ARTIFICIAL INTELLIGENCE IMPORTANT QUESTIONS

Reinforcement Learning Definitions:

(i) Policy
A **policy (π)** defines the agent's way of behaving at a given time. It maps each state **(s)** to an action **(a)**, either deterministically or probabilistically.
Types of Policies:
- **Deterministic Policy**: The same action is chosen for a given state every time.
- **Stochastic Policy**: The action selection is probabilistic.

Example:
Consider a self-driving car navigating an intersection.
- **Deterministic Policy:** If the traffic light is red, always stop.
- **Stochastic Policy:** If there is light traffic, accelerate with 70% probability and slow down with 30% probability.

In mathematical notation, a policy **π(a|s)** gives the probability of taking action **a** in state **s**.

(ii) Value Function
The **value function (V(s))** estimates the expected cumulative reward the agent will receive starting from state **s** and following a policy **π**.
Types of Value Functions:
1. **State Value Function (V(s))**: Expected reward from a given state.
2. **Action Value Function (Q(s,a))**: Expected reward from taking a specific action in a given state.

Example:
Imagine playing a chess game:
- **V(s):** The likelihood of winning if you are in a given board position **s**.
- **Q(s,a):** The likelihood of winning if you make a particular move **a** from board position **s**.

Mathematically, the state value function under policy **π** is:

$$V^{\pi}(s) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R_t | S_0 = s, \pi\right]$$

where:
- Rt is the reward at time **t**.
- γ is the discount factor.

(iii) Discount Factor (γ)
The **discount factor (γ)** determines how much future rewards are valued compared to immediate rewards. It ranges between 0 and 1:
- **γ = 0** → The agent values only immediate rewards.
- **γ = 1** → The agent considers all future rewards equally.

Example:
In a game where you earn:
- **+10 points immediately**
- **+50 points in 5 moves**

- If **γ = 0.1**, the agent prioritizes immediate rewards (+10).
- If **γ = 0.9**, the agent considers the long-term benefit of the +50 points.

Mathematically, future rewards are weighted as:

$$R = R_0 + \gamma R_1 + \gamma^2 R_2 + \gamma^3 R_3 + ...$$

where each future reward is discounted.

(iv) Action
An **action (a)** represents a decision or move that the agent takes in a given state. Actions depend on the environment and policy.
Example:
- **Robot Vacuum Cleaner:**
  - Actions: Move left, move right, clean, dock for charging.
- **Chess Game:**
  - Actions: Move pawn, move knight, castle, resign.

An agent selects an action **a** based on a policy **π(a|s)**.

(v) Reward
A **reward (R)** is the feedback signal that guides the agent's learning. It quantifies how good or bad an action is.
Example:
- **In a Chess Game:**
  - Winning: +100 reward.
  - Losing: −100 penalty.
  - Moving a piece: 0 (neutral).
- **Self-driving Car:**
  - Staying in lane: +1
  - Reaching the destination: +100
  - Crashing: −100

The agent's goal is to maximise cumulative rewards over time.

Exploration vs. Exploitation Tradeoff in the Multi-Armed Bandit Problem:

The **multi-armed bandit (MAB) problem** is a fundamental reinforcement learning problem where an agent must repeatedly choose between multiple options (or "arms") to maximize the cumulative reward. Each arm provides a different, unknown reward distribution. The agent faces a **tradeoff between exploration and exploitation** to determine the best arm to pull.

Exploration
Exploration involves selecting **less-known** actions to gather more information about their reward distributions.
- **Purpose:** To discover better actions that may yield higher rewards in the long run.
- **Example:** A gambler in a casino tries different slot machines (arms) to estimate which one pays out the most.

Exploitation

Exploitation involves choosing the **best-known** action (i.e., the arm that has given the highest reward so far) to maximize the immediate reward.

- **Purpose:** To take advantage of the best-known option and maximize short-term gains.
- **Example:** A gambler continues playing the slot machine that has given the highest winnings in past attempts.

