Name:Vamshiteja Kaspe
Znumber:z23733757

```
     google collab
link:https://colab.research.google.com/drive/1gko9AXgmUUXPfVwN1H-
DonMi7fIiOLh_?usp=sharing
```

Autonomous Delivery Drones

A forward-thinking logistics company is revolutionizing its delivery operations by incorporating autonomous drones. These drones will be responsible for picking up packages from a central hub and delivering them to delivery truck.

In the context of drone delivery, "picking" refers to the process of retrieving packages from the hub and transporting them to the designated delivery truck. Once the packages are secured, the drones must navigate the shortest route to reach their respective destinations, optimizing efficiency and ensuring timely deliveries.

To achieve this goal, the company plans to implement Q-learning, a reinforcement learning technique, enabling the drones to learn and adapt their routes dynamically for optimal performance and customer satisfaction.

```
#import libraries
import numpy as np
```

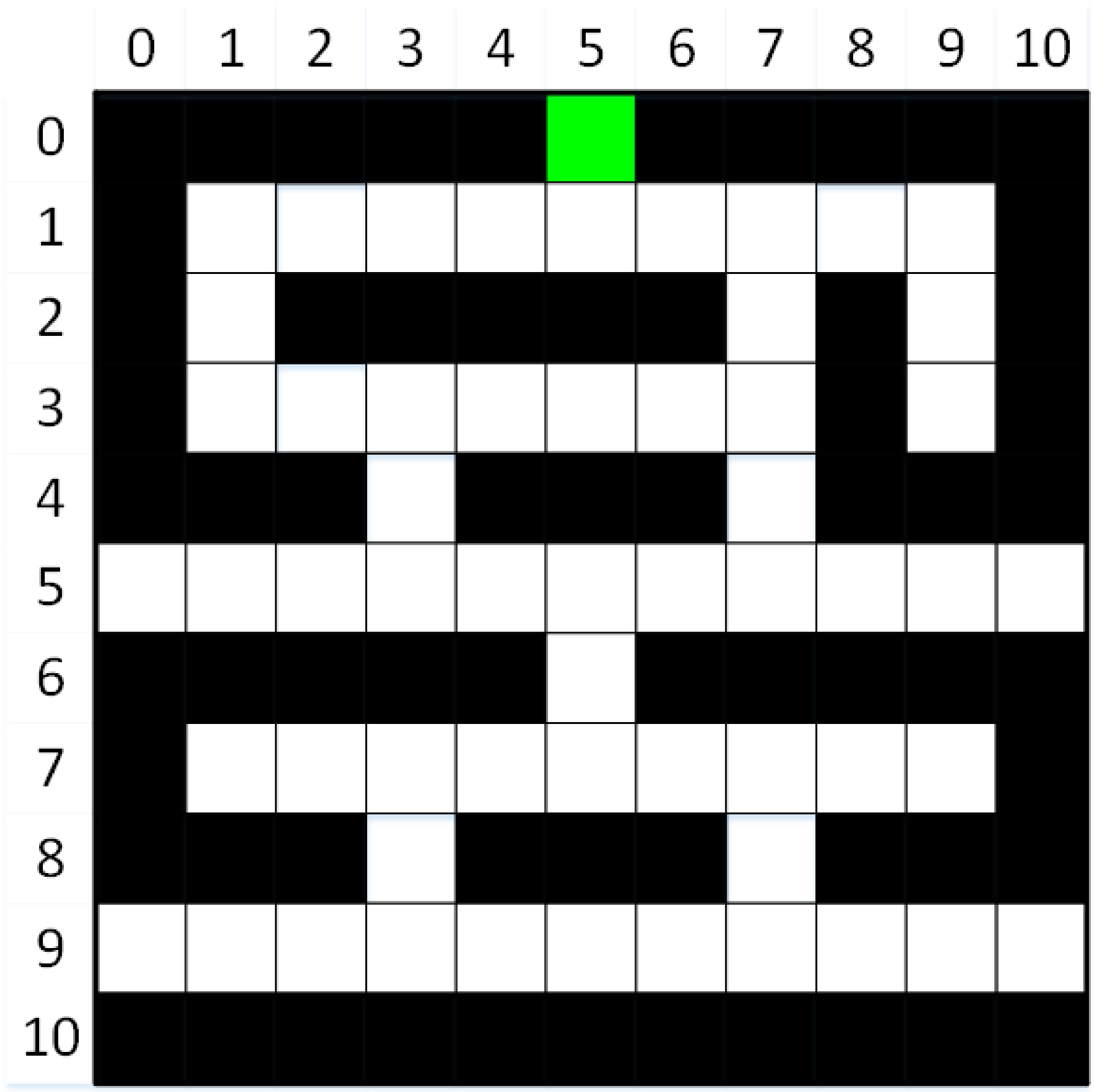
Establish the Environment

The environment in the context of autonomous drone delivery consists of **states**, **actions**, and **rewards**—essential components for the Q-learning AI agent. The AI agent's inputs are states and actions, whereas the agent's outputs are actions.

States

The states in this ecosystem indicate multiple delivery places. Some of these locations function as package pickup points (**marked in white**), while others serve as drone landing zones (**marked in green**). The **black areas** denote no-fly zones into which drones are not permitted.

The places in white and green are in **terminal states**!



The AI agent's goal is to find the most efficient route from the pick-up points to all of the other locations in the delivery area where the drone is permitted to travel.

Within the delivery region, there are 100 possible states (locations) depicted on the map. These states are organised in a grid of ten rows and ten columns, with each location designated by a row and column index.

```
# Define the dimensions of the delivery area (i.e., its states)
delivery_area_rows = 10
delivery_area_columns = 10

# Create a 3D numpy array to store the current Q-values for each state
and action pair: Q(s, a)
# The array comprises 10 rows and 10 columns (to match the shape of
the delivery area), along with a third "action" dimension.
# The "action" dimension consists of 4 layers that help us track the
Q-values for each possible action in each state.
# The initial value of each (state, action) pair is set to 0.
q_values = np.zeros((delivery_area_rows, delivery_area_columns, 4))
```

Actions

The actions that are available to the AI agent are to move the robot in one of four directions:

- Up
- Right
- Down
- Left

Obviously, the AI agent must learn to avoid driving into the item storage locations (e.g., shelves)!

```
#define actions
#numeric action codes: 0 = up, 1 = right, 2 = down, 3 = left
actions = ['up', 'right', 'down', 'left']
```

