ASSIGNMENT 3 AI

School of Mechanical and Manufacturing Engineering

Student Name: Syed Muneeb Imtiaz

Reg No: 364644

Table of Contents

Q: Complete **only Medium and Hard** challenges of Python from https://www.hackerrank.com/

1. Medium Challenges

1.1 Queens Attack II: This challenge involves calculating how many squares the queen can attack on an n×n chessboard. It's a great exercise to understand grid-based problems and edge-case handling.

Understanding the Problem

- The chessboard is n×n in size.
- The queen can move horizontally, vertically, and diagonally until she encounters an obstacle or the edge of the board.
- You are given the queen's position and the positions of obstacles.

Algorithm

Initialize Counters: Initialize eight counters for each direction (up, down, left, right, and the four diagonals).

Calculate Maximum Distances: For each direction, calculate the maximum distance the queen can move without any obstacles. This will be limited by the edges of the board.

Adjust for Obstacles: For each obstacle, determine if it blocks the queen's path. If it does, update the maximum distance for that direction to be the distance from the queen to the obstacle, minus one.

Sum the Distances: Add up the maximum distances the queen can move in each direction. This sum is the total number of squares she can attack.

Implementation in Python

```python
def queensAttack(n, k, r_q, c_q, obstacles):
    # Initialize directions: up, down, left, right, and diagonals
    up = down = left = right = up_left = up_right = down_left =
        down_right = 0

    # Calculate initial maximum distances
    up = n - r_q
    down = r_q - 1
    left = c_q - 1
    right = n - c_q
    up_left = min(up, left)
    up_right = min(up, right)
    down_left = min(down, left)
    down_right = min(down, right)

    # Adjust distances for each obstacle
    for obs in obstacles:
        r_o, c_o = obs
        # Check each direction and update if the obstacle is blocking
        # ...
```

```python

    # Sum up the distances
    total_attack_squares = up + down + left + right + up_left + up_right
        + down_left + down_right
    return total_attack_squares# Online Python compiler (interpreter) to
        run Python online.
# Write Python 3 code in this online editor and run it.
print("Hello world")
```

1.2 Sherlock and Anagrams*:* In this problem, you need to determine the number of anagrammatic pairs of substrings in a string. It's a good challenge for practicing string manipulation and hash tables.

Understanding the Problem

- A substring is anagrammatic to another if they contain the same characters, regardless of the order.
- You need to count pairs of substrings that are anagrams of each other.

Algorithm

Iterate Over All Possible Substrings: Generate all possible substrings of the given string.

Sort & Hash Substrings: For each substring, sort its characters (so anagrams will be identical) and use a hash table to count identical sorted substrings.

Count Anagrammatic Pairs: For each group of identical sorted substrings in the hash table, calculate the number of pairs. If a group has n identical substrings, the number of anagrammatic pairs is n * (n - 1) / 2 (combinatorial formula for pairs).

Implementation in Python

```python
# Onfrom collections import defaultdict

def sherlockAndAnagrams(s):
    # Hash table to store counts of sorted substrings
    substrings = defaultdict(int)

    # Generate all substrings and count their sorted forms
    for i in range(len(s)):
        for j in range(i + 1, len(s) + 1):
            substring = ''.join(sorted(s[i:j]))
            substrings[substring] += 1

    # Count anagrammatic pairs
    count = 0
    for key in substrings:
        count += substrings[key] * (substrings[key] - 1) // 2

    return count

# Example usage
```

```python
s = "abba"
print(sherlockAndAnagrams(s))  # Output will be the number of anagrammatic pairs
line Python compiler (interpreter) to run Python online.
# Write Python 3 code in this online editor and run it.
print("Hello world")
```

2. Hard Challenges

2.1 Almost Sorted: The "Almost Sorted" challenge on HackerRank asks to determine if an array can be sorted by either swapping just two elements or reversing a contiguous segment of the array. It's a great problem to test your understanding of sorting algorithms and array manipulation.

Understanding the Problem

- You are given an unsorted array.
- The goal is to check if the array can be sorted by a single swap or a single reverse of a contiguous segment.
- If possible, you need to identify the indices that need to be swapped or the start and end indices of the segment to be reversed.

Algorithm

- ❖ Check if Already Sorted: If the array is already sorted, print 'yes' and return.
- ❖ Find Unsorted Segments: Identify the segments of the array that are not in sorted order.
- ❖ Check for Swap or Reverse:
- ❖ If there is only one unsorted segment and swapping the first and last elements of this segment sorts the array, print 'yes', 'swap', and the indices.
- ❖ If the entire unsorted segment can be reversed to sort the array, print 'yes', 'reverse', and the start and end indices of the segment.
- ❖ If neither of the above conditions is met, print 'no'.