2. Consequences of More Exploration vs. More Exploitation

Strategy	Advantage	Disadvantage
More Exploration	Finds the best action in the long run	Short-term losses due to suboptimal choices
More Exploitation	Immediate high rewards	Risk of missing better alternatives

Consequences of Too Much Exploration:

1. **Delayed Rewards:** The agent wastes too much time exploring suboptimal arms.
2. **Inefficiency:** The agent might keep testing low-reward actions, even after enough data suggests a better choice.
3. **Slow Convergence:** The agent takes longer to find the best strategy.

Consequences of Too Much Exploitation:

1. **Suboptimal Strategy:** The agent may get stuck in a **local optimum** and miss better arms.
2. **Lack of Adaptability:** If reward distributions change over time (non-stationary problems), the agent won't adapt to new, better actions.
3. **Short-Term Bias:** The agent focuses too much on **immediate** rewards instead of maximizing long-term returns.

ε-Greedy Algorithm in Multi-Armed Bandit Problem

The **ε-greedy algorithm** is a simple but effective method for balancing **exploration** (trying new arms) and **exploitation** (choosing the best-known arm) in the **multi-armed bandit (MAB) problem**.

1. How ε-Greedy Works

1. The agent initializes each arm's estimated value to **0** or some prior estimate.
2. At each step **t**:
   - With probability **ε**, the agent **explores** by selecting a random arm.
   - With probability **1 - ε**, the agent **exploits** by selecting the arm with the highest estimated reward.
3. The reward from the selected arm is observed, and the estimate of that arm's value is updated.

Too little exploration (ε = 0 or very low ε) → Can get stuck in a suboptimal choice.
Too much exploration (high ε, e.g., 0.5) → Wastes time on low-reward arms.
Optimal balance → allows the agent to find the best arm efficiently.

Epsilon-Greedy Method

- Epsilon-Greedy is a simple method to balance exploration and exploitation by choosing between exploration and exploitation randomly.
- The epsilon-greedy, where epsilon refers to the probability of choosing to explore (randomly picking an action), exploits most of the time with a small chance of exploring.

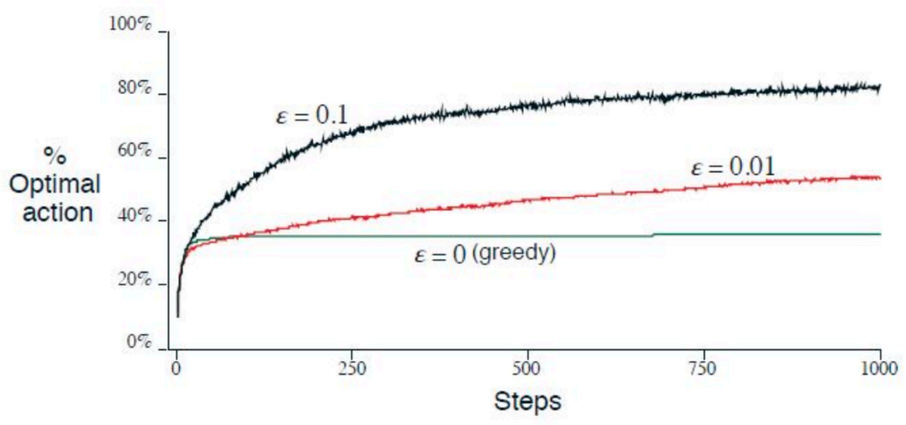
$$\text{Action at time(t)} \begin{cases} \max Q_t(a) & \text{with probability } 1-\epsilon \\ \text{any action (a)} & \text{with probability } \epsilon \end{cases}$$

```
p = random()

if p < ε:
    pull random action
else:
    pull current-best action
```

Algorithm of Epsilon-Greedy

```
Initialize the estimated values of all arms to zero or a small positive number.
For each trial:
    Generate a random number between 0 and 1.
    If the number is less than $\epsilon$, select a random arm (exploration).
    Otherwise, select the arm with the highest estimated reward (exploitation).
    Update the estimated reward of the selected arm based on the observed reward.
```



The greedy method found the optimal action in only approximately one-third of the tasks.