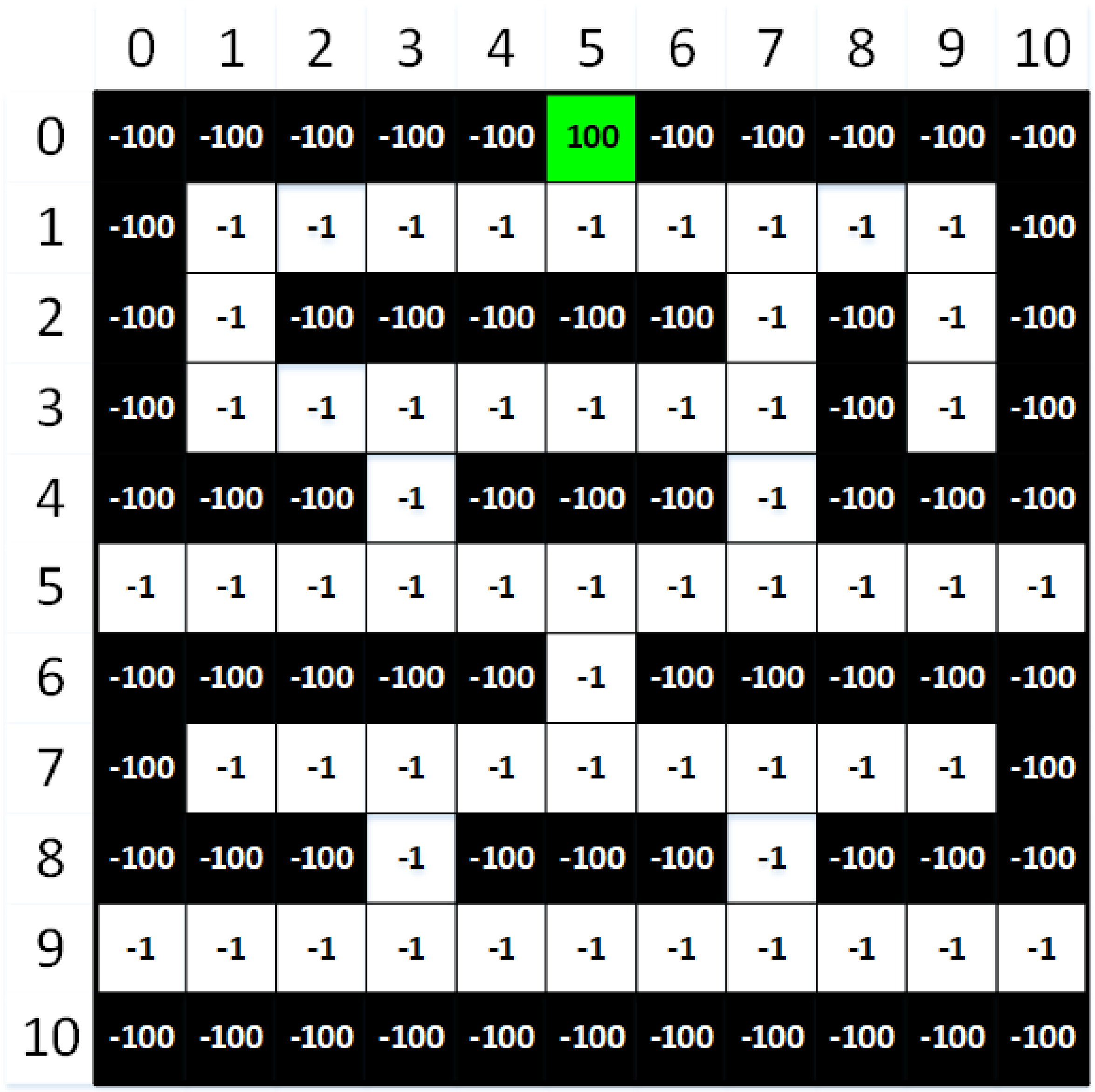
Rewards

In the field of autonomous drone delivery, defining **rewards** is critical for moulding the AI agent's behaviour.

To facilitate the agent's learning process, each site in the delivery region is connected with a certain reward value.

The agent is free to begin its travel from any designated pick-up site. The final goal, however, stays constant: ***maximise its cumulative rewards***!

Negative rewards (also known as **penalties**) are imposed on all states except the desired state. This method requires the AI agent to find the *most efficient routes* to the target while *minimising its penalties*!



	0	1	2	3	4	5	6	7	8	9	10
0	-100	-100	-100	-100	-100	100	-100	-100	-100	-100	-100
1	-100	-1	-1	-1	-1	-1	-1	-1	-1	-1	-100
2	-100	-1	-100	-100	-100	-100	-100	-1	-100	-1	-100
3	-100	-1	-1	-1	-1	-1	-1	-1	-100	-1	-100
4	-100	-100	-100	-1	-100	-100	-100	-1	-100	-100	-100
5	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
6	-100	-100	-100	-100	-100	-1	-100	-100	-100	-100	-100
7	-100	-1	-1	-1	-1	-1	-1	-1	-1	-1	-100
8	-100	-100	-100	-1	-100	-100	-100	-1	-100	-100	-100
9	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
10	-100	-100	-100	-100	-100	-100	-100	-100	-100	-100	-100

