Problem statement

For this problem, we will be looking at 4 different approaches for searching through a decimal and binary string to check if it's a palindrome or not. A palindrome is a string in which it is the same backwards as it is frontwards, for example, 55355 is a palindrome, and 1001001 is a palindrome. We will be testing each values from 1-1,000,000 in both decimal and binary to check for palindromes. We will be using 4 different techniques and recording the complexity of each method, taking into account the time each method takes, the number of primitive operation each method takes, and the complexity of each method as the numbers passed in get bigger, and how complex the algorithm gets over time. The 4 methods will use the following techniques, method 1 will simply use a loop to reverse the index of the input string and store each index into a new reversed string and then compare both strings, method 2 will compare the first and last element, then the second and second last, and so on until the two current indexes reach the middle, the third method will store the string into a stack and a queue, pop each element from the stack and queue into 2 new strings and compare the strings, as the string from the stack will naturally come out backwards, and the 4th method will call a recursive function which will keep calling itself to add the first element to the end of the string until the string is fully reversed, and then compare the reversed string with the original string. Once the code is functional and we have all our data, we will graph our data and compare each method on it's complexity.

Analysis and design notes.

For this assignment, I will firstly talk about the flow of the main testing method. I will first create 4 global variables to store the number of primitive operations for each method. For the testing, I will create 4 separate for loops that run until I <= 1,000,000 , as I want to record the time for each method separately and will do so at the start and end of each loop. Therefore I will also create a starttime, endtime, and totaltime variable for each of the 4 methods. I will also create 3 variables for each method, that will store the number of binary and decimal palindromes found, and one that will store the number of instances in which both binary and decimal values are palindromes. The reason I will create 3 for each method is so I can compare the results for each method with eachother to ensure each methods are getting the same results. So after starting each for loop, I will then call the respective method and pass in the decimal value, the binary value, obtained from passing the decimal value through the decimalTobinary()method, and add an if statement that will increment the binary and decimal palindrome counters if the methods return true. I will also add another if statement that will check if both (method(decimal) && method(binary)0 return true, and if it does it will increment the counter for the bothpalidromes variable. I will also add an interval at 50,000 runs that will print out the current number of operations in the loop, I will do so by using the modulo operator- if(i%50000 == 0) { print(operations);}. Once the loops have finished running, I will get the total time by subtracting startTime from the endrTime and print all of the results to the screen. I will do this 4 times for each method. I will now explain how I will design each method.

The decimal to binary method will run like this:
It will take in the decimal string as a parameter. I will then convert the decimal string to an integer using the Integer.parseInt() built in function. I will also create a stringbuilder object to store the binary output. I will add checks for special cases where, for example, if the number is 0 we do not need to perform operations on it and will simply return 0. Then inside a while loop that runs until the decimal number is not bigger than 0, I will perform a modulo 2 operation on the decimal number to get either a 1 or a 0 and add that to the beginning of the binary string, and then divide the decimal number by 2, and continue doing this until the binary string is complete and the decimal number is empty. The method will than return the binary string.

For the first method it will simply take in the decimal/binary string as a parameter and store it backwards in a new reversed string. It will do this by creating a new empty string called "reversed". It will then enter a for loop that have I = input.length()-1 and will run until i >= 0, and will add the ith character of the string to the reversed string. It will then compare the input string to the reversed string and return a Boolean based on if theyre equal or not.
For the second method, it will take in the decimal or binary string as a parameter and using a loop, compare the ith element with the input.length() – ith element. If at any point they don't match, the method will return false. If it checks the whole string with no mismatches, it will return true. I believe this method will be the fastest as I will only have to check half the array, as I will start at the first and last index and compare them, and work my way up and down the array until both indexes reach the middle. The loop will look like this:

```
For (I <= (input.length()-1)/2); i++) {
If (input.charAt(i) != input.charAt(input.length-1-i) {
Return false;
}
Else { return true}
```

