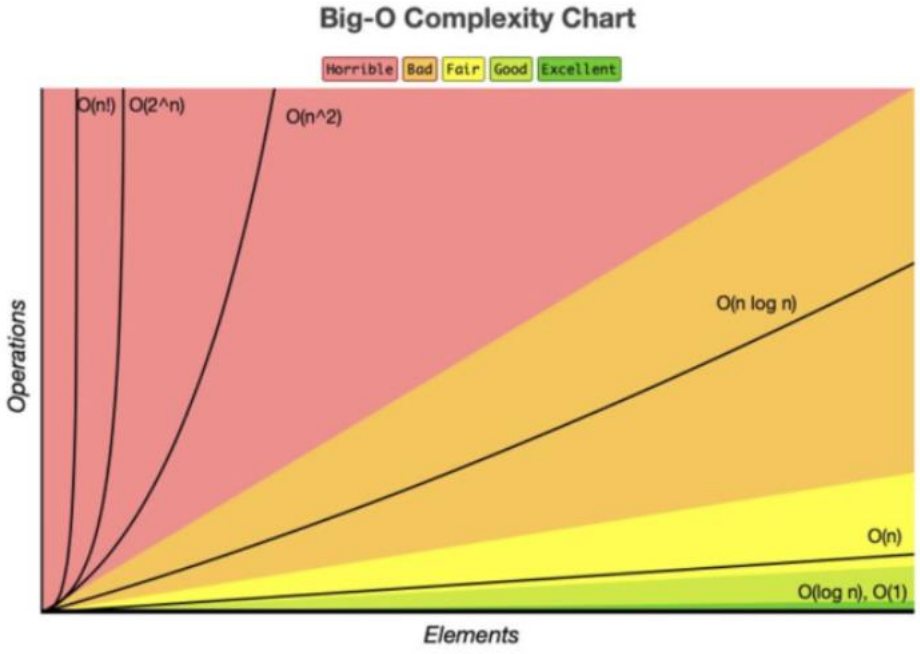
Assignment 3 OOP – Palindromes

Problem Statement:

In this assignment we are dealing with palindromes. Palindromes are sequences of characters that read the same backwards and forwards i.e. race car. We are asked to create four different Boolean methods that should all carry out the same function in different ways that is testing for binary and decimal palindromes. In order to complete this assignment in the best way possible we must have a good understanding of palindromes, stacks and queues along with recursion.

The overall goal of this assignment is (1) To find the time it takes to carry out each method. (2) Find the number of operations at a given interval up to 1,000,000. (3) To find how many numbers are both palindromes in decimal and binary form in the range of 1 - 1,000,000. Finally we must graph our methods into excel so sufficient excel knowledge is also required.

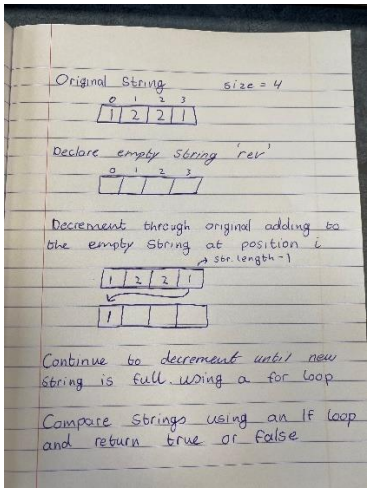
When we get to the graph, the basic knowledge of algorithms, Big O Notation and time complexity will be of use as we need to be able to interpret the graph correctly in order to determine the Big O Notation. Here is a perfect representation of what we will need to know.



Analysis and Design Notes:

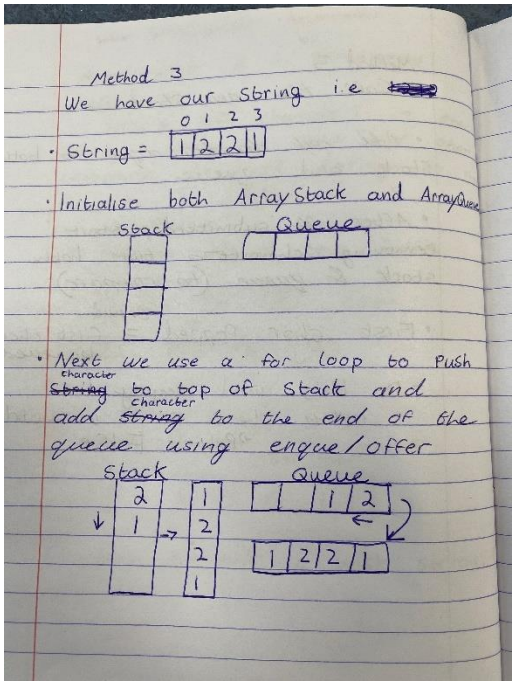
The first steps I will carry out will be walking through each method testing each time with temporary print statements to test basic palindromes, this will ensure the methods are correct so I wont have to come back to them.

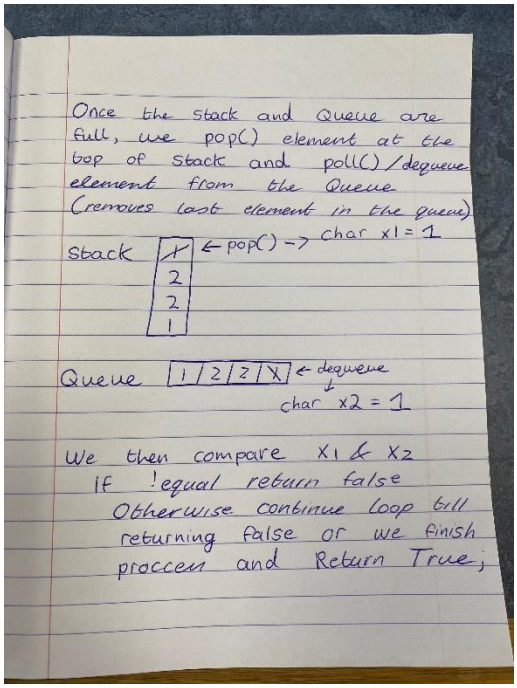
Method 1: will be a method that will reverse all the characters in a String. In order to do this method I will create an empty String and a for loop that will loop through the string backwards (i--) and adding the char at position i to the new empty string. This will place the last element at the start of the new string. Once I >= 0 the loop will stop and we will create an if loop to test if the new string is equal to the original string. Here is a rough sketch of what the process looks like.



Method 2: in this method we are asked to compare the characters element-by-element. We are comparing the elements in the same string I.e. the first character will be compared to the last element. In order to do this we will have to initialise two different variables, left will equal the first element in the string (int left = 0) and right will equal the last element (str.length() – 1). We will then compare left to right as long as right is greater than left as once they become equal to each other we have compared all the elements. We use a while loop to do this and if at any stage string at left position is not equal to the string at the right position we will return false. We can do this using str.charAt(left) != str.charAt(right). If we don't return false, increment left and right until they become equal in size. Return true.

Method 3: in this method we have to use both stack and queue implementations. This method will work by adding the characters to both the stack and the queue and comparing them using the appropriate removing methods. I will initialise two structures, the stack and queue to store the characters. A for loop can be used to loop through the string and each time the character at position i is pushed to the top of the stack and added to the end of the queue. A while loop is used to loop through until there are no characters left in the stack and queue. The top character is popped in the stack and the last character is removed from the queue. These two characters will be compared and if they are not equal the process will immediately return false. If the while loop continues without returning false that means the string is a palindrome and the method will return true. Here is a simple representation on the code.





Method 4: in this method we are using recursion which is an effective way to avoid having to write code a number of times. Create two methods RecursicePalindrome and reverse. The recursivePalindrome method will take in a string and checks if it is recursive or not. It does this by calling the recursive reverse method and comparing the reversed string with the original using the equals method. If equal it will return true. The recursive method takes in the string checks if the input String is empty or not. If it is empty, then it returns the empty String. Otherwise, it recursively calls itself with a substring of the input String and then appends the first character of the input String to the end of the reversed substring. This process is repeated until the entire String is reversed.

Decimal to Binary method: we must create a decimal to binary method in order to test both decimal and binary numbers. This will also be needed so we can find the number of matching palindromes. The method will take in a string parameter. Create a stack and convert to integer using parseInt, create a string builder too. If the input is not equal to 0 the method will enter into a while loop until the input is 0.

