CMSC 331 - Principles of programming language
Homework - 4

1. (15 pts) Write a Ruby program that defines a method called filtered_sum, which accepts an array of integers, filters out all integers less than 10 or greater than 100, and returns the sum of the remaining integers. The method should return 0 if the array is empty or if no numbers meet the filtering criteria.
   Samples:
   filtered_sum([-12, -15, 155, 250, 2950]) #output: 0
   filtered_sum([1, 5, 10, 15, 25, 150]) #output: 50
   filtered_sum([]) #output: 0

2. (15 pts) Write a Ruby program that takes a single string and returns a new string containing only the characters at prime indices (indices 2, 3, 5, 7, 11, 13, ...) of the input string.
   Samples:
   Input:
   input_string_1 = "123456789"
   Output:
   "2357"

   Input:
   input_string_2 = "Check for primes"
   Output:
   "hekfpi"

3. (20 pts) Write a Ruby program that takes a list of strings. Then removes all vowels from the strings. Then, find the shortest string that is a palindrome. If no palindrome exists, return nil.
   Note: If there are two or more shortest strings of the same length that are palindromes after removing vowels then return the first occurrence.
   Samples:
   Input:
   strings = ["car", "clock", "machine", "madam", "magnet"]
   Output:
   mdm

   Input:
   strings = ["apple", "banana", "orange", "grape", "kiwi"]
   Output:
   nil

Input:
strings = ["hello", "world", "abcba", "roar", "deed"]
Output:
rr

4. (50 pts) Design and implement a bank management system in Ruby that handles multiple bank accounts identified by unique account numbers. Your system should be able to perform operations such as deposits, withdrawals, balance inquiries, account with highest balance, and average balance of the bank along with proper error handling

   Requirements:

   Your implementation should consist of two main classes: **BankAccount** and **Bank** as follows:

   BankAccount class:

   It will be used internally by the **Bank** class. The **BankAccount** class will hold details for individual accounts, such as the account number, name, and balance.

   - Initializer:
     - The class should have an initializer that accepts an account number, an account holder's name, and an initial balance. The initial balance should not be negative.
   - Accessibility:
     - The **account_number** and **name** should be read-only after the account is initialized, ensuring that these details cannot be modified.
     - The **balance** should be both readable and writable, allowing for updates through deposits and withdrawals.

   Bank Class:
   - Initializer: Starts with an empty list of accounts.
   - Methods:
     - **add_account(account_number, name, initial_balance)**: Adds a new **BankAccount** with a unique account number, name, and initial balance to the bank.
       The initial balance must be non-negative; otherwise, returns "Invalid initial balance."

Ensures that account numbers are unique(if the account number is not unique, then return “Account number already exists”).
Ensure account number length should be 10. In case of any other length, it should return “Account number length should be 10”)

- **deposit(account_number, amount)**: Deposits the specified amount into the account associated with the given account number and return account balance after deposit. Returns "Account not found" if no account matches the account number.
- **withdraw(account_number, amount)**: Withdraws the specified amount from the account associated with the given account number and returns the account balance after withdrawal. Returns "Account not found" if no account matches the account number. Returns “Insufficient balance” if withdrawal balance is greater than account balance.
- **check_balance(account_number)**: Returns the balance of the account associated with the given account number. Returns "Account not found" if no account matches the account number.
- **highest_balance**: Identifies the account with the highest balance and returns its account number. Returns "No accounts" if the bank has no accounts.
- **average_balance**: Calculates and returns the average balance across all accounts in the bank. Returns 0 if there are no accounts.

Your Tasks:

- Implement the **BankAccount** and **Bank** classes with the functionality and error handling as described above.
- Perform the below operations:
    - Create three accounts with unique account numbers and initial balances.
    - Perform operations to generate errors while creating account(by giving input as a non-unique account number and negative balance)
    - Perform a mix of deposits and withdrawals, including scenarios expected to generate error messages (e.g., exceeding account balance, Account not found, using negative amounts).
    - Inquire the balances of specific accounts after transactions. Include scenarios expected to generate error messages(Account not found) while checking balance.

- Determine the account with the highest balance.
- Compute the average balance of all accounts.

You can refer the below sample operations(You can use same or different operations but you have to cover all the above scenarios) :

```
# Creating three accounts
puts bank.add_account("1234567890", "Harry", 20000) # Should create account
puts bank.add_account("9988998899", "Nicholas", 25000)   # Should create account
puts bank.add_account("4455445544", "Charlie", 38000) # Should create account
# Errors while creating accounts
puts bank.add_account("1234567890", "Duplicate Test", 3500)
# Account number already exists
puts bank.add_account("12345678", "Acoount number errror test", 1000)        # Account number length should be 10
puts bank.add_account("9999944444", "Negative number test", -10000)
# Invalid initial balance
# deposits and withdrawals along with error handling
puts bank.deposit("1234567890", 3000)  # Deposit into Harry's account and shows updated balance 23000
puts bank.withdraw("1234567890", 4500) # Withdraw from Harry's account and shows updated balance 18500
puts bank.withdraw("1234567890", 28000) # Insufficient balance
puts bank.deposit("1111111111", 1200)   # Account not found
# Balance Inquiry along with error handling
puts bank.check_balance("1234567890") # Should show updated balance 18500
puts bank.check_balance("1111111111")  # Account not found
# Determining the account with the highest balance
puts "Account number with highest balance: #{bank.highest_balance}"
# Account number with highest balance: 4455445544
# Computing the average balance of all accounts
puts "Average balance: #{bank.average_balance}"
#Average balance: 27166.666666666668
```

Submission Instructions:
Submit your code and attach the screenshots of code execution along with the outputs.