Part 1: Comparing Code

1) Action Initialization & Allowed Movements

Skeleton had:

```
def init_actions(self):
    self.actions = {
        "left": (-1, 0),
        "right": (1, 0),
        "down": (0, -1),
        "up": (0, 1)
    }
    self.action_list = list(self.actions.keys())

def allowed_movements(self):
    ...
    # each state s is a (col,row)
    if s[0] < self.space_subdivisions - 1:
        self.allowed_moves[...] += [1]
    ...
```

Final has effectively the same dictionary of actions but reorganized. For example:

```
def allowed_movements(self):
    # in final code, we do something like:
    for (x, y) in self.ind2state.keys():
        state_idx = self.ind2state[(x,y)]
        valid_list = []
        if x < self.space_subdivisions - 1:
            valid_list.append(1)  # "right"
        ...
```

Why / Impact:

- We now **explicitly** separate (x, y) from `state_idx`. This is primarily for clarity.
- It ensures the final code can easily see which discrete "action indices" (0=left, 1=right, 2=down, 3=up) are valid in each (x, y) location.
- This doesn't drastically change performance but makes the code **easier to maintain**.

2) Q-Table Initialization

Skeleton snippet:

```
# ADD YOUR CODE SNIPPET BETWEEN EX. 2.1
# Initialize a numpy array with ns state rows and na state
  columns
# with float values from 0.0 to 1.0
Q = None
```

Final snippet:

```
Q = np.zeros((ns, na), dtype=np.float32)
for s in range(ns):
    valid_acts = self.allowed_moves[s]
    for a in range(na):
        if a not in valid_acts:
            Q[s,a] = np.nan
```

Why / Impact:

- We replaced the placeholder `None` with an **actual** initialization.
- Instead of random numbers [0..1], the final code uses `np.zeros`, which is simpler and more stable in Q-learning.
- Invalid actions are explicitly set to `NaN`, ensuring the agent never picks them when doing `argmax`.
- This fix is **crucial** for correct Q-learning behavior.

3) Epsilon-Greedy Implementation

Skeleton:

```
def epsilon_greedy(Q, state, all_actions, ...):
    if eps_type == 'constant':
        epsilon = epsilon_final
        # ...
        action = None
    elif eps_type == 'linear':
        # ...
        action = None
```

Final:

```python
def epsilon_greedy(
    Q, state, all_actions, current_total_steps=0,
    epsilon_initial=1.0, epsilon_final=0.2,
    anneal_timesteps=10000, eps_type="constant"
):
    if eps_type == 'constant':
        epsilon = epsilon_final
        if np.random.rand() < epsilon:
            action = np.random.choice(all_actions)
        else:
            q_vals = [Q[state,a] for a in all_actions]
            best_idx = np.nanargmax(q_vals)
            action = all_actions[best_idx]
    elif eps_type == 'linear':
        fraction = min(float(current_total_steps)/float(
            anneal_timesteps), 1.0)
        epsilon = epsilon_initial + fraction*(epsilon_final -
            epsilon_initial)
        ...
```

Why / Impact:

- We replaced the placeholder `action=None` with the standard ϵ-greedy approach.
- With probability ϵ, the agent explores randomly; otherwise, it exploits the best action.
- This ensures the agent balances exploration and exploitation, which is **essential** for good RL performance.

4) Bellman Update for Q-Learning

Skeleton:

```python
# inside q_learning while loop
# ADD YOUR CODE SNIPPET BETWEEN EX. 2.2
# Implement the Bellman Update equation: Q(s,a)     Q(s,a) +
    [... etc ...]
# ...
```

Final:

```
old_val = Q[s_current, action]
best_future = np.nanmax(Q[s_next, :])
td_target = R + self.gamma * best_future
Q[s_current, action] = old_val + self.alpha * (td_target -
    old_val)
```

Why / Impact:

- The skeleton had a placeholder comment. The final code **implements the standard Q-learning update**:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\Big[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\Big].$$

- Without this, the agent never updates its Q-values and cannot learn.

5) Convergence Criterion (`while episode < self.episode_max and diff > self.threshold`)

Skeleton did:

```
while episode <= self.episode_max:
    ...
    # diff = 100
```

Final does:

```
while (episode < self.episode_max) and (diff > self.threshold
    ):
    # ...
    diff = np.nanmean(np.abs(Q - Q_old))
```

Why / Impact:

- We changed from a naive "run until `episode_max`" to a better loop that also checks whether `diff` is below the threshold.
- `diff` measures the difference between the old and new Q-values (like a measure of how stable the Q table is).
- This helps the algorithm **stop early** if it converges, saving time.

6) Episode Structure and `R_total`

In the skeleton, `R_total` was never reset properly. In the final code, we do:

```
# each new episode:
R_total = 0
while not end_episode:
    ...
    R_total += R
```

Why / Impact:

- This **tracks** total reward each episode, so we can monitor performance.
- If we did not reset `R_total`, we might keep accumulating reward from previous episodes incorrectly, or reference it before assignment.

Observations from the Code and Environment

1. Agent's Capability of Achieving Maximum Score of 11

The Q-learning agent is theoretically capable of achieving the maximum score of 11. However, several factors influence whether this is consistently achievable:

- **Exploration vs. Exploitation Trade-off:** During exploration, the agent may not consistently choose the optimal actions, which could delay or prevent convergence to the optimal policy.
- **Convergence Criteria:** If the threshold for convergence (`diff` in Q-learning) is too lenient or the number of episodes (`self.episode_max`) is insufficient, the agent might converge to a suboptimal policy before learning the optimal one.
- **Random Initialization:** The environment starts with a random seed, leading to variability in training outcomes across runs.
- **Reward Structure:** The combination of living penalties, king fish rewards, and jellyfish penalties may lead the agent to prioritize suboptimal actions if hyperparameters like the discount factor (γ) and learning rate (α) are poorly tuned.

