LAB 1: IMPLEMENTATION OF DIFFERENT AGENTS IN SNAKE GAME

OBJECTIVE

To implement different types of AI agents - Simple, Model-Based, Goal-Based, Utility-Based and Learning Agent - in a snake game environment to study their behavior, decision-making, and performance.

THEORY:

An **agent** is any entity that can:

- **Perceive** its environment through sensors,
- **Act** upon the environment using actuators.

In Artificial Intelligence, agents are used to make decisions autonomously in response to environmental changes. Each agent type is designed with different levels of intelligence, memory, and goal-orientation.

General PEAS Framework for AI Agents in Snake Game

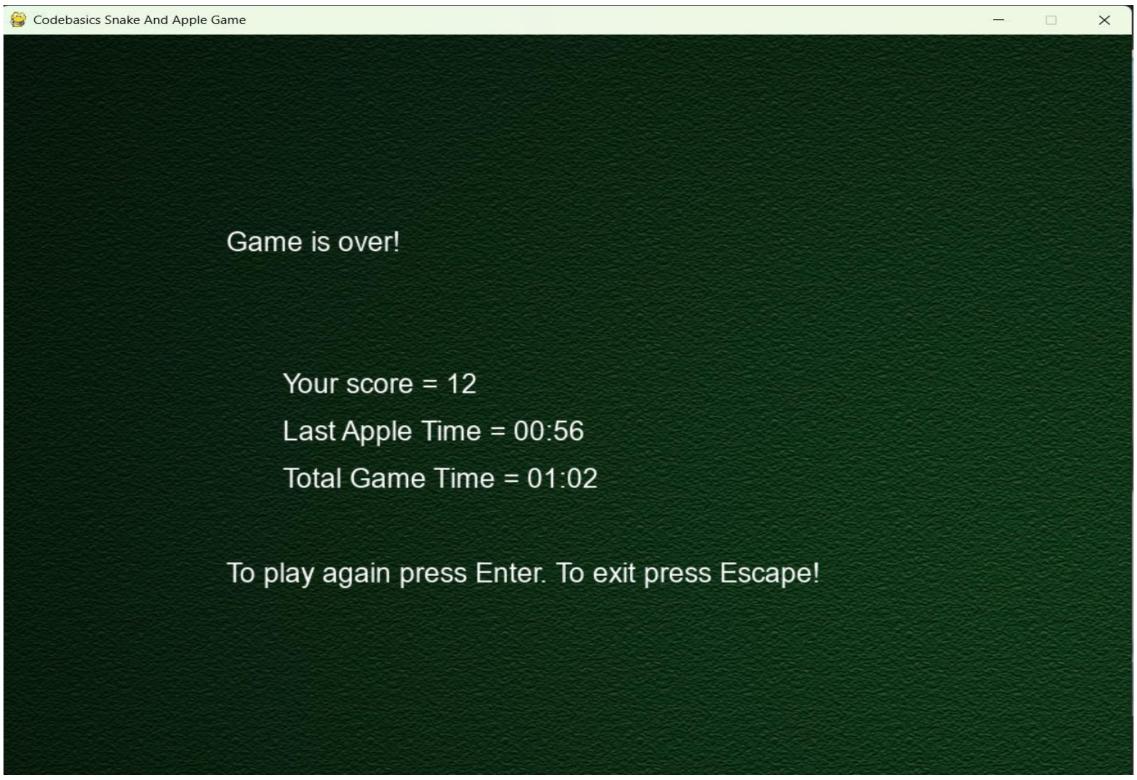
Component	Description
P - Performance Measure	- Number of apples eaten (score) - Length of survival (time or steps) - Avoiding collisions (walls, self)
E - Environment	- 2D game grid with: → Snake (head + body) → Apple → Walls or boundaries
A - Actuators	- Movement commands: → move_up() → move_down() → move_left() → move_right()
S - Sensors	- Snake's current position (head & body) - Apple's position - Grid dimensions and obstacles - Collision detection

1. SIMPLE REFLEX AGENT:

A Simple Reflex Agent operates solely based on the current percept without considering past actions or the future. It follows predefined condition-action rules like “if apple is to the left, move left.” In the Snake game, this agent simply moves toward the apple when it is directly aligned, without checking for obstacles or planning ahead. This makes it fast but highly prone to collisions and failure in complex situations.