The epsilon-greedy methods eventually perform better because they continue to explore, and to improve their chances of recognizing the optimal action. The $\varepsilon = 0.1$ method explores more, and usually finds the optimal action earlier, but never selects it more than 91% of the time.
The $\varepsilon = 0.01$ method improves more **slowly**, but eventually performs better than the $\varepsilon = 0.1$ method on both performance measures.

Hill Climbing Search in AI

Hill climbing is a heuristic search algorithm used for optimization problems. It continuously moves towards the direction of increasing value (or decreasing cost) in the search space until it reaches a peak, which represents the best possible solution based on its evaluation function. It is often used in problems such as route planning, game playing, and machine learning model tuning.

How Hill Climbing Works

1. **Start at an Initial State**: Choose a random or predefined starting point in the search space.
2. **Evaluate Neighboring States**: Examine the neighboring states using an evaluation function (also called a heuristic function).
3. **Move to the Best Neighbor**: If a neighbor has a better evaluation value, move there.
4. **Repeat Until Convergence**: Continue moving until no better neighbors are found.

Advantages of Hill Climbing

- **Memory Efficient –** Uses minimal memory as it only tracks the current state and its neighbors.
- **Faster Convergence** – Can quickly find a good solution, especially for well-structured problems.
- **Suitable for Real-Time Systems** – Works well for applications that require quick responses, like robotics and real-time decision-making.
- **Works Without Full Knowledge –** Does not require complete information about the search space, making it useful for problems with unknown environments.

Problems in Hill Climbing

Despite its simplicity, hill climbing has several **limitations**:

(i) Local Maxima Problem

- The algorithm may get stuck at a **local maximum** instead of the **global maximum**.
- **Example:** A hiker climbing a hill may stop at a peak, unaware that a taller mountain exists elsewhere.

(ii) Plateaus (Flat Regions)

- A **plateau** is a region where all neighbouring states have the same value, causing the algorithm to stall.
- **Example:** A robot searching for a higher floor may reach a long hallway with no visible incline, causing confusion.

(iii) Ridges

- A **ridge** is a narrow, elevated region where movement in a single direction doesn't improve the solution.
- The algorithm may fail to follow the ridge because it can only make small, greedy moves.

(iv) Overshooting or Step Size Issues

- If step sizes are too small, the algorithm takes too long to reach a solution.
- If step sizes are too large, it may **overshoot** the optimal solution.

3. How to Overcome These Problems

There are several **modifications and alternative approaches** to address hill climbing's limitations:

(i) Random Restarts (Solve Local Maxima & Plateaus)

- Run **multiple hill climbs** from different starting points.
- If one gets stuck, restart from another point.

- **Example:** A delivery driver explores multiple routes before choosing the best one.

(ii) Simulated Annealing (Solve Local Maxima & Plateaus)
- Introduces **random jumps** to escape local maxima.
- Uses a **temperature parameter** to allow bad moves early on and reduce them over time.
- **Example:** A hiker sometimes moves **downhill** to find a better uphill path later.

(iii) Stochastic Hill Climbing (Solve Ridges & Local Maxima)
- Instead of always choosing the best neighbour, it picks **randomly with probability proportional to the value**.
- Helps explore different directions.

(iv) Tabu Search (Solve Cycling & Plateaus)
- Keeps a list of recently visited states (**tabu list**) to avoid getting stuck in loops.
- Useful when the search space has many **plateaus**.

(v) Genetic Algorithms (Solve Most Problems)
- Uses **mutation and crossover** to explore multiple solutions simultaneously.
- Doesn't get stuck in local maxima as easily.
- **Example:** Evolving better solutions over generations, like natural selection.

Bayesian Belief Network (BBN)

A **Bayesian Belief Network (BBN)** is a **probabilistic graphical model** that represents a set of variables and their conditional dependencies using **Directed Acyclic Graphs (DAGs)**. It provides a compact way to model **uncertainty** in real-world problems.

