# Question 1

## Part 1

**B – TD value function estimation with exploring starts**

We have exploring starts since *s* is initialized randomly at the start of each episode. Further, notice that tmp represents the TD error, and we compute an estimate of *V(s)* on each step of the episode (rather than at the end of each trace as is the case in MC).

## Part 2

**M**

[3 + 2 – 4 + 4 -3, 2 – 4 + 4 – 3, 4 – 3, -2 + 3 – 3, 3 – 3] = [2, -1, 1, -2, 0] -> average 0

## Part 3

**M**

[2, -2] -> average 0

## or

## I got **K**. 3+2-4+4-3 = 2; -2+3-3 = -2; [2, 2, 2, -2, -2] -> average = 0.4

## Part 4

**I – Prioritized experience replay**

## Part 5

**L – 2, 4, 3, 1**

Each iteration of Monte-Carlo tree search is conservertive as follows:

1. Selection: sample an episode from the root node until the end of the tree.

2. Expansion: expand the final node and select a random child node.

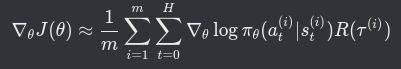
3. Simulation: sample a random sequence of actions from this node.

4. Backpropagation: update the values for all nodes in this entire episode.

## Part 6

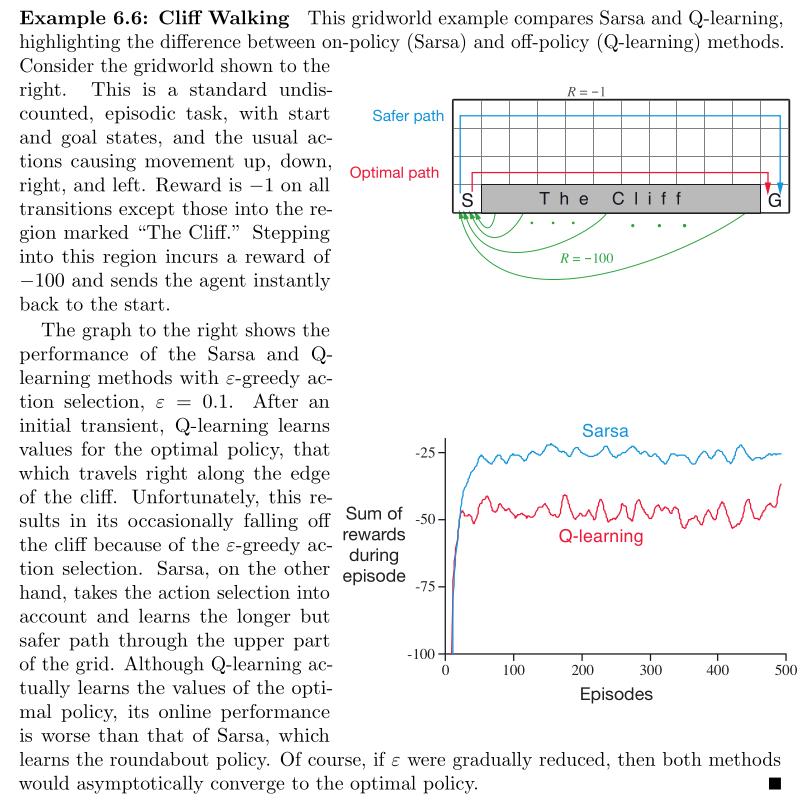
**H**

The full equation is given by



# Question 2O

For this question, refer to the **cliff-walking example** seen in lectures. In particular, observe that with Q-Learning the agent is able to learn the riskier policy of walking along the cliff’s edge, whereas with SARSA it learns a safer policy of walking far from the cliff’s edge.T



# Question 3

## Part a

* **Memory:** we have a limited number of computational resources. Combinatorically large state spaces are too big to fit in memory. With approximate solutions, we can represent the Q-values for the entire state space by a function which can be expressed by a *fixed number of parameters*.
* **Generalization:** with tabular methods, a state can only be updated once visited. With approximate methods, we can use *function approximation* to share knowledge between nearby states.

## Part b

This allows us to sample trajectories from regions close to good trajectories. More efficient than uniform sampling. This can help prevent the policy from getting stuck in a local optimum.

The notes give the following algorithm for the Cross-Entropy Method:

|  |
| --- |
| def cross\_entropy(s\_t):  # Initialise distribution P as a uniform distribution.  P = uniform\_distribution()  for \_ in range(iterations):  # Sample M trajectories from P. (using the model f(s\_t, a\_t))  trajectories = P.sample(n=M)  # Compute the total cost for each trajectory.  costs = [sum(map(lambda s\_t: C(s\_t, s\_goal), t)) for t in trajectories]  # Fit P to the top N trajectories. (where top = lowest costs)  sort(costs) # (Pedantic but python sorting is in place)  P = fit\_gaussian(costs[:N])  # The optimal trajectory is the mean of P.  Return P.mean  def mpc():  s\_t = s\_init  While s\_t != s\_goal:  policy = cross\_entropy(s\_t)  s\_t = policy.first\_step() |

Part c  
higher entropy (more randomness) in a stochastic policy encourages exploration and can lead to agent to find a better path/reward outside its optimum, as it may converge too quickly to a local optimum with a low entropy policy. By encouraging high entropy in the long term (maximum entropy reinforcement learning), the agent can more easily adapt to changes in the environment or new knowledge.

Example to illustrate entropy:   
Suppose we are at a certain state and can choose to go North, East, South or West.  
Policy A: [0.1, 0.7, 0.1, 0.1]  
Policy B: [0.4, 0.4, 0.2, 0.2]  
Policy B will have higher entropy. This means we incentivize policies where there are multiple good actions, rather than just one strict optimal path. The agent is encouraged to learn several good routes, so in a noisy environment if it is blown off its optimal path it can still follow a reasonable route, rather than relying on learning one narrow optimal path. Therefore, the agent is more robust. This also means that it can explore more throughout the learning process while still avoiding very bad actions.