CODE:

```
def agent_self(self):
head_x, head_y = self.snake.x[0], self.snake.y[0]
apple_x, apple_y = self.apple.x, self.apple.y
if head_x-apple_x == 0:
if head_y-apple_y >= 0:
self.snake.move_up()
else:
self.snake.move_down()
elif head_x-apple_x >= 0:
self.snake.move_left()
elif head_x-apple_x <= 0:
self.snake.move_right()
# Agent movement control: Uncomment one of the lines below to activate an agent.
if not pause:
self.agent_self() # Activate the random agent
# self.simple_agent() # Activate the simple, greedy agent
```



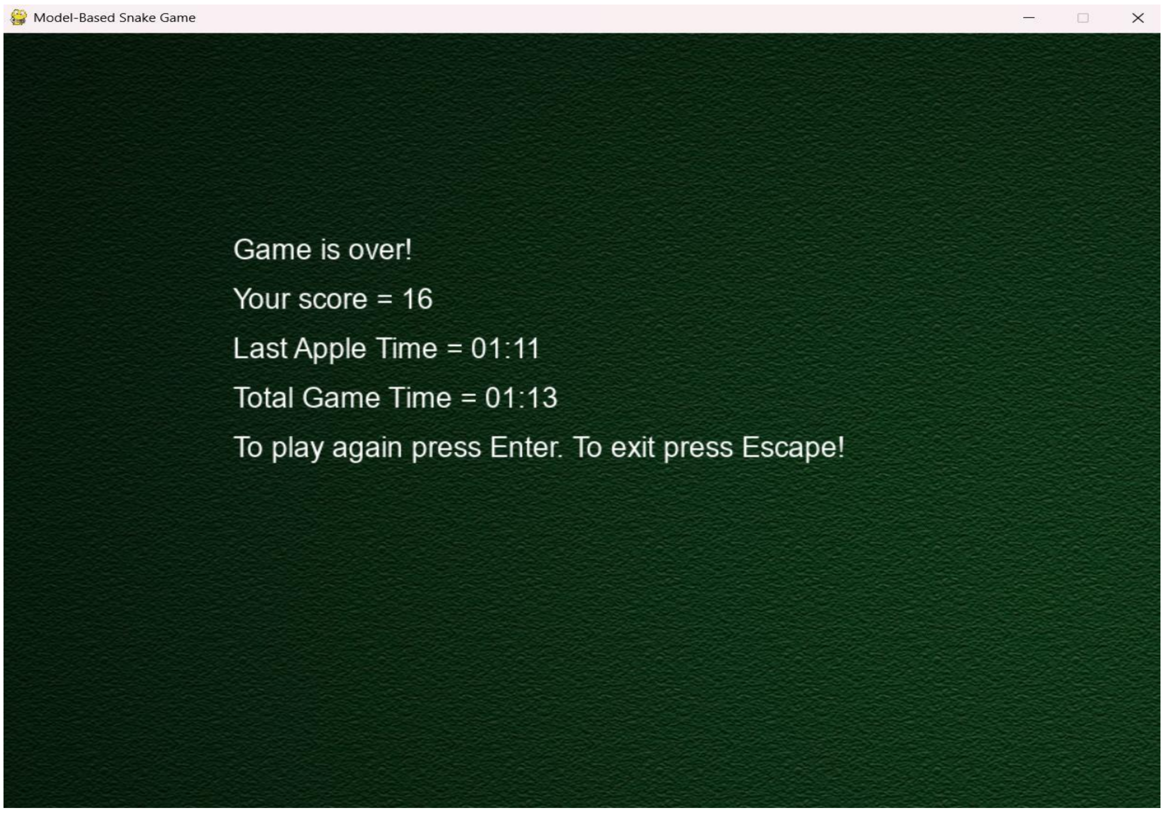
```
Game is over!

Your score = 12
Last Apple Time = 00:56
Total Game Time = 01:02

To play again press Enter. To exit press Escape!
```