For the third method, I will take the input string, create a stack and queue object, and pass the input string into both the stackand the queue. I will then pop and dequeue the stack and queue into 2 empty strings: stackstring and queuestring, and compare both. Because the stack is last in first out, the stackstring will be reversed naturally. The method will then return true or false. I predict that this method will take the longest and be the most complex as it requires the most operations and is comparing both of the entire strings.
For the 4th method, it will take in a string as a parameter. It will then return true or false based on if the string is equals to the reversed string gotten from the recursive function:
Return input.equals(recursiveFunction(input);
The recursiveFunction will take in a string as a parameter and return the reversed string. It will first check if the base case, if the input string is empty, meaning there are no characters left to reverse. If this is true, it will return the input string. If the base case isn't true, the function will create a string called reverse that will be equal to a substring of the input string from the 2nd index onwards. The function will then call itself recursively, passing in a new string consisting of the "reversed" string with the first

index of the input string appended to the end of it. This will continue to happen until the original input string is empty.
The global variables for counting the amount of primitive operations will be incremented after every operation in these methods, such as declaring variables, assigning values to variables, if statements, incrementing variables and returning values from methods.

Code

```
public class AssignmentThree {
    //variables to store the times for each method
    private static long startTime1, endTime1, totalTime1;
    private static long startTime2, endTime2, totalTime2;
    private static long startTime3, endTime3, totalTime3;
    private static long startTime4, endTime4, totalTime4;
   //variables to store the primitive operations for each method
    private static long primOps1 = 0;
    private static long primOps2 = 0;
    private static long primOps3 = 0;
    private static long primOps4 = 0;

    public static void main(String[] args) {
        AssignmentThree a3 = new AssignmentThree();
        //variables to store the number of palidromes for each method
        long decPalidromes1 = 0;
        long binPalidromes1 = 0;
        long bothPalidromes1 = 0;
        long decPalidromes2 = 0;
        long binPalidromes2 = 0;
        long bothPalidromes2 = 0;
        long decPalidromes3 = 0;
        long binPalidromes3 = 0;
        long bothPalidromes3 = 0;
        long decPalidromes4 = 0;
        long binPalidromes4 = 0;
        long bothPalidromes4 = 0;

        startTime1 = System.currentTimeMillis();
        for (int i = 0; i <= 1000000; i++) { //loop that passes through
1,000,000 values
            String decimal = String.valueOf(i); //getting the decimal value
of the current value
            String binary = decimalToBinary(decimal); //getting the binary
value of the curent value
            if (reverseString(decimal)) {
                decPalidromes1++;
            }
            if (reverseString(binary)) {
                binPalidromes1++;
            }
            if (reverseString(decimal) && (reverseString(binary))) {
                bothPalidromes1++;
            }
            if(i % 50000 == 0) { //checks the number of primitve operations
every 50,000 runs
                System.out.println("\nMethod 1 primitive Operations after
```

