1. Stack Challenge (30 Points)

Problem: **Valid Parentheses Checker**

Description:
Given a string `s` consisting only of the characters `'('`, `')'`, `'{'`, `'}'`, `'['` and `']'`, write a Python function **`is_balanced(s)`** that uses a **stack** to determine whether the parentheses/brackets in the string are **balanced**. The rules for a balanced string are:

- Every opening bracket has a corresponding closing bracket of the **same type**.
- Brackets must close in the **correct order** (e.g., `"(]"` is invalid even though each character has a match of some type).

Input: A string `s`.
Output: Return `True` if `s` is balanced, otherwise `False`.
Examples:

1. `s = "()"` → `True`
2. `s = "()[]{}"` → `True`
3. `s = "(]"` → `False`
4. `s = "([)]"` → `False`
5. `s = "{[]}"` → `True`

Constraints:

- The length of `s` can be up to 10^5 characters.
- `s` may contain only the characters `(){}[]`.

Starter Template:

```python
def is_balanced(s: str) -> bool:
    """
    Return True if the string s is a valid sequence of parentheses/brackets.
    Otherwise return False.
    """
    # HINT: Use a stack to store opening brackets.
    # When you see a closing bracket, check if it matches the top of the stack.

    # Your code here
    pass

# Sample test calls:
print(is_balanced("()"))       # Expected True
print(is_balanced("()[]{}"))   # Expected True
print(is_balanced("(]"))       # Expected False
print(is_balanced("([)]"))     # Expected False
print(is_balanced("{[]}"))     # Expected True
```

Challenge: Implement the **stack** operations (push/pop) logically. You'll need to decide how to map closing brackets to matching opening brackets and how to handle mismatches or leftover openings.

2. Treasure Hunt Game Using Queues (30 Points)

Problem:
You are leading a team of adventurers to find treasure on an island represented as a grid. The treasure is hidden at specific locations, and you must direct your team to collect them in the shortest time possible.

Game Rules:

1. The island is represented as a 2D grid of size n×nn \times nn×n.
   - `0` represents an empty cell.
   - `1` represents the adventurers' starting point.
   - `T` represents treasure locations.
   - `X` represents obstacles.
2. Adventurers can move **up, down, left, or right**, but they cannot move into obstacles.
3. Your task is to find the **shortest path** to collect all treasures.

Example Input/Output:

Grid:

```
[
    [1, 0, 0, T],
    [X, X, 0, T],
    [0, 0, 0, 0],
    [T, 0, X, T]
]
```

Output:

```
# Start at (0, 0), collect all treasures
game = TreasureHunt(grid)
print(game.find_treasures())  # Output: 10 (Shortest time to collect all treasures)
```