CS 171: Computer Programming I

Homework Assignment #6

Assignment Objectives

The purpose of this homework assignment is to practice algorithm thinking and writing recursive functions.

Assignment Notes

This assignment can be and must be solved using only the materials that have been discussed in class at the time this assignment is given. Do not look for alternative methods that have not been covered as part of this course.

For this assignment, you are only allowed to use recursion, conditional branching, and basic strings/lists (no string/list methods or functions allowed, with the exception of the len() built-in function). Do not use any global variables. Do not use loops. Do not change the function header given in the assignment instructions.

Please make sure to submit well-written programs for the programming tasks. Good identifier names, useful comments, and spacing will be some of the criteria that will be used when grading this assignment.

Materials You Are Allowed To Use:

- Lecture slides for Week 1 – 7 (with the exceptions noted above)
- Think Python Chapters 1 – 9 (with the exceptions noted above)

Academic Honesty

This is an individual assignment. No collaboration is allowed. You must be the sole original author of the solution you submit. You must compose all programs and written material yourself. All material taken from outside sources must be appropriately cited.

Overview

In the assignment, you will be practicing designing and implementing recursive functions. There won't be any input/output from the user, and you are only required to write the three functions described below. There is no need to write a main script, but feel free to write one to test your functions. You should fully test your functions and make sure they work correctly.

Part 1: Flattened lists

In week 5, we studied lists. We learned that lists could contain other lists; this is called a nested list. Flattening a list means converting a nested list structure into a single, one-dimensional list containing all the elements in sequence, without any nesting

For example, if you have a nested list like:

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

Flattening it would produce:

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

Write a function **`flattenList(myList)`**. This function receives a list, and it returns a flat list.

Test Examples

```
flattenList([]) should return []
flattenList([1, 2, 3]) should return [1, 2, 3]
flattenList([1, [2, 3], [4, 5]]) should return [1, 2, 3, 4, 5]
flattenList([1, [2, [3, 4], 5], 6]) should return [1, 2, 3, 4, 5, 6]
flattenList([1, [2, [3, [4]]], 5]) should return [1, 2, 3, 4, 5]
flattenList([[], [[], []]]) should return []
```

Part 2: The largest common prefix

Write a function, **`longestCommonPrefix(strings)`**, that takes a list of strings and returns the longest common prefix string among them. If there is no common prefix, return an empty string.

Example 1:
Input: `["flower", "flow", "flight", "flush", "flat"]`
Output: `"fl"`

Example 2:
Input: `["dog", "racecar", "car"]`
Output: `""` (There is no common prefix among the input strings)

Example 3:
Input: `["interview", "intermediate", "internal", "internet"]`
Output: `"inter"`

Tips:

- Think about how you can compare the characters of the strings at the same position recursively.
- If you have two strings, think about how you can find their common prefix character by character. How can you use this idea to build your recursive solution?
- Consider edge cases such as an empty list or a list with only one string.
- Make sure to handle cases where there is no common prefix.

Test Examples

```
longestCommonPrefix([] should return ""
longestCommonPrefix([""]) should return ""
longestCommonPrefix(["single"]) should return "single"
longestCommonPrefix(["single", "singer"]) should return "sing"
```

```
longestCommonPrefix(["flower", "flow", "flight"]) should return "fl"
longestCommonPrefix(["interview", "intermediate", "internal", "internet"])
                    should return "inter"
longestCommonPrefix(["dog", "racecar", "car"]) should return ""
longestCommonPrefix(["apple", "banana", "cherry"]) should return ""
longestCommonPrefix(["same", "same", "same", "same"]) should return "same"
longestCommonPrefix(["@hashtag", "@hash", "@has"])) should return "@has"
```

Part 3: The Hailstone sequence

The Hailstone sequence is a sequence of integer numbers generated from a starting positive integer, following a simple set of rules:

1. Start with any positive integer **`number`**.
2. If **`number`** is even, divide it by 2: n → n / 2
3. If **`number`** is odd, multiply it by 3 and add 1: n → 3n + 1
4. Repeat the process with the new value of **`number`**.
5. The sequence continues until **`number`** becomes **1**.

Here are a couple of examples:

- Starting with `number = 5`, the Hailstone sequence is: `5, 16, 8, 4, 2, 1`
- Starting with `number = 8`, the Hailstone sequence is: `8, 4, 2, 1`

Write a function **`hailstoneSequence(number)`** that takes in a positive integer number and returns a list that contains the corresponding Hailstone sequence.

Test Examples

```
hailstoneSequence(1) should return [1]
hailstoneSequence(4) should return [4, 2, 1]
hailstoneSequence(6) should return [6, 3, 10, 5, 16, 8, 4, 2, 1]
hailstoneSequence(7) should return
     [7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1]
hailstoneSequence(11) should return
     [11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1]
hailstoneSequence(20)should return
     [20, 10, 5, 16, 8, 4, 2, 1]
```

Guidelines

- Your code should be in one file:
  - **HW06_recursion.py**
- You may use Pothole Case or Camel Case, but be consistent in your file.
- Your name, date, and program description should be at the top of each file.
- Add comments to clarify how your code works.

What to Submit

- Assignments must be submitted via Blackboard Learn.
  - Please note that assignments submitted via email will not be accepted.
- For this assignment, you must submit your source code, that is, your **.py** file:

    - **HW06_recursion.py** as a single file.
    - Do not submit files in any other formats: if you do, your assignment will not be graded.
- Optional: You may submit a **readme.txt** text file with any comments for the grader.

Grading Rubric

All rubric items are graded on the following scale:

- Meets all Requirements: 100% (Full Credit)
- Good with minor issues: 66% of points rounded up to the nearest point.
- Significant issues: 33% of points round up to the nearest point.
- Not Attempted / Does not meet requirements: 0 points.

Course Assistants may not award any other percentage of points.

Rubric Item	**Points**
The `flattenList(myList)` function produces expected results when tested	9
The `flattenList(myList)` function base case(s) correctly identified and coded	9
The `flattenList(myList)` function recursive step(s) correctly identified and coded	9
The `longestCommonPrefix(strings)` function produces expected results when tested	12
The `longestCommonPrefix(strings))=` function base case(s) correctly identified and coded	12
The `longestCommonPrefix(strings)` function recursive step(s) correctly identified and coded	12
The `hailstoneSequence(number)` function produces expected results when tested	9
The `hailstoneSequence(number)` function base case(s) correctly identified and coded	9
The `hailstoneSequence(number)` function recursive step(s) correctly identified and coded	9
Code Style: Reasonable and Consistent Variable Names	3
Code Style: Reasonable Comments in code / Header Comment	3
General Coding Style (spacing and ease to read)	3
Correct File Name (checked by the autograder)	1
TOTAL	**100**