H	

S A B C	H	
S A B C H	J I	
S A B C H J	I	

S	A	B	C	H	J	I



⇨ **Now the path is:**

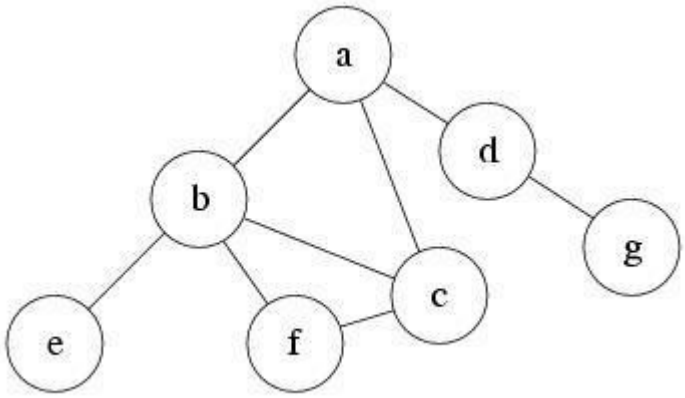


Properties of DLS

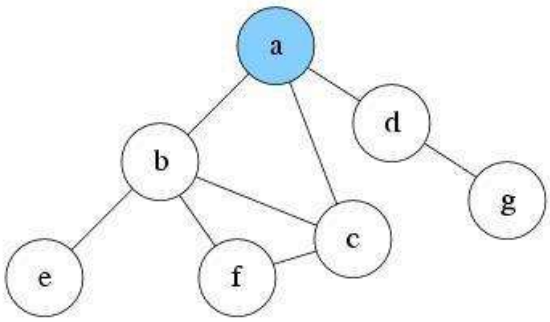
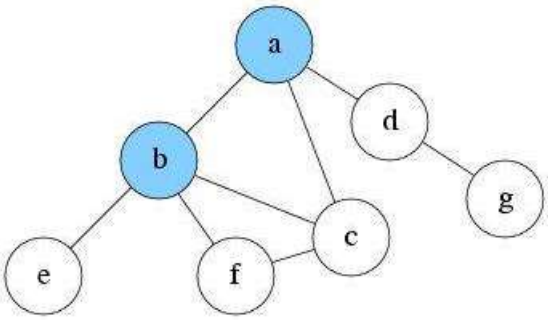
Property	Value
Completeness	No, if the solution is deeper than the limit
Optimality	No
Time	O(b^ℓ), where b = branching factor, ℓ = depth limit
Space	O(ℓ) (same as DFS – linear in depth)

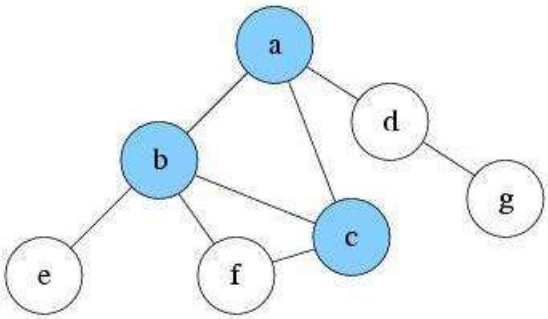
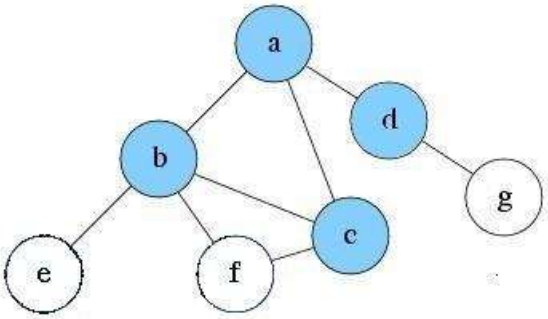
Problem2:

≫From the following graph to solve the problem where **'a' is the start node** and **'g' is the goal node**. Note that the **edges in this graph are directed**. For this problem Goal node using **Depth- Limited Search** (DLS) Where and the depth **limit is 1.**



⇨ Solutions:

Vertex Visited	Vertex in Stack	Graph
a	b c d	
a | b	c d	

a b c	d	
a b c d		
Cutoff: Goal **may exist**, but not found **within** the current depth limit.		

ITERATIVE DEEPENING SEARCH (IDS)

Iterative Deepening Search (IDS)

⇨ Iterative Deepening Search is a **search strategy** that combines the benefits of:

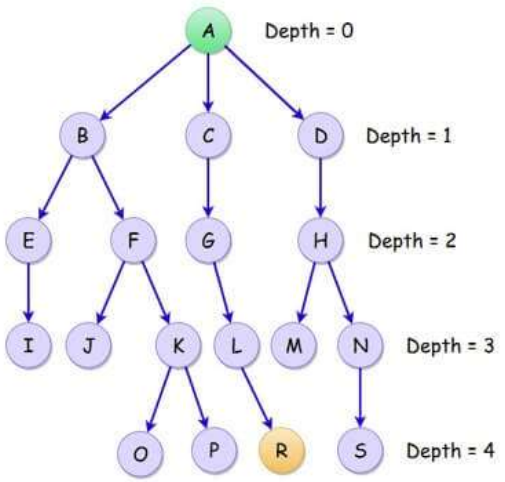
- **Depth-First Search (DFS)** (low memory usage)
- **Breadth-First Search (BFS)** (completeness and optimality for uniform step cost)

Note: It repeatedly performs depth-limited searches with increasing depth limits.

⇨ Core Idea

- Start with depth limit $\ell = 0$.
- Run **DLS** up to depth ℓ.
- If goal not found, increment ℓ and repeat.
- Stop when goal is found.

Example:

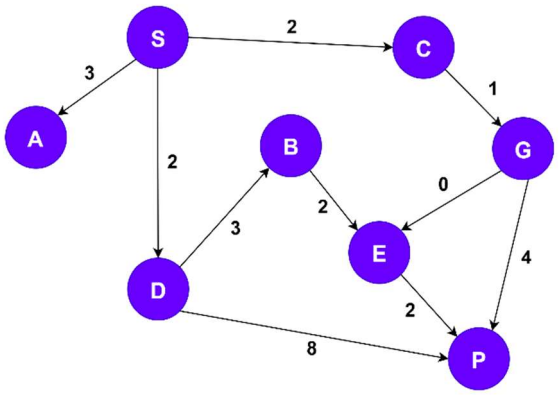
Graph	**IDS** (left to right)
	*DEPTH LIMITS* — IDDFS 0 — A 1 — A B C D 2 — A B E F C G D H 3 — A B E I F J K C G L D H M N 4 — A B E I F J K O P C G L R D H M N S

Properties of IDS

Property	Description
Completeness	Yes (will find solution if exists)
Optimality	Yes (if step cost = 1)
Time	$O(b^d)$, same as BFS
Space	O(d), same as DFS (much better than BFS)

Problem1:

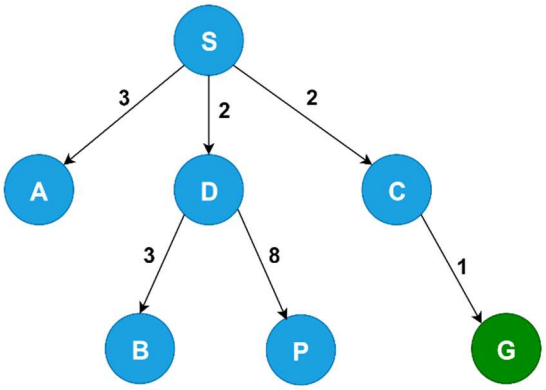
≫Show the simulation [frontier, explored set, path(solution), Path Cost, Tree] using Iterative depending. **Start node S** and **goal node G.** Insert the node in the frontier in alphabetic order.



Iterations	Depth	Frontier List	Explored List	Goal
1	0	[S]	[S]	Not Found
2	1	[A, C, D]	[S, A, C, D]	Not Found
3	2	[B, G, P]	[S, A, C, G, D, B, P]	**G found**

Term	Solution	Meaning
Explored	S, A, C, G, D, B, P	All nodes visited during the search
Solution Path	**S → C → G**	Actual path from start to goal
Path Cost	2 (S→C) + 1 (C→G) = **3**	Total weight of the solution path

Search Tree:



BIDIRECTIONAL SEARCH

Bidirectional Search

⇨ **Bidirectional Search** is a search algorithm that **runs two simultaneous searches**:

- One **forward** from the **initial state**
- One **backward** from the **goal state**

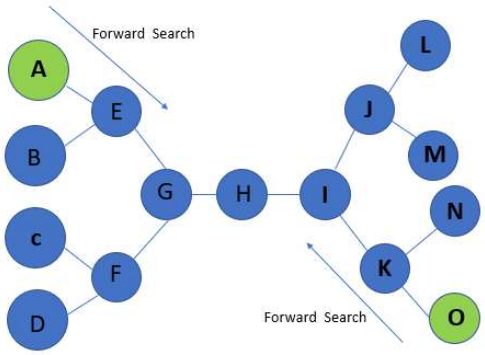
Note: The search stops when the two **searches meet**.

Algorithm Steps

1. Start two queues:
   - One from the **start node**
   - One from the **goal node**
2. Alternate between expanding:
   - Forward frontier from start
   - Backward frontier from goal
3. Stop when:
   - A node in the forward frontier also appears in the backward frontier.
4. Combine the two paths to form the complete solution.

Example:

Graph



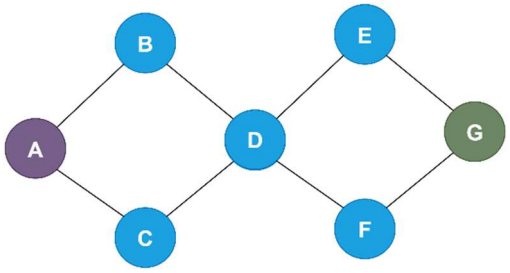
Bidirectional Search Steps

Forward Frontier		Backward Frontier	
Vertex Visited	Vertex in Queue	Vertex Visited	Vertex in Queue
A	E	O	K
A E	B G	O K	N I
A E B	G	O K N	I
A E B G	F H	O K N I	J H
A E B G F	H C D	O K N I J	H L M
A E B G F H	C D I	O K N I J H	L M G

Solution Path: A → E → B → G → F → H → J → I → N → K → O

Problem1:

≫Show the simulation using Bidirectional Search. **Start node A** and **goal node G.**



Bidirectional Search Steps			
Forward Frontier		**Backward Frontier**	
Vertex Visited	Vertex in Queue	Vertex Visited	Vertex in Queue
A	B C	G	E F
A B	C D	G E	F D
A B C	D	G E F	D
A B C D	E F	G E F D	B C
Solution Path: A → B → C → D → F → E → G			

Properties

Metric	Bidirectional Search	Breadth-First Search (BFS)
Time	$O\left(b^{\frac{d}{2}}\right)$	$O(b^d)$
Space	$O\left(b^{\frac{d}{2}}\right)$	$O(b^d)$
Completeness	Yes (if goal is reachable)	
Optimality	Yes (if BFS is used in both directions and edge cost = 1)	

BEST-FIRST SEARCH

Best-First Search (Greedy Search)

⇨ **Best-First Search** is an informed search algorithm that uses a **heuristic function** `h(n)` to select the **most promising node** to expand next.

Note: It expands the node that appears to be **closest to the goal**, based on the **heuristic estimate**.

➢ Heuristic Function (h(n))

- `h(n)` = Estimated cost from node `n` to the goal.
- It **does not** consider the cost from the start node.

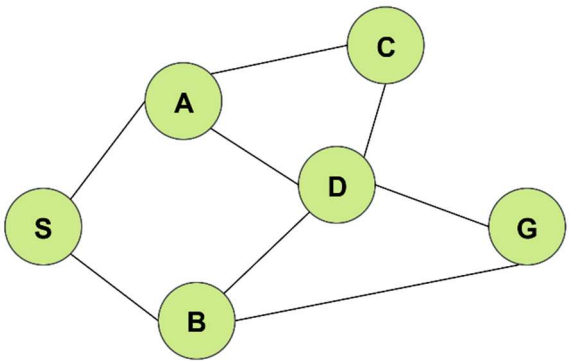
This is why it's called **Greedy Best-First Search** — it **greedily** chooses what looks best **now**, not considering the full path cost.

Algorithm Steps

1. Initialize a **priority queue** with the **start node**.
2. While the queue is not empty:
   - Remove the node with the **lowest** `h(n)` value.
   - If it's the **goal**, return the path.
   - Otherwise, expand the node and **insert its children** into the queue based on their `h(n)`.

Problem1:

≫ Consider **S** as the starting state and **G** is the goal state in given Graph. The heuristic values **h(n)** are provided beside the graph. Show the simulations for **Best First Search**.



H	V	h(H,V)
A	G	2
B	G	3
C	G	1
D	G	4
S	G	10
G	G	0

⇨ **Solutions:**

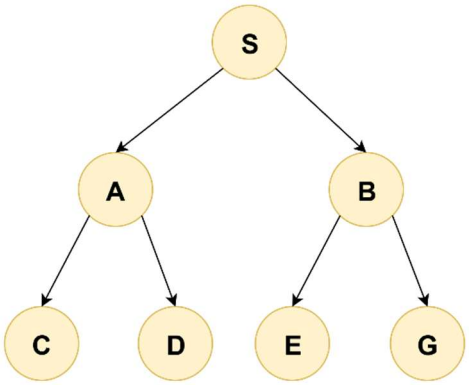
Iterations	Open	Close	Visit
1	Open [S]	Close []	NULL
2	Open [A, B]	Close [S]	S
3	Open [B, C, D]	Close [S, A]	A
4	Open [B, D]	Close [S, A, C]	C
5	Open [D, G]	Close [S, A, C, B]	B
6	Open [D]	Close [S, A, C, B, G]	G (GOAL)
Solution Path: S→ A → C → B→ G			

Properties of Best-First Search

Property	Value
Complete	Not always (may get stuck in loops)
Optimal	No (chooses based on h(n) only)
Time	$O(b^m)$ — depends on heuristic quality
Space	$O(b^m)$ — stores all nodes in memory

Problem2:

≫ Consider **S** as the starting state and **G** is the goal state in given Graph. The heuristic values **h(n)** are provided beside the graph. Show the simulations for **Best First Search**



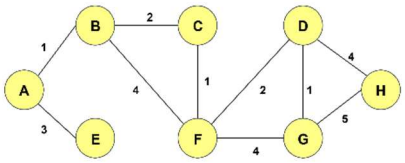
n	h(n)
A	2
B	3
C	1
D	4
S	10
E	6
G	0

Solutions:

Iterations	Open	Close	Visit
1	Open [S]	Close []	NULL
2	Open [A, B]	Close [S]	S
3	Open [B, C, D]	Close [S, A]	A
4	Open [B, D]	Close [S, A, C]	C
5	Open [D, G, E]	Close [S, A, C, B]	B
6	Open [D, E]	Close [S, A, C, B, G]	G (GOAL)
Solution Path: S⟶ A ⟶ C ⟶ B⟶ G			

☄ Problem3:

≫ Consider **A** as the starting state and **G** is the goal state in given Graph. The heuristic values **h(n)** are provided beside the graph. Show the simulations for **Best First Search**



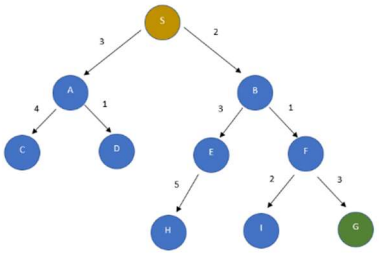
H	V	h(H,V)
A	G	9
B	G	7
C	G	4
D	G	1
E	G	10
F	G	3
H	G	5
G	G	0

⇨ Solutions:

Iterations	Open	Close	Visit
1	Open [A]	Close []	NULL
2	Open [B, E]	Close [A]	A
3	Open [E, C, F]	Close [A, B]	B
4	Open [E, C, D, G]	Close [A, B, F]	F
5	Open [E, C, D, H]	Close [A, B, F, G]	G (GOAL)
Solution Path: A → B → F → G			

☄ Problem4:

≫ Consider **S** as the starting state and **G** is the goal state in given Graph. The heuristic values **h(n)** are provided beside the graph. Show the simulations for **Best First Search**



node	H (n)
A	12
B	4
C	7
D	3
E	8
F	2
H	4
I	9
S	13
G	0

Iterations	Open	Close	Visit
1	Open [S]	Close []	NULL
2	Open [A, B]	Close [S]	S
3	Open [A, E, F]	Close [S, B]	B
4	Open [A, E, I, G]	Close [S, B, F]	F
5	Open [A, E, I]	Close [S, B, F, G]	G (GOAL)
Solution Path: S → B → F → G			

A* SEARCH ALGORITHM

A* Search Algorithm

⇨ **A* (A-Star) Search** is an informed search algorithm that finds the **least-cost path** to the goal using a combination of:

- `g(n)` = cost from start to current node `n`
- `h(n)` = heuristic estimate from `n` to goal

It uses:

$$f(n) = g(n) + h(n)$$

Note: A* selects the node with the **lowest estimated total cost** (`f(n)`) to reach the goal.

