CMPSC 132 : Fall 2024 Project

Please complete the following 3 coding questions. Upload your code in Gradescope. There is no autograder in Gradescope. Test your code using some of the provided test cases. Please note that this not a comprehensive list.

Note:

1. Complete by **Wednesday**, **Dec 11th, 11:59 PM**.
2. There is **NO automatic 24-hour grace period**.
3. This project is worth 5% of your grade.

1. Bank Account System with Tiered Interest Rates (10 Points)

Write a Python program that simulates a simple bank account system. This allows you to manage two types of accounts: **Savings Account** and a **Checking Account**. The program uses the concept of classes in Python, which lets us organize the code in a way that makes it easy to manage and expand.

Overview of the Program:

The program has three main parts:

i. A base class called **BankAccount** handles the common features of all bank accounts.
ii. A class for **Savings Accounts** that calculates interest on the balance.
iii. A class for **Checking Accounts** that allows for overdrafts (spending more than the available balance), but with a fee.

We also have **tiered interest rates** in the **SavingsAccount**, meaning the interest rate depends on how much money you have in the account. The more money you have, the higher the interest rate you get.

Classes in the Program:

i. BankAccount Class (Base Class)

The BankAccount class is the base for both the **Savings Account** and the **Checking Account.** It contains shared functionality like deposits and withdrawals.

- **Account Holder's Name:** This is the person who owns the account.
- **Balance**: This shows how much money is in the account.

Methods (functions) in this class:

- `deposit(amount)`: This method adds money to the account. It checks that the amount is positive before depositing.
- `withdraw(amount)`: This method subtracts money from the account, if there is enough balance.
- `get_balance()`: This returns the current balance from the account.
- `__str__()`: Provides a string representation of the account (useful for printing).

ii. SavingsAccount: `SavingsAccount`

This class inherits from `BankAccount` and includes additional feature for savings accounts, such as the ability to apply interest to the balance.

Tiered Interest Rates: The interest rate is dynamic, meaning it changes based on the account balance.

- **3% interest** for balances equal to or less than $1,000.
- **5% interest** for balances between $1,001 and $5,000.
- **7% interest** for balances above $5,001.

Method:

- `apply_interest()`: This method calculates the interest based on the balance and adds it to the account. The rate depends on how much money is in the account (as described above).
  Note: You are not required to use the interest rate formula. Simply add interest to the balance.
  For example, if balance is 1000,
  interest = 0.03 * 1000. #Add this amount to the balance
  balance += interest

iii. CheckingAccount: `CheckingAccount`

The `CheckingAccount` class also inherits from `BankAccount`. You can deposit money into a checking account and withdraw money from it. This is for accounts where you can spend more money than you have (known as an overdraft). However, there's a limit to how much you can go negative, and if you do, you get charged a fee.

Overdraft Feature:

- The checking account allows you to withdraw money even if you don't have enough, but only up to a certain negative balance limit (like -$500).

- If your balance goes below zero, you are charged an overdraft fee of $25.

Method:

- withdraw(amount): This method allows withdrawals even if the balance is negative (up to the overdraft limit). If you go negative, it applies a $25 overdraft fee.

Test Code

```
def test_savings_account_deposit(self):
    savings = SavingsAccount("John Doe", 500)
    savings.deposit(200)  # Deposit $200
    self.assertEqual(savings.get_balance(), 700)  # New balance should be 700

def test_savings_account_interest_below_1000(self):
    savings = SavingsAccount("John Doe", 800)
    savings.apply_interest()  # Interest at 3% for balance less than 1000
    self.assertAlmostEqual(savings.get_balance(), 824, places=2)  # 800 + 3% of 800 = 824

def test_savings_account_interest_between_1000_and_5000(self):
    savings = SavingsAccount("Jane Smith", 1500)
    savings.apply_interest()  # Interest at 5% for balance between 1000 and 5000
    self.assertAlmostEqual(savings.get_balance(), 1575, places=2)  # 1500 + 5% of 1500 = 1575

def test_checking_account_overdraft(self):
    checking = CheckingAccount("Alex Lee", 200)
    checking.withdraw(300)  # Withdraw $300, which goes negative
    self.assertEqual(checking.get_balance(), -125)  # Balance should be -125 after overdraft and fee
```

2. **(10 points)** You are given a **singly linked list**, where each node has:
   - An **integer value**.
   - A **pointer** to the next node in the list.

   Your task is to detect if the list contains a **cycle**. A **cycle** happens when a node's next pointer points back to one of the previous nodes in the list, forming a loop.
   - If a **cycle exists**, return the **length of the cycle**, which is the number of nodes involved in the cycle.
   - If there is **no cycle**, return **-1**.

   Input:
   - A singly linked list where each node has a value and a pointer to the next node.

   Output:
   - If a cycle is found, return the length of the cycle.
   - If no cycle is found, return -1.

   Example:
   Input:
   1 → 2 → 3 → 4 → 5→ 6 →7 →4
   Here, the node with value 4 points back to itself, forming a cycle.
   Output:
   4

   Starter Code:

```
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def find_cycle_length(head):
    pass
```

Task:

1. Implement the function find_cycle_length(head) to detect the cycle and return its length or -1 if no cycle is present.
2. Create a sample linked list for testing purposes.

3. **(5 Points):** You are given an array of integers nums and an integer target. Your task is to implement a function that finds the **smallest index** i such that **all elements** after this index (i.e., from index i+1 to the end) are **strictly greater** than the target.
If no such index exists, return -1.

Example 1:
Input:
nums = [1, 3, 3, 5, 7]
target = 3
Output:
3
Explanation:

- Starting from index 3, the elements after it are 5 and 7, both of which are strictly greater than 3. Therefore, the function should return index 3.

Example 2:
Input:
nums = [1, 3, 5, 7]
target = 8
Output:
-1
Explanation:

- There is no index where all numbers after it are strictly greater than 8. Hence, the function should return -1.

Example 3:
Input:
nums = [5, 6, 7, 8, 9]
target = 5
Output:
4
Explanation:

- Starting from index 4, the element after it (none) is greater than 5. Therefore, index 4 is the valid answer.

Example 4:
Input:
nums = [1, 2, 3, 4, 5]
target = 5
Output:
-1
Explanation:

- All elements in the array are either less than or equal to 5. There is no index where all the following elements are greater than 5. Thus, the function should return -1.