PytechArena buildathon

Level 1 Beginner

Problem Set

Duration: 60 Minutes

Total Questions: 10

Points: 100

Department of Information Technology
February 10, 2026

Instructions

- **Duration:** 60 minutes
- **Submission:** Fork the provided GitHub repository and submit your solutions
- **File Naming:** Name your solution files as `q1.py`, `q2.py`, ..., `q10.py`
- **Testing:** Each question includes sample test cases. Your code will be evaluated against additional hidden test cases
- **Scoring:** Questions are weighted differently based on difficulty (indicated in each problem)
- **Code Quality:** Write clean, readable code with appropriate comments
- **Edge Cases:** Handle all edge cases mentioned in the problem statement
- **Academic Integrity:** Solutions must be your own work

Difficulty Distribution

- Questions 1-3: Easy (5 points each)
- Questions 4-6: Medium (10 points each)
- Questions 7-8: Hard (15 points each)
- Questions 9-10: Very Hard (20 points each)

Submission Guidelines

1. Fork the provided repository:
2. Create your solution files: `q1.py` through `q10.py`
3. Each file should contain only the required function(s) for that question
4. Include Team name and Team ID as a comment at the top of each file
5. Test your code with the provided test cases
6. Push your solutions to your forked repository

Evaluation Criteria

Your solutions will be evaluated based on:

- **Correctness (60%):** Passing all test cases (visible and hidden)
- **Code Quality (20%):** Readability, comments, and proper naming
- **Efficiency (10%):** Time and space complexity
- **Edge Case Handling (10%):** Robustness of solution

Good Luck!

1 Question 1: String Transformer (5 points)

Difficulty: Easy

Write a function `transform_string(s)` that takes a string and returns a new string where:

- Vowels (a, e, i, o, u) are converted to uppercase
- Consonants are converted to lowercase
- Digits and special characters remain unchanged

Function Signature

```python
def transform_string(s: str) -> str:
    """
    Transform string based on character type.

    Args:
        s: Input string

    Returns:
        Transformed string
    """
    pass
```

Sample Test Cases

```python
# Test Case 1
assert transform_string("Hello World!") == "hEllO wOrld!"

# Test Case 2
assert transform_string("Python123") == "pythOn123"

# Test Case 3
assert transform_string("AEIOU") == "AEIOU"

# Test Case 4
assert transform_string("bcdfg") == "bcdfg"

# Test Case 5
assert transform_string("") == ""
```

Constraints

- $0 \leq$ length of string ≤ 1000
- String may contain letters, digits, spaces, and special characters

2 Question 2: List Analyzer (5 points)

Difficulty: Easy

Write a function `analyze_list(numbers)` that takes a list of integers and returns a dictionary with the following keys:

- `'sum'`: Sum of all numbers
- `'mean'`: Average of all numbers (rounded to 2 decimal places)
- `'even_count'`: Count of even numbers
- `'odd_count'`: Count of odd numbers

Function Signature

```
def analyze_list(numbers: list) -> dict:
    """
    Analyze a list of integers.

    Args:
        numbers: List of integers

    Returns:
        Dictionary with analysis results
    """
    pass
```

Sample Test Cases

```
# Test Case 1
assert analyze_list([1, 2, 3, 4, 5]) == {
    'sum': 15,
    'mean': 3.0,
    'even_count': 2,
    'odd_count': 3
}

# Test Case 2
assert analyze_list([10, 20, 30]) == {
    'sum': 60,
    'mean': 20.0,
    'even_count': 3,
    'odd_count': 0
}

# Test Case 3
assert analyze_list([7]) == {
    'sum': 7,
    'mean': 7.0,
    'even_count': 0,
    'odd_count': 1
}
```

Constraints

- List will always contain at least one integer

- $-1000 \leq$ each number ≤ 1000
- Round mean to 2 decimal places

3 Question 3: Pattern Matcher (5 points)

Difficulty: Easy

Write a function `count_pattern(text, pattern)` that counts how many times a pattern appears in the text. The search should be **case-insensitive** and **overlapping matches should be counted**.