➢ Why A* is Powerful

- It balances:
  - **Cost so far** (g(n))
  - **Estimated cost to goal** (h(n))
- **Guarantees optimality**, if `h(n)` is **admissible** (never overestimates).

Algorithm Steps

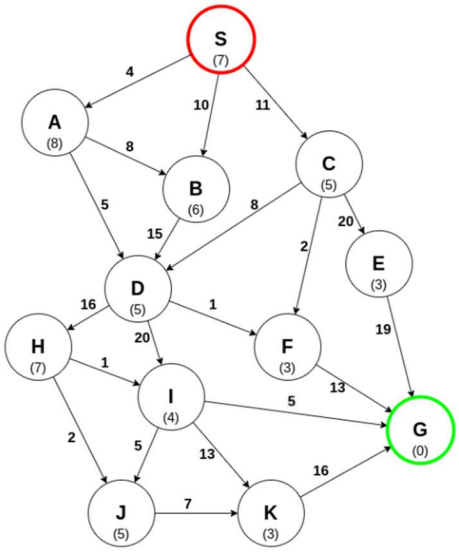
1. Add the **start node** to a **priority queue**, with `f(start) = g(start) + h(start).`
2. Loop:
   - Pick the node with the **lowest f(n)** from the queue.
   - If it is the goal, return the path.
   - Else, expand its children.
   - For each child:
     - Calculate `g(child)` and `f(child).`
     - Add to the queue if it has a lower cost path.

Properties of A*

Property	Description
Complete	Yes (if step cost ≥ ε > 0)
Optimal	Yes (if heuristic is **admissible** & **consistent**)
Time	$O(b^d)$ in worst case
Space	$O(b^d)$ — stores all paths in memory

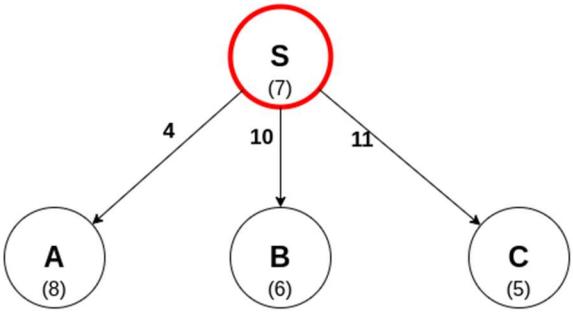
Problem1:

≫ Consider **S** as the starting state and **G** is the goal state in given Graph. The heuristic values **h(n)** is given in the **Node** and the step costs are given in the **edges**. Construct the search tree to find the **least cost path** from the start state S to the goal state G **using A* search** for the given graph. Clearly mention the **expand sequence** and, **path with path cost.**



⇨ Solutions:

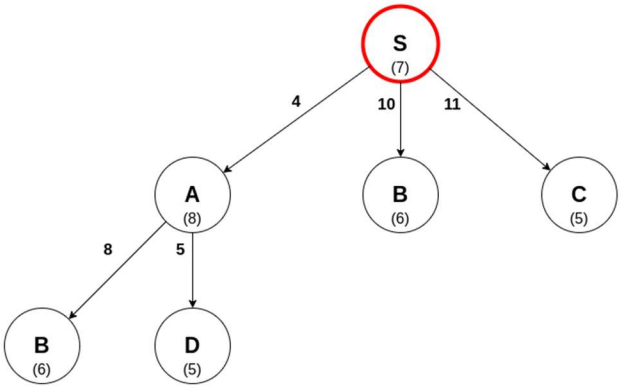
Exploring S:



Node[cost]
A[12]
B[16]
C[16]

Closed List
S

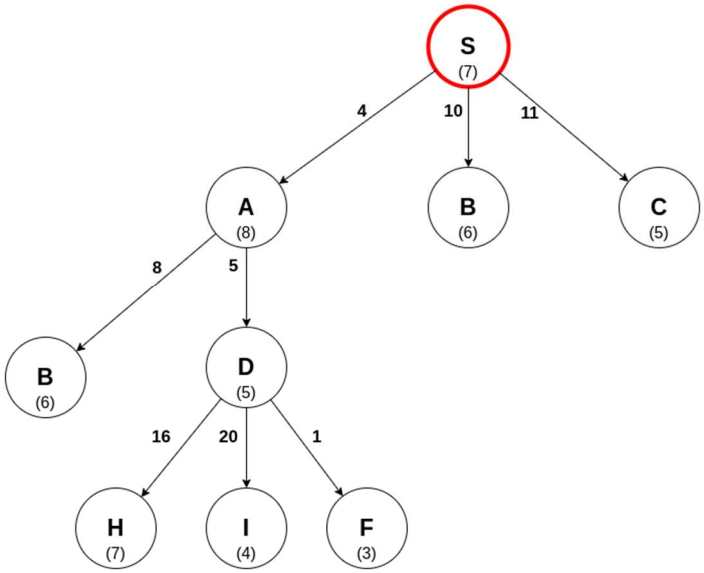
A is the current most promising path, so it is explored next:



Node[cost]
D[14]
C[16]
B[16]
B[18]

Closed List
S
A

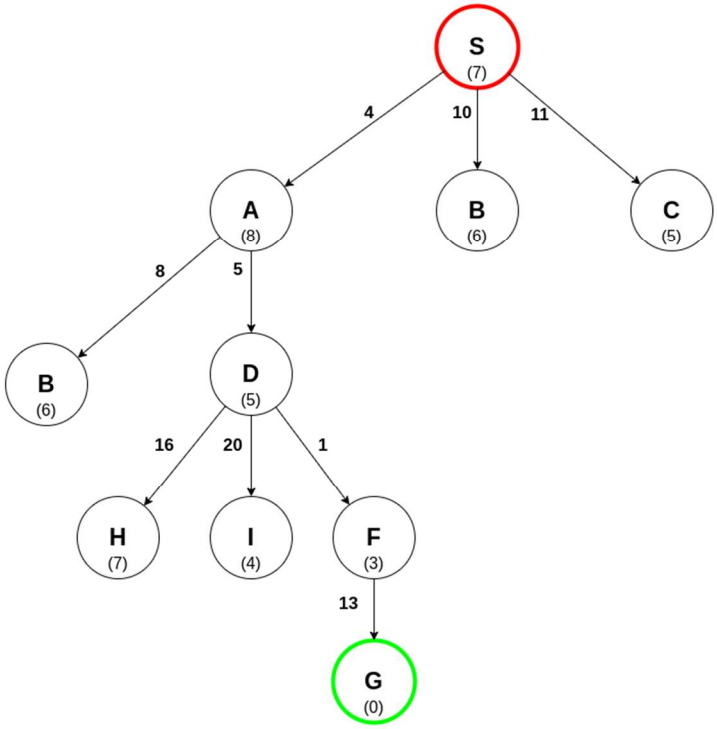
Exploring **D**:



Node[cost]
F[13]
C[16]
B[16]
H[32]
I[33]
B[18]

Closed List
S
A
D

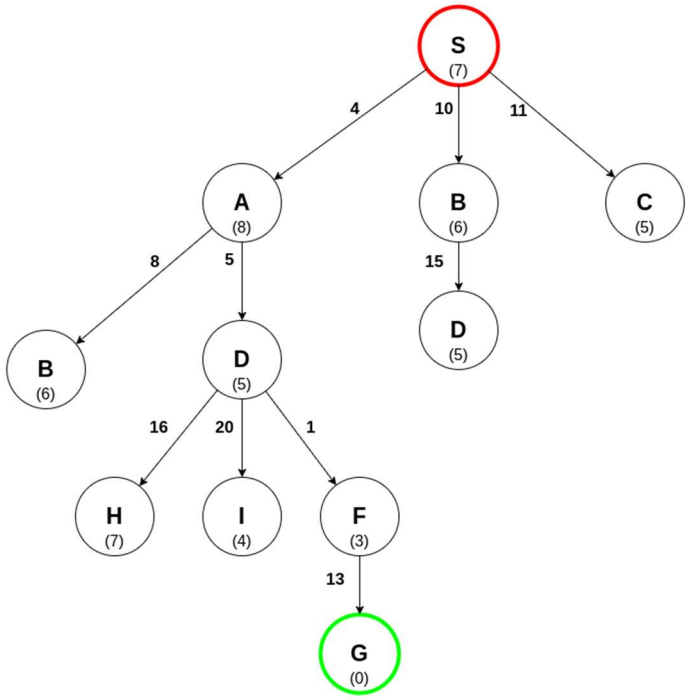
Exploring **F**:



Node[cost]
B[16]
C[16]
B[18]
H[32]
I[33]
G[23]

Closed List
S
A
D
F

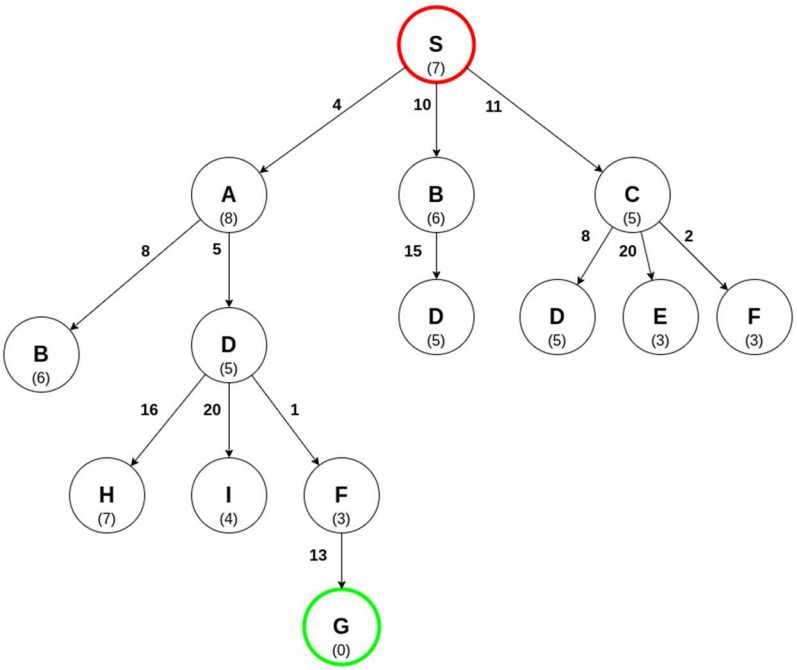
The goal node G is found but not yet explored, so the algorithm continues in case there's a shorter path. Node B appears twice in the open list (cost 16 from S, cost 18 from A); the lower-cost entry is explored next.



Node[cost]
C[16]
G[23]
B[18]
H[32]
I[33]

Closed List
S
A
D
F
B

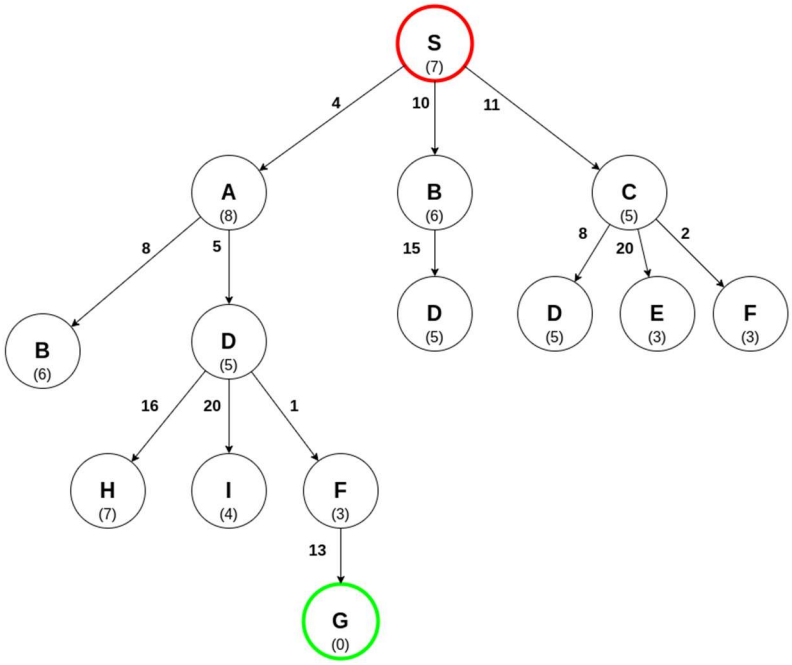
Exploring **C:**



Node[cost]
B[18]
G[23]
I[33]
H[32]
E[36]

Closed List
S
A
D
F
B
C

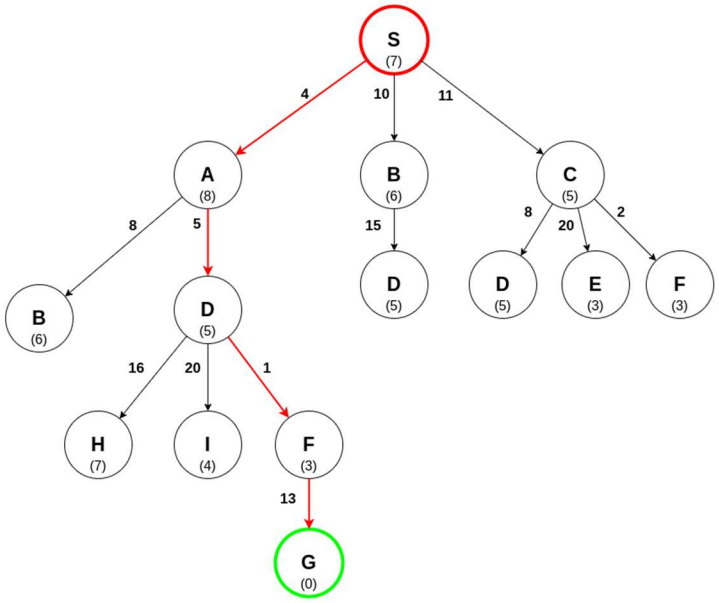
The next node in the open list is again **B**. However, because **B** has already been explored, meaning the shortest path to **B** has been found, it is not explored again, and the algorithm continues to the next candidate.



Node[cost]
G[23]
I[33]
H[32]
E[36]

Closed List
S
A
D
F
B
C

The next node to be explored is the **goal node G**, meaning the shortest path to G has been found! The path is constructed by tracing the graph backwards from G to S:



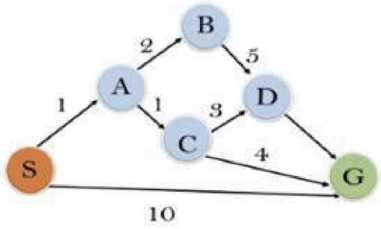
Node[cost]
I[33]
H[32]
E[36]

Closed List
S
A
D
F
B
C
G

Solution Path: S→ A → D → F → G	Solution Path Cost: 23

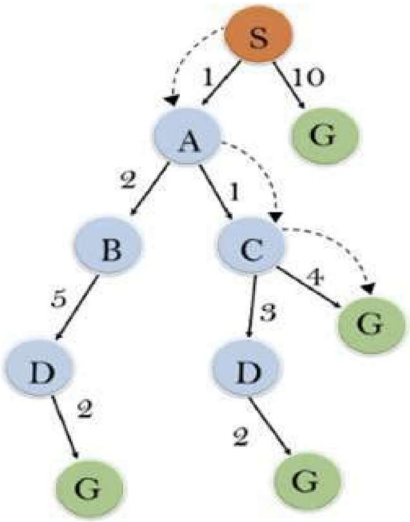
Problem2:

≫ Consider **S** as the starting state and **G** is the goal state in given Graph. The heuristic values **h(n)** are provided beside the graph and the step cost are labeled on the edges. Construct the search tree to find the **least cost path** from the start state S to the goal state G **using A* search** for the given graph. Clearly mention the **expand sequence** and, **path with path cost.**



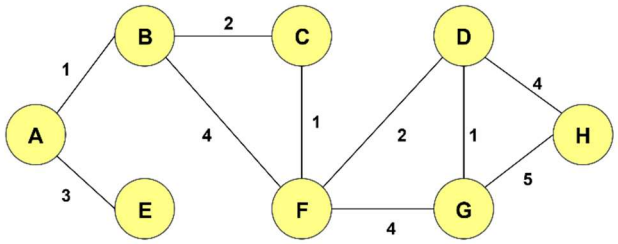
State	h(n)
S	5
A	3
B	4
C	2
D	6
G	0

Iterations	Expand Node	Expand Sequence
1	S	$\{(S \rightarrow A, 4), (S \rightarrow G, 10)\}$
2	A	$\{(S \rightarrow A \rightarrow C, 4), (S \rightarrow A \rightarrow B, 7), (S \rightarrow G, 10)\}$
3	C	$\{(S \rightarrow A \rightarrow C \rightarrow G, 6), (S \rightarrow A \rightarrow C \rightarrow D, 11), (S \rightarrow A \rightarrow B, 7), (S \rightarrow G, 10)\}$
4	G	$(S \rightarrow A \rightarrow C \rightarrow G, 6)$
Solution Path: S→ A → C → G		**Solution Path Cost: 6**
Tree:		



Problem3:

≫ Consider **A** as the starting state and **G** is the goal state in given Graph. The heuristic values **h(n)** are provided beside the graph and the step cost are labeled on the edges. Show the simulations for **A* Search**.



