Lab 3 - Assignment 1

Author: Aashay Pawar
NUID: 002134382
Mail: pawar.aa@northeastern.edu

In this assignment, you need to write programs to solve the following problems. The source project of the problems is shared on the Canvas. Please import it to the Eclipse to write your codes.

Problem 1

Credit card numbers follow certain patterns. A credit card number must have between 13 and 16 digits. It must start with:

- 4 for Visa cards
- 5 for Master cards
- 37 for American Express cards
- 6 for Discover cards

Solution:

The problem at hand revolves around verifying the legitimacy of a credit card number through pattern checks and validation procedures. The objective is to ascertain whether a given lengthy number corresponds to a valid credit card. This verification process holds significance in promptly providing users with feedback regarding the validity of their card information. As a result, users can avoid submitting erroneous details repeatedly, thereby lessening the server load. By addressing this problem, the occurrence of invalid requests impacting the API rate limit is minimized, leading to enhanced overall efficiency.

The Luhn algorithm, also known as the modulus 10 or mod 10 algorithm, is a widely used method to validate identification numbers, including credit card numbers. It involves calculating a check digit based on a partial account number, which is then appended as the rightmost digit of the complete account number. By using this algorithm, valid numbers can be distinguished from random digit combinations.

Implementing the Luhn algorithm may present some challenges:

1. Understanding the algorithm: Although the Luhn method is a simple arithmetic formula, comprehending its workings may require some effort. Familiarizing oneself with the algorithm and its steps is crucial before utilizing it effectively.
2. Handling different formats: Validating credit card numbers can be complex due to variations in formats used by different card companies. It is important to investigate and accommodate various formats to ensure accurate validation.
3. Input validation: Prior to performing calculations, it is essential to validate the input. This involves checking for non-digit characters, ensuring appropriate input length, and handling such cases to prevent errors.

While the Luhn algorithm is widely employed, it does have limitations. It can only detect single-digit errors and adjacent number transpositions, but it cannot identify transpositions involving the first and last valid characters.

Other considerations include:

1. Adapting to evolving requirements: As credit card industry standards change over time, software utilizing the Luhn algorithm may need updates to support new card prefixes or modified card number lengths.
2. Data security: When dealing with sensitive information like credit card details, precautions must be taken to protect data. Encryption techniques should be employed when storing and transmitting such information, and care should be exercised when displaying it within software interfaces.

Regarding finding a better solution, the efficacy of the Luhn algorithm depends on specific requirements and constraints. In certain situations, optimization or alternative approaches may be necessary to achieve more efficient and accurate results.

Source Code:

```
// Author: Aashay Pawar
// NUID: 002134382

package edu.northeastern.csye6200;

import java.util.Scanner;

public class LAB3_P1 {
	public static void main(String[] args) {
		// TODO: write your code here
		System.out.print("Enter a credit card number as a long integer: ");
		try (Scanner sc = new Scanner(System.in)) {
			long cardNo = sc.nextLong();
			boolean check = isValid(cardNo);
			if(check)
				System.out.println(cardNo + " is valid");
			else
				System.out.println(cardNo + " is invalid");
		}
	}

	/** Return true if the card number is valid */
	public static boolean isValid(long number) {
		// TODO: write your code here
		int size = getSize(number);
		boolean pValue = (prefixMatched(number, 4) || prefixMatched(number, 5) ||
prefixMatched(number, 37) || prefixMatched(number, 6));
		int evenSum = sumOfDoubleEvenPlace(number);
		int oddSum = sumOfOddPlace(number);
		if((size >= 13 && size <= 16) && (pValue) && (pValue) &&
((evenSum+oddSum)%10 == 0))
			return true;
		return false;
	}
```

```
/** Get the result from Step 2 */
public static int sumOfDoubleEvenPlace(long number) {
	// TODO: write your code here
	String mString = number+"";
	int evenSum = 0;
	for(int i = mString.length()-2; i>=0; i-=2)
		evenSum += getDigit(Integer.parseInt(mString.charAt(i) + "")*2);
	return evenSum;
}

/**
 * Return this number if it is a single digit, otherwise, return the sum of
 * the two digits
 */
public static int getDigit(int number) {
	// TODO: write your code here
	if(number < 9)
		return number;
	return number/10 + number%10;
}

/** Return sum of odd place digits in number */
public static int sumOfOddPlace(long number) {
	// TODO: write your code here
	String mString = number+"";
	int oddSum = 0;
	for(int i = mString.length()-1; i>=0; i-=2)
		oddSum += Integer.parseInt(mString.charAt(i) + "");
	return oddSum;
}

/** Return true if the digit d is a prefix for number */
public static boolean prefixMatched(long number, int d) {
	// TODO: write your code here
	long n = getPrefix(number, getSize(d));
	if(n == d)
		return true;
	return false;
}

/** Return the number of digits in d */
public static int getSize(long d) {
	// TODO: write your code here
	String mString = d + "";
	return mString.length();
}

/**
 * Return the first k number of digits from number. If the number of digits
 * in number is less than k, return number.
```

```
 */
public static long getPrefix(long number, int k) {
        // TODO: write your code here
        if(getSize(number) > k) {
                String mString= number + "";
                return Long.parseLong(mString.substring(0, k));
        }
        return number;
```