2. MODEL-BASED AGENT:

A Model-Based Agent improves upon the simple agent by keeping track of the environment using an internal model. It uses this model to avoid dangers like walls and the snake's own body. In the Snake game, the agent evaluates all possible directions and checks which moves are safe before deciding. This helps it survive longer and make more intelligent choices, even if the apple is not directly aligned.

CODE:

```
def model_based_agent(self):
directions = ['left', 'right', 'up', 'down']
safe_moves = []
for d in directions:
nx, ny = self._get_potential_head(d)
if not self._is_potential_move_colliding(nx, ny):
dist = self._calculate_distance(nx, ny, self.apple.x, self.apple.y)
safe_moves.append((dist, d))
safe_moves.sort()
if safe_moves:
_, best = safe_moves[0]
getattr(self.snake, f"move_{best}")()
if not pause:
self.model_based_agent()
```



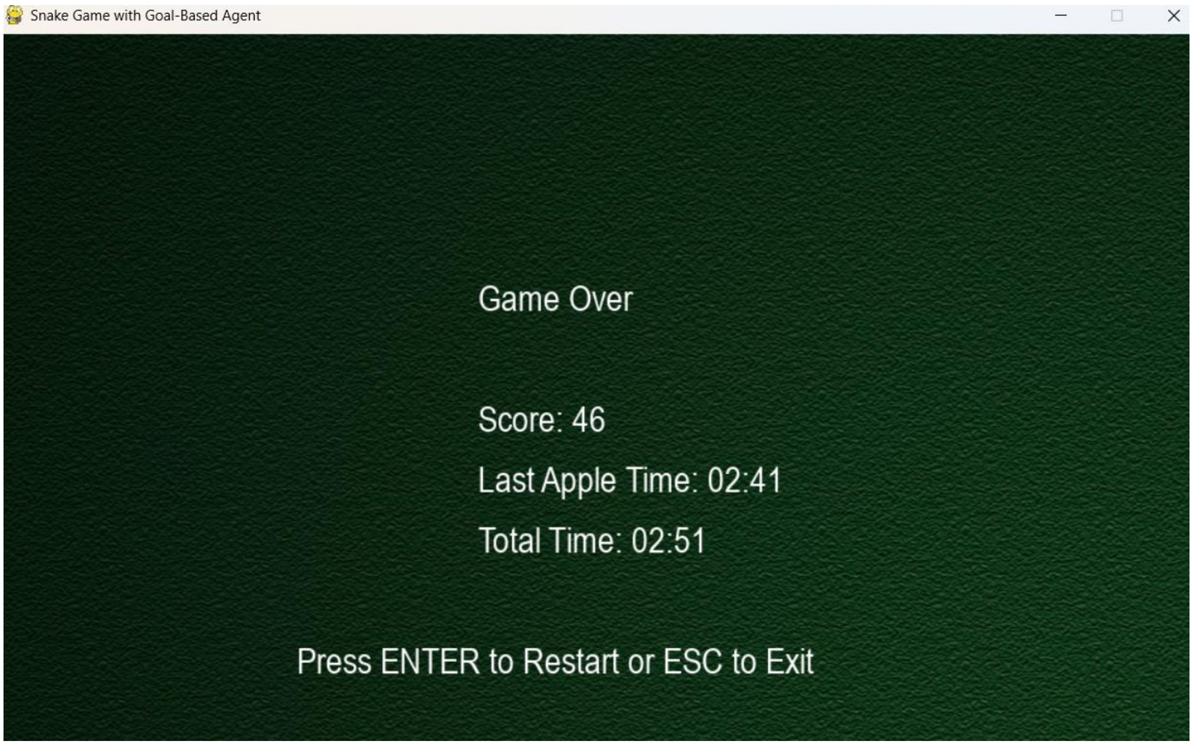
```
Game is over!
Your score = 16
Last Apple Time = 01:11
Total Game Time = 01:13
To play again press Enter. To exit press Escape!
```