1. Structure of a Bayesian Belief Network

A Bayesian Belief Network consists of:
- **Nodes:** Represent **random variables** (e.g., Disease, Symptoms, Weather).
- **Edges:** Represent **dependencies** (directed edges show cause-effect relationships).
- **Conditional Probability Tables (CPTs):** Define the probability of each variable given its parents.

Example: Medical Diagnosis

Imagine a **network for diagnosing a disease** based on symptoms.
markdown

```
Disease → Fever
        → Cough
```

- The **"Disease"** node affects **"Fever"** and **"Cough"**.
- The CPT defines:
  - P(Fever∣Disease) → Probability of fever if the patient has the disease.
  - P(Cough∣Disease) → Probability of cough if the patient has the disease.
  - P(Disease) → Prior probability of having the disease.

2. How Does a Bayesian Belief Network Work?

BBNs use **Bayes' Theorem** to update probabilities when new evidence is available:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

- **Forward Inference:** Predict effects from causes (e.g., given Disease, what is the probability of Fever?).
- **Backward Inference:** Infer causes from effects (e.g., given Fever, what is the probability of Disease?).

3. Key Advantages of Bayesian Belief Networks

- **Handles Uncertainty**: Useful in real-world domains like medicine, finance, and robotics.
- **Compact Representation**: Reduces complexity using conditional independence
- **Allows Probabilistic Inference**: Can answer **what-if** questions based on observed data.

4. Challenges & Solutions

Challenge	Solution
Computational Complexity	Use Approximate Inference (Monte Carlo Methods)
Difficult to Define CPTs	Learn CPTs from data using Machine Learning
Requires Expert Knowledge	Use automated structure learning

What is Heuristic?

1. Definition of Heuristic in AI

A **heuristic** is a function that estimates the cost or value of a solution in an optimization or search problem. It is a **rule of thumb** used to speed up problem-solving when an exact solution is computationally expensive.

Key Characteristics of Heuristics

- **Approximates Optimal Solutions**: Finds "good enough" solutions quickly.
- **Domain-Specific**: Designed based on knowledge of the problem.
- **Trade-off Between Accuracy & Speed**: Faster but may not guarantee the best solution.

2. Role of Heuristics in AI Search Algorithms

Heuristics guide AI in **search problems** like pathfinding and game playing. Examples include:

- **A* Algorithm** → Uses a heuristic to find the shortest path efficiently.
- **Greedy Best-First Search** → Selects the path with the lowest heuristic value at each step.
- **Hill Climbing** → Moves towards higher heuristic values to reach an optimal state.

3. Example Calculation: Heuristic in A* Algorithm

Let's calculate a **heuristic function** h(n) for a state in a pathfinding problem.

Scenario: Shortest Path in a Grid (Manhattan Distance Heuristic)

Imagine an **agent** trying to reach a **goal (G)** from a **starting position (S)** in a **grid-based map**.

Given Grid (Numbers represent the heuristic values):

```
S → 3 → 2 → 1 → G
↓   ↓   ↓   ↓
4   3   2   1
5   4   3   2
```

Heuristic Function:
We use **Manhattan Distance**, defined as:
$h(n)=|x_{current} - x_{goal}|+|y_{current} - y_{goal}|$

Example Calculation

- **Start Position** S
  S(0,0)
- **Goal Position** G
  G(3,3)

$h(S)=|0-3|+|0-3|=3+3=6$

- **State at (1,0):**

$h(1,0)=|1-3|+|0-3|=2+3=5$

- **State at (2,2):**

$h(2,2)=|2-3|+|2-3|=1+1=2$

4. Choosing a Good Heuristic
A **good heuristic** should:
- **Be Admissible** → Never overestimate the actual cost.
- **Be Consistent** → Maintain logical ordering between states.
- **Reduce Computation** → Speed up search without unnecessary calculations.