2. Performance Variability

Testing multiple times with the same number of episodes can result in different outcomes due to:

- **Stochastic Exploration:** The agent's actions during exploration are probabilistic, driven by epsilon-greedy behavior.
- **Environment Dynamics:** Random placements of the king fish and jellyfish (if applicable) may create scenarios where achieving the maximum score is easier or harder.

3. Key Parameters to Tune for Consistency

- **Alpha (α) and Gamma (γ):**
  - Higher α accelerates learning but may lead to high variance in Q-value updates.
  - A well-balanced γ ensures that the agent values long-term rewards appropriately, which is critical for catching the king fish.
- **Epsilon Annealing:**
  - An effective annealing schedule (e.g., linear or exponential) ensures the agent transitions smoothly from exploration to exploitation, improving the likelihood of consistently achieving the optimal policy.

4. Possible Limitations Preventing Maximum Score

- **Suboptimal Policy:** If the agent learns a local optimum, it won't achieve the maximum score.
- **Reward Penalties:** Excessive penalties from jellyfish collisions or living steps may offset the king fish reward, particularly in scenarios where the king fish is hard to reach.

Part 2: Hyperparameter Configurations and Observations

3.1 Hyperparameters (student_3_2_1.py and student_3_2_2.py)

The parameters we modified include:

- **Rewards array** (the environment's R-living, R-jelly, R-king fish).
- **Learning rate** α.
- **Discount factor** γ.
- **Epsilon** scheduling: $\epsilon_{\text{initial}}$, ϵ_{final}, and the annealing timesteps.
- **Threshold** for convergence checks.

For instance:

`student_3_2_1.py`:

- `rewards = [10, -10, -10, 10, -10, 10, -10, 10, -10, 10, -10, 10, -10]`
  (Large positive for certain fish, negative for others, producing a "fast catch" or "mixed" environment.)
- $\alpha = 0$, $\gamma = 0$
  (No real learning from future rewards, so the agent basically sees immediate reward only and does not update much.)
- $\epsilon_{\text{initial}} = 1$, $\epsilon_{\text{final}} = 1$
  (Agent stays random at all times, never becomes greedy.)
- `annealing_timesteps` = 1, `threshold` = 1e-6

This combination effectively results in a policy that **does not** improve, because $\alpha = 0$ (no updates) and $\epsilon = 1$ (always random).

`student_3_2_2.py`:

- `rewards = [-10, -10, -10, -10, ... , -10]`
  (All negative, so the agent is heavily penalized for basically anything, including catching the "King Fish.")
- $\alpha = 0.02$, $\gamma = 0.12$
  (A small learning rate and a relatively small discount factor.)

- $\epsilon_{\text{initial}} = 1$, $\epsilon_{\text{final}} = 1$
  (Again, no decay, so the agent remains random in practice, but it *can* update the Q-table slightly.)
- `annealing_timesteps` $= 1$, `threshold` $=$ 1e-6

Why Changing Reward Values? How They Affect the Final Result

The assignment's instructions mention that the total reward is a sum $R = R_{\text{living}} + R_{\text{jelly}} + R_{\text{king}}$. By altering these terms:

- If we **increase the King fish reward**, the agent has a stronger incentive to find and catch it.
- If we **decrease (make negative) the Jelly fish reward**, the agent tries harder to avoid collisions with jelly fish.
- If we **add a living penalty** each time step, the agent is encouraged to move quickly and not idle.
- If we set all rewards to **negative**, the agent sees every action as "punishing," so it might freeze or wander randomly (depending on α, γ).

Hence, changing each reward can drastically shift the agent's learned strategy or whether it learns a stable policy at all.

Analysis of the Long-Term Return for the Optimal Policy

Reward Structure Analysis

- **King Fish Reward (`rewards[0]`):**
  - Value: 50
  - A positive reward for catching the King Fish.
- **Jellyfish Reward (`rewards[1:n-1]`):**
  - Value: -10
  - A negative reward for colliding with a jellyfish.

- **Living Reward (`rewards[n]`):**
  - Value: -2
  - A small negative reward for each step taken.

Optimal Policy

The optimal policy aims to maximize the cumulative return:

1. **Goal:** Catch the King Fish as quickly as possible while avoiding jellyfish.
2. **Living Reward Penalty:** Penalizes excessive steps, so the shortest path to the King Fish is optimal.
3. **Jellyfish Penalty:** Avoiding jellyfish is critical since each collision significantly reduces the reward.

Initial Position Analysis

- **Diver's initial position:** $[1, 8]$
- **King Fish position:** $[8, 5]$

The diver must navigate from $[1, 8]$ to $[8, 5]$, avoiding jellyfish located at multiple positions.

Long-Term Return Calculation

Assuming the optimal policy:

1. **Path:** The diver takes the shortest path to $[8, 5]$, avoiding jellyfish entirely.
2. **Steps:** Let S represent the number of steps to reach the King Fish.
3. **Return:**

$$R = 50 + (-2) \cdot S$$

   where:

   - 50: Reward for catching the King Fish.
   - $-2 \cdot S$: Accumulated living reward penalty for S steps.

Example Calculation

For an estimated path requiring 15 steps:

$$R = 50 + (-2) \cdot 15 = 50 - 30 = 20$$

This value could vary slightly depending on the exact path optimization and whether any jellyfish collisions occur. For the optimal policy, jellyfish collisions are expected to be zero.

Final Answer

The **long-term return** of the optimal policy is:

$$R = 50 + (-2 \cdot \text{minimal steps}),$$

which depends on the specific shortest path and obstacle configuration. For typical configurations like this, it is expected to be around 20 to 30.