H	V	h(H,V)
A	G	9
B	G	7
C	G	4
D	G	1
E	G	10
F	G	3
H	G	5

⇨ Solutions:

Iterations	Expand Node	Expand Sequence
1	A	$\{(A \rightarrow B, 8), (A \rightarrow E, 13)\}$
2	B	$\{(A \rightarrow B \rightarrow C, 7), (A \rightarrow B \rightarrow F, 8), (A \rightarrow E, 13)\}$
3	C	$\{(A \rightarrow B \rightarrow C \rightarrow F, 7), (A \rightarrow B \rightarrow F, 8), (A \rightarrow E, 13)\}$
4	F	$\{(A \rightarrow B \rightarrow C \rightarrow F \rightarrow D, 7), (A \rightarrow B \rightarrow C \rightarrow F \rightarrow G, 8),$ $(A \rightarrow B \rightarrow F, 8), (A \rightarrow E, 13)\}$
5	D	$\{(A \rightarrow B \rightarrow C \rightarrow F \rightarrow D \rightarrow G, 7), (A \rightarrow B \rightarrow C \rightarrow F \rightarrow D \rightarrow H, 15),$ $(A \rightarrow B \rightarrow C \rightarrow F \rightarrow G, 8), (A \rightarrow B \rightarrow F, 8), (A \rightarrow E, 13)\}$
6	G	$(A \rightarrow B \rightarrow C \rightarrow F \rightarrow D \rightarrow G, 7)$
Solution Path: A→ B→ C → F→ D→ G		**Solution Path Cost: 7**

8-PUZZLE PROBLEM

8-puzzle problem (Uninformed Search)

Problem1:

≫ Given an initial state of an 8-puzzle problem and the final state to be reached, find the solution path from the initial state to the final state **using an appropriate uninformed search algorithm (e.g., BFS, UCS, or IDS)** and show the sequence of moves required.

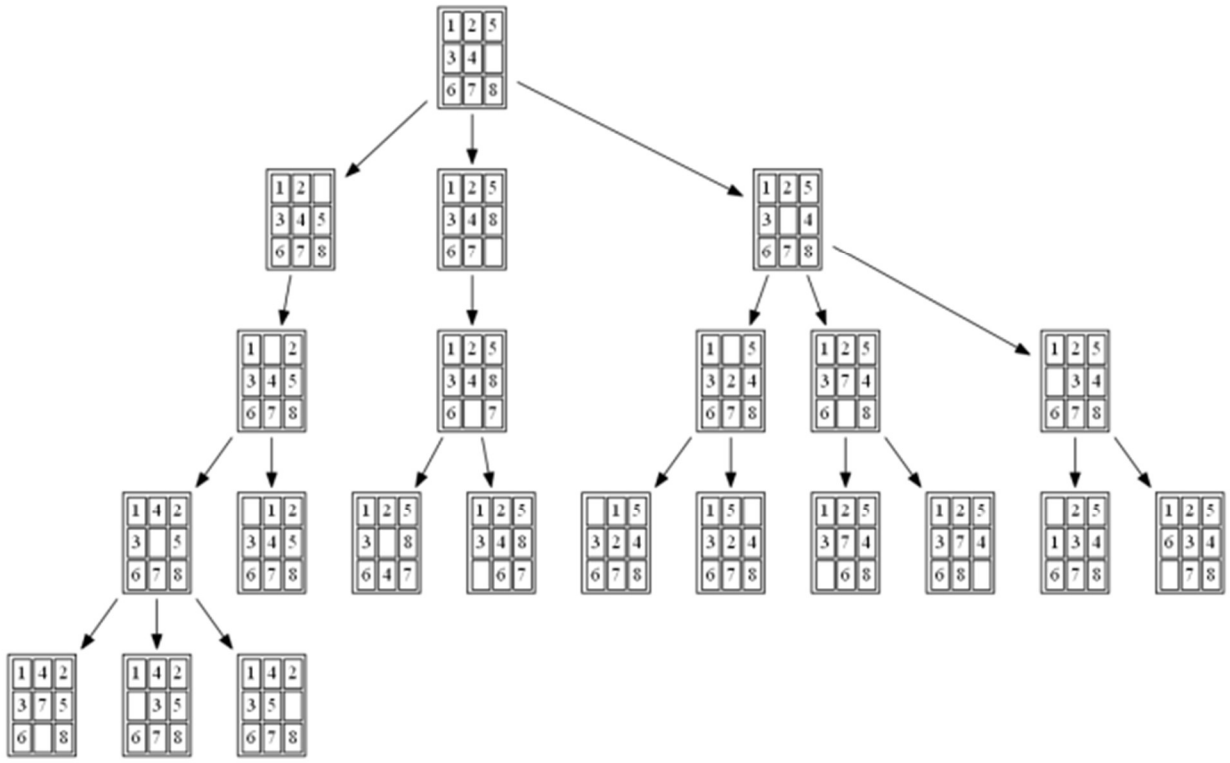
1	2	5
3	4	
6	7	8

Initial state

1	4	2
3	5	
6	7	8

Final state

Solution- using BFS:



8-puzzle problem (Informed Search)

Problem1:

≫ Given an initial state of a 8-puzzle problem and final state to be reached-

2	8	3
1	6	4
7		5

Initial State

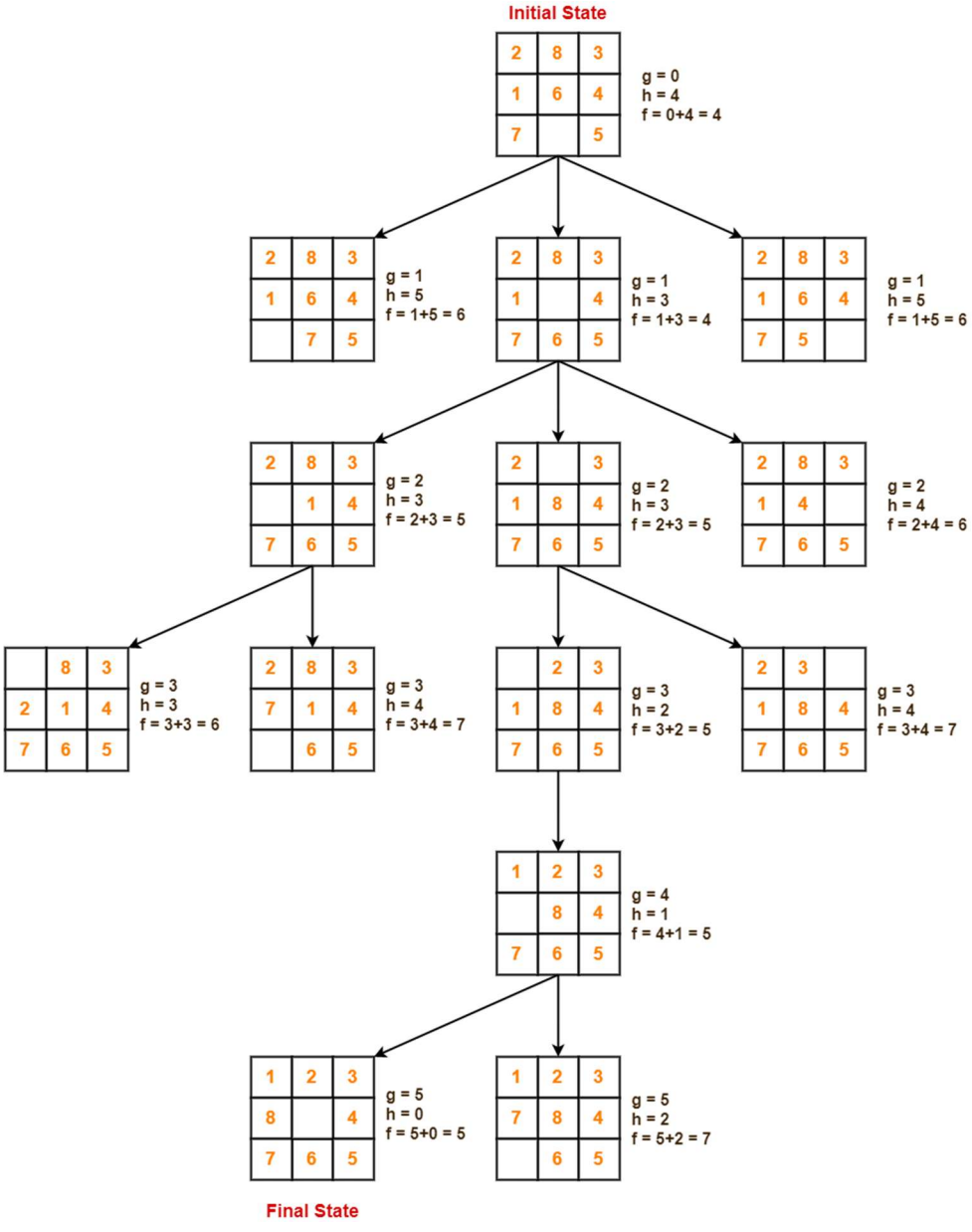
1	2	3
8		4
7	6	5

Final State

- Find the most cost-effective path to reach the final state from initial state using A* Algorithm.
- Consider $g(n) = \textbf{Depth of node}$ & $h(n) = \textbf{Number of misplaced tiles}$.

Solution-

- A* Algorithm maintains a tree of paths originating at the initial state.
- It continues until final state is reached.



Problem2:

≫ Find the heuristics for the 8-puzzel problem using the initial state of the given puzzle in Figure.

5	4	
6	1	8
7	3	2

Initial state

1	2	3
8		4
7	6	5

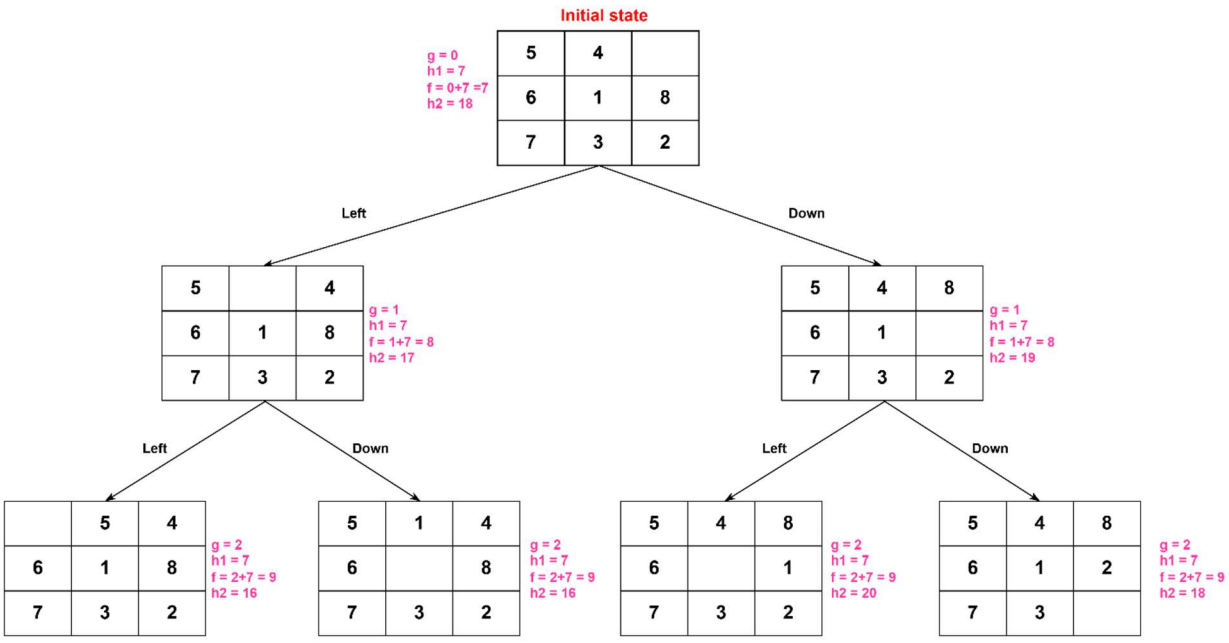
Final state

i. $\mathbf{h_1}$ = The number of misplaced tiles (Count how many tiles are not in their goal position)
ii. $\mathbf{h_2}$ = The sum of distances of the tiles from their goal position (use Manhattan distance for each tile)
iii. Draw the partial state space tree for the 8-puzzle problem based on the figure, up to depth level 2. Start from the initial state and generate all possible states up to two moves (depth = 2)

(i & ii)

h_1 = number of misplaced tiles	h_2 = sum of Manhattan distances
$h_1 = 1_5 + 1_4 + 1_8 + 1_2 + 1_3 + 1_6 + 1_1 = 7$	$h_2 = 4_5 + 2_4 + 2_8 + 3_2 + 3_3 + 2_6 + 2_1 = 18$

(iii)



Practice Problem

Problem1:

≫ Draw the schematic diagram of a Utility-Based Agent and describe its working procedure.

Solution-

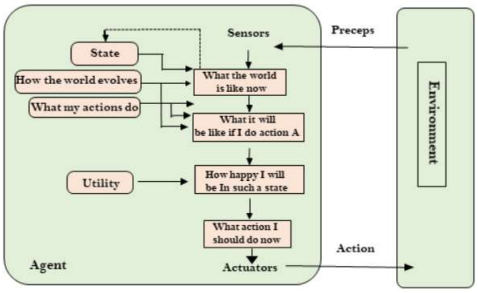


Figure 2.14 A model-based, utility-based agent. It uses a model of the world, along with a utility function that measures its preferences among states of the world. Then it chooses the action that leads to the best expected utility, where expected utility is computed by averaging over all possible outcome states, weighted by the probability of the outcome.

Problem2:

≫ Explain the properties of "Completeness" and "Cost Optimality" for the DFS and BFS search algorithms

Completeness:

⇨ **A search algorithm is** complete **if it is guaranteed to find a solution** if one exists**.**

Algorithm	Completeness	Reason
BFS	Yes	Explores all nodes level-by-level, so it will reach the goal if reachable in a finite graph.
DFS	No	May go infinitely deep in infinite state spaces or get stuck in loops without reaching the goal.

Cost Optimality:

⇨ A search algorithm is cost-optimal if it always returns the solution with the **lowest path cost**.

Algorithm	Cost Optimal	Reason
BFS	Yes (when all step costs are equal)	Finds the shallowest (fewest moves) solution, which is also the least cost if each step has cost = 1.
DFS	No	May find a deeper (more costly) solution before a shorter one, so not guaranteed optimal.

Problem3:

≫ Explain the "Completeness" and "Optimality" properties of **Iterative Deepening Search** algorithm.

Completeness:

- An algorithm is **complete** if it guarantees to find a solution **if one exists**.
- **IDS: Complete**
  - Because it performs **Depth-Limited Search** repeatedly with increasing limits ($\ell = 0, 1, 2, \ldots$), it will eventually reach the shallowest goal node.
  - Works for **finite state spaces** and step cost > 0.

Optimality:

- An algorithm is **optimal** if it always finds the **least-cost (or shallowest)** solution.
- **IDS: Optimal** (for **unit step costs**)
  - Because it finds the goal **at the smallest depth first**, just like BFS.
  - If costs vary (non-unit), IDS is **not guaranteed optimal** — in that case, UCS is used.

Problem4:

≫ Develop PEAS description of the task environment for a Candy Sorting Robot that can sort candies to the appropriate jars.

PEAS Element	Description
P – Performance Measure	Accuracy, Speed, Safety, Efficiency
E – Environment	Conveyor, Sorting Table, Jars, Lighting
A – Actuators	Robotic Arm, Conveyor Motor, Jar Mechanism
S – Sensors	Camera, Weight Sensor, Position Sensor, Proximity Sensor

Problem5:

≫ Describe the Turing Test that can be used to test the **'Acting humanly'** perspective.

Turing Test – Acting Humanly:

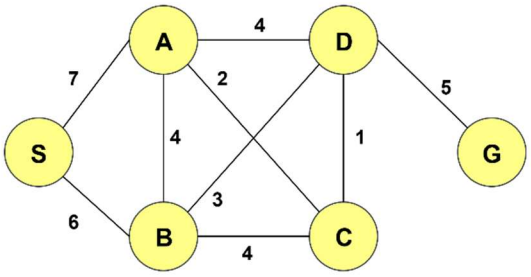
⇨ Proposed by **Alan Turing (1950)** to check if a machine can **act like a human**.
In the test:

- A **human judge** chats (text only) with a **human** and a **machine**.
- The judge asks any questions and tries to guess which is the machine.
- If the judge **can't reliably tell** which is which, the machine **passes**.

It matches the **"Acting Humanly"** perspective because it focuses on **human-like responses** rather than how the machine thinks.
Passing doesn't prove the machine is intelligent — only that it can **mimic human behavior** well enough to fool a person.

