CT2109: Assignment 3

Problem Statement

We are required to create a program that tests whether a sequence of numbers is a palindrome or not. A palindrome is a sequence of numbers that reads the same backwards as forwards. Four different methods are required to check whether a number is a palindrome or not:

Method One: Reserve all characters in the String using a loop and compare the reversed String to the original to determine if it is a palindrome, returning a Boolean indicating this.

Method Two: Compare the characters using an element-by-element basis using a loop. In other words, the method should compare the first element to the last, the second element to the second last and so on, returning a Boolean indicting whether it is palindrome or not.

Method Three: Using the ArrayStack and ArrayQueue implementations, add each character to both a stack and a queue as it is read. After all the characters have been added to each implementation, compare them both until the stack and queue are empty, returning a Boolean indicating whether the String is a palindrome or not.

Method Four: Using recursion, reverse the characters in the String and compare the original String to the output of the previous operation, returning a Boolean depending on the palindrome status.

The four above methods must accept both decimal and binary numbers. Therefore, a method that converts decimal to binary and returns a String of the equivalent binary number is required.

Analysis and Design Notes

The application will feature a Palindrome interface that defines the required functions to complete all tests. A class called PalindromeTester will contain the methods that perform their respective functionality. The four methods for testing whether a String is a palindrome or not are described below:

Method 1: This method requires the reversal of all characters in a specified String using a loop and comparison to the original to determine if the String is a palindrome. This can be done using the following pseudocode:

```
for (int i = input.length() - 1; i >= 0; i--) {
  reversed.append(input.charAt(i));
}
return reversed.equals(input);
```

Method 2: This method requires the comparison of two characters using an element-by-element loop. To achieve this, a loop that runs the length of the String can be used, which can compare two characters from the correct positions:

```
char x = input.charAt(i);
char y = input.charAt(input.length() - 1 - i);
if (x != y) return false;
```

Method 3: Using the ArrayStack and ArrayQueue implementations, this method will push and enqueue each character of the String to their respective implementation. Then, the method can use a for-loop and compare the character from the stack pop result and queue dequeue result:

```
stack.push(character);
queue.enqueue(character);

for (int i = 0; i < input.length(); i++) {
    char x = (char) stack.pop();
    char y = (char) queue.dequeue();
    if (x != y) return false;
}
```

Method 4: This method will utilise another method, named 'reverse' to recursively reverse the String and compare it with the original String. The following pseudocode demonstrates the functionality to reverse a String using recursion. This functionality will be contained in the 'reverse' method, which will be called from another method that will compare it with the original String:

```
if (input.isEmpty()) {
    return input;
}
return reverse(input.substring(1)) + input.charAt(0);
```

Conversion from decimal to binary: A simple method that utilises the ArrayStack implementation will be used to convert a decimal String into binary. The String will be cast as a number and the stack will be filled with the remainders of the conversion algorithm. The stack can then be popped, and the resulting String returned:

```
while (number > 0) {
    stack.push((char) number % 2);
    number = number / 2;
}
while (!stack.isEmpty()) {
    result.append(stack.pop());
}
```

A class, namely ComplexityAnalysis will contain the required testing functions for the application. A test function will call the four methods, and use Integers to count the amount of decimal, binary and equivalent numbers each algorithm returns. Also contained within the test function will be a timer that can time the period required to run each method a specified number of times. This can be later used for the complexity analysis in our tests. The methods will be contained inside a for-loop so each method can count palindromes in a specific range.

The final design also implemented primitive operation counters, which are in the PalidromeTester class. These can be accessed by the testing class.

Testing

The efficiency of each method was tested. To achieve this, a test method called `efficiencyTester` was created that could be invoked from the main method in the class. For each method, the following items were tested: identification of decimal palindromes, identification of binary palindromes and numbers which both had a decimal and binary equivalence. A timer was used to record the duration it took for the method to run. The test method also allowed the recording of the number of primitive operations it took for each respective algorithm to complete its tasks. The test method yielded the following results in the application console:

```
Method One: loopCharacter()
0: 44
50000: 9145530
100000: 19147214
150000: 30036000
200000: 41142500
250000: 52249000
300000: 63620765
350000: 75077615
400000: 86534465
450000: 97991315
500000: 109448165
550000: 121085188
600000: 132892388
650000: 144699588
700000: 156506788
750000: 168313988
800000: 180121188
850000: 191928388
900000: 203735588
950000: 215542788
1000000: 227349752

> Time taken: 885 ms
> Decimal palindromes: 1999
> Binary palindromes: 2000
> Decimal & binary equivalent palindromes: 20
```