It calculates the remainder of 'in' when divided by 2 using the modulus operator % and pushes it onto the stack s. Update the value of 'in' by dividing it by 2 using integer division /. After the first while loop ends, the method enters another while loop that pops each element from the stack s and appends it to the StringBuilder output. This reverses the order of the binary digits. If the input String is equal to "0", then the method simply appends it to the StringBuilder output. Finally, the method returns the binary representation of the input decimal number as a String by converting the StringBuilder output to a String using the toString method.

Test Method: this is where I will call each method in order to test the efficiency. Using 'System.currentTimeMillis()' to calculate the time it takes for each method to carry out it's operations the formula will be end time minus the start time. We must create a different variable for each method to store the start and end time i.e startTimeM1 and endTimeM1. This function gets the exact time of the day in milliseconds (not a timer).

We are testing numbers in the range of 1-1,000,000 so we must create a for loop to iterate through numbers in that range. While looping through we are constantly checking if the binary equivalent to the decimal number if palindrome is also a palindrome if so we add I to an empty array that will end up storing all the numbers that fit these requirements

[*reversePalindrome*(String.*valueOf*(i)) && *reversePalindrome*(*decimal2Binary*(String.*valueOf*(i)]. (M1List.add(i)) we do this for each method to ensure they are all working properly and returning the same number. (NOTE: we create an empty array list for each method)

We should also create an interval loop to print out the number of operations every 50,000 iterations, this will make it easy for me to transfer my data over to excel and plot the graph.

Our test method should print the time taken for the operations, the operations at each interval up to 1,000,000 and the number of matching palindromes for the binary and decimal numbers. We must make a test call for each of the methods. Note we must create specific variable for each method.

Operations: in order to count the primitive operations, create unique variables for each method i.e. countM1, countM2…… Increment the counter (count++;) each time with the appropriate amount after each primitive operation within the methods and the testing as each time we will loop through both.