3. GOAL-BASED AGENT:

A Goal-Based Agent takes actions specifically designed to achieve a particular goal. Unlike model-based agents, it actively reasons about which direction will bring it closer to the goal—in this case, reaching the apple. In the Snake game, the agent checks all valid directions and chooses the one that minimizes the distance to the apple. It is smarter than reactive agents and uses short-term planning.

CODE:

```
def goal_based_agent(self):
valid_moves = self._get_valid_moves()
if not valid_moves:
return # No valid move
best_direction = min(valid_moves, key=valid_moves.get)
if best_direction == 'left':
self.snake.move_left()
elif best_direction == 'right':
self.snake.move_right()
elif best_direction == 'up':
self.snake.move_up()
elif best_direction == 'down':
self.snake.move_down()
if not paused:
self.goal_based_agent()
```



Game Over

Score: 46

Last Apple Time: 02:41

Total Time: 02:51

Press ENTER to Restart or ESC to Exit

4. UTILITY-BASED AGENT:

A Utility-Based Agent not only aims for the goal but also evaluates the usefulness (utility) of each possible action to choose the best one. It often uses techniques like pathfinding (e.g., Breadth-First Search) to calculate the safest and shortest path to the apple. In the Snake game, this agent finds an optimal route while avoiding obstacles, making it the most intelligent and reliable among all.

In the **Snake game**, the utility-based agent evaluates not just which direction brings it closer to the apple, but **which path is safest and most efficient**. It often uses algorithms like **Breadth-First Search (BFS)** to explore all possible paths from the current snake position to the apple, while avoiding collisions with walls or its own body. This approach ensures the agent makes smart decisions that balance **shortest distance**, **safety**, and **survival**.