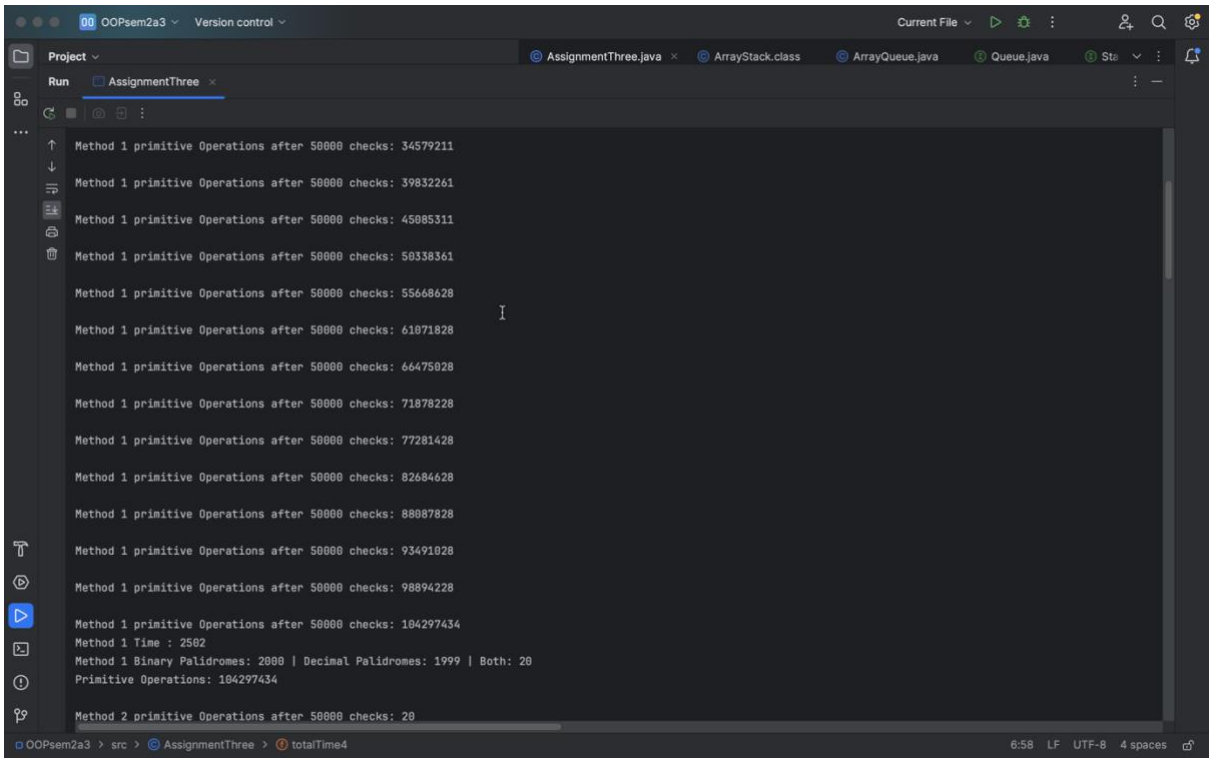
```
50000 checks: " + primOps1);
            }
        }
        endTime1 = System.currentTimeMillis();
        totalTime1 = endTime1 - startTime1; //total time taken for the
function
        System.out.println("Method 1 Time : " + totalTime1);
        System.out.println("Method 1 Binary Palidromes: " + binPalidromes1
+ " | Decimal Palidromes: " + decPalidromes1 + " | Both: " +
bothPalidromes1);
        System.out.println("Primitive Operations: " + primOps1);
        //the following 4 loops work the same as the first, except they all
call 1 of each 4 methods
        startTime2 = System.currentTimeMillis();
        for (int i = 0; i <= 1000000; i++) {
            String decimal = String.valueOf(i);
            String binary = decimalToBinary(decimal);
            if (compareElements(decimal)) {
                decPalidromes2++;
            }
            if (compareElements(binary)) {
                binPalidromes2++;
            }
            if (compareElements(decimal) && (compareElements(binary))) {
                bothPalidromes2++;
            }
            if(i % 50000 == 0) {
                System.out.println("\nMethod 2 primitive Operations after
50000 checks: " + primOps2);
            }
        }
        endTime2 = System.currentTimeMillis();
        totalTime2 = endTime2 - startTime2;
        System.out.println("Method 2 Time : " + totalTime2);
        System.out.println("Method 2 Binary Palidromes: " + binPalidromes2
+ " | Decimal Palidromes: " + decPalidromes2 + " | Both: " +
bothPalidromes2);
        System.out.println("Primitive Operations: " + primOps2);

        startTime3 = System.currentTimeMillis();
        for (int i = 0; i <= 1000000; i++) {
            String decimal = String.valueOf(i);
            String binary = decimalToBinary(decimal);
            if (stackCompToQueue(decimal)) {
                decPalidromes3++;
            }
            if (stackCompToQueue(binary)) {
                binPalidromes3++;
            }
            if (stackCompToQueue(decimal) && (stackCompToQueue(binary))) {
                bothPalidromes3++;
            }
            if(i % 50000 == 0) {
                System.out.println("\nMethod 3 primitive Operations after
50000 checks: " + primOps3);
            }
        }
        endTime3 = System.currentTimeMillis();
        totalTime3 = endTime3 - startTime3;
        System.out.println("Method 3 Time : " + totalTime3);
        System.out.println("Method 3 Binary Palidromes: " + binPalidromes3
```