Code:

```
import java.util.*;
import java.util.Queue;
import java.util.Stack;

public abstract class Palindrome {
    static int countM1 = 0, countM2 = 0, countM3 = 0, countM4 = 0; //
primitive operations counters
    static long startTimeM1, endTimeM1, startTimeM2, endTimeM2,
startTimeM3, endTimeM3, startTimeM4, endTimeM4; // timer
    // Array lists to store Palindromes
    static List<Integer> M1List = new ArrayList<>();
    static List<Integer> M2List = new ArrayList<>();
    static List<Integer> M3List = new ArrayList<>();
    static List<Integer> M4List = new ArrayList<>();

    //Main Method
    public static void main(String[] args) {
        testMethods();
    }
    static void testMethods() {
        startTimeM1 = System.currentTimeMillis(); // start time
        countM1++; // increment primitive operations
        System.out.print("Method1\nInterval\tNumber of operations\n");
        for (int i = 0; i <= 1000000; i++) {
            countM1 += 2; // init i && i < num check
            countM1 += 5;
            if (reversePalindrome(String.valueOf(i)) &&
reversePalindrome(decimal2Binary(String.valueOf(i)))) { // finds matching
binary and decimal palindromes
                M1List.add(i); // adding to array
                countM1++;
            }
            if (i % 50000 == 0) { // interval for the graph
                System.out.printf("\t%d\t\t %d\n", i, countM1);
            }
            countM1++;
        }
        endTimeM1 = System.currentTimeMillis() - startTimeM1;
        System.out.printf("Time taken: %d ms \n", endTimeM1);
```

```
        System.out.printf("Number of matching decimal and binary numbers:
%d\n", M1List.size());
        System.out.printf("-----------------------------------------------
----------------\n");


        startTimeM2 = System.currentTimeMillis();
        countM2++;
        System.out.print("Method2\nInterval\tNumber of operations\n");
        for (int i = 0; i <= 1000000; i++) {
            countM2 += 2; // init i && i < num check
            countM2 += 5;
            if (comparePalindrome(String.valueOf(i)) &&
reversePalindrome(decimal2Binary(String.valueOf(i)))) {
                M2List.add(i); // adding to array
                countM2++;
            }
            if (i % 50000 == 0) { // interval for the graph
                System.out.printf("\t%d\t\t %d\n", i, countM2);
            }
            countM2++;
        }

        endTimeM2 = System.currentTimeMillis() - startTimeM2;
        System.out.printf("Time taken: %d ms \n", endTimeM2);
        System.out.printf("Number of matching decimal and binary numbers:
%d\n", M2List.size());
        System.out.printf("-----------------------------------------------
----------------\n");


        startTimeM3 = System.currentTimeMillis();
        countM3++;
        System.out.printf("Method3\nInterval\tNumber of operations\n");
        for (int i = 0; i <= 1000000; i++) {
            countM3 += 2; // init i && i < num check
            countM3 += 5;
            if (stackAndQueue(String.valueOf(i)) &&
stackAndQueue(decimal2Binary(String.valueOf(i)))) {
                M3List.add(i);
                countM3++;
            }
                if (i % 50000 == 0) { // interval for the graph
                    System.out.printf("\t%d\t\t %d\n", i, countM3);
                }
                countM3++; // increment i

        }
        endTimeM3 = System.currentTimeMillis() - startTimeM3;
        System.out.printf("Time Taken: %d ms\n", endTimeM3);
        System.out.printf("Number of matching decimal and binary numbers:
%d\n", M3List.size());
        System.out.printf("-----------------------------------------------
----------------\n");


        startTimeM4 = System.currentTimeMillis();
        countM4++;
        System.out.print("Method4\nInterval\tNumber of operations\n");
        for (int i = 0; i <= 1000000; i++) {
           countM4 += 2; // init i && i < num check
```

```
            countM4 += 5;
            if (recursivePalindrome(String.valueOf(i)) &&
recursivePalindrome(decimal2Binary(String.valueOf(i)))) { // find matching
binary and decimal palindromes
                M4List.add(i);
                countM4++;
            }
            if (i % 50000 == 0) { // interval for the graph
                System.out.printf("\t%d\t\t %d\n", i, countM4);
            }
            countM4++; // increment i
        }
        endTimeM4 = System.currentTimeMillis() - startTimeM4;
        System.out.printf("Time Taken: %d ms\n", endTimeM4);
        System.out.printf("Number of matching decimal and binary numbers:
%d\n", M4List.size());
        System.out.printf("-----------------------------------------------
----------------\n");
    }

    //Static method for: Palindrome Method 1 (give it a name based on how
