# 1. Identifying Agents and Environments – List five real-world AI agents with their environment, sensors, actuators and goals.

List of 5 real-world AI agents, along with their environments, sensors, actuators, and goals

### 1. Autonomous Car (e.g., Tesla Autopilot)

- **Environment:** Roads, traffic, pedestrians, weather conditions
- **Sensors:** Cameras, LiDAR, radar, GPS, ultrasonic sensors
- **Actuators:** Steering, throttle, brake, signal indicators
- **Goal:** Safely navigate to a destination while obeying traffic rules and avoiding collisions

### 2. Smart Home Thermostat (e.g., Nest)

- **Environment:** Indoor home environment (temperature, humidity, occupancy)
- **Sensors:** Temperature sensor, humidity sensor, motion detector, Wi-Fi module
- **Actuators:** HVAC controls (heater/cooler on/off)
- **Goal:** Maintain comfortable temperature efficiently, save energy based on occupancy

### 3. Voice Assistant (e.g., Amazon Alexa)

- **Environment:** Human-inhabited space (home, office)
- **Sensors:** Microphones, sometimes cameras
- **Actuators:** Audio speakers, smart device signals (e.g., lights, alarms)
- **Goal:** Understand and fulfill user voice commands (e.g., play music, control devices, answer questions)

### 4. Warehouse Robot (e.g., Amazon Kiva System)

- **Environment:** Warehouse floor with shelves and other robots
- **Sensors:** Cameras, RFID readers, proximity sensors, wheel encoders
- **Actuators:** Wheels, lifting arms, directional motors
- **Goal:** Pick and deliver inventory items to specific locations efficiently and safely

### 5. Email Spam Filter

- **Environment:** Incoming email stream on a mail server
- **Sensors:** Email text, metadata (sender, subject, links), user feedback
- **Actuators:** Classifier label (spam/ham), email routing (inbox or spam folder)
- **Goal:** Classify and filter out unwanted/spam messages with high accuracy

### 6. Self-Checkout Kiosk (e.g., in supermarkets)

- **Environment:** Store checkout area, interacting with products and customers
- **Sensors:** Barcode scanner, weight sensors, cameras, touchscreen input
- **Actuators:** Display screen, speaker, payment terminal, receipt printer
- **Goal:** Accurately scan items, calculate total, handle payment, and reduce checkout time without human cashiers

### 7. Drone Delivery System (e.g., Zipline or Amazon Prime Air)

- **Environment:** Outdoor airspace, wind/weather conditions, GPS-based location data
- **Sensors:** GPS, altimeter, gyroscope, cameras, proximity sensors
- **Actuators:** Propellers, navigation control surfaces, package drop mechanism
- **Goal:** Deliver packages autonomously to precise locations safely and efficiently

### 8. Autonomous Vacuum Cleaner (e.g., Roomba)

- **Environment:** Home floors with furniture, pets, and humans
- **Sensors:** Infrared sensors, bump sensors, cliff sensors, gyroscope, cameras
- **Actuators:** Wheels, vacuum motor, rotating brushes, sound system
- **Goal:** Navigate and clean floors thoroughly while avoiding obstacles and stairs

### 9. Facial Recognition System (e.g., airport security systems)

- **Environment:** Public access points like airports, office buildings, or phones
- **Sensors:** Camera input (static or video), sometimes infrared sensors
- **Actuators:** Access control system (e.g., gates), alerting systems
- **Goal:** Identify or verify individuals based on facial features for security or personalization

### 10. Stock Trading Bot (e.g., algorithmic trading systems)

- **Environment:** Digital financial markets and trading platforms
- **Sensors:** Real-time financial data, market news feeds, economic indicators
- **Actuators:** Trade execution APIs (buy/sell orders), portfolio rebalancing tools
- **Goal:** Maximize profit or minimize risk through rapid and data-driven trading decisions