Problem6:

≫Simulate Depth-First Search (DFS) on the graph shown in below, **starting from the node S** and **aiming to reach the goal node G**. (push nodes into the frontier in alphabetic order)

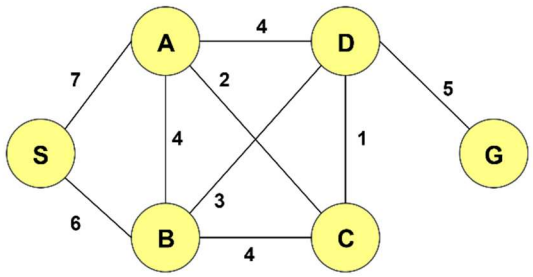


Steps	Vertex in Stack	Pop & Expand	Vertex Visited	Goal Node
1	S	S	Empty	Null
2	B A	B	S	**S** Not Goal
3	D C A	D	S B	**B** Not Goal

4	G C A	G	S B D	**D** Not Goal
5	C A	Null	S B D G	**G** Goal

Problem7:

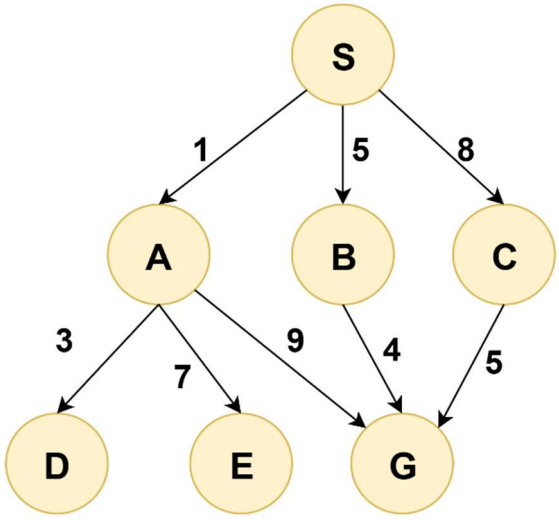
≫Simulate Breadth-First Search (BFS) on the graph shown in below, **starting from the node S** and **aiming to reach the goal node G**. (push nodes into the frontier in alphabetic order)



Steps	Vertex in Queue	Dequeue & Expand	Vertex Visited	Goal Node
1	S	S	Empty	Null
2	A B	A	S	**S** Not Goal
3	B C D	B	S A	**A** Not Goal
4	C D	C	S A B	**B** Not Goal
5	D	D	S A B C	**C** Not Goal
6	G	G	S A B C D	**D** Not Goal
7	Empty	Null	S A B C D G	**G** Goal

Problem8:

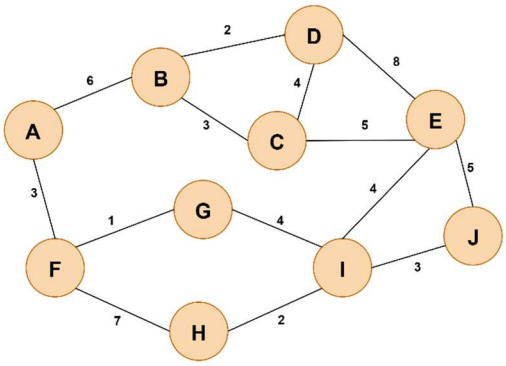
≫ Consider the given graph below, where you need to reach the destination node G starting from node S. Node {A, B, C, D & E} are the intermediate nodes. Your task is to find the path from S to the destination node G with the least cumulative cost using Uniform Cost Search. Show the steps of searching the best path.



Step	Frontier List	Expand List	Explored List
1	{(S,0)}	S	NULL
2	{(S-**A**,1), (S-**B**,5), (S-**C**,8)}	A	{S}
3	{(S-**B**,5), (S-**C**,8), (S-A-**D**,4), (S-A-**E**,8), (S-A-**G**,10)}	B	{S, A}
4	{(S-**C**,8), (S-A-**D**,4), (S-A-**E**,8), (S-A-**G**,10), (S-B-G,9)}	D	{S, A, B}
5	{(S-**C**,8), (S-A-**E**,8), (S-A-**G**,10), (S-B-**G**,9)}	C	{S, A, B, D}
6	{(S-A-**E**,8), (S-A-**G**,10), (S-B-**G**,9), (S-C-**G**,13)}	E	{S, A, B, D, C}
7	{(S-A-**G**,10), (S-B-**G**,9), (S-C-**G**,13)}	G	{S, A, B, D, C, E}
8	(S-B-**G**,9)	NULL	{S, A, B, D, C, E, G} # GOAL Found!
Solution Path: S→ B→ G		**Solution Path Cost: 9**	
Travers Path: S→ A→ B→ D→ C→ E→ G			

Problem9:

≫Perform Uniform Cost Search starting (UCS) form **Node A** with the goal of reaching Node **J.** If priority is same for different nodes, tie break using alphabetical order.



Step	Frontier List	Expand List	Explored List
1	{(A,0)}	A	NULL
2	{(A-**B**,**6**), (A-**F**,**3**)}	F	{A}
3	{(A-**B**,**6**), (A-F-**G**,**4**), (A-F-**H**,**10**)}	G	{A, F}
4	{(A-**B**,**6**), (A-F-**H**,**10**), (A-F-G-**I**,**8**)}	B	{A, F, G}
5	{(A-F-**H**,**10**), (A-F-G-**I**,**8**), (A-B-**D**,**8**), (A-B-**C**,**9**)}	D	{A, F, G, B}
6	{(A-F-**H**,**10**), (A-F-G-**I**,**8**), (A-B-**C**,**9**), (A-B-D-**E**,**16**)}	I	{A, F, G, B, D}
7	{(A-F-**H**,**10**), (A-B-**C**,**9**), (A-B-D-**E**,**16**), (A-F-G-I-**E**,**12**), (A-F-G-I-**J**,**11**)}	C	{A, F, G, B, D, I}
8	{(A-F-**H**,**10**), (A-B-D-**E**,**16**), (A-F-G-I-**E**,**12**), (A-F-G-I-**J**,**11**), (A-B-C-**E**,**14**)}	H	{A, F, G, B, D, I, C}
9	{(A-B-D-**E**,**16**), (A-F-G-I-**E**,**12**), (A-F-G-I-**J**,**11**), (A-B-C-**E**,**14**), ~~(A-F-H-**I**,**12**)~~}	J	{A, F, G, B, D, I, C, H}
10	(A-F-G-I-**J**,**11**)	NULL	{A, F, G, B, D, I, C, H, J} **# GOAL Found!**
Solution Path: A→ F→ G → I→ J		**Solution Path Cost: 11**	
Travers Path: A→ F→ G→ B→ D→ I→ C→ H→ J			

Problem10:

≫Which algorithm is comparatively faster (Expend less node) A* or UCS? Briefly explain.

⇨ **A*** is generally faster than **UCS** because it uses both the path cost from the start and a heuristic estimate to the goal, guiding the search more efficiently. This focused search means **A*** expands fewer nodes.

On the other hand, **UCS** only considers the path cost from the start, ignoring the goal direction, so it may explore many unnecessary nodes.

With an admissible and consistent heuristic, **A*** finds the optimal path faster by reducing the search space compared to **UCS**.

Problem11:

≫ The A* algorithm is complete and optimal, why?

⇨ **Complete:** It always finds a solution if one exists, as it systematically explores nodes by increasing estimated total cost and never ignores any possible paths.

⇨ **Optimal:** When using an **admissible heuristic** (one that never overestimates the true cost to the goal), A* guarantees the found path is the least-cost (optimal) path because it expands nodes in order of the lowest estimated total cost (actual cost so far + heuristic).

Problem12:

≫ The heuristic based algorithm in which the objective function is f(n) = (2- w)g(n) + wh(n) What kind of search does this perform when w = 0? When w = 1? When w =2?

When $w = 0$:

$$f(n) = (2 - 0)g(n) + 0 \times h(n) = 2g(n)$$

Since h(n) is ignored, the algorithm only considers the path cost so far. This behaves like **Uniform Cost Search (UCS)** (the factor 2 is just a constant scaling and does not affect the order of node expansions).

When $w = 1$:

$$f(n) = (2 - 1)g(n) + 1 \times h(n) = g(n) + h(n)$$

This is exactly the classic **A*** search formula.

When $w = 2$:

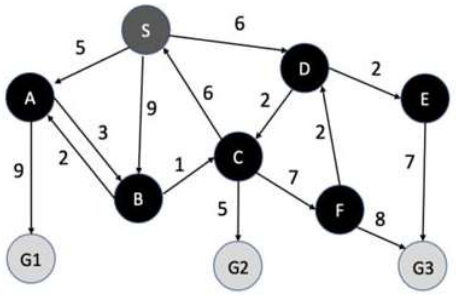
$$f(n) = (2 - 2)g(n) + 2 \times h(n) = 2h(n)$$

The algorithm ignores the path cost so far and only uses the heuristic. This behaves like a **Greedy Best-First Search**

Problem11:

(a) Consider the given graph where you need to reach any of the destination nodes {G1, G2, G3} starting from node S. Find the path from S to any of the destination nodes with the least cumulative cost using a Uniform Cost Search. (Lec 3.1)

(b) Is it possible to find the solution for the given graph using A* search? Explain your answer. [Lec 3.2]



(a)

Step	Frontier List	Expand List	Explored List
1	{(S,0)}	S	NULL
2	{(S-A,5), (S-B,9), (S-D,6)}	A	{S}
3	{(S-B,9), (S-D,6), (S-A-G1,14), (S-A-B,8)}	D	{S, A}
4	{(S-B,9), (S-A-G1,14), (S-A-B,8), (S-D-C,8), (S-D-E,8)}	B	{S, A, D}
5	{~~(S-B,9),~~ (S-A-G1,14), (S-D-C,8), (S-D-E,8), (S-A-B-C,9)}	C	{S, A, D, B}
6	{(S-A-G1,14), (S-D-E,8), ~~(S-A-B-C,9),~~ (S-D-C-G2,13), (S-D-C-F,15)}	E	{S, A, D, B, C}
7	{(S-A-G1,14), (S-D-C-G2,13), (S-D-C-F,15) (S-D-E-G3,15)}	G2	{S, A, D, B, C, E}
8	(S-D-C-G2,13)	NULL	{S, A, D, B, C, E, G2} **# GOAL Found!**
Solution Path: S→ D→ C → G2		**Solution Path Cost: 13**	
Travers Path: S→ A→ D→ B→ C→ E→ G2			

(b)

Yes, if we use an **admissible heuristic** (never overestimates the cost to the nearest goal). For multiple goals, use $h(n)$ =min of estimates to each goal. With $h = 0$, A* becomes UCS and still finds the optimal path.

Problem12:

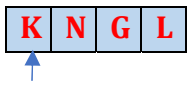
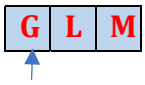
≫ Consider a grid where S is the starting state and G is the goal state. Movement is allowed in four directions: top, left, down, tight. provided the adjacent cell is an open box. Black boxes are obstacles and cannot be entered. You are to **construct the search tree** and solve the problem using: BFS. DFS. UCS

The default cost to move from or o another is **1**. If the target cell is on the boundary of the of the grid (but not a corner) the actions cost becomes **2.** If the target cell is in the corner, the actions cost become **3.**

S	A	B	
C		D	E
F	H	I	
	J	K	L
M	N	G	O

Using BFS:

Steps	Vertex in Queue	Expand	Explored List	Path
1	S	S	Empty	Null
2	A C	A	S	S
3	C B	C	S A	**S-A**
4	B F	B	S A C	**S-C**
5	F D	F	S A C B	**S-A-B**
6	D H	D	S A C B F	**S-C-F**
7	H E I	H	S A C B F D	**S-A-B-D**
8	E I J	E	S A C B F D H	**S-C-F-H**

9		I	S A C B F D H E	**S-A-B-D**
10		J	S A C B F D H E I	**S-C-F-H-I**
11		K	S A C B F D H E I J	**S-C-F-H-J**
12		N	S A C B F D H E I J K	**S-C-F-H-I-K**
13		G	S A C B F D H E I J K N	**S-C-F-H-J-N**
14		L	S A C B F D H E I J K N G	**S-C-F-H-I-K-G** **goal**
$\boldsymbol{Cost}$: $\mathbf{0+2+2+1+1+1+2=9}$				

Using DFS:

Steps	Vertex in Stack	Expand	Explored List	Path
1	**S**	S	**Empty**	Null
2	**C** **A**	C	S	S
3	**F** **A**	F	S C	S-C
4	**H** **A**	H	S C F	S-C-F
5	**J** **I** **A**	J	S C F H	S-C-F-H

6	N K I A	N	S C F H J	S-C-F-H-J
7	G M K I A	G	S C F H J N	S-C-F-H-J-N
8	O M K I A	O	S C F H J N G	S-C-F-H-J-N-G goal
$Cost: 0+2+2+1+1+2+2 = 10$				

Using UCS:

Steps	Vertex in Queue	Expand	Explored List	Path
1	(S,0)	S	Empty	Null
2	(C,2), (A,2)	A	S	S=0
3	(C,2), (B,4)	C	S A	**S-A=2**
4	(F,4), (B,4)	B	S A C	**S-C=2**
5	(F,4), (D,5)	F	S A C B	**S-A-B=4**
6	(D,5), (H,5)	D	S A C B F	**S-C-F=4**

7	(H,5), (I,6), (E,7)	H	S A C B F D	**S-A-B-D=5**
8	(J,6), (I,6), (E,7)	I	S A C B F D H	**S-C-F-H=5**
9	(J,6), (K,7), (E,7)	J	S A C B F D H I	**S-A-B-D-I=6**
10	(K,7), (E,7)	E	S A C B F D H I J	**S-C-F-H-J=6**
11	(K,7)	K	S A C B F D H I J E	**---------**
12	(G,9), (L,9)	G	S A C B F D H I J E K	**S-A-B-D-I-K=7**
13	-	-	S A C B F D H I J E K G	**S-A-B-D-I-K-G=9 goal**
$\boldsymbol{Cost}: \mathbf{0+2+2+1+1+1+2=9}$				

Problem13:

≫ What is the difference between tree search and graph search?

Tree Search – Expands nodes without tracking visited states. This can cause repeated state exploration and infinite loops in cyclic graphs. Uses less memory but may waste time revisiting nodes. Not always complete or optimal in graphs.

Graph Search – Maintains an *explored set* (visited states list) to avoid revisiting nodes. Prevents loops, improves efficiency in cyclic or large graphs, but requires more memory. Can be complete and optimal depending on the search strategy (e.g., BFS, UCS).

Problem14:

≫ What is the difference between goal agent and the utility agent

Aspect	Goal-Based	Utility-Based
Decision	Achieves a goal.	Maximizes utility value.
Evaluation	Success/failure.	Quality of outcome.
Flexibility	Any goal state is fine.	Chooses best among options.
Example	Deliver package anywhere on time.	Deliver package fastest & safest.

Problem15:

Design a painting agent for a 12×12 grid (144 squares). The agent may **move its arm to another square only after fully painting its current square**.

(a) Formulate this as a search problem by specifying the **initial state, actions, transition model, goal test, and path cost**.

(b) **Calculate and justify the total state-space size**.

(c) Classify the task environment as: **single vs. multi-agent, fully vs. partially observable, deterministic vs. stochastic, dynamic vs. static, episodic vs. sequential** (justify briefly).

(a)

Problem Formulation:

- **State:** Agent's position **(x, y)** and the set of painted squares **P**.
- **Initial State:** Start at **(1, 1)** with no painted squares.
- **Actions:**
  - If current square is unpainted → **Paint**.
  - If painted → **Move** (North, South, East, West) within grid.
- **Transition Model:**
  - **Paint** → mark square painted, stay in place.
  - **Move** → go to neighbor square, painted set unchanged (only from a painted square).
- **Goal Test:** All **144 squares** painted.
- **Path Cost:** 1 per action (paint or move).

(b)

State-space size

- **Positions:** 12×12 = 144 possible locations.
- **Paint patterns:** Each square painted/unpainted ⇒ 2^{144} possibilities.
- **Total states:** 144×2^{144}
- **Note:** "Paint-before-move" limits actions but not state count; many states may be unreachable.

(c)

Environment properties

- **Single vs. Multi-agent: Single-agent** (only the painter acts).
- **Observability: Fully observable** (assumed the agent can sense its position and which squares are painted).
- **Determinism: Deterministic** (Paint and Move have predictable outcomes).
- **Dynamics: Static** (environment doesn't change unless the agent acts).
- **Episodic vs. Sequential: Sequential** (current choices affect future options via the painted set and position).
- **Discrete:** Yes (discrete states, actions, and time steps).
- **Known vs. Unknown model (optional): Known**, if grid layout and rules are given.

Mohsin Iban Hossain

AIUB, Python Notes

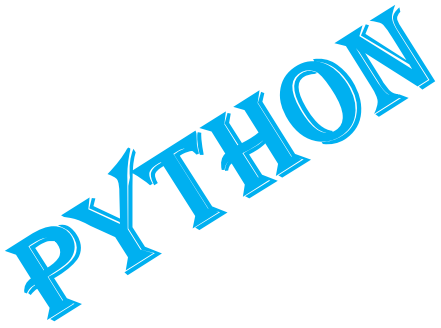


Table of Contents

INTRODUCTION TO PYTHON

Introduction

What is Python?

➔ **Python** is a high-level, easy-to-read programming language used for web development, data science, automation, and more. It is beginner-friendly, interpreted, and supports multiple programming styles.

➢ Our First Program in python:

Code:	Output:
`print("Hello World")`	Hello World

➢ Python Character Set:

➔ The **character set** in Python refers to all valid characters that can be used in writing Python programs.