it works)
    //Takes a String as a parameter and return a boolean value
    public static Boolean reversePalindrome(String str) {
        String rev = ""; // initialise empty String to store reversed
string
        countM1++;

        boolean answer = false;
        countM1++;

        // reverse through the string
        for (int i = str.length() - 1; i >= 0; i--) { // decrement through
the string
            countM1 += 5;
            rev = rev + str.charAt(i); // add char to the reverse string
            countM1++;
        }

        if (str.equals(rev)) { // compare reversed string to the original
string
            answer = true;
            countM1 += 2;
        }
        countM1++;
        return answer;
    }


    //Static method for: Palindrome Method 2 (give it a name based on how
it works)
    //Takes a String as a parameter and return a boolean value
    public static boolean comparePalindrome(String str) {

        int left = 0; // start of the string
        int right = str.length() - 1; // end of the string
        countM2 += 3;

        while (left < right) { // when they meet in the middle or there are
no more characters to match
            if (str.charAt(left) != str.charAt(right)) { // compare element
```

```
by element
                countM2 += 5;
                return false;
            }
            left++; // increment through the string
            right--; // decrement through the string
            countM2 += 2;
        }
        countM2++;
        return true;
    }


    //Static method for: Palindrome Method 3 (give it a name based on how
it works)
    //Takes a String as a parameter and return a boolean value
    public static boolean stackAndQueue(String str) {
        Stack<Character> stack = new Stack<Character>(); // initialise
stack
        Queue<Character> queue = new ArrayDeque<Character>(); // initialise
queue
        countM3 += 2;

        for (int i = 0; i < str.length(); i++) { // loop through string
            countM3 += 3;
            char x = str.charAt(i);
            stack.push(x); // push the string to the stack
            queue.offer(x); // adds element to the end of the queue
            countM3 += 3;
        }

        while (!stack.isEmpty() && !queue.isEmpty()) { // while stack and
queue arent empty
            countM3 += 2;
            char x1 = stack.pop(); // pops element at the top of the stack
            char x2 = queue.poll(); // removes last element from the queue
            countM3 +=2;
            if (x1 != x2) { // compare element popped from stack and
removed from queue
                countM3 += 2;
                return false; // mismatch
            }
        }
        countM3++;
        return true;
    }

    //Static method for: Palindrome Method 4 (give it a name based on how
it works)
    //Takes a String as a parameter and return a boolean value
    public static boolean recursivePalindrome(String str){
        countM4 += 3;
        String reversed = reverse(str); // call reverse recursion
        return str.equals(reversed); // compare strings
    }

    //Static method for: Recursively reversing a String (to be used by
Method 4)
    //Takes a String and returns a String value of it reversed (must use
recursion)
    public static String reverse(String str) {
```

```
        countM4++; // if statement
        if (str.isEmpty()) {
            countM4++;
            return str;
        }
        countM4 += 5; // return,reverse,substring, +, charAt
        return reverse(str.substring(1)) + str.charAt(0);
    }

    //Static method for: Converting a decimal number into its equivalent
binary representation
    //Takes a String representation of a number as a parameter and return a
String value
    public static String decimal2Binary(String input){
        Stack<Integer> s = new Stack<>(); // create a stack to hold the
binary digits
        int in = Integer.parseInt(input); // converts the input to an
integer
        StringBuilder output = new StringBuilder(); // create a
StringBuilder object to hold the output binary digits

        // convert the input integer to binary by continuously
        // dividing it by 2 and pushing the remainder onto the stack
        if(!input.equals("0")){
            while(in != 0){
                s.push(in % 2);
                in /= 2;
            }
            // pop each binary digit off the stack and add it to the output
StringBuilder
            while (!s.isEmpty()){
                output.append(s.peek());
                s.pop();
            }
        }else {
            output.append(input);  // if the input is 0, add "0" to the
output StringBuilder
        }
        return output.toString(); // return binary number as a string
    }
}
```

