Assignment 3: Double-base Palindromes and Complexity Analysis

Cathal Lawlor - 21325456

Palindrome: a sequence that reads the same backwards as forwards

Four methods
Each take: string input ->   Returns boolean -> is string a palindrome T/F

Method one:
Reverse all the characters in the String using a loop
Compare the reversed String to the original, determining if it is a palindrome.
.equals Comparison returns T/F

Method two:
Using a loop we compare the characters on a element by element basis.
Compare first element to the last, second to second last etc.
If there is any false matches, return false - Else return false

Method three:
ArrayStack and ArrayQueue implentations from before
Add each character to a stack and a queue as it is read.
After all characters have been added to both,
Remove characters from both, comparing them.

If a palindrome, first char popped == first dequeued   So on etc.

If mismatch, return false,
true after

Method four:
Seperate method called reverse,
Uses recursion to reverse the chars in a provided string - returns answer (reversed string) – returns substring(index 1) + charAt(0).
In method four, compare the original string to the output that was returned from reverse(recursive method)

Return T/F from comparison

Testing note - Using both decimal & binary numbers

Utility method:
Converts a decimal to equivalent binary returning it

One parameter - String of decimal number

Returns - String of binary number representation

Time Complexity analysis of each function – before code

Method one:

The loop runs the length n, with other operations the O(n) remains at n, even though it might be xn with x being 5 etc. with operations such as appending on the character to the reverse string.

Going to take longer than method 2 as it will do minimum 2n operations through reversing the string an then checking each string against each other.

Method two:

Will run the length n, with n/2 + n/2 as it scans from each end of the string the characters to the mid-point of input string. There shouldn't be much more operands in this method. It remains O(n) = n.

Method three:

This will have lots of operations as arrayStack.push / .pop will have their own operations within them. arrayQueue will also have operations in them as well as an n length operation moving the index's of the elements in the queue after deQueue. enQueue also has its own primitive operations.

With this the method will have it's own n operations combined with these operations above it will be considerably more than method 2. It still remains O(n) as the deQueue is O(n) as the string length n decreases. I am still not decided if this is O(n) n^2 or a very high O(n) n with the way deQueue length decreases.

Method four:

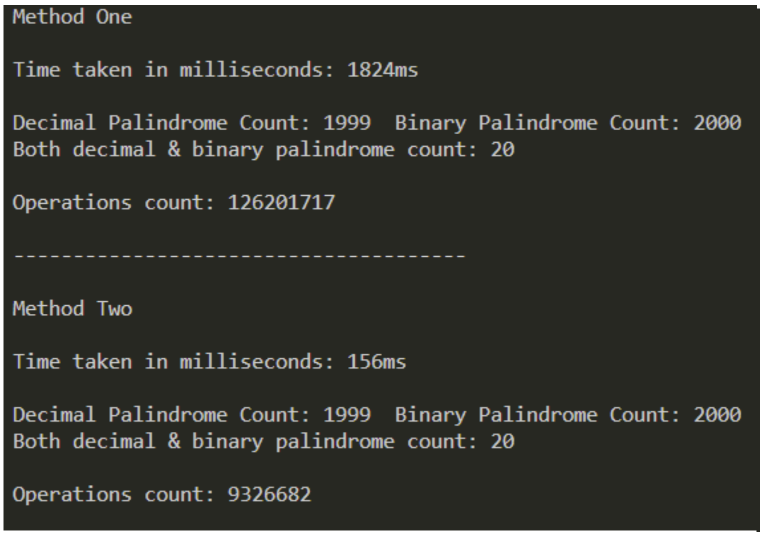
Recursively this will have a lot of operations as it will need to split the string with substrings and append the first character. This will eat a huge amount of memory as the cpu will be sitting and waiting as the method recursively calls itself. This is an efficient method for it being recursive but it will not be the fastest as it still will have lots of primitive operations.

NOTE – Utility – decimalToBinary(String decimalString):