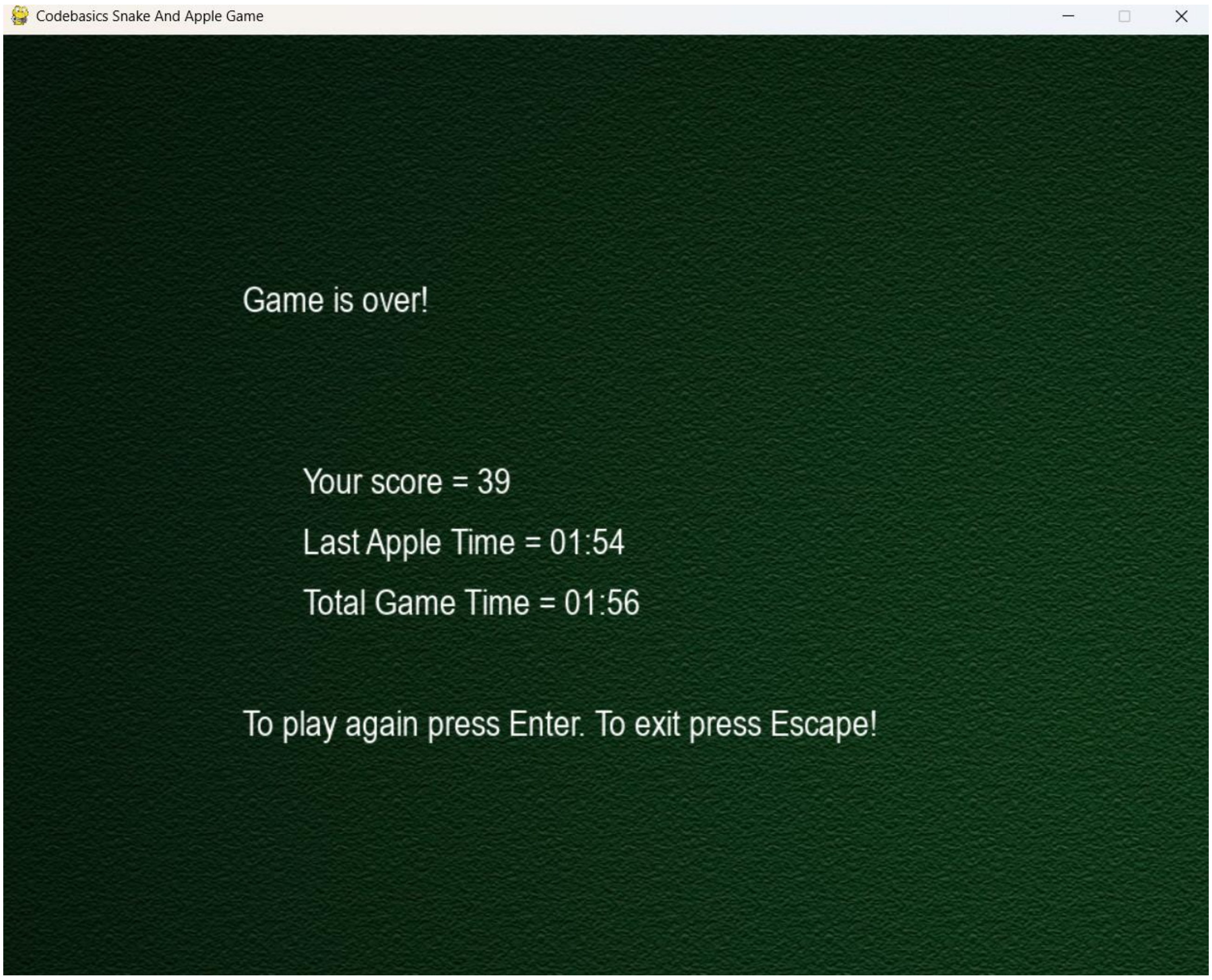
CODE:

```
def utility_agent_bfs(self):
path = self.bfs_path()
if path and len(path) > 1:
next_pos = path[1] # next step
head_x, head_y = self.snake.x[0], self.snake.y[0]
nx, ny = next_pos
if nx < head_x:
self.snake.move_left()
elif nx > head_x:
self.snake.move_right()
elif ny < head_y:
self.snake.move_up()
elif ny > head_y:
self.snake.move_down()
else:
# No path found, fallback to safe random move
self.safe_random_move()
def safe_random_move(self):
possible_moves = []
head_x, head_y = self.snake.x[0], self.snake.y[0]
# Check each direction if it's safe
if self._is_safe_position(head_x - SIZE, head_y) and self.snake.direction != 'right':
possible_moves.append('left')
if self._is_safe_position(head_x + SIZE, head_y) and self.snake.direction != 'left':
possible_moves.append('right')
if self._is_safe_position(head_x, head_y - SIZE) and self.snake.direction != 'down':
possible_moves.append('up')
if self._is_safe_position(head_x, head_y + SIZE) and self.snake.direction != 'up':
possible_moves.append('down')
```

```
if possible_moves:
move = random.choice(possible_moves)
if move == 'left':
self.snake.move_left()
elif move == 'right':
self.snake.move_right()
elif move == 'up':
self.snake.move_up()
elif move == 'down':
self.snake.move_down()
else:
# If no safe moves (very rare), keep current direction or move down by default
Pass
if not pause:
# Use the BFS utility agent to move snake smartly towards apple
self.utility_agent_bfs()
```