Testing:

1. **Operations per interval up to 1,000,000. The intervals make it easy to plot the graph with the data already given**
2. **The time taken in milliseconds to execute the operations**
3. **The number of matching pairs of binary and decimal palindromes in the range of 1 – 1,000,000.**

```
Method1
Interval    Number of operations
    0        31
    50000        2035468
    100000       4138053
    150000       6493517
    200000       8849167
    250000       11204817
    300000       13560695
    350000       15916645
    400000       18272595
    450000       20628545
    500000       22984495
    550000       25340601
    600000       27696854
    650000       30053104
    700000       32409354
    750000       34765604
    800000       37121854
    850000       39478104
    900000       41834354
    950000       44190604
    1000000      46546860
```

```
Time taken: 161 ms
Number of matching decimal and binary numbers: 20
```

```
Method2
Interval    Number of operations
    0        13
    50000        808614
    100000       1617617
    150000       2428517
    200000       3239417
    250000       4050317
    300000       4861217
    350000       5672117
    400000       6483017
    450000       7293917
    500000       8104817
    550000       8915717
    600000       9726618
    650000       10537518
    700000       11348418
    750000       12159318
    800000       12970218
    850000       13781118
    900000       14592018
    950000       15402918
    1000000      16213818
```

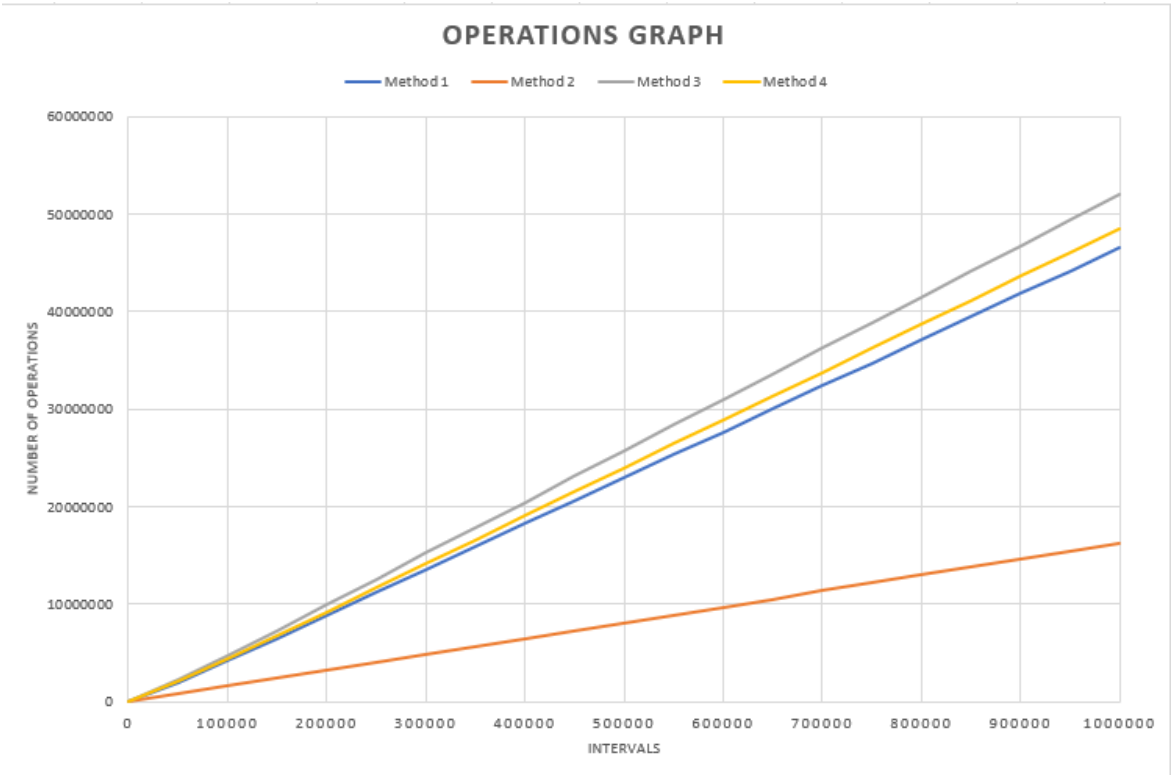
```
Time taken: 34 ms
Number of matching decimal and binary numbers: 20
```

```
Method3
Interval    Number of operations
    0        35
    50000        2315199
    100000       4697287
    150000       7325823
    200000       9954569
    250000       12582919
    300000       15211497
    350000       17840499
    400000       20469517
    450000       23098167
    500000       25726817
    550000       28356035
    600000       30985389
    650000       33614339
    700000       36243289
    750000       38872595
    800000       41501949
    850000       44130899
    900000       46759849
    950000       49389171
    1000000      52018571
```

```
Time Taken: 146 ms
Number of matching decimal and binary numbers: 20
```

```
Method4
Interval    Number of operations
    0        31
    50000        2135438
    100000       4338017
    150000       6793481
    200000       9249131
    250000       11704781
    300000       14160659
    350000       16616609
    400000       19072559
    450000       21528509
    500000       23984459
    550000       26440565
    600000       28896816
    650000       31353066
    700000       33809316
    750000       36265566
    800000       38721816
    850000       41178066
    900000       43634316
    950000       46090566
    1000000      48546822
```

```
Time Taken: 152 ms
Number of matching decimal and binary numbers: 20
```



- From our graph we can see method 2 was by far the slowest and method 3 was the fastest with method 1 and 4 not far behind.
- We plot our intervals on the x-axis and the number of operations on the y-axis.
- From our graphs we can identify the Big O Notation of each method
- Methods 1, 3 and 4 have a Big O Notation of O(nlogn). This is a loglinear complexity. O(nlogn) implies that logn operations will occur n times. O(nlogn) time is common in recursive sorting algorithms, binary tree sorting algorithms and most other types of sorts.
- Method 2 has a Big O Notation of O(n). This is a linear complexity.
- Method 2 is the most efficient of the methods as it will take less time to execute. If a new operation or iteration is needed every time n increases by one, then the algorithm will run in O(n) time.