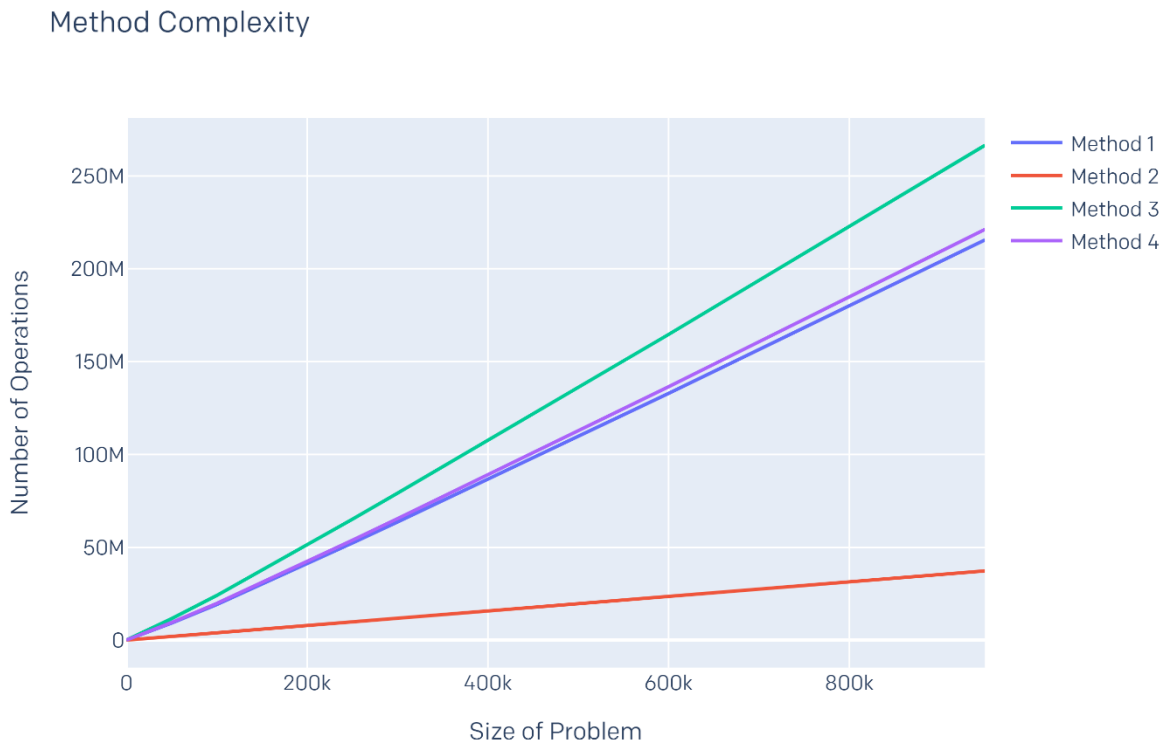
```
Method Two: characterCompare()
0: 24
50000: 1977367
100000: 3951527
150000: 5913322
200000: 7873302
250000: 9832802
300000: 11792627
350000: 13752997
400000: 15713442
450000: 17673372
500000: 19633407
550000: 21593002
600000: 23551757
650000: 25510017
700000: 27468252
750000: 29427002
800000: 31385732
850000: 33343987
900000: 35302257
950000: 37260952
1000000: 39219794

> Time taken: 677 ms
> Decimal palindromes: 1999
> Binary palindromes: 2000
> Decimal & binary equivalent palindromes: 20
```

```
Method Three: stackQueue()
0: 60
50000: 11527848
100000: 24031470
150000: 37539609
200000: 51295051
250000: 65050109
300000: 79108586
350000: 93264739
400000: 107420953
450000: 121576754
500000: 135732641
550000: 150094061
600000: 164649275
650000: 179204092
700000: 193758888
750000: 208314098
800000: 222869290
850000: 237424103
900000: 251978928
950000: 266534092
1000000: 281089116

> Time taken: 2496 ms
> Decimal palindromes: 1999
> Binary palindromes: 2000
> Decimal & binary equivalent palindromes: 20
```

```
Method Four: reverseString()
0: 52
50000: 9446734
100000: 19749418
150000: 30938304
200000: 42344904
250000: 53751504
300000: 65423369
350000: 77180319
400000: 88937269
450000: 100694219
500000: 112451169
550000: 124388292
600000: 136495592
650000: 148602892
700000: 160710192
750000: 172817492
800000: 184924792
850000: 197032092
900000: 209139392
950000: 221246692
1000000: 233353750

> Time taken: 1785 ms
> Decimal palindromes: 1999
> Binary palindromes: 2000
> Decimal & binary equivalent palindromes: 20
```

This test function was run multiple times to ensure the results used were correct.