To maximise its overall rewards (while minimising its total penalties), the AI agent must find the shortest paths between the pick-up spots (blue squares) and all other locations inside the delivery area where the drone is allowed to travel (green and white squares). Furthermore, the agent must learn to avoid colliding with any no-fly zones (black zones) while performing delivery tasks.

```
# Create a 2D numpy array to store the rewards for each state.
# The array consists of 10 rows and 10 columns (matching the shape of
the delivery area),
# with each value initialized to -100.
rewards = np.full((delivery_area_rows, delivery_area_columns), -100.)
rewards[0, 5] = 100.  # Set the reward for the packaging area (i.e.,
the goal) to 100
```

```
# Define aisle locations (i.e., white squares) for rows 1 through 9
aisles = {}  # Store locations in a dictionary
aisles[1] = [i for i in range(1, 10)]
aisles[2] = [1, 7, 9]
aisles[3] = [i for i in range(1, 8)]
aisles[3].append(9)
aisles[4] = [3, 7]
aisles[5] = [i for i in range(10)]
aisles[6] = [5]
aisles[7] = [i for i in range(1, 10)]
aisles[8] = [3, 7]
aisles[9] = [i for i in range(10)]

# Set the rewards for all aisle locations (i.e., white squares)
for row_index in range(1, 10):
    for column_index in aisles[row_index]:
        rewards[row_index, column_index] = -1.

# Print rewards matrix
for row in rewards:
    print(row)
```

```
[-100. -100. -100. -100. -100.  100. -100. -100. -100. -100.]
[-100.   -1.   -1.   -1.   -1.   -1.   -1.   -1.   -1.   -1.]
[-100.   -1. -100. -100. -100. -100. -100.   -1. -100.   -1.]
[-100.   -1.   -1.   -1.   -1.   -1.   -1.   -1. -100.   -1.]
[-100. -100. -100.   -1. -100. -100. -100.   -1. -100. -100.]
[-1. -1. -1. -1. -1. -1. -1. -1. -1. -1.]
[-100. -100. -100. -100. -100.   -1. -100. -100. -100. -100.]
[-100.   -1.   -1.   -1.   -1.   -1.   -1.   -1.   -1.   -1.]
[-100. -100. -100.   -1. -100. -100. -100.   -1. -100. -100.]
[-1. -1. -1. -1. -1. -1. -1. -1. -1. -1.]
```

Our next challenge will be to train our AI agent with a Q-learning model. The following steps will guide you through the learning process:

1. To begin a new episode, start the agent at a random, non-terminal state (white square).
2. Choose an action for the current state (such as *up*, *right*, *down*, or *left*). An *epsilon greedy algorithm* will be used to pick actions. This algorithm prioritises the most promising action, but it also investigates less promising choices to stimulate environment exploration.
3. Carry out the selected action, transitioning to the next state (i.e., moving to the next place).
4. Receive the transition reward and compute the time difference.
5. Refresh the Q-value associated with the preceding state and action pair.

6.Return to step 1 if the new (current) state is a terminal state. Otherwise, move on to step 2.

This process will be repeated 1000 times, giving the AI agent plenty of opportunities to learn the best routes between the pick-up places and all other authorised locations within the delivery area. At the same time, the agent will learn to avoid accidents with no-fly zones, assuring effective and efficient deliveries!

Define Helper Functions

```
# Define a function to determine if the specified location is a
terminal state
def is_terminal_state(current_row_index, current_column_index):
    # If the reward for this location is -1, it is not a terminal
state (i.e., a 'white square')
    if rewards[current_row_index, current_column_index] == -1.:
        return False
    else:
        return True

# Define a function to choose a random, non-terminal starting location
def get_starting_location():
    current_row_index = np.random.randint(delivery_area_rows)
    current_column_index = np.random.randint(delivery_area_columns)
    # Continue choosing random row and column indexes until a non-
terminal state is identified
    while is_terminal_state(current_row_index, current_column_index):
        current_row_index = np.random.randint(delivery_area_rows)
        current_column_index =
np.random.randint(delivery_area_columns)
    return current_row_index, current_column_index

# Define an epsilon-greedy algorithm to choose the next action (i.e.,
where to move next)
def get_next_action(current_row_index, current_column_index, epsilon):
    if np.random.random() < epsilon:
        return np.argmax(q_values[current_row_index,
current_column_index])
    else:
        return np.random.randint(4)

# Define a function to get the next location based on the chosen
action
def get_next_location(current_row_index, current_column_index,
action_index):
    new_row_index = current_row_index
    new_column_index = current_column_index
    if actions[action_index] == 'up' and current_row_index > 0:
        new_row_index -= 1
    elif actions[action_index] == 'right' and current_column_index <
delivery_area_columns - 1:
        new_column_index += 1
    elif actions[action_index] == 'down' and current_row_index <
delivery_area_rows - 1:
        new_row_index += 1
    elif actions[action_index] == 'left' and current_column_index > 0:
        new_column_index -= 1
    return new_row_index, new_column_index

# Define a function to get the shortest path between any location
within the delivery area that
# the drone is allowed to travel and the item packaging location.
def get_shortest_path(start_row_index, start_column_index):
    if is_terminal_state(start_row_index, start_column_index):
        return []
    else:
        current_row_index, current_column_index = start_row_index,
start_column_index
        shortest_path = []
        shortest_path.append([current_row_index,
current_column_index])
        while not is_terminal_state(current_row_index,
current_column_index):
            action_index = get_next_action(current_row_index,
current_column_index, 1.)
            current_row_index, current_column_index =
get_next_location(current_row_index, current_column_index,
action_index)
            shortest_path.append([current_row_index,
current_column_index])
        return shortest_path
```