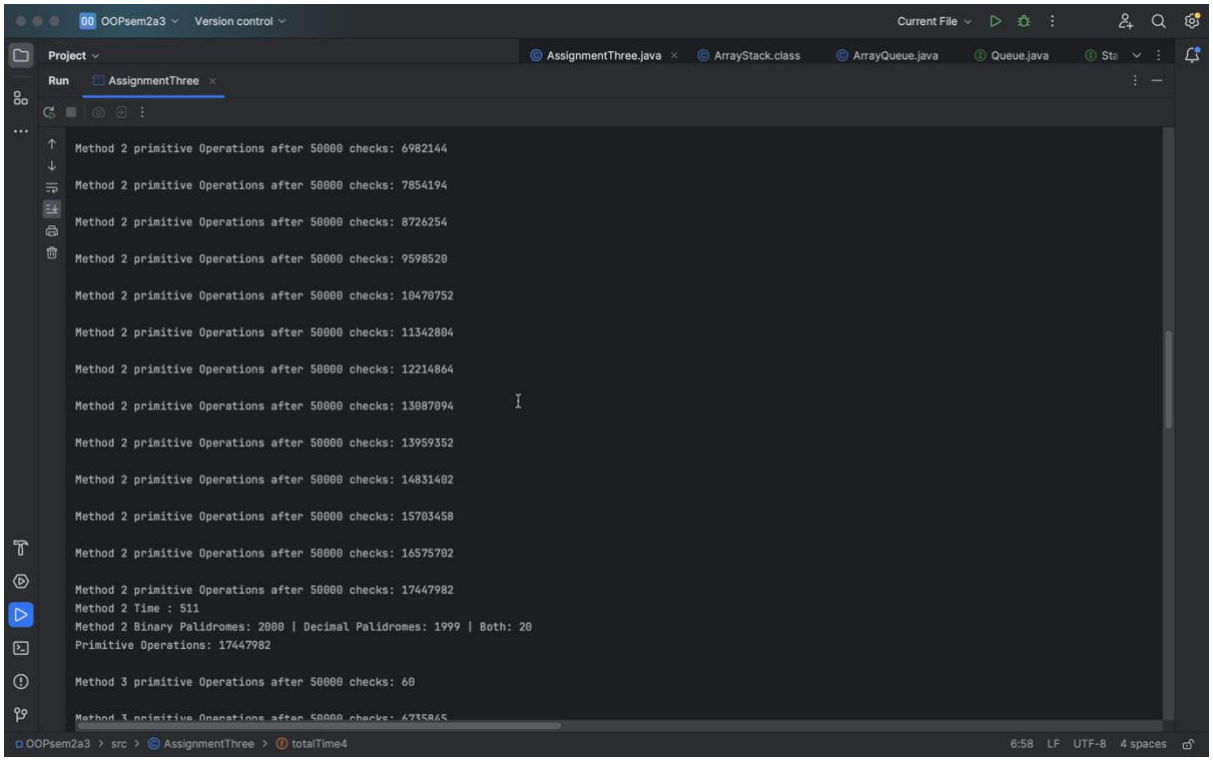
```
+ " | Decimal Palidromes: " + decPalidromes3 + " | Both: " +
bothPalidromes3);
        System.out.println("Total primitive Operations: " + primOps3);

        startTime4 = System.currentTimeMillis();
        for (int i = 0; i <= 1000000; i++) {
            String decimal = String.valueOf(i);
            String binary = decimalToBinary(decimal);
            if (recursion(decimal)) {
                decPalidromes4++;
            }
            if (recursion(binary)) {
                binPalidromes4++;
            }
            if (recursion(decimal) && (recursion(binary))) {
                bothPalidromes4++;
            }
            if(i % 50000 == 0) {
                System.out.println("\nMethod 4 primitive Operations after
50000 checks: " + primOps4);
            }
        }
        endTime4 = System.currentTimeMillis();
        totalTime4 = endTime4 - startTime4;
        System.out.println("Method 4 Time : " + totalTime4);
        System.out.println("Method 4 Binary Palidromes: " + binPalidromes4
+ " | Decimal Palidromes: " + decPalidromes4 + " | Both: " +
bothPalidromes4);
        System.out.println("Total primitive Operations: " + primOps4);


    }

    public static boolean reverseString(String input) {
        String reversed = ""; primOps1 += 2; //2 primitive operations for
instansiating the reversed string and the int i
        for(int i = input.length() - 1; i >= 0; i--) { //work through the
string starting at the end and add each character to the reversed string
            reversed += input.charAt(i); primOps1 += 3; //3 primitive
operations for adding a character to the reversed string, checking that i
<= 0, and decrementing i
        }
        if(input.equals(reversed)) {
            primOps1 += 2; //2 primitive operations when we compare the
strings, and return a value
            return true;
        }
        else {
            primOps1 += 2; //2 primitive operations when we compare the
strings, and return a value
            return false;
        }
    }
    public static boolean compareElements(String input) {
        primOps2 += 1; //primitive operation for instansiating i
        for (int i = 0; i <= (input.length() - 1)/2; i++) {
            primOps2 += 2; //primitive operations for checking i<input
length and incrementing i
            if(input.charAt(i) != input.charAt((input.length() - 1) - i)) {
//if at any points the elements don't match, return false
                primOps2 += 2; //primitive operations for checking the ith
```