1. Letters

- **Uppercase: `A` to `Z`**
- **Lowercase: `a` to `z`**

2. Digits

- **`0` to `9`**

3. Special Symbols

- `+ - * / % = < > ! & | ^ ~`
- `@ # $ _ ? : ; , . ' " \ ( ) [ ] { }`

4. Whitespace Characters

- **Space**
- **Tab (`\t`)**
- **Newline (`\n`)**
- **Carriage return (`\r`)**

Variables

➔ **Variable** is a name used to **store data** in memory. It acts as a container for values.

How to Declare a Variable:

Syntax:	Note:
x = 10 name = "Alice"	No need to declare the **type**—Python figures it out.
Rules for Naming Variables:	**Example Code:**
• Must start with a letter or underscore (_) • Cannot start with a digit • Can contain letters, digits, and underscores • Case-sensitive (`age` and `Age` are different)	`a = 5 # Integer` `pi = 3.14 # Float` `is_valid = True # Boolean` `name = "Mohsin" # String`

Data Types

➔ In Python, **data types** define the kind of value a variable hold.

Basic Data Types:

Type	Example	Description
`int`	`x = 10`	Whole numbers
`float`	`pi = 3.14`	Decimal numbers
`str`	`name = "Mohsin"`	Text (string of characters)
`bool`	`is_valid = True`	Boolean (True or False)

How to See the Data Type in Python:

Syntax:

```
type(variable)
```

Example Code:	Output:
`x = 10` `print(type(x))  # <class 'int'>` `name = "Mohsin"` `print(type(name))  # <class 'str'>` `price = 3.99` `print(type(price))  # <class 'float'>` `is_valid = True` `print(type(is_valid))  # <class 'bool'>`	<class 'int'> <class 'str'> <class 'float'> <class 'bool'>

Keywords

➔ **Keywords** are reserved words in Python. We **cannot** use them as variable names or identifiers.

List of Common Python Keywords:

False	class	finally	is	return	None
continue	for	lambda	try	True	def
from	nonlocal	while	and	del	global
not	with	as	elif	if	or
yield	assert	else	import	pass	break
except	in	raise	async	await	

Example Code:

Correct:	Wrong:
`if True:` `    print("This is a keyword example.")`	`class = "Math"` `# ❌ Error! 'class' is a keyword`

- **Problem1:** Write a Python program that assign values to variables num1, num2 and perform addition operations. Also print the results.

Code:	Output:
`num1 = 10` `num2 = 5` `sum = num1+num2` `print("Sum:", sum)`	Sum: 15

- **Problem2:** Write a Python program that assign values to variables num1, num2 and perform subtraction operations. Also print the results.

Code:	Output:
`num1 = 10` `num2 = 5` `sub = num1-num2` `print("Sub:", sub)`	Sub: 5

- **Problem3:** Write a Python program that assign values to variables num1, num2 and perform multiplication & division operations. Also print the results.

Code:	Output:
`num1 = 10` `num2 = 5` `mult = num1*num2` `div = num1/num2` `print("Mult:",mult)` `print("Div:",div)`	Mult: 50 Div: 2.0

Comments in Python

➔ **Comments** are notes in the code that are ignored by Python during execution. They help explain the code for humans.

Types of Comments:

Single-line Comment: starts with #

```
# This is a single-line comment
print("Hello")  # This prints Hello
```

Multi-line Comment:
use triple quotes ''' ''' or """ """

```
'''
This is a
multi-line comment
'''
print("Hi")
```

Note: Triple quotes are technically multi-line strings, but often used as multi-line comments.

Types of Operators

➔ Python supports **several types of operators** to perform operations on variables and values.

Type	Example Operators	Description
1. Arithmetic	+ - * / % // **	Used for math operations
2. Assignment	= += -= *= /=	Assign or update values
3. Comparison	== != > < >= <=	Compare values (returns True/False)
4. Logical	and or not	Combine conditions
5. Bitwise	`&	^ ~ << >>`

Example Code:

Types of Operators:

```
x = 10              # Assignment
y = 5               # Assignment

print(x + y)        # Arithmetic
print(x > y)        # Comparison
print(x != y)       # Comparison
print(x and y)      # Logical
```

Type Conversion

➔ **Type Conversion** means changing the data type of a value or variable.

Types of Type Conversion:

Implicit Conversion (Automatic):

➔Python automatically converts one data type to another when needed.

```
x = 10        # int
y = 2.5       # float
result = x + y
print(result)       # 12.5 (float)
print(type(result)) # <class 'float'>
```

Explicit Conversion (Type Casting):

➔We manually convert types using built-in functions:

Syntax:

```
int()    # Converts to integer
float()  # Converts to float
str()    # Converts to string
bool()   # Converts to boolean
```

Example code:	Output:
`x = 5.9` `print(int(x))  # 5` `print(type(x))  # <class 'float'>` `y = 100` `print(str(y))  # "100"` `print(type(y))  # <class 'int'>` `z = "45"` `print(int(z))  # 45` `print(type(z))  #<class 'str'>`	`5` `<class 'float'>` `100` `<class 'int'>` `45` `<class 'str'>`

Input in Python

➔ The `input()` function is used to **take user input** in Python.

Syntax:

```
variable = input("Enter something: ")
```

Example Code:

Example code:

```
name = input("Enter your name: ")
print("Hello,", name)
```

Output:

```
Enter your name: Mohsin Ibna Hossain
Hello, Mohsin Ibna Hossain
```

Convert input to other types:

```
# Converts input to integer
age = int(input("Enter your age: "))

# Converts input to float
price = float(input("Enter price: "))

print("Age:",age)
print("Price:",price)
```

Output:

```
Enter your age: 23
Enter price: 99.99
Age: 23
Price: 99.99
```

Practice Problem

ø **Problem1:** Write a Program to input 2 numbers & print their sum.

Code:

```
num1 = int(input("Enter number 1: "))
num2 = int(input("Enter number 2: "))
sum = num1+num2

print("Sum:",sum)
```

Output:

```
Enter number 1: 10
Enter number 2: 20
Sum: 30
```

- **Problem2:** Write a Python program to take five float variables as input from the user. Find out the summation and average of five numbers

Code:	Output:

```
num1 = float(input("Enter number 1: "))
num2 = float(input("Enter number 2: "))
num3 = float(input("Enter number 3: "))
num4 = float(input("Enter number 4: "))
num5 = float(input("Enter number 5: "))

summ = num1 + num2 + num3 + num4 + num5
average = summ / 5

print("Summation:", summ)
print("Average:", average)
```

```
Enter number 1: 10
Enter number 2: 3
Enter number 3: 7
Enter number 4: 6
Enter number 5: 8
Summation: 34.0
Average: 6.8
```

- **Problem3:** Write a python program that takes an integer variable, Z and computes the result of $Z^5 + Z^7 + Z^9 + Z^4 - Z^3 * Z^2$. Do not use any built-in function.

Code:	Output:

```
z = int(input("Enter an integer number: "))

z2 = z * z        # Z^2
z3 = z2 * z       # Z^3
z4 = z2 * z2      # Z^4
z5 = z4 * z       # Z^5
z7 = z4 * z3      # Z^7
z9 = z5 * z4      # Z^9

result = z5 + z7 + z9 + z4 - (z3 * z2)

print("Z⁵ + Z⁷ + Z⁹ + Z⁴ - Z³*Z² =", result)
```

```
Enter an integer number: 4
Z⁵ + Z⁷ + Z⁹ + Z⁴ - Z³*Z² = 278784
```

- **Problem4:** Write a python program that has two variables, Base and Height and it computes the area of a triangle.

Code:	Output:
`a=float(input("Enter Base: "))` `b=float(input("Enter Height: "))` `area=0.5*a*b` `print("Area is:",area)`	`Enter Base: 5` `Enter Height: 4` `Area is: 10.0`

- **Problem5:** Write a Python program to input two integer numbers, a and b. The program should print True if a is greater than or equal to b. Otherwise, print False.

Code:	Output:
`a = int(input("Enter the integer (a): "))` `b = int(input("Enter the integer (b): "))` `result = a >= b` `print(result)`	`Enter the integer (a): 10` `Enter the integer (b): 8` `True`
	`Enter the integer (a): 10` `Enter the integer (b): 20` `False`

- **Problem6:** Write a python program that has one variables, Side and it computes the area of a square.

Code:	Output:
`side = float(input("Enter side: "))` `area = side*side` `print("Area is:",area)`	`Enter side: 5` `Area is: 25.0`


STRING

Strings

➔ A **string** is a sequence of characters enclosed in **single**, **double**, or **triple quotes**.

Common String Operations:

Operation	Example	Output
Concatenation	`"Hello" + " World"`	`'Hello World'`
Repetition	`"Hi" * 3`	`'HiHiHi'`
Indexing	`"Hello"[1]`	`'e'`
Slicing	`"Hello"[1:4]`	`'ell'`
Length	`len("Hello")`	`5`
Uppercase	`"hello".upper()`	`'HELLO'`
Lowercase	`"HELLO".lower()`	`'hello'`
Replace	`"apple".replace("a", "A")`	`'Apple'`
Strip	`" hello ".strip()`	`'hello'`

Concatenation

➔ **Concatenation** means **joining two or more strings** together using the + operator.

Example code:	Output:
`str1 = "Hello"` `str2 = "World"` `result = str1 + " " + str2` `print(result)`	`Hello World`

Repetition

➔ **Repetition** means repeating a string multiple times using the * operator

Example code:	Output:
`text = "Hi"` `result = text * 3` `print(result)`	`HiHiHi`
Note: Works only with string and integer	

Indexing

➔ **Indexing** is used to access individual characters in a string using their position (index).

Example code:	Output:
`text = "Python"` `print(text[0])  # Output: P` `print(text[3])  # Output: h`	P h

Note: Indexing starts at 0

IndexError: Trying to access an index that doesn't exist will raise an error

```
text = "Python"
print(text[10])  # ❌ IndexError
```

Negative indexing: Negative indexing starts from the end

Example code:	Output:
`text = "Python"` `print(text[-1])  # Output: n (last character)` `print(text[-3])  # Output: h`	n h

Slicing

➔ **Slicing** allows you to extract a **portion of a string** using a range of indices.

Syntax:

```
string[start : end]
```

Example code:	Output:
`text = "Python"` `print(text[1:4])  # Output: yth` `print(text[:3])  # Output: Pyt` `print(text[2:])  # Output: thon` `print(text[-4:-1]) # Output: tho`	yth Pyt thon tho

Note: Includes characters from **start index** up to, but **not including, end.**

Length

➔ To find the **length** (number of characters) in a string, use the built-in `len()` function.

Syntax:

```
len(string)
```

Example code:	Output:
`msg = "Hi there!"` `print(len(msg))  # Output: 9`	**9**

Note: Counts **all characters**, including spaces, symbols, and numbers.

Uppercase

➔ To convert all characters in a string to **uppercase**, use the `.upper()` method.

Syntax:

```
string.upper()
```

Example code:	Output:
`text = "python"` `print(text.upper())  # Output: PYTHON`	**PYTHON**

Notes:

- Doesn't change the original string (strings are immutable).
- Returns a **new string** with all letters in uppercase.

Lowercase

➔ To convert all characters in a string to **uppercase**, use the `.upper()` method.

Syntax:

```
string.lower()
```

Example code:	Output:
`text = "HELLO"` `print(text.lower())  # Output: hello`	**hello**

Notes:

- Returns a new string in lowercase.
- Useful for case-insensitive comparisons.

Replace

➔ The `.replace()` method is used to **replace parts of a string** with another string.

Syntax:

```
string.replace(old, new)
```

Example code:	Output:
`text = "I like apples"` `print(text.replace("apples", "oranges"))` `# Output: I like oranges`	**I like oranges**

Notes:

- It does not change the original string.
- We can chain **.replace()** multiple times if needed.

Strip

➔ The `.strip()` method is used to **remove whitespace or specified characters** from the **start and end** of a string.

Syntax:

```
string.strip()              # removes spaces
string.strip(chars)         # removes specified characters
```

Example code:

```
text = "  Hello  "
print(text)
print(text.strip())

name = "***Mohsin***"
print(name)
print(name.strip("*"))
```

Output:

```
  Hello
Hello

***Mohsin***
Mohsin
```

Notes:

- Use **.lstrip()** to remove from the left only.
- Use **.rstrip()** to remove from the right only.

String Functions

String Functions:

```
str1 = "I am a coder."

print(str1.endswith("er."))          # returns True if string ends with "er."
print(str1.capitalize())             # capitalizes the first character
print(str1.find("a"))                # returns index of first occurrence of "a"
print(str1.count("am"))              # counts the occurrence of "am"
```

Practice Problem

- **Problem1:** Write a Program to input user's first name & print its length.

Code:	Output:
`` first_name = input("Enter first name: ") length = len(first_name) print("Length of first name is:", length) ```	``` Enter first name: Mohsin Length of first name is: 6 ```

- **Problem2:** Write a Python program to initialize a string and find how many times the character '$' appears in it. text = "He bought a pen for $5 and a book for $10, total $15."

Code:	Output:
`` text = "He bought a pen for $5 and a book for $10, total $15." count = text.count('$') print("The character '$' appears", count, "time(s).") ```	``` The character '$' appears 3 time(s). ```

- **Problem3:** Write a Python program to initialize a string containing a sentence, and then convert and print the entire string in **uppercase letters** using a string function.

Code:	Output:
`` sentence = "Learning Python is fun." upper_sentence = sentence.upper() print("Uppercase:", upper_sentence) ```	``` Uppercase: LEARNING PYTHON IS FUN ```

CONDITIONAL STATEMENTS

Conditional Statements

➔ **Conditional statements** let you execute different blocks of code based on conditions.

Common String Operations:

`if` Statement:

```
x = 10
if x > 0:
    print("Positive number")
```

`if...else` Statement:

```
x = -5
if x >= 0:
    print("Positive")
else:
    print("Negative")
```

`if...elif...else` Statement:

```
x = 0
if x > 0:
    print("Positive")
elif x == 0:
    print("Zero")
else:
    print("Negative")
```

Notes:

- Conditions use **comparison operators** (==, !=, >, <, >=, <=)
- Python uses **indentation** to define blocks

Nested Conditional Statements

➔ A **nested conditional** means using one `if` or `else` block **inside another**.
It allows for **more complex decision making**.

Syntax:

```
if condition1:
    if condition2:
        # Block 1
    else:
        # Block 2
else:
    # Block 3
```

Example code:

```
age = 20
has_id = True

if age >= 18:
    if has_id:
        print("You are allowed to enter.")
    else:
        print("Please show your ID.")
else:
    print("You are not allowed to enter.")
```

Output:

```
You are allowed to enter.
```

Notes:

- Use indentation correctly (4 spaces recommended)
- Helps when multiple levels of checking are required
- Can combine with elif if needed

Practice Problem

- **Problem1:** Write a Python program to check whether a number entered by the user is **odd** or **even**, and print the result.

Code:	Output:
`num = int(input("Enter a number: "))` `if num % 2 == 0:` `    print("Even")` `else:` `    print("Odd")`	`Enter a number: 10` `Even` `Enter a number: 5` `Odd`

- **Problem2:** Write a Python program to **grade students based on their marks** using the following criteria:

- If marks are **90 or above**, grade = "A"
- If marks are **80 to 89**, grade = "B"
- If marks are **70 to 79**, grade = "C"
- If marks are **below 70**, grade = "D"

Then print the grade.

Code:	Output:
`marks = int(input("Enter student's marks: "))` `if marks >= 90:` `    grade = "A"` `elif marks >= 80:` `    grade = "B"` `elif marks >= 70:` `    grade = "C"` `else:` `    grade = "D"` `print("Grade:", grade)`	`Enter student's marks: 73` `Grade: C`

- **Problem3:** Write a Python program to input three numbers from the user and print the **greatest** among them.

Code:

```
a = int(input("Enter first number: "))
b = int(input("Enter second number: "))
c = int(input("Enter third number: "))

if a >= b and a >= c:
    greatest = a
elif b >= a and b >= c:
    greatest = b
else:
    greatest = c

print("The greatest number is:", greatest)
```

Output:

```
Enter first number: 10
Enter second number: 20
Enter third number: 30
The greatest number is: 30
```

```
Enter first number: 15
Enter second number: 10
Enter third number: 5
The greatest number is: 15
```

```
Enter first number: 7
Enter second number: 21
Enter third number: 14
The greatest number is: 21
```

- **Problem6: Write a Python program to check whether a number is divisible by 2, 5, and 11.** Also, **demonstrate the use of both OR and AND operators** in the program.

Code:

```
num = int(input("Enter a number: "))

# Using AND operator to check if divisible by 2, 5, and 11
if num % 2 == 0 and num % 5 == 0 and num % 11 == 0:
    print("number is divisible by 2, 5, and 11.")
else:
    print("number is NOT divisible by 2, 5, and 11.")

# Optional: Demonstrating OR operator (for learning)
if num % 2 == 0 or num % 5 == 0 or num % 11 == 0:
    print("number is divisible by at least one of 2, 5, or 11.")
else:
    print("number is not divisible by 2, 5, or 11.")
```

Output:

```
Enter a number: 110
number is divisible by 2, 5, and 11.
number is divisible by at least one of 2, 5, or 11.
```

```
Enter a number: 37
number is NOT divisible by 2, 5, and 11.
number is not divisible by 2, 5, or 11.
```

Loops

➔ **Loops** are used to execute a block of code repeatedly.