Implementation in Python

```
# Odef almostSorted(arr):
    # Function to check if the array is sorted
    def is_sorted(a):
        return all(a[i] <= a[i + 1] for i in
            range(len(a) - 1))

    sorted_arr = sorted(arr)
    if arr == sorted_arr:
        print("yes")
        return

    # Identifying the indices where arr differs from sorted_arr
    diff = [i for i in range(len(arr)) if arr[i] !=
        sorted_arr[i]]

    # Checking if it's possible to sort by swapping
    if len(diff) == 2:
        print("yes")
        print("swap", diff[0] + 1, diff[1] + 1)
        return

    # Checking if it's possible to sort by reversing
    if arr[diff[0]:diff[-1] + 1] == sorted_arr[diff[0]:diff[-1]
        + 1][::-1]:
        print("yes")
        print("reverse", diff[0] + 1, diff[-1] + 1)
        return
```

```
25
26      print("no")
27
28  # Example usage
29  almostSorted([1, 5, 4, 3, 2, 6])
30  nlidef almostSorted(arr):
31      # Function to check if the array is sorted
32      def is_sorted(a):
33          return all(a[i] <= a[i + 1] for i in range(len(a) - 1))
34
35      sorted_arr = sorted(arr)
36      if arr == sorted_arr:
37          print("yes")
38          return
39
```

```
40      # Identifying the indices where arr differs from sorted_arr
41      diff = [i for i in range(len(arr)) if arr[i] != sorted_arr[i]]
42
43      # Checking if it's possible to sort by swapping
44      if len(diff) == 2:
45          print("yes")
46          print("swap", diff[0] + 1, diff[1] + 1)
47          return
48
49      # Checking if it's possible to sort by reversing
50      if arr[diff[0]:diff[-1] + 1] == sorted_arr[diff[0]:diff[-1] + 1][
            ::-1]:
51          print("yes")
52          print("reverse", diff[0] + 1, diff[-1] + 1)
53          return
54
55      print("no")
```

```
56
57  # Example usage
58  almostSorted([1, 5, 4, 3, 2, 6])
59  ne Python compiler (interpreter) to run Python online.
60  # Write Python 3 code in this online editor and run it.
```

2.2 **Matrix Layer Rotation:** This challenge involves rotating the layers of a matrix. It's a complex problem that will test your skills in multidimensional array traversal and manipulation.

Understanding the Problem

- Given a matrix, you need to rotate each layer of the matrix r times.
- A layer in a matrix is a set of elements forming a ring at the periphery of the matrix, gradually moving towards the center.
- The rotation is circular, so elements at one end move to the beginning.

Algorithm

- Decompose Matrix into Layers: Break down the matrix into its constituent layers or rings.
- Rotate Each Layer: For each layer, rotate the elements by r positions. This rotation needs to be modular to handle cases where r is greater than the number of elements in the layer.
- Reconstruct the Matrix: After rotating each layer, reconstruct the matrix with the rotated layers.

Implementation in Python

```python
# Ondef rotateMatrixLayer(matrix, r):
    rows, cols = len(matrix), len(matrix[0])
    layers = min(rows, cols) // 2

    # Function to get the elements of a layer
    def getLayer(layer):
        elements = []
        for j in range(layer, cols - layer):
            elements.append(matrix[layer][j])
        for i in range(layer + 1, rows - layer):
            elements.append(matrix[i][cols - layer - 1])
        for j in range(cols - layer - 2, layer - 1, -1):
            elements.append(matrix[rows - layer - 1][j])
        for i in range(rows - layer - 2, layer, -1):
            elements.append(matrix[i][layer])
        return elements

    # Function to set the elements of a layer
    def setLayer(layer, elements):
        idx = 0
```

```python
        for j in range(layer, cols - layer):
            matrix[layer][j] = elements[idx]
            idx += 1
        for i in range(layer + 1, rows - layer):
            matrix[i][cols - layer - 1] = elements[idx]
            idx += 1
        for j in range(cols - layer - 2, layer - 1, -1):
            matrix[rows - layer - 1][j] = elements[idx]
            idx += 1
        for i in range(rows - layer - 2, layer, -1):
            matrix[i][layer] = elements[idx]
            idx += 1

    # Rotate each layer
    for layer in range(layers):
        layer_elements = getLayer(layer)
```

```
37          rotation = r % len(layer_elements)
38          rotated_elements = layer_elements[rotation:] + layer_elements[
                :rotation]
39          setLayer(layer, rotated_elements)
40
41      return matrix
42
43  # Example usage
44  matrix = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12], [13, 14, 15,
        16]]
45  rotated_matrix = rotateMatrixLayer(matrix, 2)
46  for row in rotated_matrix:
47      print(row)
48  line Python compiler (interpreter) to run Python online.
49  # Write Python 3 code in this online editor and run it.
50  print("Hello world")
```