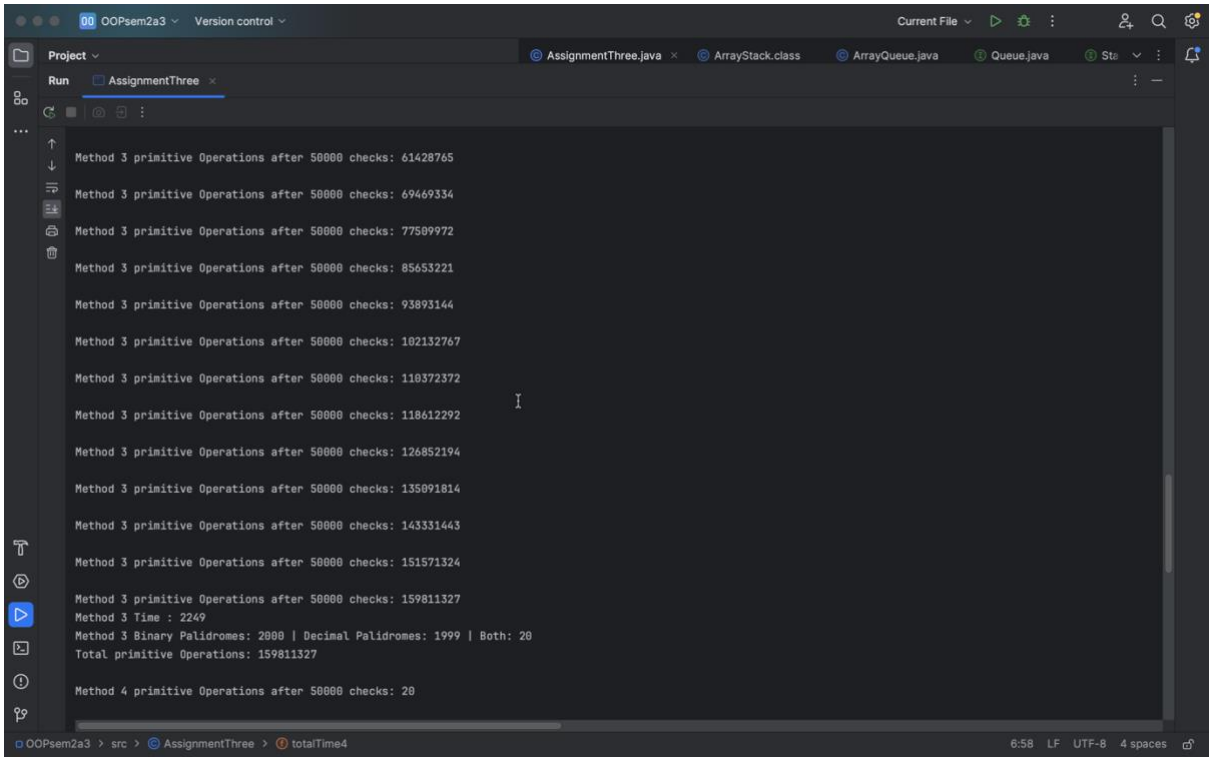
```
element and the inputlength-1-ith element, and returning a value
                return false;
            }
        }
        primOps2 += 2; //primitive operations for checking the ith element
and the inputlength-1-ith element, and returning a value
        return true; //if all the elements match
    }
    public static boolean stackCompToQueue(String input) {
        String stacknum = "";
        String queuenum = ""; primOps3 += 2; //2 primitive operations for
instansiating the strings that will be used for holding the popped and
dequeued strings
        ArrayStack stack = new ArrayStack(input.length());
        ArrayQueue queue = new ArrayQueue(input.length()); primOps3 += 3;
//3 primiitive operations for instansiating the stack and queue objects and
instansiating i
        for (int i = 0; i < input.length(); i++) {
            stack.push(input.charAt(i)); //push each character from the
string to the stack
            primOps3 += 2; //for incrementing i and pushing the input
string
        }
        primOps3 += 1; //for instansiating k
        for (int k = 0; k < input.length(); k++) {
            queue.enqueue(input.charAt(k)); //enqueue each character from
the string from the queue
            primOps3 += 2; //for incrementing k and enqueing the input
string
        }
        while (!stack.isEmpty() && !queue.isEmpty()) { //pop and dequque to
the two strings
            stacknum += (char) stack.pop();
            queuenum += (char) queue.dequeue();
            primOps3 += 3; //for checking that the queue and stack arent
empty, and popping and dequeuing each character
                if (!stacknum.equals(queuenum)) { primOps3 += 2; //for
comparing both the strings, and returning a value
                    return false; //if the strings arent equal, not a
palidrome
                }
            }
        primOps3 += 2; //for comparing both the strings, and returning a
value
        return true;
    }
    public static boolean recursion(String input) {
        primOps4 += 1; //for retrning a value, returns true or false based
on if the strings match
        return input.equals(reverse(input)); //if the original and reversed
string are equal, return true or false
    }
    public static String reverse(String input) {
        if (input.isEmpty()) {
            primOps4 += 2; //for checking if the input string is empty and
returning it
            return input;
        } else {
            String reverse = reverse(input.substring(1));
            primOps4 += 2; //primitive operation for declaring the reversed
string as a substring of the input string, excluding the first element ,
```