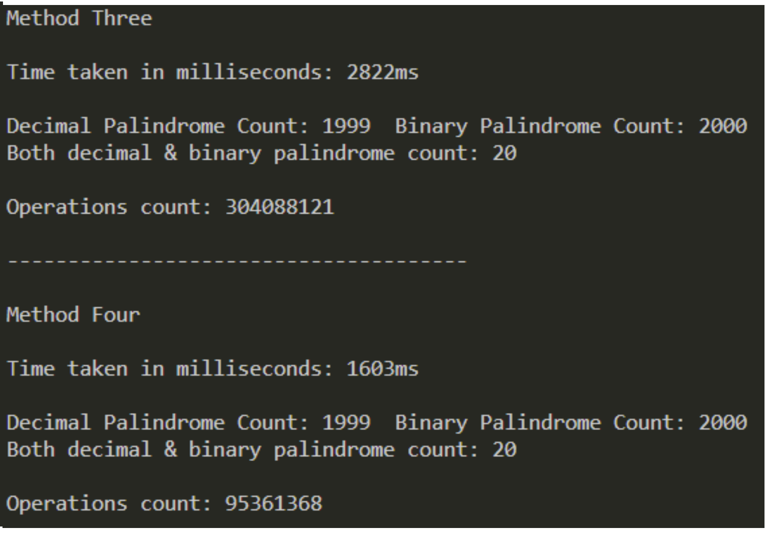
I had this method set as my own manual code doing % 2 and carrying the remainder and using exponents to calculate the binary. I found it added 5 seconds to the timing and rendered timing useless as it increased the margin of error.

I now use inbuilt java functions which are much faster. I don't know if this impacts assignment brief.