Using the above output, the complexity of each method can be analysed against the range of numbers the test method ran. The following graph plots the number of operations against the range of numbers checked for each method, allowing a comparison between the four methods used in this application.



The graph clearly indicates that the second method considerably requires less operations then the other three methods. Intuitively, this is correct as that method only requires a for-loop and an if-statement that compares opposite values in a character array without having to use an additional data structure. The low slope of method two suggests that the growth rate of its respective algorithm remains quite low in comparison to its counterparts.

This contrasts with method three, which results in over 250 million operations for the same size of the problem. This result is expected as this method utilises the ArrayStack and ArrayQueue implementations to locate palindromes. The method must first push and enqueue each character into each implementation. Running a comparison requires the method to pop and dequeue each character. This is comparably more tedious to do as it requires the usage of multiple abstract data type implementations to achieve its goal.

Method one and four performed considerably worst than method two, but better than method three. This performance was, again expected as both methods implemented a palindrome-finder algorithm that required more primitive operations, for handling strings, for example and the utilisation of recursion in method four. The first method reversed a String using a loop and then compared it to the original. This was more inefficient in contrast to method two, as it created the whole string before it compared it, whereas method two could terminate its execution exactly at the character which did not match its opposite, thus not being a palindrome. In the fourth method, more operations were required, as it utilised recursion and multiple primitive operations.

PalindromeTester.java

```java
public class PalindromeTester implements Palindrome
{
    public int primitiveCount1 = 0;
    public int primitiveCount2 = 0;
    public int primitiveCount3 = 0;
    public int primitiveCount4 = 0;

    public boolean loopCharacter(String input) {
        StringBuilder res = new StringBuilder();
        primitiveCount1++;

        // Reserve the input string using a for loop
        for (int i = input.length() - 1; i >= 0; i--) {
            primitiveCount1 += 7;
            res.append(input.charAt(i));
        }

        primitiveCount1 += 3;
        return res.toString().equals(input);
    }

    public boolean characterCompare(String input) {
        // Compare the characters on an element-by-element basis
        for (int i = 0; i < input.length(); i++) {
            primitiveCount2 += 5;

            if (input.charAt(i) != input.charAt(input.length() - 1 - i)) {
                primitiveCount2 += 6;
                return false;
            }
        }

        primitiveCount2++;
        return true;
    }

    public boolean stackQueue(String input) {
        ArrayStack stack = new ArrayStack();
        ArrayQueue queue = new ArrayQueue();
        primitiveCount3 += 2;

        // Push the characters of the string to both a stack and queue
        for (char x: input.toCharArray()) {
            primitiveCount3 += 4;
            stack.push(x);
            queue.enqueue(x);
            primitiveCount3 += 4;
        }

        // Compare each character in the stack and queue
        for (int i = 0; i < input.length(); i++) {
            primitiveCount3 += 4;

            if (stack.pop() != queue.dequeue()) {
                primitiveCount3 += 4;
                return false;
            }
```

```
        }

        primitiveCount3++;
        return true;
    }

    public boolean reverseString(String input) {
        primitiveCount4 += 3;
        return reverse(input).equals(input);
    }

    public String reverse(String input) {
        primitiveCount4 += 2;
        if (input.isEmpty()) {
            primitiveCount4 += 1;
            return input;
        }

        primitiveCount4 += 5;
        return reverse(input.substring(1)) + input.charAt(0);
    }

    public String decimalBinary(String input) {
        ArrayStack stack = new ArrayStack();

        // Check if the number provided is zero
        if (input.equals("0")) return "0";

        int n = Integer.parseInt(input);

        // Create the binary number
        while (n > 0) {
            stack.push((char) n % 2);
            n = n / 2;
        }

        // Build the binary string
        StringBuilder res = new StringBuilder();

        while (!stack.isEmpty()) {
            res.append(stack.pop());
        }

        return res.toString();
    }
}
```