Function Signature

```python
def count_pattern(text: str, pattern: str) -> int:
    """
    Count occurrences of pattern in text (case-insensitive, overlapping).

    Args:
        text: Input text
        pattern: Pattern to search for

    Returns:
        Count of pattern occurrences
    """
    pass
```

Sample Test Cases

```python
# Test Case 1
assert count_pattern("aaaa", "aa") == 3  # Overlapping: positions 0, 1, 2

# Test Case 2
assert count_pattern("Hello World", "o") == 2

# Test Case 3
assert count_pattern("Programming", "GRAM") == 1  # Case-insensitive

# Test Case 4
assert count_pattern("abcabc", "abc") == 2

# Test Case 5
assert count_pattern("test", "xyz") == 0
```

Constraints

- Pattern length ≥ 1
- Text length ≥ 0
- Both text and pattern contain only alphanumeric characters and spaces

4 Question 4: List Deduplicator (10 points)

Difficulty: Medium

Write a function `remove_duplicates(lst)` that removes duplicates from a list while **preserving the original order** of first occurrences.

Constraint: You **CANNOT** use Python's `set()` or `dict.fromkeys()` methods.

Function Signature

```
def remove_duplicates(lst: list) -> list:
    """
    Remove duplicates while preserving order (no set() or dict.fromkeys()).

    Args:
        lst: Input list

    Returns:
        List with duplicates removed
    """
    pass
```

Sample Test Cases

```
# Test Case 1
assert remove_duplicates([1, 2, 2, 3, 3, 3, 4]) == [1, 2, 3, 4]

# Test Case 2
assert remove_duplicates(['a', 'b', 'a', 'c', 'b']) == ['a', 'b', 'c']

# Test Case 3
assert remove_duplicates([1, 1, 1, 1]) == [1]

# Test Case 4
assert remove_duplicates([]) == []

# Test Case 5
assert remove_duplicates([5, 4, 3, 2, 1]) == [5, 4, 3, 2, 1]
```

Constraints

- List can contain integers, strings, or mixed types
- $0 \leq$ list length ≤ 1000
- **Cannot use:** `set()`, `dict.fromkeys()`

5 Question 5: Smart Calculator (10 points)

Difficulty: Medium

Write a function `calculate(expression)` that evaluates a mathematical expression given as a string. The expression contains:

- Integers (positive or negative)
- Operators: +, -, *, /
- Spaces (should be ignored)

Follow standard operator precedence (* and / before + and -). Return the result as a float rounded to 2 decimal places.

Constraint: You **CANNOT** use `eval()`, `exec()`, or any similar built-in evaluation functions.

Function Signature

```python
def calculate(expression: str) -> float:
    """
    Evaluate mathematical expression without using eval().

    Args:
        expression: Mathematical expression as string

    Returns:
        Result rounded to 2 decimal places
    """
    pass
```

Sample Test Cases

```python
# Test Case 1
assert calculate("2 + 3") == 5.0

# Test Case 2
assert calculate("10 - 5 * 2") == 0.0

# Test Case 3
assert calculate("20 / 4 + 3 * 2") == 11.0

# Test Case 4
assert calculate("100 / 3") == 33.33

# Test Case 5
assert calculate("5") == 5.0
```

Constraints

- Expression will always be valid
- No parentheses in the expression
- Division by zero will not occur
- **Cannot use:** `eval()`, `exec()`, `compile()`

6 Question 6: Code Debugger (10 points)

Difficulty: Medium

The following function is supposed to find all prime numbers up to `n`, but it has **multiple bugs**.