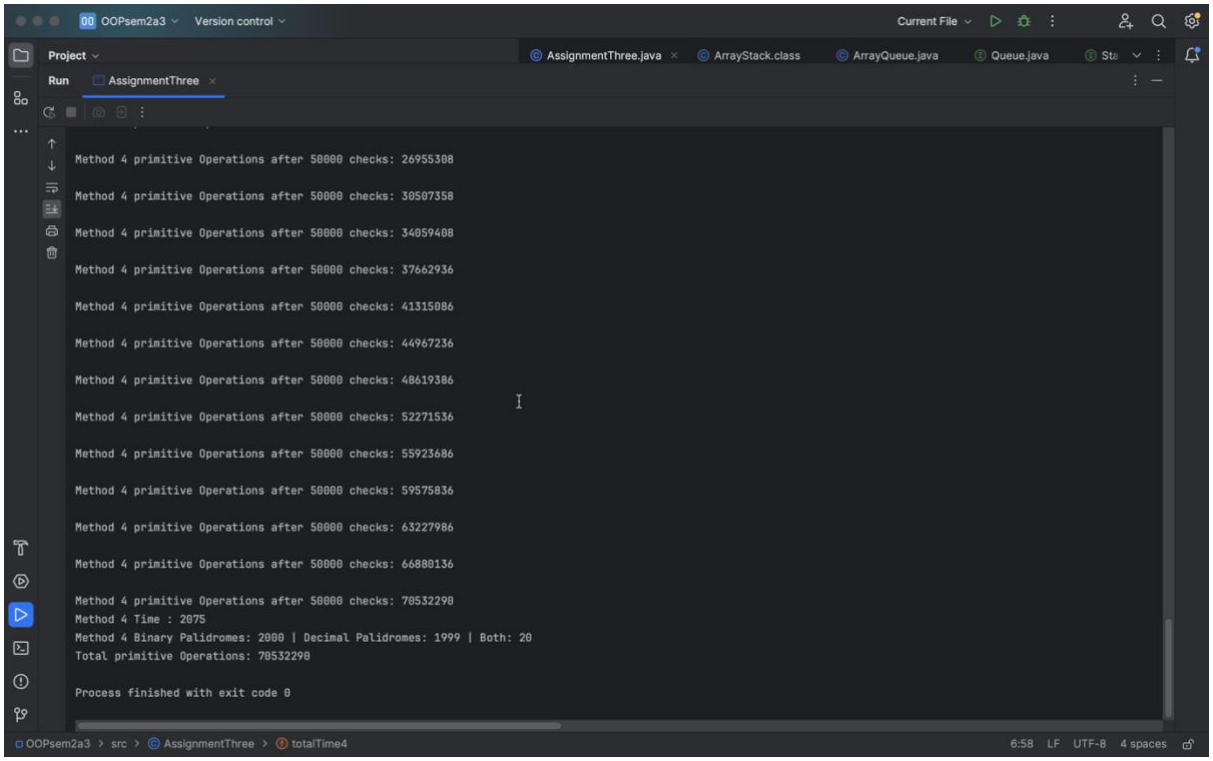
```
and another primitive operation for recalling the function and appending
the first character to the end of the reversed substring
            return reverse + input.charAt(0); //
        }
    }

    public static String decimalToBinary(String decimal) {
        int decimalNum = Integer.parseInt(decimal); //converting the string
to a decimal number
        StringBuilder binary = new StringBuilder(); //instansiating a new
string builder
        if (decimalNum == 0) { //in the special case the number is ), we
dont need to perform any operations on it
            return "0";
        }
        else {
            while(decimalNum > 0) { //loop will run until the decimal
number hasnt been reduced to 0
                binary.insert(0, decimalNum % 2); //perform modulo
operation on the number and insert the 1 or 0 at the beginning of the
string builder, at index 0
                decimalNum = decimalNum / 2; //half the decimal number
            }
        }
        return binary.toString(); //convert the stringbuilder to a string
    }
}
```