Types of Loops:

`for` Loop: iterate over a sequence	`while` Loop: runs while a condition is `True`
`for i in range(0?,5,2?):` `  print(i)  # Output: 0 2 4`	`count = 0` `while count < 3:` `  print(count)  # Output: 0 1 2` `  count += 1`
Notes: **range(Start?, End, Step?)** functions returns a sequence of numbers, starting from start (**0 by default),** and **increments by Step (1 by default),** and stops before a specified number.	

Practice Problem

- **Problem1:** Write a Python program to **print numbers from 1 to 5 using a `while` loop**.

Code:	Output:
`i = 1` `while i <= 5:` `  print(i)` `  i += 1`	1 2 3 4 5

- **Problem2:** Write a Python program to **print numbers from 5 to 1** using a while loop.

Code:	Output:
`i = 5` `while i >= 1:` `  print(i)` `  i -= 1`	5 4 3 2 1

- **Problem3:** Write a Python program to **input a number n** and print the **sum of all numbers from 1 to n** using a while loop.

Code:	Output:
`n = int(input("Enter a number: "))` `i = 1` `total = 0` `while i <= n:` `  total += i` `  i += 1` `print("Sum from 1 to", n, "is:", total)`	`Enter a number: 5` `Sum from 1 to 5 is: 15`

- **Problem4:** Write a Python program to input a number n and print its **multiplication table** using a while loop.

Code:	Output:
`` n = int(input("Enter a number: ")) i = 1 while i <= 10: print(n, "x", i, "=", n * i) i += 1 ```	Enter a number: 11 11 x 1 = 11 11 x 2 = 22 11 x 3 = 33 11 x 4 = 44 11 x 5 = 55 11 x 6 = 66 11 x 7 = 77 11 x 8 = 88 11 x 9 = 99 11 x 10 = 110

- **Problem5:** Write a Python program to **print numbers from 1 to 5 using a `using for` loop `& range().`**

Code:	Output:
`` for i in range(1, 6): print(i) ```	1 2 3 4 5

- **Problem6:** Write a Python program to **print numbers from 5 to 1 using a `using for` loop `& range().`**

Code:	Output:
`` for i in range(5, 0, -1): print(i) ```	5 4 3 2 1

- **Problem7:** Write a Python program to **input a number n** and print the **sum of all numbers from 1 to n** using a `for` loop `& range().`

Code:	Output:
`` n = int(input("Enter a number: ")) total = 0 for i in range(1, n + 1): total += i print("Sum from 1 to", n, "is:", total) ```	Enter a number: 5 Sum from 1 to 5 is: 15

BREAK & CONTINUE

Break & Continue

➔ Used to **control the flow** inside loops.

`break` Statement:	`while` **Loop**: runs while a condition is `True`
`for i in range(5):` `  if i == 3:` `  break` `  print(i)  # Output: 0 1 2`	`for i in range(5):` `  if i == 3:` `  continue` `  print(i)  # Output: 0 1 2 4`
Note: **Stops** the loop immediately.	**Note:** **Skips** the current iteration and moves to the next.

Pass Statement

➔ The `pass` statement is a **placeholder** used when a statement is required syntactically, but no code needs to run.

Syntax:	**Example:**
`if True:` `  pass  # Do nothing`	`for i in range(5):` `  if i == 3:` `  pass  # Placeholder, does nothing` `  print(i)`
Use Cases: • Empty function or class definitions • Placeholder for future code • Avoid syntax errors when a block is needed	

Printing Output in a Single Line

➔ By default, `print()` adds a **newline** (`\n`) after each call. To print on the **same line**, use the `end` parameter.

Syntax:

```
print(value, end=" ")
```

Example code:	Output:
`for i in range(3):` `  print(i, end=" ")`	0 1 2

Notes:

- end=" " adds a space instead of a new line
- You can use any character: end="-", end="", etc.

Practice Problem

- **Problem1:** Write a Python program that takes numbers as input from the user using a `while` loop. The loop should **stop if the user enters** `-1`. Print the **sum** of all entered numbers (excluding `-1`).

Code:	Output:
`total = 0` `while True:` `  num = int(input("Enter a number (-1 to stop): "))` `  if num == -1:` `  break` `  total += num` `print("Total sum is:", total)`	Enter a number (-1 to stop): 10 Enter a number (-1 to stop): 5 Enter a number (-1 to stop): 3 Enter a number (-1 to stop): 2 Enter a number (-1 to stop): 1 Enter a number (-1 to stop): -1 Total sum is: 21

- **Problem2:** Write a Python program to print all numbers from **1 to 10 in a line**, except for number **5**, using a `for` loop and the `continue` statement.

Code:	Output:
`for i in range(1, 11):` `  if i == 5:` `  continue` `  print(i , end=" ")`	1 2 3 4 6 7 8 9 10

LISTS

Lists

➔ A **list** is an **ordered**, **changeable (mutable)** collection that allows **duplicate** values. Just like **Array.**

Syntax:	Example:
`llistVariableName = ["item1", "item2"]`	`fruits = ["apple", "banana", "cherry"]`

Key Features:

- Can contain **mixed data types** (e.g., numbers, strings, etc.)

Common List Operations:

Operation	Example	Description
Access item	`fruits[0]`	`'apple'`
Change item	`fruits[1] = "mango"`	Replaces `'banana'` with `'mango'`
Add item	`fruits.append("orange")`	Adds to end
Insert item	`fruits.insert(1, "kiwi")`	Inserts at index 1
Remove item	`fruits.remove("apple")`	Removes `'apple'`
Pop item	`fruits.pop()`	Removes last item
Length	`len(fruits)`	Number of items
Sort	`fruits.sort()`	Sorts list
Reverse	`fruits.reverse()`	Reverses list
Slicing	`fruits[1:3]`	`'banana', 'cherry'`

Access item

➔ We can access elements in a list using **indexing**.

Syntax:

```
list_name[index]   #Indexing starts at 0 (first item), negative indexing starts from the end.
```

Example code:	Output:
`fruits = ["apple", "banana", "cherry"]` `print(fruits[0])  # Output: apple` `print(fruits[1])  # Output: banana` `print(fruits[-1])  # Output: cherry`	**apple** **banana** **cherry**

Change item

➔ Lists are **mutable**, so you can **change the value** of an item by referring to its index.

Syntax:

```
list_name[index] = new_value
```

Example code:

```
fruits = ["apple", "banana", "cherry"]
print("Before:",fruits)

fruits[1] = "mango"
print("After:",fruits)
```

Output:

```
Before: ['apple', 'banana', 'cherry']
After: ['apple', 'mango', 'cherry']
```

Notes:

- Index **must be valid** (within range)
- Only one item is updated at a time

Add item

➔ You can add items to a list using `append()` or `insert()` methods.

`extend()` – Add multiple items:

```
fruits = ["apple", "banana"]
fruits.extend(["grape", "melon"])

print(fruits)
# Output: ['apple', 'banana', 'grape', 'melon']
```

`append()` – Add at the end:

```
fruits = ["apple", "banana"]
fruits.append("cherry")

print(fruits)
# Output: ['apple', 'banana', 'cherry']
```

`insert(index, item)` – Add at specific position:

```
fruits = ["apple", "banana"]
fruits.insert(1, "orange")

print(fruits)
# Output: ['apple', 'orange', 'banana']
```

Inserting item

➔ To add an item at a **specific position** in a list, use the **insert()** method.

Syntax:

```
list_name.insert(index, item)
```

Example code:

```
fruits = ["apple", "banana", "cherry"]
print("Before:",fruits)

fruits.insert(1, "orange")
print("After:",fruits)
```

Output:

```
Before: ['apple', 'banana', 'cherry']
After: ['apple', 'orange', 'banana', 'cherry']
```

Notes:

- Index starts at 0
- Shifts elements to the right
- If index is greater than the length, item is added at the end

Remove item

➔ We can **remove** items from a list using the `remove()` method.

Syntax:

```
list_name.remove(item)
```

Example code:

```
fruits = ["apple", "banana", "cherry"]
fruits.remove("banana")
print(fruits)
```

Output:

```
['apple', 'cherry']
```

Notes:

- Removes first **occurrence** of the specified item.
- If the item is **not found**, it raises a **ValueError**.

Pop item

➔ The **pop()** method removes an item from the list **by index** and returns it.

Syntax:

```
list_name.pop(index)  # index is optional
```

Example code:	Output:
`fruits = ["apple", "banana", "cherry"]` `fruits.pop()` `print(fruits)`	`['apple', 'banana']`

Notes:

- If index is out of range, it raises an IndexError.
- Unlike remove(), pop() can be used to get the removed value.

List Methods

List Methods:

```
list = [2, 1, 3]

list.append(4)  #adds one element at the end [2, 1, 3,4]
list.sort( )  #sorts in ascending order [1, 2, 3]
list.sort( reverse=True )  #sorts in descending order [3, 2, 1]
list.reverse( )  #reverses list [3, 1, 2]
```

Practice Problem

- **Problem1:** Write a Python program to **ask the user to enter the names of their 3 favorite movies** and **store them in a list**.

Code:	Output:
`` movies = [] for i in range(3): movie = input("Enter the name of movie: ") movies.append(movie) print("Your favorite movies are:\n", movies) ```	Enter the name of movie: Inception Enter the name of movie: Interstellar Enter the name of movie: The Matrix Your favorite movies are: ['Inception', 'Interstellar', 'Matrix']

- **Problem2:** Write a Python program to **check if a list is a palindrome**, meaning the list reads the same forward and backward. **(Hint: Use the copy() method)**

Code:	Output:
`` elements = input("Enter elements: ").split() original_list = elements.copy() original_list.reverse() reversed_list = original_list if elements == reversed_list: print("The list is a palindrome.") else: print("The list is not a palindrome.") ```	Enter elements: 12121 The list is a palindrome. Enter elements: 1212 The list is a palindrome.

Taking User Input in a List

➔ We can take list input from a user using a loop or by splitting a string input.

Method 1: Using `split()` (for space-separated input)

Example code:

```
names = input("Enter Name: ").split()
print(names)
```

Output:

```
Enter Name: JAM, MOHSIN, NIAHN, ALIF
['JAM,', 'MOHSIN,', 'NIAHN,', 'ALIF']
```

Method 2: Using a loop (for multiple inputs)

Example code:

```
n = int(input("How many items? "))
items = []
for i in range(n):
    item = input(f"Enter item {i+1}: ")
    items.append(item)

print(items)
```

Output:

```
Enter item 1: apple
Enter item 2: banana
Enter item 3: cherry
['apple', 'banana', 'cherry']
```

Notes: `input(f"Enter item {i+1}: ")` uses an **f-string** to format the prompt correctly, displaying the item number to the user.

Practice Problem

ꕥ **Problem1:** Write a Python program that takes **5 numbers as input from the user**, **stores them in a list**, and then **prints the list** along with the **sum of all the numbers**.

Code:

```
numbers = []
sum = 0

for i in range(5):
    num = int(input("Enter a number: "))
    numbers.append(num)
    sum +=num

print("Your list:", numbers)
print("Sum of all numbers:",sum)
```

Output:

```
Enter a number: 10
Enter a number: 20
Enter a number: 30
Enter a number: 40
Enter a number: 50
Your list: [10, 20, 30, 40, 50]
Sum of all numbers: 150
```

TUPLES

Tuples

➔ A **tuple** is an **ordered**, **immutable** (unchangeable) collection of items. It is similar to a list but cannot be modified after creation.

Syntax:	Example:
`my_tuple = (item1, item2, item3)`	`fruits = ("apple", "banana", "cherry")` `print(fruits[0])  # Output: apple`

Key Features:

- Defined using **parentheses** `()`
- **Immutable**: We cannot change, add, or remove items after creation
- Allows **duplicate** values
- Can contain **mixed data types** (e.g., numbers, strings, etc.)

Tuple with One Item:

Example code:	Output:
`one_item = ("apple",)` `print(one_item)` `print(type(one_item))` `one_item = ("apple")` `print(one_item)` `print(type(one_item))`	`('apple',)` `<class 'tuple'>` `apple` `<class 'str'>`

Notes: Must include comma!

Tuples Methods

Tuples Methods:

```
fruits = ("apple", "banana", "cherry")

len(fruits)             # Length of the tuple
fruits.index("banana")  # Find index of an item
fruits.count("apple")   # Count how many times an item appears
```

Practice Problem

Problem1: **Write a Python program to count how many students received an `"A"` grade** in the following **list**: ["C", "D", "A", "A", "B", "B", "A"]

Code:	Output:
`grades = ("C", "D", "A", "A", "B", "B", "A")` `count_A = grades.count("A")` `print("Number of 'A' grade:", count_A)`	**Number of 'A' grade: 3**

Problem2: Write a Python program to **input 5 numbers from the user**, store them in a **tuple**, and then **print the maximum and minimum values**.

Code:	Output:
`numbers = []` `for i in range(5):` `  num = int(input("Enter a number: "))` `  numbers.append(num)` `number_tuple = tuple(numbers)` `print("Tuple:", number_tuple)` `print("Maximum value:", max(number_tuple))` `print("Minimum value:", min(number_tuple))`	**Enter a number: 10** **Enter a number: 20** **Enter a number: 30** **Enter a number: 40** **Enter a number: 50** **Tuple: (10, 20, 30, 40, 50)** **Maximum value: 50** **Minimum value: 10**

Problem3: Create a tuple with **mixed data types** (e.g., `("Mohsin", 21, "CSE")`) and **print each item along with its data type**.

Code:	Output:
`info = ("Mohsin", 21, "CSE")` `for item in info:` `  print(item, "=>", type(item))`	**Mohsin => <class 'str'>** **21 => <class 'int'>** **CSE => <class 'str'>**

Dictionary

➔ A **dictionary** is an **unordered**, **mutable** collection that stores **key:value pairs**.

Syntax:

```
my_dict = {
    "key1": "value1",
    "key2": "value2"
}
```

Example:

```
student = {
    "name": "Mohsin",
    "age": 21,
    "department": "CSE"
}
print(student["name"])  # Output: Mohsin
```

Key Features:

- Keys must be **unique**
- Values can be **any data type**
- Defined using **curly braces {}**

Common Dictionary Methods:

Method	Description
`dict.keys()`	Returns all keys
`dict.values()`	Returns all values
`dict.items()`	Returns all key-value pairs
`dict.get("key")`	Gets value for key, returns `None` if absent
`dict["key"] = value`	Adds or updates a key-value pair
`dict.pop("key")`	Removes a key-value pair

dict.keys() Method

➔ The `.keys()` method returns a **view object** containing all the **keys** from a dictionary.

Syntax:

```
dictionary.keys()
```

Example code:

```
student = {
    "name": "Alice",
    "age": 21,
    "dept": "CSE"
}

print(student.keys())
```

Output

```
dict_keys(['name', 'age', 'dept'])
```

dict.values() Method

➔ The `.values()` method returns a **view object** containing all the **values** in the dictionary.

Syntax:

```
dictionary.values()
```

Example code:

```
student = {
    "name": "Alif",
    "age": 21,
    "dept": "CSE"
}

print(student.values())
```

Output:

```
dict_values(['Alif', 21, 'CSE'])
```

dict.get("key") Method

➔ The `.get()` method returns the **value** for the specified key. If the key is not found, it returns `None` (or a default value if provided), **without causing an error**.

Syntax:

```
dictionary.get("key", default_value?)
```

Example code:

```
student = {
    "name": "Nihan",
    "age": 21
}

print(student.get("name"))
print(student.get("dept"))
print(student.get("dept", "N/A"))
```

Output:

```
Nihan
None
N/A
```

Difference from `dict["key"]`:

- `student["dept"]` → ✗ Raises `KeyError` if key **doesn't exist**
- `student.get("dept")` → ☑ Returns `None` (or default)

dict.items() Method

➔ The `.items()` method returns a **view object** containing all **key-value pairs** as tuples.

Syntax:

```
dictionary.items()
```

Example code:

```
student = {
    "name": "Jam",
    "age": 21,
    "dept": "CSE"
}

print(student.items())
```

Output:

```
dict_items([('name', 'Jam'), ('age', 21), ('dept', 'CSE')])
```

dict["key"] = value

➔ The **dict["key"] = value** syntax is used to **add** a new key-value pair or **update** the value of an existing key in a dictionary.

Syntax:

```
dictionary["key"] = value
```

Example code:

```
student = {
    "name": "Mohsin",
    "age": 21
}
# Add new key-value
student["dept"] = "CSE"

# Update existing value
student["age"] = 22

print(student)
```

Output:

```
{'name': 'Mohsin', 'age': 22, 'dept': 'CSE'}
```

Use Case:

- If the key exists → updates the value
- If the key doesn't exist → adds the key-value pair

dict.pop("key") Method

➔ The `.pop("key")` method **removes** the specified key from the dictionary and **returns its value**.

Syntax:

```
dictionary.pop("key", default_value?)
```

Example code:	Output:

```
student = {
    "name": "Alice",
    "age": 21,
    "dept": "CSE"
}

removed_value = student.pop("age")
print(removed_value)
print(student)
```

```
21
{'name': 'Alice', 'dept': 'CSE'}
```

Nested Dictionary

➔ A **nested dictionary** is a dictionary **inside another dictionary**. It allows you to represent complex structured data.