Your task:

1. Identify all the bugs
2. Write the corrected version
3. Add a docstring explaining what the function does

Buggy Code

```
def find_primes(n):
    primes = []
    for num in range(2, n):
        is_prime = True
        for i in range(2, num):
            if num % i = 0:
                is_prime = False
        if is_prime:
            primes.append(num)
    return primes
```

Expected Output

```
# Test Case 1
assert find_primes(10) == [2, 3, 5, 7]

# Test Case 2
assert find_primes(20) == [2, 3, 5, 7, 11, 13, 17, 19]

# Test Case 3
assert find_primes(2) == []

# Test Case 4
assert find_primes(3) == [2]

# Test Case 5
assert find_primes(30) == [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

Bugs to Find

Write your corrected code in `q6.py` and include comments explaining each bug you found.

Constraints

- $2 \leq n \leq 1000$
- Function should be reasonably efficient

7 Question 7: Matrix Spiral (15 points)

Difficulty: Hard

Write a function `spiral_order(matrix)` that returns all elements of a 2D matrix in spiral order (clockwise from outside to inside).

Function Signature

```python
def spiral_order(matrix: list[list[int]]) -> list[int]:
    """
    Return matrix elements in spiral order.

    Args:
        matrix: 2D list of integers

    Returns:
        List of integers in spiral order
    """
    pass
```

Sample Test Cases

```python
# Test Case 1
matrix1 = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
assert spiral_order(matrix1) == [1, 2, 3, 6, 9, 8, 7, 4, 5]

# Test Case 2
matrix2 = [
    [1, 2, 3, 4],
    [5, 6, 7, 8],
    [9, 10, 11, 12]
]
assert spiral_order(matrix2) == [1, 2, 3, 4, 8, 12, 11, 10, 9, 5, 6, 7]

# Test Case 3
matrix3 = [[1]]
assert spiral_order(matrix3) == [1]

# Test Case 4
matrix4 = [[1, 2, 3]]
assert spiral_order(matrix4) == [1, 2, 3]

# Test Case 5
matrix5 = [
    [1],
    [2],
    [3]
]
assert spiral_order(matrix5) == [1, 2, 3]
```

Constraints

- $1 \leq \text{rows} \leq 100$

- $1 \leq$ columns ≤ 100
- Matrix contains integers

8 Question 8: Palindrome Partitioning (15 points)

Difficulty: Hard

Write a function `is_palindrome_possible(s)` that determines if a string can be rearranged to form a palindrome. Then write `make_palindrome(s)` that returns one possible palindrome arrangement if it exists, otherwise returns an empty string.

Constraint: You **CANNOT** use string reversal (`[::-1]`) or `reversed()` function.

Function Signatures

```python
def is_palindrome_possible(s: str) -> bool:
    """
    Check if string can be rearranged into a palindrome.

    Args:
        s: Input string

    Returns:
        True if palindrome arrangement exists, False otherwise
    """
    pass

def make_palindrome(s: str) -> str:
    """
    Create a palindrome from string if possible (no [::-1] or reversed()).

    Args:
        s: Input string

    Returns:
        A valid palindrome arrangement or empty string
    """
    pass
```

Sample Test Cases

```python
# Test Case 1
assert is_palindrome_possible("aab") == True
assert make_palindrome("aab") in ["aba", "baa"]  # Either is valid

# Test Case 2
assert is_palindrome_possible("abc") == False
assert make_palindrome("abc") == ""

# Test Case 3
assert is_palindrome_possible("aabbcc") == True
# Valid: "abccba", "bacbcb", "cabcac", etc.

# Test Case 4
assert is_palindrome_possible("a") == True
assert make_palindrome("a") == "a"

# Test Case 5
assert is_palindrome_possible("aabbccd") == False
```

Constraints

- String contains only lowercase letters

- $1 \leq$ string length ≤ 1000
- **Cannot use:** `[::-1]`, `reversed()`

9 Question 9: Nested List Flattener (20 points)

Difficulty: Very Hard

Write a function `flatten(nested_list)` that flattens a deeply nested list of arbitrary depth. The list can contain integers, strings, or other lists.

Additionally, write `flatten_with_depth(nested_list, max_depth)` that flattens only up to a specified depth level.

Function Signatures

```python
def flatten(nested_list: list) -> list:
    """
    Completely flatten a nested list of arbitrary depth.

    Args:
        nested_list: List with possible nested lists

    Returns:
        Flattened list
    """
    pass

def flatten_with_depth(nested_list: list, max_depth: int) -> list:
    """
    Flatten nested list up to max_depth levels.