ComplexityAnalysis.java

```java
public class ComplexityAnalysis
{
    static PalindromeTester palindrome = new PalindromeTester();

    public static void main(String[] args) {
        // Run the efficiency tester
        efficiencyTester();
    }

    public static void printMethodResult(long duration, int decimalCount, int binaryCount, int
decimalBinaryEquivalent) {
        System.out.println("\n> Time taken: " + duration + " ms");
        System.out.println("> Decimal palindromes: " + decimalCount);
        System.out.println("> Binary palindromes: " + binaryCount);
        System.out.println("> Decimal & binary equivalent palindromes: " + decimalBinaryEquivalent);
    }

    public static void efficiencyTester() {
        // Method 1
        long startTime = System.nanoTime();
        int decimalCount = 0; int binaryCount = 0; int decimalBinaryEquivalent = 0;

        System.out.println("\nMethod One: loopCharacter()");
        for (int i = 0; i < 1000000; i++) {
            if (palindrome.loopCharacter(Integer.toString(i))) {
                decimalCount++;
            }

            if (palindrome.loopCharacter(palindrome.decimalBinary(Integer.toString(i)))) {
                binaryCount++;
            }

            if (
                palindrome.loopCharacter(Integer.toString(i)) &&
                palindrome.loopCharacter(palindrome.decimalBinary(Integer.toString(i)))
            ) decimalBinaryEquivalent++;

            if (i % 50000 == 0) {
                System.out.println(i + ": " + palindrome.primitiveCount1);
            }
        }

        long endTime = System.nanoTime();
        long duration = ((endTime - startTime) / 1000000);
        System.out.println(1000000 + ": " + palindrome.primitiveCount1);

        printMethodResult(duration, decimalCount, binaryCount, decimalBinaryEquivalent);

        // Method 2
        startTime = System.nanoTime();
        decimalCount = 0; binaryCount = 0; decimalBinaryEquivalent = 0;

        System.out.println("\nMethod Two: characterCompare()");
        for (int i = 0; i < 1000000; i++) {
            if (palindrome.characterCompare(Integer.toString(i))) {
                decimalCount++;
            }
```

```
    if (palindrome.characterCompare(palindrome.decimalBinary(Integer.toString(i)))) {
        binaryCount++;
    }

    if (
        palindrome.characterCompare(Integer.toString(i)) &&
        palindrome.characterCompare(palindrome.decimalBinary(Integer.toString(i)))
    ) decimalBinaryEquivalent++;

    if (i % 50000 == 0) {
        System.out.println(i + ": " + palindrome.primitiveCount2);
    }
}

endTime = System.nanoTime();
duration = ((endTime - startTime) / 1000000);
System.out.println(1000000 + ": " + palindrome.primitiveCount2);

printMethodResult(duration, decimalCount, binaryCount, decimalBinaryEquivalent);

// Method 3
startTime = System.nanoTime();
decimalCount = 0; binaryCount = 0; decimalBinaryEquivalent = 0;

System.out.println("\nMethod Three: stackQueue()");
for (int i = 0; i < 1000000; i++) {
    if (palindrome.stackQueue(Integer.toString(i))) {
        decimalCount++;
    }

    if (palindrome.stackQueue(palindrome.decimalBinary(Integer.toString(i)))) {
        binaryCount++;
    }

    if (
        palindrome.stackQueue(Integer.toString(i)) &&
        palindrome.stackQueue(palindrome.decimalBinary(Integer.toString(i)))
    ) decimalBinaryEquivalent++;

    if (i % 50000 == 0) {
        System.out.println(i + ": " + palindrome.primitiveCount3);
    }
}

endTime = System.nanoTime();
duration = ((endTime - startTime) / 1000000);
System.out.println(1000000 + ": " + palindrome.primitiveCount3);

printMethodResult(duration, decimalCount, binaryCount, decimalBinaryEquivalent);

// Method 4
startTime = System.nanoTime();
decimalCount = 0; binaryCount = 0; decimalBinaryEquivalent = 0;

System.out.println("\nMethod Four: reverseString()");
for (int i = 0; i < 1000000; i++) {
    if (palindrome.reverseString(Integer.toString(i))) {
```

```
                decimalCount++;
            }

            if (palindrome.reverseString(palindrome.decimalBinary(Integer.toString(i)))) {
                binaryCount++;
            }

            if (
                palindrome.reverseString(Integer.toString(i)) &&
                palindrome.reverseString(palindrome.decimalBinary(Integer.toString(i)))
            ) decimalBinaryEquivalent++;

            if (i % 50000 == 0) {
                System.out.println(i + ": " + palindrome.primitiveCount4);
            }
        }

        endTime = System.nanoTime();
        duration = ((endTime - startTime) / 1000000);
        System.out.println(1000000 + ": " + palindrome.primitiveCount4);

        printMethodResult(duration, decimalCount, binaryCount, decimalBinaryEquivalent);
    }
}
```

Palindrome.java

```
interface Palindrome
{
    /**
     * 1: Reverse all characters in the string using a loop and then compare
     * the reversed string to the original
     */
    public boolean loopCharacter(String input);

    /**
     * 2: Compare characters on an element-by-element basis using a loop
     */
    public boolean characterCompare(String input);

    /**
     * 3: Use ArrayStack and ArrayQueue implementations
     */
    public boolean stackQueue(String input);

    /**
     * 4: Use recursion to reverse the characters and compare with original
     * string
     */
    public boolean reverseString(String input);

    /**
     * Utility function to convert decimal into binary
     */
    public String decimalBinary(String input);
}
```