Syntax:

```
nested_dict = {
    "key1": {
        "subkey1": value1,
        "subkey2": value2
    },
    "key2": {
        "subkey1": value1,
        "subkey2": value2
    }
}
```

Example code:	Output:
	Mohsin 22

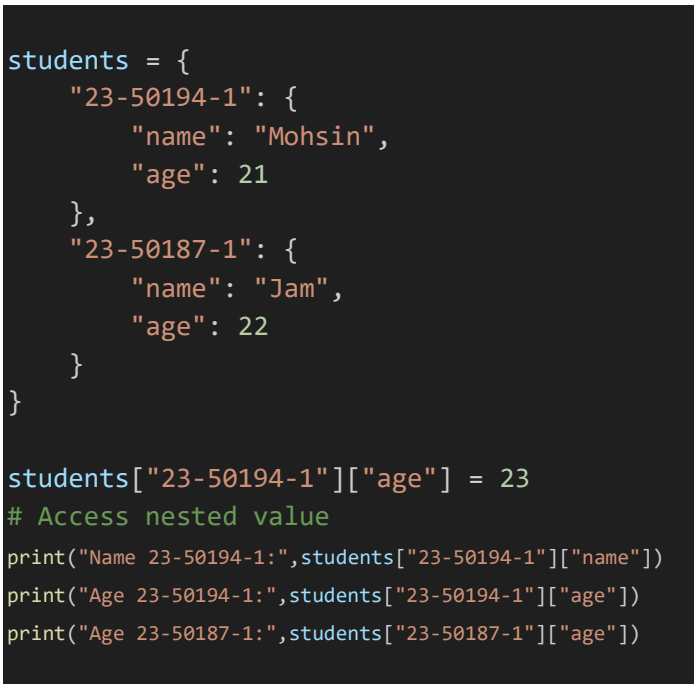
```
students = {
    "23-50194-1": {
        "name": "Mohsin",
        "age": 21
    },
    "23-50187-1": {
        "name": "Jam",
        "age": 22
    }
}

# Access nested value
print(students["23-50194-1"]["name"])
print(students["23-50187-1"]["age"])
```

Modify Nested Value

Syntax:

```
students["23-50194-1"]["age"] = 23
```

Example code:	Output:
	Name 23-50194-1: Mohsin Age 23-50194-1: 23 Age 23-50187-1: 22

```
students = {
    "23-50194-1": {
        "name": "Mohsin",
        "age": 21
    },
    "23-50187-1": {
        "name": "Jam",
        "age": 22
    }
}

students["23-50194-1"]["age"] = 23
# Access nested value
print("Name 23-50194-1:",students["23-50194-1"]["name"])
print("Age 23-50194-1:",students["23-50194-1"]["age"])
print("Age 23-50187-1:",students["23-50187-1"]["age"])
```

Add New Inner Dictionary

Syntax:

```
students["23-50200-1"] = {
    "name": "Nihan",
    "age": 23
}
```

Example code:

```
students = {
    "23-50194-1": {
        "name": "Mohsin",
        "age": 24
    },
    "23-50187-1": {
        "name": "Jam",
        "age": 22
    }
}

students["23-50200-1"] = {
    "name": "Nihan",
    "age": 23
}

students["23-50186-1"] = {
    "name": "Alif",
    "age": 21
}

# Access nested value
print("23-50186-1:",students["23-50186-1"])
print("23-50187-1:",students["23-50187-1"])
print("23-50194-1:",students["23-50194-1"])
print("23-50200-1:",students["23-50200-1"])
```

Output:

```
23-50186-1: {'name': 'Alif', 'age': 21}
23-50187-1: {'name': 'Jam', 'age': 22}
23-50194-1: {'name': 'Mohsin', 'age': 24}
23-50200-1: {'name': 'Nihan', 'age': 23}
```

Practice Problem

Problem1: Store the following word meanings in a **Python dictionary**, where each word can have **multiple meanings**.

`table → "a piece of furniture","list of facts & figures"`

`cat → "a small animal"`

Code:

```
dictionary = {
    "table": ["a piece of furniture", "list of facts & figures"],
    "cat": ["a small animal"]
}

# Printing the dictionary
print(dictionary)
print()

print("Table:",dictionary["table"])
print("Cat:",dictionary["table"])
```

Output:

```
{'table': ['a piece of furniture', 'list of facts & figures'], 'cat': ['a small animal']}

Table: ['a piece of furniture', 'list of facts & figures']
Cat: ['a piece of furniture', 'list of facts & figures']
```

Problem2: Write a Python program to **enter marks of 3 subjects** from the user and **store them in a dictionary**. Start with an **empty dictionary** and add subject–marks **one by one** using the subject name as the key and the marks as the value.

Code:

```
marks_dict = {}

for i in range(3):
    subject = input("Enter subject name: ")
    marks = int(input(f"Enter marks for {subject}: "))
    marks_dict[subject] = marks

print("Marks Dictionary:", marks_dict)
```

Output:

```
Enter subject name: Physics
Enter marks for Physics: 99
Enter subject name: English
Enter marks for English: 78
Enter subject name: Math
Enter marks for Math: 100
Marks Dictionary: {'Physics': 99, 'English': 78, 'Math': 100}
```


SET

Set

➔ A **Set** is an **unordered**, **unindexed** collection of **unique elements**. It is useful when we want to store items without duplicates.

Syntax:	Example:
`my_set = {1, 2, 3}`	`numbers = {1, 2, 3, 2, 1}` `print(numbers)` `# Output: {1, 2, 3} → duplicates are removed`

Common Set Methods:

Operation	Code Example	Result
Add item	`my_set.add(4)`	Adds 4 to the set
Remove item	`my_set.remove(2)`	Removes 2, raises error if not found
Discard item	`my_set.discard(2)`	Removes 2, **no error** if not found
Pop random item	`my_set.pop()`	Removes a random element
Clear set	`my_set.clear()`	Removes all items
Union	`a.union(b)` or `a | b`	Common, Uncommon items
Intersection	`a.intersection(b)` or `a & b`	Common items
Difference	`a.difference(b)` or `a - b`	Items in `a` but not in `b`

Add item

➔ To add a new element to a set, use the `.add()` method.

Syntax:

```
set_name.add(item)
```

Example code:	Output:
`fruits = {"apple", "banana"}` `fruits.add("cherry")` `print(fruits)`	`{'apple', 'banana', 'cherry'}` `{'cherry', 'apple', 'banana'}`

Notes:

- If the item already exists, it **won't be added again** (sets store unique values only).
- Order is **not guaranteed** when printing a set.

Remove item

➔ Python provides **two main ways** to remove items from a set: `.remove()` and `.discard()`.

Syntax: Using `.remove()`

```
set_name.remove (item)
```

Example code:

```
fruits = {"apple", "banana", "cherry"}
fruits.remove("banana")

print(fruits)
```

Output:

```
{'cherry', 'apple'}
```

Notes:

- Removes a specific element.
- Raises an **error** if the element is not found.

Discard item

Syntax: Using `.discard()`

```
set_name. discard(item)
```

Example code:

```
fruits = {"apple", "banana", "cherry"}
fruits.discard("banana")
fruits.discard("orange")  # No error!

print(fruits)
```

Output:

```
{'cherry', 'apple'}
```

Notes:

- Also removes a specific element.
- **Does NOT raise an error** if the item is missing.

Pop random item

➔ The `.pop()` method **removes and returns a random element** from a set because sets are unordered.

Syntax:

```
set_name.pop()
```

Example code:

```
colors = {"red", "green", "blue"}
removed_item = colors.pop()

print("Removed:", removed_item)
print("Remaining:", colors)
```

Output:

```
Removed: blue
Remaining: {'green', 'red'}
```

Notes: Raises `KeyError` if the set is **empty**.

Union sets

➔ The **union** of two or more sets returns a new set containing **all unique elements** from the sets involved.

Syntax:

```
set1.union(set2, set3, ...)
# or
set4 = set1 | set2 | set3
```

Example code:

```
a = {"apple", "banana"}
b = {"banana", "cherry"}
c = {"Mango"}
d = a.union(b)
e = a | b | c

print(d)
print(e)
```

Output:

```
{'cherry', 'banana', 'apple'}
{'cherry', 'banana', 'Mango', 'apple'}
```

Notes:

- Duplicates are automatically removed.
- Original sets remain unchanged.

Intersection sets

➔ The **intersection** of sets returns a new set containing **only elements common** to all sets.

Syntax:

```
set1.intersection(set2, set3, ...)
# or
set4 = set1 & set2 & set3
```

Example code:

```
a = {"apple", "banana", "cherry"}
b = {"banana", "cherry", "grape"}
c = {"Mango", "cherry", "grape"}

d = a.intersection(b)
e = a & b & c

print(d)
print(e)
```

Output:

```
{'cherry', 'banana'}
{'cherry'}
```

Notes:

- Does **not modify** the original sets.
- Result contains **only shared elements**.

Clear set

➔ To remove **all elements** from a set, use the `.clear()` method.

Syntax:

```
set_name.clear()
```

Example code:

```
fruits = {"apple", "banana", "cherry"}
fruits.clear()

print(fruits)
```

Output:

```
set()
```

Intersection sets

➔ The **difference** returns a new set containing elements that are **only in the first set** and **not in the others**.

Syntax:

```
set1.difference(set2, set3, ...)
# or
set4 = set1 - set2 - set3
```

Example code:

```
a = {"apple", "banana", "cherry"}
b = {"banana", "grape"}
c = {"Mango", "cherry", "grape"}

d = a.difference(b)
e = a - b - c

print(d)
print(e)
```

Output:

```
{'apple', 'cherry'}
{'apple'}
```

Notes:

- The original sets are not changed.
- Order doesn't matter in sets, but the **set from which we subtract comes first.**

Practice Problem

Problem1: You are given a list of subjects chosen by students. Each **unique subject** needs **one classroom**. **Write a Python program** to calculate **how many classrooms** are needed. ["python", "java", "C++", "python", "javascript", "java", "python", "java", "C++", "C"]

Code:

```
subjects = ["python", "java", "C++", "python",
"javascript","java", "python", "java", "C++", "C"]

unique_subjects = set(subjects)
classrooms_needed = len(unique_subjects)

print("Classrooms needed:", classrooms_needed)
```

Output:

```
Classrooms needed: 5
```

Problem2: Figure out a way to **store both 9 and 9.0 as separate values** in a Python set.

Usually, `9` (int) and `9.0` (float) are considered **equal in value**, so a set will treat them as the same.

Example code:

```
set = {9, 9.0}
print(set)
```

Output:

```
{9}
```

Why this happens: Because,

```
9 == 9.0  # True
type(9) != type(9.0)  # True, but doesn't matter in set
```

Solution: Using string and int

Example code:

```
my_set = {9, "9.0"}
print(my_set)
```

Output:

```
{9, '9.0'}
```

Better Solution: Store with type info using tuple

Example code:

```
my_set = {(9, int), (9.0, float)}
print(my_set)
```

Output:

```
{(9, <class 'int'>), (9.0, <class 'float'>)}
```

Functions

➔ A **function** is a block of reusable code that performs a specific task.

Define a Function:	Call a Function:
`def function_name():` `    # code block`	`function_name()`
Example code:	**Output:**
`def greet():` `    print("Hello, welcome to Python!")` `greet()`	`Hello, welcome to Python!`
Notes: Use `def` to define a function.	

Function with Parameters

Define a Function:	Call a Function:
`def function_name(Parameters):` `    # code block`	`function_name(Parameters)`
Example code:	**Output:**
`def greet(name):` `    print("Hello", name)` `greet("Mohsin")` `greet("Jam")` `greet("Alif")` `greet("Nihan")`	`Hello Mohsin` `Hello Jam` `Hello Alif` `Hello Nihan`
Notes: Functions help make code **modular** and **reusable**.	

Function with Return Value:

Example code:	Output:
``def add(a, b): return a + b result = add(3, 5) print(result)```	**8**

Notes: `return` sends a value back to the caller.

Practice Problem

Problem1: Write a function in Python that takes a list as a parameter and **prints its length**.

Code:	Output:
``def print_list_length(lst): print("Length of the list:", len(lst)) my_list = [10, 20, 30, 40] print_list_length(my_list)```	**Length of the list: 4**

Problem2: Write a function that takes a list as a parameter and **prints all its elements** in a single line.

Code:	Output:
``def print_elements_single_line(lst): for item in lst: print(item, end=' ') print() my_list = [1, 2, 3, 4, 5] print_elements_single_line(my_list)```	**1 2 3 4 5**

Problem3: Write a function in Python that **takes a number** `n` as a parameter and **prints its factorial.**

Code:	Output:
`` def print_factorial(n): factorial = 1 for i in range(1, n + 1): factorial *= i print("Factorial of", n, "is", factorial) num = int(input("Enter a Number: ")) print_factorial(num) ```	Enter a Number: 5 Factorial of 5 is 120

Problem4: Write a function in Python that **takes an amount in USD** and **converts it to BDT.** Assume 1 USD = 117 BDT.

Code:	Output:
`` def convert_usd_to_bdt(usd): bdt = usd * 117 print(f"{usd} USD = {bdt} BDT") amount = int(input("Enter Amount: ")) convert_usd_to_bdt(amount) ```	Enter Amount: 50 50 USD = 5850 BDT


RECURSION

Recursion

➔ **Recursion** is a method of solving problems where a function calls itself.

Structure of Recursive Function:

```
def function_name():
    if base_condition:
        return result
    else:
        return function_name()  # recursive call
```

Example: Factorial Using Recursion

```
def factorial(n):
    if n == 0 or n == 1:
        return 1  # base case
    else:
        return n * factorial(n - 1)  # recursive case

print(factorial(5))  # Output: 120
```

Output:

```
120
```

Key Points:

- Every recursive function must have a **base case** to stop the recursion.
- Without a base case, it will lead to **infinite recursion** (and crash with an error).
- Useful for problems like:
  - Factorial
  - Fibonacci
  - Tree traversal
  - Tower of Hanoi

Practice Problem

Problem1: Write a recursive function to calculate the sum of the first n natural numbers.

Code:

```
def sum_natural(n):
    if n == 1:
        return 1
    else:
        return n + sum_natural(n - 1)
num = int(input("Enter a Number: "))
print("Sum: ", sum_natural(num))
```

Output:

```
Enter a Number: 5
Sum:  15
```

OBJECT-ORIENTED PROGRAMMING (OOP)

Object-Oriented Programming (OOP)

➔ OOP is a programming paradigm that organizes code into **classes** and **objects**.

Key Concepts of OOP:

Concept	Description
Class	Blueprint for creating objects
Object	Instance of a class
Attribute	Variables inside a class
Method	Functions inside a class
Constructor	`__init__()` method called when object is created
Self	Refers to the current object

Basic OOP Example:

```
class Student:
    def __init__(self, name, age):
        self.name = name        # attribute
        self.age = age

    def greet(self):            # method
        print("Hello, my name is", self.name)
        print("And my age is", self.age)

# Create object
s1 = Student("Mohsin", 22)

# Access method
s1.greet()
```

Output:

```
Hello, my name is Mohsin
And my age is 22
```

🧠 OOP Features:

Feature	Example
Encapsulation	Hiding data (using private variables)
Inheritance	One class inherits from another
Polymorphism	Same method behaves differently in different classes
Abstraction	Hiding complex details using abstract classes (via `abc` module)

Encapsulation

➔ **Encapsulation** is the OOP principle of **hiding internal details** and protecting object data from direct modification.

How It Works:

- ⇨ **Private attributes**: Defined using **underscore _** or **double underscore __**.
- ⇨ Accessed using **getter and setter methods**.

Example code:

```
class Student:
    def __init__(self, name, age):
        self.__name = name   # private attribute
        self.__age = age

    def get_name(self):      # getter method
        return self.__name


    def set_name(self, name):  # setter method
        self.__name = name


# Create object
s1 = Student("Mohsin", 22)

print(s1.get_name())       # Output: Mohsin

s1.set_name("Alif")
print(s1.get_name())       # Output: Alif
```

Output:

```
Mohsin
Alif
```

Why Use Encapsulation?

- Prevents unauthorized access.
- Makes the class modular and secure.
- Allows validation before changing data.

Inheritance

➔ **Inheritance** allows a class (child) to **acquire properties and methods** from another class (parent), promoting code reusability.

Syntax:

```
class Parent:
    # Parent class content

class Child(Parent):
    # Child class inherits Parent
```

Example code:

```
class A:
    def methodA(self):
        print("A method")

class B:
    def methodB(self):
        print("B method")

class C(A, B):
    pass

obj = C()
obj.methodA()
obj.methodB()
```

Output:

```
A method
B method
```

🧠 Types of Inheritance in Python:

Type	Description
Single Inheritance	One child class inherits one parent
Multiple Inheritance	One class inherits from multiple parents
Multilevel Inheritance	Inheritance across multiple levels
Hierarchical Inheritance	Multiple child classes from one parent
Hybrid Inheritance	Combination of multiple types

super() Keyword

➔ The `super()` keyword is used to **access methods and constructors of a parent class** from the child class.

Why Use super()?