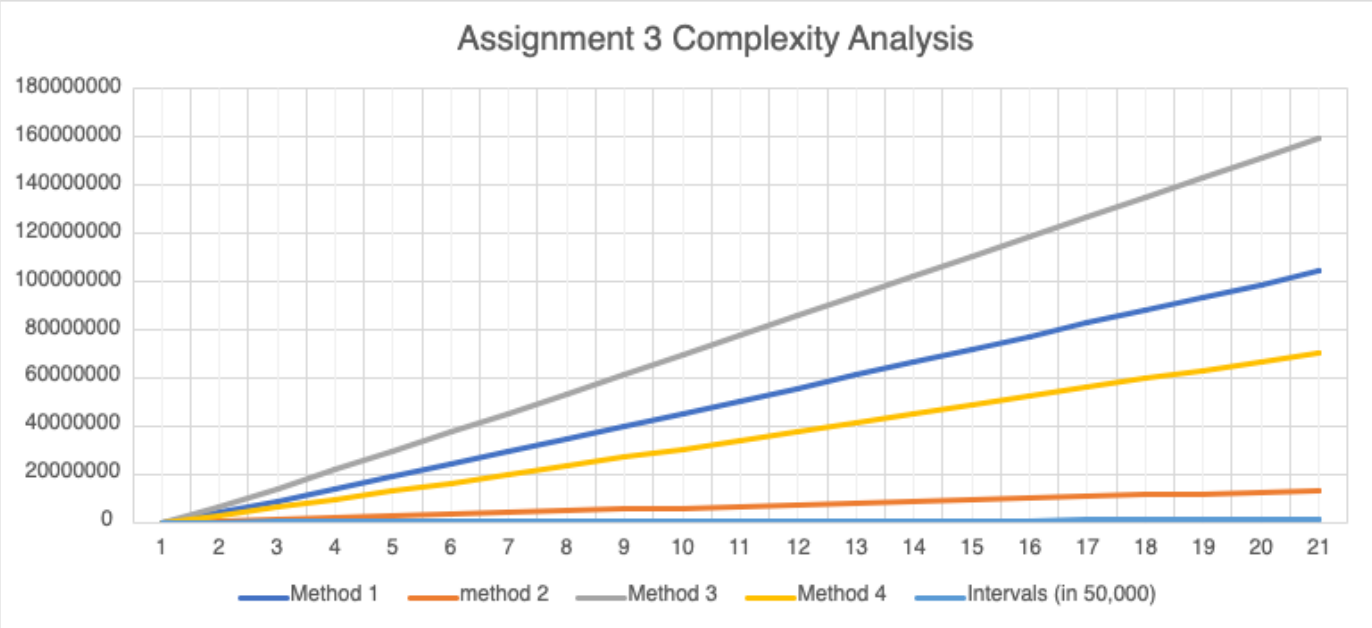
testing.