Train the AI Agent using Q-Learning

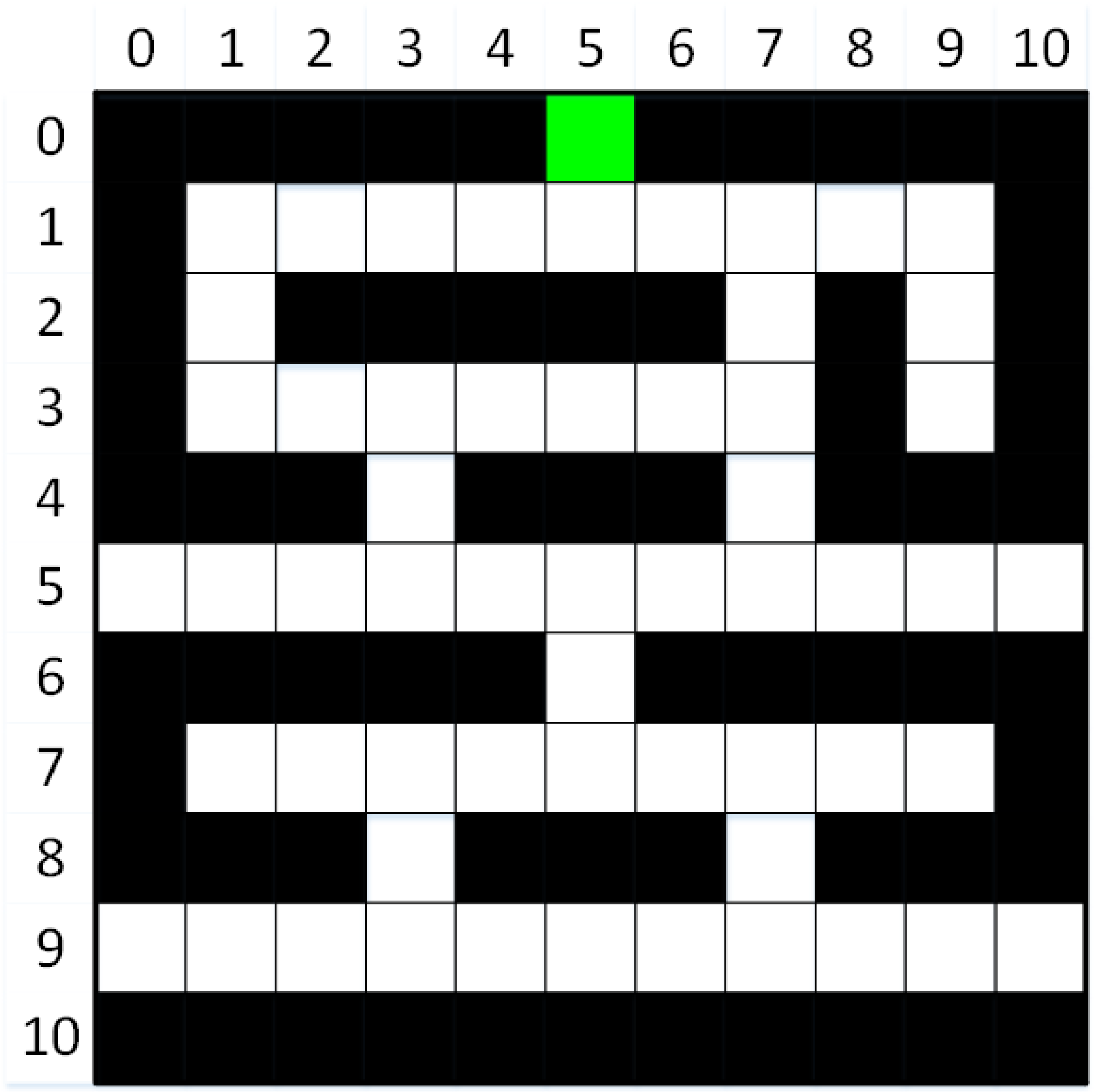
```
# Define training parameters
exploration_rate = 0.9  # The percentage of time when we explore
(choose a random action) instead of exploiting the best action
discount_factor = 0.9  # Discount factor for future rewards
learning_rate = 0.9  # The rate at which the AI agent learns

# Run through 1000 training episodes
for episode in range(1000):
    # Get the starting location for this episode
    row_index, column_index = get_starting_location()

    # Continue taking actions (i.e., moving) until reaching a terminal
state
    # (i.e., reaching the item packaging area or crashing into an item
storage location)
    while not is_terminal_state(row_index, column_index):
        # Choose which action to take (i.e., where to move next)
        action_index = get_next_action(row_index, column_index,
exploration_rate)

        # Perform the chosen action and transition to the next state
(i.e., move to the next location)
```

```
        old_row_index, old_column_index = row_index, column_index  #
Store the old row and column indexes
        row_index, column_index = get_next_location(row_index,
column_index, action_index)

        # Receive the reward for moving to the new state and calculate
the temporal difference
        reward = rewards[row_index, column_index]
        old_q_value = q_values[old_row_index, old_column_index,
action_index]
        temporal_difference = reward + (discount_factor *
np.max(q_values[row_index, column_index])) - old_q_value

        # Update the Q-value for the previous state and action pair
        new_q_value = old_q_value + (learning_rate *
temporal_difference)
        q_values[old_row_index, old_column_index, action_index] =
new_q_value

print('Training complete!')
```

```
Training complete!
```