- ⇨ To reuse the parent class's **methods or __init__() constructor**.
- ⇨ Avoids hardcoding the parent class name.
- ⇨ Useful in **inheritance** (especially with multiple inheritance).

Example code:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
        print(f"Person: {self.name}, Age: {self.age}")

class Student(Person):
    def __init__(self, name, age, student_id):
        super().__init__(name, age)   # Call parent constructor with parameters
        self.student_id = student_id
        print(f"Student ID: {self.student_id}")

# Create object
s = Student("Mohsin", 20, "23-50194-1")
```

Output:

```
Person: Mohsin, Age: 20
Student ID: 23-50194-1
```

Summary:

- super() helps in calling **parent class methods or constructors.**
- Makes code **clean and maintainable**.
- Essential **in constructor chaining** and **method overriding.**

Single Inheritance

➔ **Single Inheritance** is when a **child class inherits** from **only one parent class**.

Syntax:

```
class Parent:
    # parent class content

class Child(Parent):
    # inherits from Parent
```

Example code:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age


class Student(Person):
    def __init__(self, name, age, student_id):
        super().__init__(name, age)
        self.student_id = student_id


# Create object
s1 = Student("Mohsin", 22, "23-50194-1")
print(s1.name)
print(s1.age)
print(s1.student_id)

s2 = Student("Jam", 20, "23-50187-1")
print(s2.name)
print(s2.age)
print(s2.student_id)
```

Output:

```
Mohsin
22
23-50194-1

Jam
20
23-50187-1
```

Key Points:

- Promotes code reusability
- Only one parent class
- The child can use all non-private members of the parent

Multiple Inheritance

➔ **Multiple Inheritance** is when a child class inherits from **more than one parent class**.

Syntax:

```
class Parent1:
    # code

class Parent2:
    # code

class Child(Parent1, Parent2):
    # inherits from both Parent1 and Parent2
```

Example code:

```
class Father:
    def skill(self):
        print("Father: Farming")

class Mother:
    def talent(self):
        print("Mother: Cooking")

class Child(Father, Mother):   # Multiple Inheritance
    def hobby(self):
        print("Child: Drawing")

c = Child()
c.skill()     # Inherited from Father
c.talent()    # Inherited from Mother
c.hobby()     # Own method
```

Output:

```
Father: Farming
Mother: Cooking
Child: Drawing
```

Key Points:

- Child class gets properties from **multiple parent classes**.
- Python uses **Method Resolution Order (MRO)** to decide which method to use if there's a conflict.
- Helps combine features from multiple sources.

Multilevel Inheritance

➔ **Multilevel Inheritance** means a class is derived from a **class that is already derived from another class**.
In simple words: **Child → Parent → Grandparent**

Syntax:

```
class Grandparent:
    # code

class Parent(Grandparent):
    # code

class Child(Parent):
    # inherits from Parent (and indirectly Grandparent)
```

Example code:

```
class Grandfather:
    def house(self):
        print("Grandfather: Owns a house")

class Father(Grandfather):
    def car(self):
        print("Father: Owns a car")

class Son(Father):
    def laptop(self):
        print("Son: Owns a laptop")

s = Son()
s.house()    # From Grandfather
s.car()      # From Father
s.laptop()   # From Son
```

Output:

```
Grandfather: Owns a house
Father: Owns a car
Son: Owns a laptop
```

Key Points:

- Inherits features in **a chain** (like 3 generations).
- Child has access to all **public members of Parent** and **Grandparent.**
- Helps build **stepwise structure** in OOP design.

Abstraction

➔ **Abstraction** means **hiding the internal implementation** and **showing only essential features** to the user.

In Python, abstraction is achieved using:

- **Abstract classes**
- **Abstract methods**

👉 Done using the `abc` module (`abc` = Abstract Base Class)

Syntax:

```
from abc import ABC, abstractmethod

class MyAbstractClass(ABC):
    @abstractmethod
    def abstract_method(self):
        pass
```

Example code:

```
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def sound(self):
        pass

class Dog(Animal):
    def sound(self):
        print("Dog: Woof!")

class Cat(Animal):
    def sound(self):
        print("Cat: Meow!")

# a = Animal()  ❌ Cannot create object of abstract class
d = Dog()
d.sound()

c = Cat()
c.sound()
```

Output:

```
Dog: Woof!
Cat: Meow!
```

Key Points:

- Abstract class cannot be instantiated.
- All child classes must override the abstract methods.
- Encourages design consistency.
- Promotes security and cleaner code.

Polymorphism

➔ **Polymorphism** means **"many forms"** — the ability of different classes to respond to the same method in different ways.

It allows:

- ⇨ Same method name to behave differently on different classes.
- ⇨ Flexibility in code design.

Example code:

```
class Cat:
    def sound(self):
        print("Cat: Meow")

class Dog:
    def sound(self):
        print("Dog: Woof")

# Polymorphism
def make_sound(animal):
    animal.sound()

c = Cat()
d = Dog()

make_sound(c)  # Cat: Meow
make_sound(d)  # Dog: Woof
```

Output:

```
Cat: Meow
Dog: Woof
```

Key Points:

- Allows same **interface, different behavior.**
- Achieved via **method overriding** or **common method** names across classes.
- Increases **code reusability and scalability**.

Static Methods

➔ A **static method** is a method that **does not take** `self` **or** `cls` as the first argument.
It does **not depend on class or instance state**.

Why Use Static Methods?

⇨ Used when some functionality is **logically related to the class**, but **doesn't need access to class or instance variables**.

Defined using:

```
@staticmethod
```

Example code:

```
class MathTools:
    @staticmethod
    def add(x, y):
        return x + y

    @staticmethod
    def multiply(x, y):
        return x * y

# Accessing without creating object
print(MathTools.add(5, 3))       # 8
print(MathTools.multiply(4, 6))  # 24

# Can also access via object
tool = MathTools()
print(tool.add(2, 2))            # 4
```

Output:

```
8
24
4
```

Key Points:

- Decorated with **@staticmethod**
- No **self** or **cls** parameter
- Can be called using class name or object
- Ideal for **utility/helper functions** related to a class

Class Methods

➔ A **class method** is a method that **works with the class itself**, not the instance. It's defined using the `@classmethod` decorator, and it takes `cls` (class) as the first parameter.

Syntax:

```
@classmethod
def method_name(cls):
    # code
```

Example code:

```
class Student:
    University_name = "AIUB"

    def __init__(self, name):
        self.name = name

    @classmethod
    def change_University(cls, new_name):
        cls.University_name = new_name

# Before changing
print(Student.University_name)  # AIUB

# Using class method
Student.change_University("BUET")

# After changing
print(Student.University_name)  # BUET
```

Output:

```
AIUB
BUET
```

Key Points:

- Uses `@classmethod`
- First argument is `cls` (refers to the class)
- Can **access or modify class variables**
- Can be called using **class name or instance**

Practice Problem

Problem1: Write a Python program to define a class named `BankAccount` that models a simple bank account. The class should have the following:

- **Attributes**:
  - `name` (the name of the account holder)
  - `balance` (the current account balance)
- **Methods**:
  - `deposit(amount)` – adds the specified amount to the balance
  - `withdraw(amount)` – deducts the amount from the balance if sufficient funds are available, otherwise display an error
  - `display_balance()` – displays the current balance of the account

Create an object of the class and demonstrate the use of each method.

Code:	Output:

```python
def sum_natural(n):
class BankAccount:
    def __init__(self, name, balance):
        self.name = name
        self.balance = balance

    def deposit(self, amount):
        self.balance += amount
        print(f"{amount} BDT deposited successfully.")

    def withdraw(self, amount):
        if amount <= self.balance:
            self.balance -= amount
            print(f"{amount} BDT withdrawn successfully.")
        else:
            print("Insufficient balance!")

    def display_balance(self):
        print(f"Account Holder: {self.name}")
        print(f"Current Balance: {self.balance} BDT")


# Example usage
account1 = BankAccount("Mohsin", 10000)

account1.display_balance()
account1.deposit(2000)
account1.withdraw(5000)
account1.display_balance()
```

```
Account Holder: Mohsin
Current Balance: 10000 BDT
2000 BDT deposited successfully.
5000 BDT withdrawn successfully.
Account Holder: Mohsin
Current Balance: 7000 BDT
```

Problem2: Write a Python program to implement **inheritance** using object-oriented programming.

- Create a base class named `Shape` with a method `area()` that returns 0.
- Derive the following subclasses from `Shape`:
  - `Square` – with a constructor to accept `side` and override the `area()` method.
  - `Circle` – with a constructor to accept `radius` and override the `area()` method.
  - `Triangle` – with a constructor to accept `base` and `height` and override the `area()` method.

Demonstrate polymorphism by calling the `area()` method for each shape.

Code:	Output:

```python
import math

class Shape:
    def area(self):
        return 0

class Square(Shape):
    def __init__(self, side):
        self.side = side

    def area(self):
        return self.side * self.side

class Triangle(Shape):
    def __init__(self, base, height):
        self.base = base
        self.height = height

    def area(self):
        return 0.5 * self.base * self.height


# Create a list of different shapes
shapes = [Square(4), Triangle(6, 5)]

# Loop through each shape and print its name and area
print("Shape Areas:\n" + "-"*20)
for shape in shapes:
    name = shape.__class__.__name__  # Get the class name
    area = shape.area()              # Call the area method
    print(f"{name} area: {area:.2f}")
```

```
Shape Areas:
--------------------
Square area: 16.00
Triangle area: 15.00
```

Problem3: Write a Python program to demonstrate the concept of **inheritance** and the use of the `super()` function.

- Create a base class `Person` with attributes `name` and `age`.
- Create a derived class `Employee` that inherits from `Person` and adds `salary` and `department` attributes.
- Use the `super()` function to initialize the base class attributes from the derived class constructor.
- Display the details of the employee.

Code:	Output:

```python
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def display(self):
        print(f"Name: {self.name}")
        print(f"Age: {self.age}")

class Employee(Person):
    def __init__(self, name, age, salary, department):
        super().__init__(name, age)
        self.salary = salary
        self.department = department

    def display(self):
        super().display()
        print(f"Salary: {self.salary}")
        print(f"Department: {self.department}")

# Example usage
emp1 = Employee("Alif Hasan Khan", 25, 50000, "IT")
emp1.display()

print("-"*20)
emp2 = Employee("Mohsin Ibna Hossain", 27, 70000, "IT")
emp2.display()
```

```
Age: 25
Salary: 50000
Department: IT
--------------------
Name: Mohsin Ibna Hossain
Age: 27
Salary: 70000
Department: IT
```

Problem4: Design a Movie Ticket Booking System using OOP in Python:

- Create a base class `Ticket` with attributes: `_price` (private), `seat_no`.
- Include methods to **get the price**, **set the price**, and **display ticket info**.
- Create a derived class `StudentTicket` that inherits from `Ticket` and applies a discount to the ticket price.
- Demonstrate **encapsulation** (via private attributes and getter/setter methods) and use of **`super()`** to initialize base class attributes.

Code:	Output:

```python
class Ticket:
    def __init__(self, price, seat_no):
        self._price = price     # Encapsulated (private) attribute
        self.seat_no = seat_no

    def get_price(self):
        return self._price

    def set_price(self, price):
        if price >= 0:
            self._price = price
        else:
            print("Invalid price!")

    def display_info(self):
        print(f"Seat No: {self.seat_no}")
        print(f"Ticket Price: BDT {self._price}")

class StudentTicket(Ticket):
    def __init__(self, price, seat_no, discount):
        super().__init__(price, seat_no)
        self.discount = discount

    def apply_discount(self):
        discounted_price = self.get_price() * (1 -
self.discount / 100)
        self.set_price(discounted_price)

    def display_info(self):
        self.apply_discount()
        print("Student Ticket Info:")
        super().display_info()

ticket1 = StudentTicket(500, "A12", 20)
ticket1.display_info()
```

```
Student Ticket Info:
Seat No: A12
Ticket Price: BDT 400.0
```

FILE I/O

File I/O

➔ File I/O lets your Python program **read from** or **write to** files.

Key keyword of File I/O:

Mode	Description
`'r'`	Read (default)
`'w'`	Write (overwrite)
`'a'`	Append
`'x'`	Create
`'b'`	Binary mode
`'t'`	Text mode (default)
`'r+'`	Read and write

Open, read & close File:

Reading a File:

```
file = open("demo.txt", "r")
print(file.read())
file.close()
```

Writing to a File:

```
file = open("demo.txt", "w")
file.write("Hello, World!")
file.close()
```

Appending to a File:

```
file = open("demo.txt", "a")
file.write("\nAppended line.")
file.close()
```

Best Practice: Use `with` statement, automatically closes the file:

```
with open("demo.txt", "r") as file:
    data = file.read()
    print(data)
```

Common Methods:

Method	Description
`read()`	Reads entire content
`readline()`	Reads a single line
`readlines()`	Returns list of lines
`write(text)`	Writes string to file
`writelines(list)`	Writes list of strings to file
`close()`	Closes the file

Writing a File

➔ Overwrites the file **if it exists**. **Creates a new** file **if it doesn't.**

Syntax:

Method1: Using `"w"` mode (Write)

```
with open("example.txt", "w") as file:
    file.write("This is the first line.\n")
    file.write("This is the second line.\n")
```

Notes:

- **\n** adds a new line.
- Using with is safer as it auto-closes the file.

Reading a File

Syntax:

Method1: Using `open()` and `read()`

```
file = open("example.txt", "r")
content = file.read()
print(content)
file.close()

#reads entire file as a single string.
```

Method2: Using `with` (Best Practice)

```
with open("example.txt", "r") as file:
    content = file.read()
    print(content)

#Automatically closes the file after reading.
```

Notes:

- File must exist, or you'll get a **FileNotFoundError.**
- Always close the file using **file.close()** unless you use **with.**

Deleting a File

➔ To delete a file, you use the `os` module.

Syntax:

Step1: Import `os`

```
import os
```

Step2: Delete a File using `os.remove()`

```
import os

os.remove("example.txt")
```

Best Practice: Check if File Exists First

```
import os

if os.path.exists("example.txt"):
    os.remove("example.txt")
    print("File deleted successfully.")
else:
    print("File does not exist.")
```

Notes: If the file doesn't exist and you don't check, it will raise a `FileNotFoundError`.

Practice Problem

Problem1: Write a Python program to perform the following operations using **File I/O**:

1. Create a new file named `practice.txt` and write the following content in it:

```
Hi everyone
we are learning File I/O
using Java.
I like programming in Java.
```

2. Write a function to **replace all occurrences** of the word `"Java"` with `"Python"` in the file.
3. Search whether the word `"learning"` exists in the file or not, and print the result.

Code:	Output:

```
# Step 1: Create and write data to the file
with open("practice.txt", "w") as f:
    f.write("Hi everyone\n")
    f.write("we are learning File I/O\n")
    f.write("using Java.\n")
    f.write("I like programming in Java.\n")

# Step 2: Function to replace 'Java' with 'Python'
def replace_java_with_python():
    with open("practice.txt", "r") as f:
        data = f.read()

    data = data.replace("Java", "Python")

    with open("practice.txt", "w") as f:
        f.write(data)
```

```
The word 'learning' exists in the file.
```

```python
# Step 3: Function to search for 'learning'
def search_learning():
    with open("practice.txt", "r") as f:
        content = f.read()
        if "learning" in content:
            print("The word 'learning' exists in the file.")
        else:
            print("The word 'learning' does not exist in the file.")

# Run the functions
replace_java_with_python()
search_learning()
```

Problem2:

Write a Python function to **find the line number** where the word `"learning"` occurs **first** in a given file.

- Return the **line number** (starting from 1).
- Return **-1** if the word does not exist in the file.

Code:	Output:

```python
def find_learning_line(filename):
    with open(filename, "r") as file:
        for line_number, line in enumerate(file, start=1):
            if "learning" in line:
                return line_number
    return -1

# Example usage:
filename = "practice.txt"
result = find_learning_line(filename)
print("Line number:", result)
```

```
Line number: 2
```

IMPORT IN PYTHON

import in Python

➔ Importing an Entire Module

Importing an Entire Module:

```
import math
print(math.sqrt(16))
# Output: 4.0
```

Import with Alias:

```
import math as m
print(m.pi)
# Output: 3.141592653589793
```

Import Specific Function or Variable:

```
from math import sqrt, pi
print(sqrt(25))  # Output: 5.0
print(pi)
# Output: 3.141592653589793
```

Import All Functions (Not Recommended):

```
from math import *
print(sin(90))
```

Common Modules:

Module	Purpose
`math`	Math operations
`random`	Random number generation
`datetime`	Date and time handling
`os`	File system operations
`sys`	System-specific functions

random Modules

random integer between **a** and **b** (inclusive):

```
import random
print(random.randint(1, 10))
# Output: Any integer from 1 to 10
```

random element from a **list**, **tuple**, or **string**:

```
import random
print(random.uniform(1, 5))
# Example: 2.39485
```

Shuffles the list **in place** (modifies original list):

```
import random
cards = [1, 2, 3, 4, 5]
random.shuffle(cards)
print(cards)
# Output: e.g. [3, 5, 1, 2, 4]
```

random float between **a** and **b**:

```
import random
colors = ["red", "green", "blue"]
print(random.choice(colors))
 # Output: "green", etc.
```