**3. Demonstrate basic problem-solving using Breadth-First Search on a simple grid.**

**Program/Code:**

```
from collections import deque

grid = [

    ['.', '.', '.', '#', '.', '.'],

    ['#', '#', '.', '#', '.', '#'],

    ['.', '.', '.', '.', '.', '.'],

    ['.', '#', '#', '#', '#', '.'],

    ['.', '.', '.', '.', '#', '.']

]

start = (0, 0)
goal = (4, 5)
directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

def bfs(grid, start, goal):
    rows, cols = len(grid), len(grid[0])
    visited = set()
    queue = deque()
    queue.append((start, [start]))

    while queue:

        (x, y), path = queue.popleft()

        if (x, y) == goal:

            return path

        for dx, dy in directions:

            nx, ny = x + dx, y + dy
```

```
            if (0 <= nx < rows and 0 <= ny < cols and
                grid[nx][ny] != '#' and (nx, ny) not in visited):
                visited.add((nx, ny))
                queue.append(((nx, ny), path + [(nx, ny)]))

    return None

path = bfs(grid, start, goal)

if path:
    print("Shortest Path:")
    for step in path:
        print(step)
else:
    print("No path found.")
```

**Output:**

```
Shortest Path:
(0, 0)
(0, 1)
(0, 2)
(1, 2)
(2, 2)
(2, 3)
(2, 4)
(2, 5)
(3, 5)
(4, 5)
```

## 4. Implement Depth-First Search (DFS) on a small graph.

**Program/Code:**

```
graph = {
        'A': ['B', 'C'],
        'B': ['D', 'E'],
        'C': ['F'],
        'D': [],
        'E': ['F'],
        'F': []
    }

def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()
    visited.add(start)
    print(start)

    for neighbor in graph[start]:
        if neighbor not in visited:
            dfs(graph, neighbor, visited)
# Start DFS from node 'A'
print("DFS Traversal Starting from Node A:")
dfs(graph, 'A')
```

**Output:**

DFS Traversal Starting from Node A:

A

B

D

E

F

C

## 7. Apply the A* Search algorithm to find the shortest path in a 4x4 grid.

**Program/Code:**

```
import heapq

    grid = [
        ['.', '.', '.', '.'],
        ['.', '#', '#', '.'],
        ['.', '.', '.', '.'],
        ['#', '.', '#', '.']
    ]

    start = (0, 0)
    goal = (3, 3)

    # Directions: Up, Down, Left, Right
    directions = [(-1,0), (1,0), (0,-1), (0,1)]

    # Heuristic: Manhattan distance
    def heuristic(a, b):
        return abs(a[0] - b[0]) + abs(a[1] - b[1])


    def a_star(grid, start, goal):
        rows, cols = len(grid), len(grid[0])
        open_set = []
        heapq.heappush(open_set, (0 + heuristic(start, goal), 0, start, [start]))
        visited = set()

        while open_set:
            f, g, current, path = heapq.heappop(open_set)
            if current == goal:
                return path
            if current in visited:
                continue
```

```python
        visited.add(current)

        for dx, dy in directions:
            nx, ny = current[0] + dx, current[1] + dy
            neighbor = (nx, ny)

        if 0 <= nx < rows and 0 <= ny < cols and grid[nx][ny] != '#' and neighbor not
in visited:
                heapq.heappush(open_set, (
                    g + 1 + heuristic(neighbor, goal),
                    g + 1,
                    neighbor,
                    path + [neighbor]
                ))
    return None

shortest_path = a_star(grid, start, goal)
if shortest_path:
    print("Shortest path from start to goal:")
    for step in shortest_path:
        print(f"Step: {step}")
else:
    print("No path found.")
```

**Output:**

```
Shortest path from start to goal:
Step: (0, 0)
Step: (0, 1)
Step: (0, 2)
Step: (0, 3)
Step: (1, 3)
Step: (2, 3)
Step: (3, 3)
```

## 5. Solve the Water Jug Problem using Breadth First Search (BFS).

**Program/Code:**

```
from collections import deque

# Define jug capacities and goal

jug1_capacity = 4
jug2_capacity = 3
goal = 2

def is_goal(state):
    x, y = state
    return x == goal or y == goal

def get_next_states(x, y):
    states = []

    # Fill either jug
    states.append((jug1_capacity, y))  # Fill jug1
    states.append((x, jug2_capacity))  # Fill jug2

    # Empty either jug
    states.append((0, y))  # Empty jug1
    states.append((x, 0))  # Empty jug2

    # Pour jug1 -> jug2
    pour = min(x, jug2_capacity - y)
    states.append((x - pour, y + pour))

    # Pour jug2 -> jug1
    pour = min(y, jug1_capacity - x)
    states.append((x + pour, y - pour))
    return states

def bfs():
    visited = set()
    queue = deque()
```

```
    start_state = (0, 0)
    queue.append((start_state, [start_state]))

    while queue:

        (x, y), path = queue.popleft()
        if (x, y) in visited:
            continue
        visited.add((x, y))
        if is_goal((x, y)):
            return path

        for next_state in get_next_states(x, y):
            if next_state not in visited:
                queue.append((next_state, path + [next_state]))
    return None

solution_path = bfs()
if solution_path:
    print("Steps to reach the goal (2 liters in one jug):")
    for step in solution_path:
        print(f"Jug1: {step[0]}L, Jug2: {step[1]}L")
else:
    print("No solution found.")
```