Retrieve Optimal Routes

With our AI agent fully trained, let's uncover its newfound knowledge by visualizing the shortest routes from any location within the delivery area where the drone is permitted to travel to the designated package pick-up points.



Execute the code cell below to explore various starting points and witness the AI agent's proficiency in finding the most efficient paths!

```
# Display a few shortest paths
print(get_shortest_path(3, 9))  # Starting at row 3, column 9
print(get_shortest_path(5, 0))  # Starting at row 5, column 0
print(get_shortest_path(9, 5))  # Starting at row 9, column 5
```

```
[[3, 9], [2, 9], [1, 9], [1, 8], [1, 7], [1, 6], [1, 5], [0, 5]]
[[5, 0], [5, 1], [5, 2], [5, 3], [4, 3], [3, 3], [3, 4], [3, 5], [3,
6], [3, 7], [2, 7], [1, 7], [1, 6], [1, 5], [0, 5]]
[[9, 5], [9, 4], [9, 3], [8, 3], [7, 3], [7, 4], [7, 5], [6, 5], [5,
5], [5, 6], [5, 7], [4, 7], [3, 7], [2, 7], [1, 7], [1, 6], [1, 5],
[0, 5]]
```

Last but Not Least...

Indeed, our drone's ability to find the shortest path from any 'legal' warehouse location to the item packaging area is impressive. **However, what about the reverse journey?**

To clarify, our drone is currently adept at delivering items from any point in the warehouse ***to*** the packaging area. Yet, after delivering an item, it must travel ***from*** the packaging area to another warehouse location to pick up the next item!

Rest assured, addressing this challenge is straightforward—simply ***reverse the order of the shortest path***.

Execute the code cell below to observe an illustrative example:

```
# Display an example of a reversed shortest path
example_path = get_shortest_path(5, 2)  # Go to row 5, column 2
example_path.reverse()
print(example_path)
```

```
[[0, 5], [1, 5], [1, 6], [1, 7], [2, 7], [3, 7], [3, 6], [3, 5], [3,
4], [3, 3], [4, 3], [5, 3], [5, 2]]
```

##Convergence Process In this step, you will monitor the convergence of your Q-network. During training, you should keep an eye on the mean square error (MSE) of your Q-values and the weight trajectories of the Q-network. This displays the neural network's learning progress and convergence. Here's how to go about it:

MSE Convergence: Over time, plot the MSE of your Q-network's predictions. As the Q-network learns and converges on accurate Q-values, it should steadily decrease.

Weight Trajectories: Consider how your Q-network's weights fluctuate during training. Plot individual layer or neuron weights to show how they evolve.

By displaying the convergence process, you can demonstrate that your Q-network is learning to effectively approximate the Q-values and is converging to an optimal policy in your complex grid world environment.