    Args:
        nested_list: List with possible nested lists
        max_depth: Maximum depth to flatten (1 = flatten one level)

    Returns:
        Partially flattened list
    """
    pass
```

Sample Test Cases

```python
# Test Case 1
assert flatten([1, [2, 3], [4, [5, 6]]]) == [1, 2, 3, 4, 5, 6]

# Test Case 2
assert flatten([1, [2, [3, [4, [5]]]]]) == [1, 2, 3, 4, 5]

# Test Case 3
assert flatten([]) == []

# Test Case 4
assert flatten([1, 2, 3]) == [1, 2, 3]

# Test Case 5
assert flatten_with_depth([1, [2, [3, [4]]]], 1) == [1, 2, [3, [4]]]

# Test Case 6
assert flatten_with_depth([1, [2, [3, [4]]]], 2) == [1, 2, 3, [4]]

# Test Case 7
assert flatten(['a', ['b', ['c']]]) == ['a', 'b', 'c']
```

Constraints

- List can be nested to arbitrary depth
- Elements can be integers, strings, or lists
- $0 \leq max\_depth \leq 100$
- Use recursion for an elegant solution

10 Question 10: Transaction Validator (20 points)

Difficulty: Very Hard

You are given a list of bank transactions. Each transaction is a dictionary with:

- `'id'`: Unique transaction ID (string)
- `'type'`: Either `'credit'` or `'debit'`
- `'amount'`: Transaction amount (float)
- `'timestamp'`: Unix timestamp (integer)
- `'category'`: Category like `'food'`, `'salary'`, etc.

Write a function `analyze_transactions(transactions, initial_balance)` that returns a dictionary with:

- `'final_balance'`: Balance after all transactions
- `'largest_expense'`: Category with highest total debits
- `'monthly_summary'`: Dictionary mapping month (YYYY-MM) to net change
- `'suspicious'`: List of transaction IDs where a single debit $> 50\%$ of balance at that point

Function Signature

```python
def analyze_transactions(transactions: list[dict],
                         initial_balance: float) -> dict:
    """
    Analyze bank transactions and detect suspicious activity.

    Args:
        transactions: List of transaction dictionaries
        initial_balance: Starting account balance

    Returns:
        Dictionary with analysis results
    """
    pass
```

Sample Test Case

```python
transactions = [
    {'id': 'T1', 'type': 'credit', 'amount': 1000,
     'timestamp': 1704067200, 'category': 'salary'},  # Jan 1, 2024
    {'id': 'T2', 'type': 'debit', 'amount': 600,
     'timestamp': 1704153600, 'category': 'rent'},    # Jan 2, 2024
    {'id': 'T3', 'type': 'debit', 'amount': 50,
     'timestamp': 1704240000, 'category': 'food'},    # Jan 3, 2024
    {'id': 'T4', 'type': 'credit', 'amount': 500,
     'timestamp': 1706832000, 'category': 'salary'},  # Feb 2, 2024
    {'id': 'T5', 'type': 'debit', 'amount': 800,
     'timestamp': 1706918400, 'category': 'rent'},    # Feb 3, 2024
]

result = analyze_transactions(transactions, 500)
```

```python
assert result['final_balance'] == 550.0
assert result['largest_expense'] == 'rent'
assert result['monthly_summary'] == {'2024-01': 350.0, '2024-02': -300.0}
assert result['suspicious'] == ['T2', 'T5']
# T2: 600 debit when balance was 1500 (40% - not suspicious)
# Actually T2 is NOT suspicious. Let me recalculate:
# Initial: 500, After T1: 1500, T2 debit 600 = 40% (< 50%, OK)
# After T2: 900, T3 debit 50 = 5.5% (OK)
# After T3: 850, After T4: 1350, T5 debit 800 = 59% (> 50%, SUSPICIOUS)
assert result['suspicious'] == ['T5']
```

Constraints

- Transactions are sorted by timestamp
- $1 \leq$ number of transactions ≤ 1000
- All amounts are positive
- Timestamps are valid Unix timestamps
- Initial balance ≥ 0