Output:

```
Enter a credit card number as a long integer: 5117275325077359
5117275325077359 is valid
```

```
Enter a credit card number as a long integer: 4388576018402626
4388576018402626 is invalid
```

Problem 2: Detection of Consecutive Quadruplets in a Sequence

Build an application that prompts the user to input a sequence of integers and determines whether the sequence contains four consecutive numbers with the same value. The program should first ask the user to specify the size of the sequence, indicating the total number of integers in the series.

Solution:

To solve this problem, we can follow the following steps:

1. Obtain the size of the sequence from the user.
2. Create an array of the specified size.
3. Use a loop to populate the array with the integers entered by the user.
4. Call a method where the array is passed as an argument, and the main logic is implemented.
5. Initialize a counter for consecutive quadruplets.
6. Iterate over the array inside the method to check if there exist four consecutive elements with the same value.
7. If four consecutive elements have the same value, increment the quadruplet counter.
8. If the quadruplet counter is greater than or equal to one, display a message indicating whether the sequence has consecutive quadruplets or not.

Potential challenges during implementation include:

1. Clear understanding of the problem: It is crucial to comprehend the problem requirements accurately to devise an appropriate approach.
2. Handling user input: Managing user input correctly can be challenging, particularly if the input format is not explicitly defined. Ensuring the user provides the correct number of values and valid integers is essential.
3. Error handling and debugging: Mistakes in the code might lead to unexpected outcomes. Identifying and rectifying errors can be time-consuming.

4. Optimization and efficiency: There can be multiple ways to implement the solution, and optimizing the code for efficiency and memory usage can be complex.
5. Maintainability and scalability: Writing clear, understandable, and scalable code is crucial for future updates or modifications.

The proposed solution effectively addresses the given problem. However, depending on the specific requirements and constraints, alternative approaches or optimizations may exist that could provide better performance or additional functionality.

Source Code:

```
// Author: Aashay Pawar
// NUID: 002134382

package edu.northeastern.csye6200;

import java.util.Scanner;

public class LAB3_P2 {
	public static void main(String[] args) {
		// TODO: write your code here
		System.out.println("Enter the number of values:");
		try (Scanner sc = new Scanner(System.in)) {
			int mVal[] = new int[sc.nextInt()];
			System.out.println("Enter the numbers: ");
			for(int i=0; i<mVal.length; i++)
				mVal[i] = sc.nextInt();
			boolean check = isConsecutiveFour(mVal);
			if(check)
				System.out.println("The list has consecutive fours");
			else
				System.out.println("The list has no consecutive fours");
		}
	}

	public static boolean isConsecutiveFour(int[] values) {
		// TODO: write your code here
		int mCount = 0;
		int mLast = values[0];
		for (int i = 1; i < values.length; i++){
			if (values[i] == mLast) {
				mCount++;
			} else {
				mCount = 1;
				mLast = values[i];
			}
			if (mCount == 4) {
				return true;
				}
			}
```

```
                return false;
            }
}
```