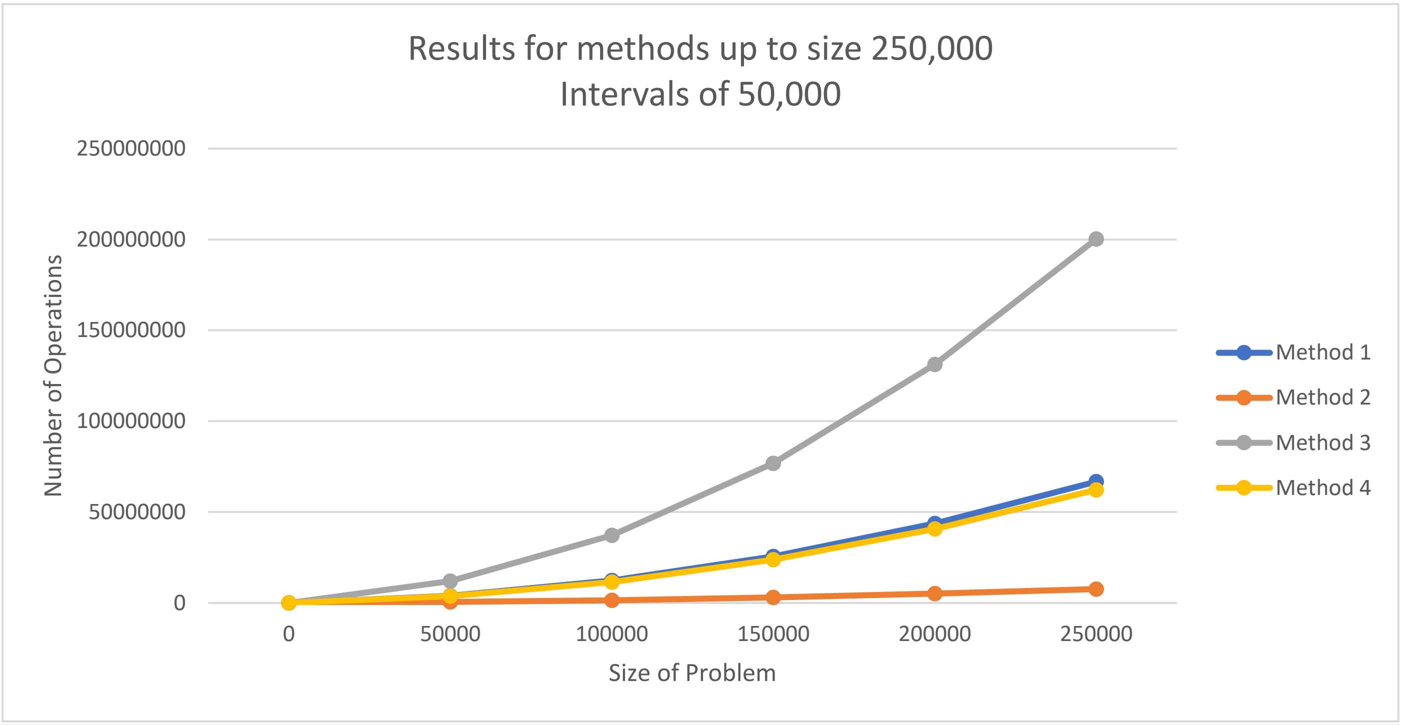
Testing

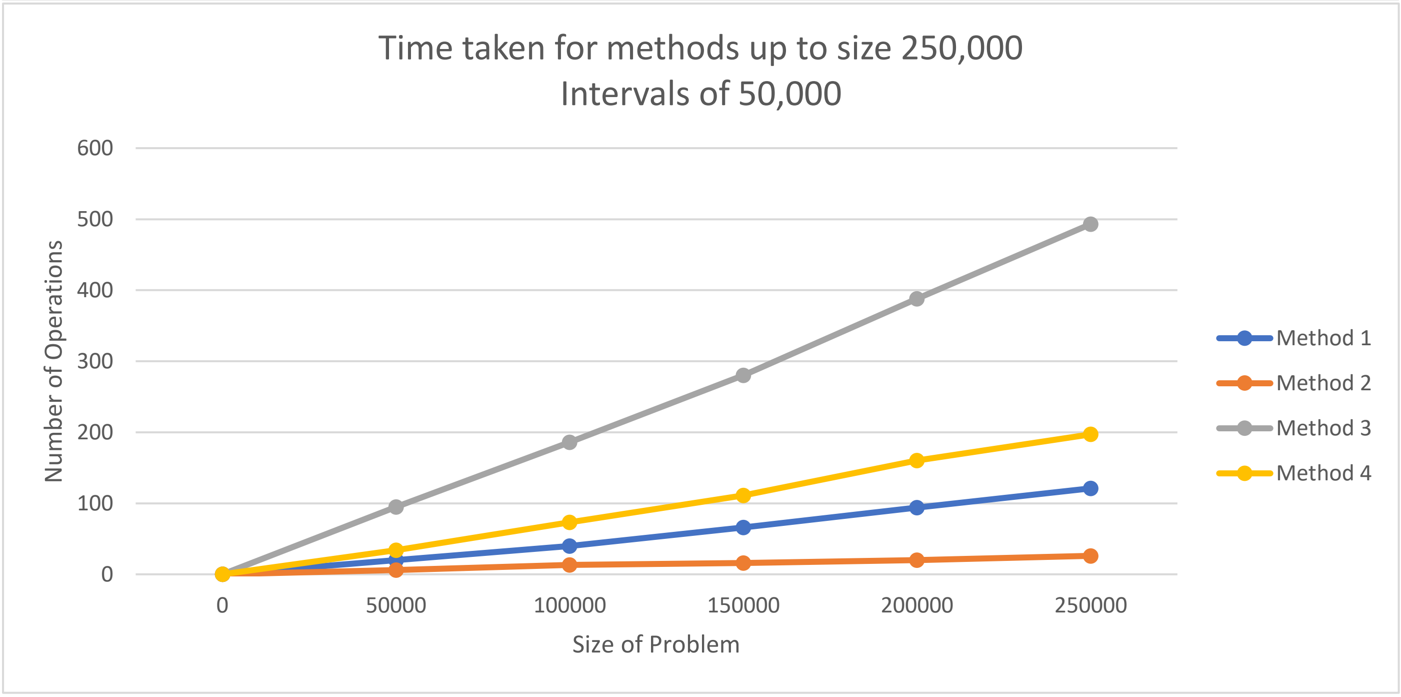


```
Method One

Time taken in milliseconds: 1824ms

Decimal Palindrome Count: 1999  Binary Palindrome Count: 2000
Both decimal & binary palindrome count: 20

Operations count: 126201717

-------------------------------------

Method Two

Time taken in milliseconds: 156ms

Decimal Palindrome Count: 1999  Binary Palindrome Count: 2000
Both decimal & binary palindrome count: 20

Operations count: 9326682
```



```
Method Three

Time taken in milliseconds: 2822ms

Decimal Palindrome Count: 1999  Binary Palindrome Count: 2000
Both decimal & binary palindrome count: 20

Operations count: 304088121

-------------------------------------

Method Four

Time taken in milliseconds: 1603ms

Decimal Palindrome Count: 1999  Binary Palindrome Count: 2000
Both decimal & binary palindrome count: 20

Operations count: 95361368
```





Note – while counting inbuilt functions such as .substring as one operation, the time taken to complete that versus a much simpler operation such as x = 2 + 1; is going to be more, however saying that, the function is optimised at the compiler I believe so I cannot just find the time complexity inside of each of these inbuilt functions. This may be why time taken for method two and three are closer to method four than one would expect looking at total operations for each.
I have chosen to take a more simplistic approach and count each call as one 'operation' even though this isn't 100% accurate.

Testing Analysis

We see that as predicted earlier method two is by far the fastest. It has a quite low operation count making it exceedingly fast compared to other methods. With it being iterative it remains O(n) = n. As it only loops over the string of length n once, and it only has a few operations in this loop, it only has a 3n number of operations making it quite efficient. This is the best implementation out of all the methods.