BFS LOGIC:

```
# BFS pathfinding to get shortest path from snake head to apple
def bfs_path(self):
start = (self.snake.x[0], self.snake.y[0])
goal = (self.apple.x, self.apple.y)
queue = deque()
queue.append([start]) # queue of paths
visited = set()
visited.add(start)
while queue:
path = queue.popleft()
current = path[-1]
if current == goal:
return path
# Return the path from start to goal
# Explore neighbors (up, down, left, right)
neighbors = [
(current[0] - SIZE, current[1]),
(current[0] + SIZE, current[1]),
(current[0], current[1] - SIZE),
(current[0], current[1] + SIZE)
]
for nx, ny in neighbors:
if (0 <= nx < self.surface.get_width() and 0 <= ny < self.surface.get_height())
and \
self._is_safe_position(nx, ny) and (nx, ny) not in visited:
visited.add((nx, ny))
queue.append(path + [(nx, ny)])
return None
```


Codebasics Snake And Apple Game

Game is over!

Your score = 39

Last Apple Time = 01:54

Total Game Time = 01:56

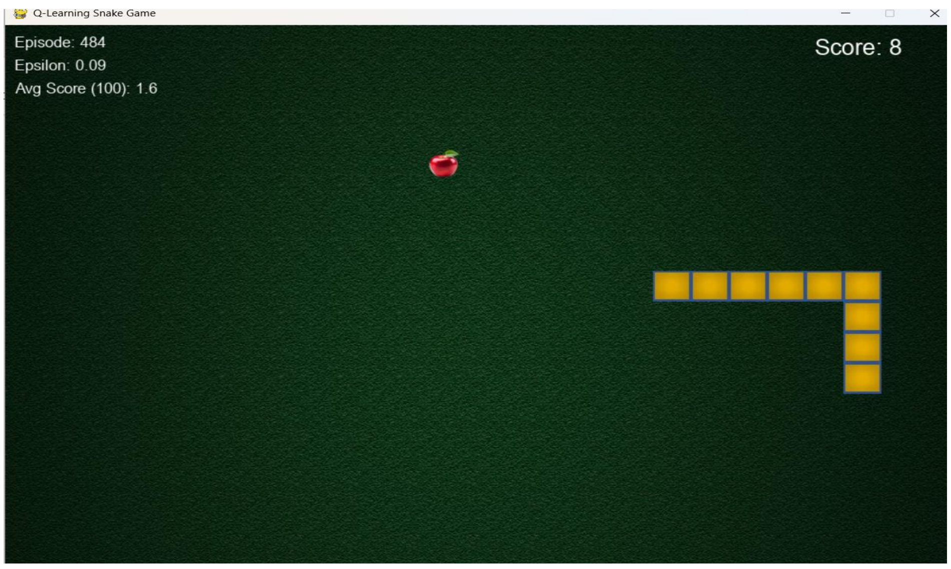
To play again press Enter. To exit press Escape!