**Output:**

Steps to reach the goal (2 liters in one jug):

Jug1: 0L, Jug2: 0L

Jug1: 0L, Jug2: 3L

Jug1: 3L, Jug2: 0L

Jug1: 3L, Jug2: 3L

Jug1: 4L, Jug2: 2L

## 10. Use constraint propagation to solve a Magic Square puzzle.

**Program/Code:**

```
from itertools import permutations
def is_magic(square):
    return (
        sum(square[0:3]) == 15 and
        sum(square[3:6]) == 15 and
        sum(square[6:9]) == 15 and
        sum(square[0::3]) == 15 and
        sum(square[1::3]) == 15 and
        sum(square[2::3]) == 15 and
        sum(square[0::4]) == 15 and
        sum(square[2:8:2]) == 15
    )

def solve_magic_square():
    digits = [1, 2, 3, 4, 5, 6, 7, 8, 9]

    for square in permutations(digits):
        if is_magic(square):
            return square
    return None
```

```
solution = solve_magic_square()

if solution:
    print("Magic Square Solution:")
    for i in range(0, 9, 3):
        print(solution[i], solution[i+1], solution[i+2])
else:
    print(" No solution found.")
```

**Output:**

```
Magic Square Solution:
2 7 6
9 5 1
4 3 8
```

## 9. Solve the 4 – Queens Problem as a CSP backtracking problem

**Program/Code:**

```
def is_safe(queen_positions, row, col):
    for r in range(row):
        c = queen_positions[r]
        if c == col or abs(c - col) == abs(r - row):
            return False
    return True

def solve_n_queens(n):
    solutions = []
    queen_positions = [-1] * n

    def backtrack(row):
        if row == n:
            solutions.append(queen_positions[:])
            return

        for col in range(n):
            if is_safe(queen_positions, row, col):
                queen_positions[row] = col
                backtrack(row + 1)
                queen_positions[row] = -1

    backtrack(0)
    return solutions

solutions = solve_n_queens(4)


def print_boards(solutions):
    for sol in solutions:
        print(" Solution:")
```

```
    for i in sol:
        row = ['.'] * len(sol)
        row[i] = 'Q'
        print(" ".join(row))
    print()

print_boards(solutions)
```

**Output:**

```
Solution:
. Q . .
. . . Q
Q . . .
. . Q .

Solution:
. . Q .
Q . . .
. . . Q
. Q . .
```

## 11. Apply optimization techniques to find the maximum value in a list.

**Simple optimization technique** to find the **maximum value in a list** — using a **linear scan** (also called **brute force search**).

**Program/Code:**

```
def find_max_value(data):
    max_val = data[0]

    for val in data:
        if val > max_val:
            max_val = val

    return max_val
data = [12, 5, 23, 7, 34, 9, 30]
print("Max value found using Brute Force:", find_max_value(data))
```

**Output:**

```
Max value found using Brute Force: 34
```

**Another simple technique** to find the **maximum value in a list** — this time using **Python’s built-in max() function**, which is the most concise and efficient method.

```
# Example list
    data = [12, 5, 23, 7, 34, 9, 30]

# Use Python's built-in max() function
    max_value = max(data)

    print("Max value found using built-in max():", max_value)
```

**Output:**

```
Max value found using built-in max(): 34
```

## 8. Implement the Minimax search algorithm for 2-player games. You may use a game tree with3 plies.

**Program/Code:**

```
# Leaf node values (3 plies: root -> children -> grandchildren)

# Structure: MAX -> MIN -> Leaf values


    tree = [

        [3, 5],

        [2, 9],

        [0, -1]

    ]


    def minimax(tree):
    min_values = []

  for branch in tree:
    min_val = min(branch)
    min_values.append(min_val)

  return max(min_values)


best_value = minimax(tree)
print(f"Best value for MAX: {best_value}")
```

**Output:**

```
Best value for MAX: 3
```