Method four is quite fast considering it is recursive, but we can see the clear distinction between method one and itself. At each recursive call, the method is creating a new substring which is the input string length - 1, so the number of recursive calls is n where n is the length of the input string.
Each recursive call takes constant time to execute the base case check, and then it makes a recursive call. Therefore, the total number of operations performed by the reverse method is proportional to the length of the input string.

This will take a lot of memory though and is not recommended as a good implementation.

Method one is quite similar to method four in operations, with it being iterative it remains O(n) = n, but it is more inefficient with the extra operations of looping over the whole string reversing it, and only then comparing it on top of the earlier operations completed. I would not implement this in practice.

Method three by far is the most inefficient. We can see it is O(n) = n^2 with a very inefficient number of operations. This is most likely due to the deQueue method mentioned earlier, shifting the index of all the methods one by one. The maths corroborates this with $50{,}000^2$ = 2.5 million, $100{,}000^2$ = 1.E10, $150{,}000^2$ = 2.25E13. This lines up with the graph. You would quickly run into problems using this implementation. It is a good learning tool for stacks and queues though 😊.

Overall, the predictions stuck mostly to what we were expecting. The time taken reflects the number of operations. With each method still ranking in the same place as their operation count had placed them on the graph. Taking into the note I mentioned earlier the results line up correctly in my opinion.

Code

MyApplication

```java
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.util.Scanner;

public class MyApplication {

    public MyApplication() {

    }

    public static long testPalindromeMethod(MethodInterface method, int testNum) {
        int decimalPalindromes = 0, binaryPalindromes = 0, bothPalindomes = 0; // counts
for amount of palindromes for
                                                                                // Decimal,
Binary & both for the same
                                                                                // num
        boolean isDecPal, isBinPal = false;
        long timerStart = System.currentTimeMillis(); //timing how long to do testNum
palindrome checks

        for (int i = 0; i <= testNum; i++) {
            isDecPal = false;
            isBinPal = false;
            String testDecString = Integer.toString(i); //dec string
            String testBinString = decimalToBinary(testDecString); //bin string

            if (method.method(testDecString)) {
                decimalPalindromes++;
                isDecPal = true;
            }

            if (method.method(testBinString)) {
                binaryPalindromes++;
                isBinPal = true;
            }

            if (isDecPal && isBinPal) { //if both are palindromes then we have a match
                bothPalindomes++;
            }
        }
        long timerStop = System.currentTimeMillis(); //stop timing

        //Comment this out if you are doing excel automation
        System.out.println("Time taken in milliseconds: " + (timerStop - timerStart) +
"ms");
        System.out.println("\nDecimal Palindrome Count: " + decimalPalindromes + "\tBinary
Palindrome Count: "
                + binaryPalindromes);
        System.out.println("Both decimal & binary palindrome count: " +
bothPalindomes);
        System.out.println("\nOperations count: " + method.getOperationsCount());
        System.out.println("\n--------------------------------------");


        return (method.getOperationsCount()); //operation counter for testNum numbers

    }

    public static String decimalToBinary(String decimalString) { //using java inbuilt
functions cut down on time taken as manually doing it can add 5 seconds over a million
numbers :)
        int decimal = Integer.parseInt(decimalString);
        String binaryString = Integer.toBinaryString(decimal);

        return binaryString;
    }

    public static void testAutomation(MethodInterface method) { //automation for excel
numbers
        Long[] operations = {0, 0, 0, 0, 0, 0}; //array to count operation count
        for (int i = 0; i <= 250000; i += 50000) { //up to a quarter of a million we check
in 50,000 increments
            Long averageOPs = 0; //average operations
            for (int j = 0; j < 4; j++) { //we check 4 times to get an average
                averageOPs += testPalindromeMethod(method, i);
            }
            operations[i / 50000] = (averageOPs / 4);
        }

        for (int i = 0; i <= 250000; i += 50000) {
            System.out.println("Length " + i + " requires: " + operations[i / 50000] + "
operations");
        }

    }

    public static void main(String[] args) {
        int testIterations = 1000000;
        MethodInterface methodOne = new MethodOne();
        MethodInterface methodTwo = new MethodTwo();
        MethodInterface methodThree = new MethodThree();
        MethodInterface methodFour = new MethodFour();

        /*  // test the methods work if you like
        System.out.println("Enter a sample to test: ");
        Scanner s = new Scanner(System.in);
        String userString = s.nextLine();
        System.out.println("M1: " + methodOne.method(userString) + " M2: " +
methodTwo.method(userString) + " M3: "
                + methodThree.method(userString) + " M4: " +
methodFour.method(userString));
         */
        System.out.println("\nMethod One\n");
        testPalindromeMethod(methodOne, testIterations);
        System.out.println("\nMethod Two\n");
        testPalindromeMethod(methodTwo, testIterations);
        System.out.println("\nMethod Three\n");
        testPalindromeMethod(methodThree, testIterations);
        System.out.println("\nMethod Four\n");
        testPalindromeMethod(methodFour, testIterations);

        // automation of tests for excel - comment out testPalindromeMethod prints if
        // you use these
        /*
        System.out.println("\nMethod One\n");
        testAutomation(methodOne);
        System.out.println("\nMethod Two\n");
        testAutomation(methodTwo);
        System.out.println("\nMethod Three\n");
        testAutomation(methodThree);
        System.out.println("\nMethod Four\n");
        testAutomation(methodFour);
        */
    }
}
```