5. LEARNING AGENT

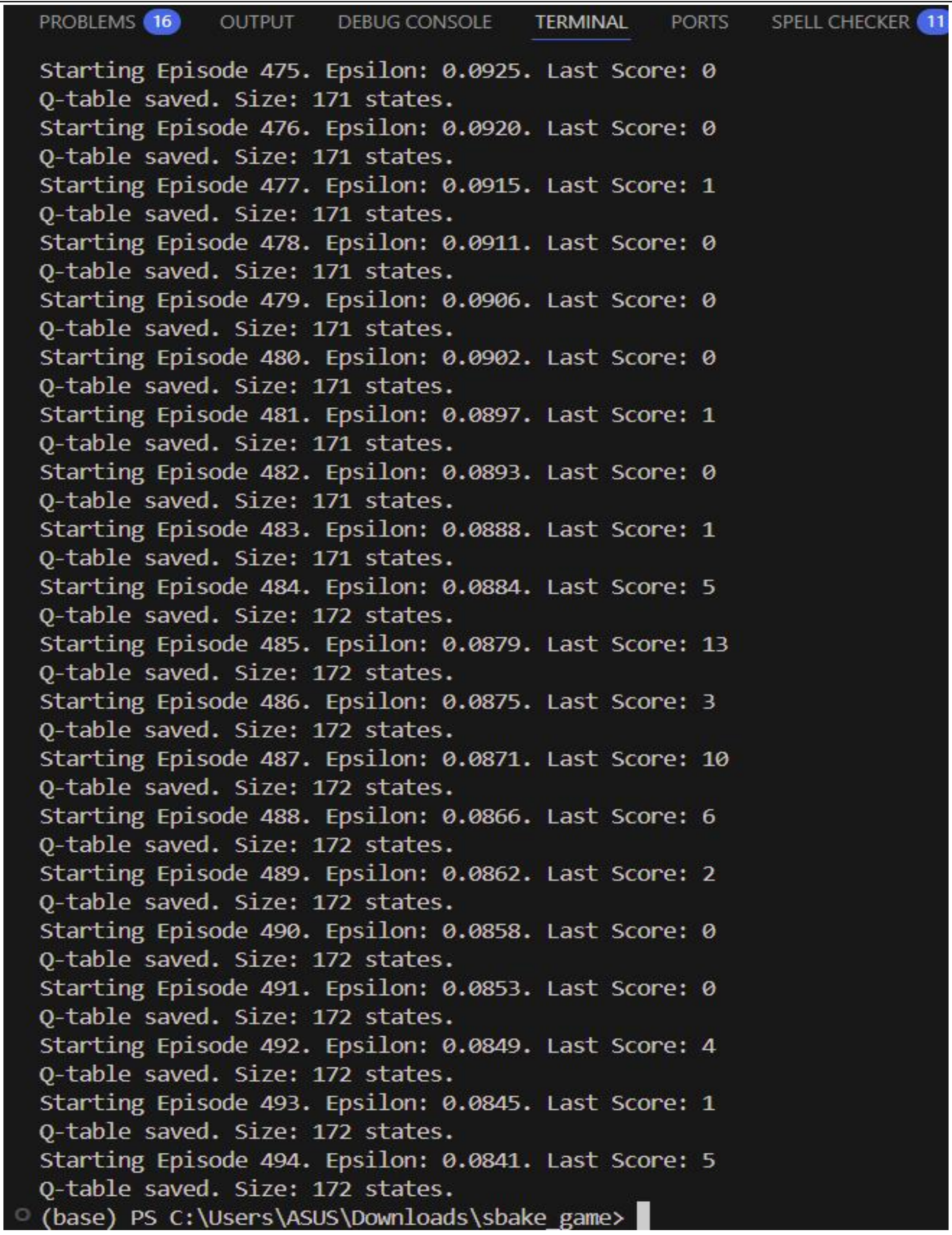
A **Learning Agent** is an intelligent agent that improves its performance over time by learning from past experiences. It has four main components: a learning element, a performance element, a critic, and a problem generator. The learning element helps the agent adapt to changes, while the critic provides feedback based on performance. In the context of games like Snake, a learning agent (e.g., using reinforcement learning) can discover optimal strategies by playing repeatedly and adjusting its behavior based on rewards and penalties. This makes it highly adaptable and capable of handling complex, unseen situations.

