CS 171
Week 7 Lab

Overview

This lab consists of a collection of problems that you may solve in any order. To earn full credit, you must submit solutions to all problems at one time.

Important:

- You must be present in the lab session to earn credit
    - If you are recorded as absent but still complete the work, you will not earn credit. If you arrived after the CA/TA took attendance, be sure to check in so you are recorded as present/late.
    - If you miss the lab meeting, contact your instructor via email right away to request a make-up session.
- You may collaborate with at most one lab partner; Feel free to switch partners next week if you wish
- Each student must submit their own work to Gradescope for credit, even if you worked with a partner
- At most 100 points are possible. To get a combined score, submit all of your solutions at once.
- Please only use features of the language that have been covered in reading and lectures
    - Solutions which make use of language features not yet covered will not receive credit.
    - Students whose submissions repeatedly make use of additional skills not covered in the course so far may be suspected of violating academic integrity.
    - *Even if you know the language already, please work within the subset of the language covered up to this point in the course.*
- You may not use artificial intelligence tools to solve these problems.
- Your lab work is due at the end of the lab period. Late work will be accepted with a 30-point penalty if it is submitted within 12 hours of the end of your lab.

Helpful to Know:

- The last score earned will be recorded for your lab score
- To get a combined lab score, you must submit multiple files at the same time
- Lab assistants are here for your support. Ask them questions if you are stuck!

Materials to Use:

- Lecture slides for Weeks 1 – 7
- Think Python Chapters 1 – 10
- Note: Solutions to this week’s problems should use recursion. Solutions which use loops will not earn credit, even if the auto-grader scores them as being correct.

Problem A

Submission File: `lab07a.py`

The Fibonacci sequence begins with 0 and then 1 follows. All subsequent values are the sum of the previous two, for example: 0, 1, 1, 2, 3, 5, 8, 13. Complete the **fibonacci(n)** function, which takes in an index (starting at 0), n, and returns the nth value in the sequence. Any negative index values should return -1.

Ex: If the input is:

```
7
```

the output is:

```
fibonacci(7) is 13
```

Problem B

Submission File: `lab07b.py`

Write a recursive function, **`decimalToBinary(number)`**, that takes a positive integer number (in base 10) as its parameter and returns a string containing its binary representation.

Recall that in order to convert a decimal number to binary, we repeatedly divide by 2 and then read the remainders backwards.

Converting 11 to Binary

- 11 divided by 2 is 5 remainder 1
- 5 divided by 2 is 2 remainder 1
- 2 divided by 2 is 1 remainder 0
- 1 divided by 2 is 0 remainder 1

The binary representation of 11 is 1011, found by reading the remainders from bottom to top.

Before you attempt to write the code for this problem, make sure that you identify the base case and the recursive case. Let the recursive case solve a simpler problem.

Below are some examples:

- `decimalToBinary(8) returns “1000”`
- `decimalToBinary(2) returns “10”`
- `decimalToBinary(25) returns “11001”`

Problem C

Submission File: `lab07c.py`

Write a recursive function, **`is_balanced(expression)`** to determine whether the parentheses in a string containing some expression is balanced. An expression is considered balanced if every opening parenthesis (has a corresponding closing parenthesis) and the pairs are properly nested. A string containing no parentheses is considered balanced.

Examples:

- `is_balanced("")  # Returns True`
- `is_balanced("(())")  # Returns True`
- `is_balanced("()()")  # Returns True`
- `is_balanced("(()")   # Returns False`
- `is_balanced(")(")    # Returns False`
- `is_balanced("(a(b)c)(d)") # Returns True`
- `is_balanced("hello (world)")  # Returns True`

Notes:

- The function **must use recursion** to solve the problem. No loops (for, while, etc.) are allowed.
- The function should ignore non-parenthesis characters (e.g., letters, numbers, spaces)
- You may implement one or more "helper functions" if needed. In the example below, check_complementary_chars(str, d) is a helper function which assists is_palindrome(str).

```
def check_complementary_chars(str, d):
    # Checks that characters at a distance of d from
    # the beginning and end of a string are the same
    return str[d] == str[(d+1)*-1]

def is_palindrome(str):
    # Check each pair of characters from the outside to the middle
    for d in range(0, len(str)//2):
        if check_complementary_chars(str, d) == False:
            return False
    return True
```