MethodInterface

```java
public interface MethodInterface {

    public boolean method(String testString);
    public long getOperationsCount();
}
```

MethodOne

```java
public class MethodOne implements MethodInterface {


    protected long oCount = 0; //operations counter


    public boolean method(String testString) {
        String revString = ""; //reverse string initialised
        oCount++; // 1

        //starting at the end of testString we append reverse it's testString backwards
        for(int i = testString.length() - 1; i >= 0; i--) {  //2n
            revString += testString.charAt(i); // 2n
            oCount += 4; //2n + 2n -> O(n)
        }

        if(testString.equals(revString)) { //if both strings match
            return true; //return we have a palindrome
        }
        else { return false; } //else - false

    }

    public long getOperationsCount(){
        return this.oCount;
    }
}
```

MethodTwo

```
public class MethodTwo implements MethodInterface {


    protected long oCount = 0; //operations counter

    public boolean method(String testString) {
        for(int i = 0; i < testString.length() / 2; i++) { // 2n / 2
            if(testString.charAt(i) != testString.charAt(testString.length() - 1 - i)) {
// 4n / 2
                oCount += 4;
                return false; //if each individual elements aren't the same - return a
false
            }
            // 2n / 2 - only runs half the array
            oCount += 2; //(n / 2) = O(n)
        }
        oCount += 2;
        return true; //else we have a palindrome
    }

    public long getOperationsCount(){
        return this.oCount;
    }
}
```

MethodThree

```java
public class MethodThree implements MethodInterface {


    protected long oCount = 0; //operations counter

    @Override
    public boolean method(String testString) {
        ArrayStack arrayStack = new ArrayStack(); // 1
        ArrayQueue arrayQueue = new ArrayQueue(); // 1
        oCount += 2;

        //arrayStack / queue yuck
        for(int i = 0; i < testString.length(); i++) {  //2n
            //Taking it as the push and enqueue is one operation
            arrayStack.push(testString.charAt(i)); // 3n - 3 operations in ArrayStack.push
            arrayQueue.enqueue(testString.charAt(i)); // 3n - 3 operations in
ArrayQueue.enqueue
            oCount += 9;
        }

        while(!arrayStack.isEmpty() && !arrayQueue.isEmpty()){ // 2n - Singular operation
in both methods
            //3n + 7 - Pop has 5 operations and dequeue has 3n + 3 operations (as it
shifts the indexs)
            if(arrayStack.pop() != arrayQueue.dequeue()){
                for(int i = 0; i < arrayQueue.rear; i++) {
                    oCount += 3; //the 3n in arrayQueue
                }
                oCount += 7; //the remaining + 7 for pop and enqueue
                return false;
            }
            oCount += 3; //O(n) - 8n + 2
        }
        oCount += 2;

        return true;
    }

    public long getOperationsCount(){
        return this.oCount;
    }
}
```

MethodFour

```java
public class MethodFour implements MethodInterface {


    protected long oCount = 0; //operations counter

    @Override
    public boolean method(String testString) {
        String reverseString = reverse(testString);
        return testString.equals(reverseString);
    }

    public String reverse(String testString) {
        //if string inputted is empty or length one - we know it's a palindrome - base case
        if (testString.length() <= 1) { //2 - checks if smaller than 1, and if equals to 1.
            oCount += 2;
            return testString;
        }
        oCount += 2;
        //n - the method is making n recursive calls, and for each call, it is creating a new substring which is the input string length - 1
        String reverseSubStr = reverse(testString.substring(1)); // 2 - substring & initialising a string
        oCount += 2;
        return reverseSubStr + testString.charAt(0);
    }


    public long getOperationsCount(){
        return this.oCount;
    }

}
```