CODE:

```
# --- Q-Learning Agent Class (NEW) ---
class QLearningAgent:
    def __init__(self, game):
        self.game = game  # Reference to the game instance to access snake/apple state
        self.q_table = self.load_q_table()  # Load Q-table from file if it exists
        self.learning_rate = 0.1   # How much new information overrides old information (Increased from 0.01)
        self.discount_factor = 0.95  # Importance of future rewards
        self.epsilon = 1.0  # Exploration-exploitation trade-off: starts high for exploration
        self.epsilon_min = 0.01  # Minimum epsilon value
        self.epsilon_decay = 0.995 # Rate at which epsilon decays per episode (Increased from 0.999)
        self.actions = ['left', 'right', 'up', 'down']  # Possible actions
        self.action_map = {action: i for i, action in enumerate(self.actions)}  # Map action strings to indices

    def save_q_table(self):
        """Saves the current Q-table to a file using pickle."""
        try:
            with open("q_table.pkl", "wb") as f:
                pickle.dump(self.q_table, f)
            print(f"Q-table saved. Size: {len(self.q_table)} states.")
        except Exception as e:
            print(f"Error saving Q-table: {e}")

    def load_q_table(self):
        """Loads a Q-table from a file if it exists, otherwise returns an empty dictionary."""
        if os.path.exists("q_table.pkl"):
            try:
                with open("q_table.pkl", "rb") as f:
                    q_table = pickle.load(f)
                    print(f"Q-table loaded. Size: {len(q_table)} states.")
                    return q_table
            except Exception as e:
                print(f"Error loading Q-table, starting fresh. Error: {e}")
        return {}
```



Episode: 484
Epsilon: 0.09
Avg Score (100): 1.6
Score: 8



```
Starting Episode 475. Epsilon: 0.0925. Last Score: 0
Q-table saved. Size: 171 states.
Starting Episode 476. Epsilon: 0.0920. Last Score: 0
Q-table saved. Size: 171 states.
Starting Episode 477. Epsilon: 0.0915. Last Score: 1
Q-table saved. Size: 171 states.
Starting Episode 478. Epsilon: 0.0911. Last Score: 0
Q-table saved. Size: 171 states.
Starting Episode 479. Epsilon: 0.0906. Last Score: 0
Q-table saved. Size: 171 states.
Starting Episode 480. Epsilon: 0.0902. Last Score: 0
Q-table saved. Size: 171 states.
Starting Episode 481. Epsilon: 0.0897. Last Score: 1
Q-table saved. Size: 171 states.
Starting Episode 482. Epsilon: 0.0893. Last Score: 0
Q-table saved. Size: 171 states.
Starting Episode 483. Epsilon: 0.0888. Last Score: 1
Q-table saved. Size: 171 states.
Starting Episode 484. Epsilon: 0.0884. Last Score: 5
Q-table saved. Size: 172 states.
Starting Episode 485. Epsilon: 0.0879. Last Score: 13
Q-table saved. Size: 172 states.
Starting Episode 486. Epsilon: 0.0875. Last Score: 3
Q-table saved. Size: 172 states.
Starting Episode 487. Epsilon: 0.0871. Last Score: 10
Q-table saved. Size: 172 states.
Starting Episode 488. Epsilon: 0.0866. Last Score: 6
Q-table saved. Size: 172 states.
Starting Episode 489. Epsilon: 0.0862. Last Score: 2
Q-table saved. Size: 172 states.
Starting Episode 490. Epsilon: 0.0858. Last Score: 0
Q-table saved. Size: 172 states.
Starting Episode 491. Epsilon: 0.0853. Last Score: 0
Q-table saved. Size: 172 states.
Starting Episode 492. Epsilon: 0.0849. Last Score: 4
Q-table saved. Size: 172 states.
Starting Episode 493. Epsilon: 0.0845. Last Score: 1
Q-table saved. Size: 172 states.
Starting Episode 494. Epsilon: 0.0841. Last Score: 5
Q-table saved. Size: 172 states.
(base) PS C:\Users\ASUS\Downloads\sbake_game>
```

DISCUSSION:
In this lab, we implement different agent like simple reflex agent, model-based agent, goal-based agent, utility based agent and learning based agent in snake game program using python and see how all these module works in the game.
All four agents were somehow easy to implement but In Learning agent implementation we have to make several runs to make the agent learn which takes a long time and after making multiple attempts we can see the how learning agent is implemented and how it works.

CONCLUSION:
Hence, all five agent was implemented successfully in the snake game.