Output:

```
Enter the number of values:
7
Enter the numbers:
3 3 5 5 5 5 4
The list has consecutive fours
```

```
Enter the number of values:
9
Enter the numbers:
3 4 5 5 6 5 5 4 5
The list has no consecutive fours
```

Problem 3 (Optional for Extra Credit: 10 points)

You are given an integer array nums that is sorted in non-decreasing order. Your task is to modify the array in-place by removing any duplicates such that each unique element appears only once. The order of the unique elements should remain unchanged. Finally, you need to return the count of the unique elements in nums. public int removeDuplicates(int[] nums)

To ensure your solution is accepted, please adhere to the following guidelines:

- Modify the nums array in-place, so that the first k elements contain the unique elements in their original order. The remaining elements and their order are not important.
- Return the value of k, representing the count of the unique elements in nums.

Solution:

This code implements a program that removes duplicate elements from an integer array and returns the count of unique elements. Here is a breakdown of the code:

1. The code begins with author information and a package declaration.
2. The `LAB3_P3` class contains the `main` method, which serves as the entry point of the program. Inside the `main` method, an integer array named `nums` is declared and initialized with the values {1, 1, 2, 3, 3, 4, 5}.
3. The code then prints the original array `nums` using a loop. It iterates over the array, printing each element. Commas and spaces are added to improve readability.
4. The `removeDuplicates` method is called with the `nums` array as an argument, and the returned value is stored in the `uniqueCount` variable.
5. Next, the code prints the final array `nums` after removing duplicates. It uses a similar loop as before to iterate over the array and print each element. Commas and spaces are added for separation.
6. The `removeDuplicates` method takes an array of integers as input and returns an integer representing the count of unique elements.
7. Inside the `removeDuplicates` method, it first checks if the array is null or empty. If so, it returns 0, indicating that there are no unique elements.

8. Two variables, `uniqueCount` and `currentIndex`, are initialized to 1. These variables keep track of the count of unique elements and the current index for storing the next unique element.
9. The code then enters a loop that iterates over the array starting from the second element. It compares each element with the previous element. If the current element is different from the previous element, it means a new unique element is found. The current element is then stored at the `currentIndex` position in the array, and both `currentIndex` and `uniqueCount` are incremented.
10. After the loop, any remaining elements in the array (beyond the unique elements) are filled with zeroes. This ensures that the array retains its original size, and any non-unique elements are replaced with zeroes.
11. Finally, the method returns the length of the array, representing the count of unique elements.

In summary, this code effectively removes duplicates from the input array and provides the count of unique elements.

Source Code:

```
// Author: Aashay Pawar
// NUID: 002134382

package edu.northeastern.csye6200;

public class LAB3_P3 {
        public static void main(String[] args) {
      int[] nums = {1, 1, 2, 3, 3, 4, 5};

      System.out.print("Original nums: [");
      for (int i = 0; i < nums.length; i++) {
         System.out.print(nums[i]);
         if (i < nums.length - 1) {
            System.out.print(", ");
         }
      }
      System.out.println("]");

      int uniqueCount = removeDuplicates(nums);

      System.out.print("Final nums: [");
      for (int i = 0; i < uniqueCount; i++) {
         System.out.print(nums[i]);
         if (i < uniqueCount - 1) {
            System.out.print(", ");
         }
      }
      System.out.println("]");
   }

   public static int removeDuplicates(int[] nums) {
```

```
    if (nums == null || nums.length == 0) {
        return 0;
    }

    int uniqueCount = 1;
    int currentIndex = 1;

    for (int i = 1; i < nums.length; i++) {
        if (nums[i] != nums[i - 1]) {
            nums[currentIndex] = nums[i];
            currentIndex++;
            uniqueCount++;
        }
    }

    // Fill remaining elements with zeroes
    for (int i = uniqueCount; i < nums.length; i++) {
        nums[i] = 0;
    }

    return nums.length;
}
```

Output:

```
Original nums: [1, 1, 2, 3, 3, 4, 5]
Final nums: [1, 2, 3, 4, 5, 0, 0]
```