OOPsem2a3
Version control
Current File
Project
AssignmentThree.java
ArrayStack.class
ArrayQueue.java
Queue.java
Run
AssignmentThree
Method 2 primitive Operations after 50000 checks: 6982144
Method 2 primitive Operations after 50000 checks: 7854194
Method 2 primitive Operations after 50000 checks: 8726254
Method 2 primitive Operations after 50000 checks: 9598520
Method 2 primitive Operations after 50000 checks: 10470752
Method 2 primitive Operations after 50000 checks: 11342804
Method 2 primitive Operations after 50000 checks: 12214864
Method 2 primitive Operations after 50000 checks: 13087094
Method 2 primitive Operations after 50000 checks: 13959352
Method 2 primitive Operations after 50000 checks: 14831402
Method 2 primitive Operations after 50000 checks: 15703458
Method 2 primitive Operations after 50000 checks: 16575702
Method 2 primitive Operations after 50000 checks: 17447982
Method 2 Time : 511
Method 2 Binary Palidromes: 2000 | Decimal Palidromes: 1999 | Both: 20
Primitive Operations: 17447982
Method 3 primitive Operations after 50000 checks: 60
OOPsem2a3 > src > AssignmentThree > totalTime4
6:58 LF UTF-8 4 spaces


OOPsem2a3
Version control
Current File
Project
AssignmentThree.java
ArrayStack.class
ArrayQueue.java
Queue.java
Run
AssignmentThree
Method 3 primitive Operations after 50000 checks: 61428765
Method 3 primitive Operations after 50000 checks: 69469334
Method 3 primitive Operations after 50000 checks: 77509972
Method 3 primitive Operations after 50000 checks: 85653221
Method 3 primitive Operations after 50000 checks: 93893144
Method 3 primitive Operations after 50000 checks: 102132767
Method 3 primitive Operations after 50000 checks: 110372372
Method 3 primitive Operations after 50000 checks: 118612292
Method 3 primitive Operations after 50000 checks: 126852194
Method 3 primitive Operations after 50000 checks: 135091814
Method 3 primitive Operations after 50000 checks: 143331443
Method 3 primitive Operations after 50000 checks: 151571324
Method 3 primitive Operations after 50000 checks: 159811327
Method 3 Time : 2249
Method 3 Binary Palidromes: 2000 | Decimal Palidromes: 1999 | Both: 20
Total primitive Operations: 159811327
Method 4 primitive Operations after 50000 checks: 20
OOPsem2a3 > src > AssignmentThree > totalTime4
6:58 LF UTF-8 4 spaces





From the graph here, we can see that method 3 which uses the array stack has the highest number of operations, and has a bad complexity analysis graph line, and is the least effective. From the graph we can see that it's big O complexity is O(n log n), which is bad. Method 2 is the most effective and is also the quickest, with a big O complexity of O(1)/O(log n), which is good. Method 1 and 4 are somewhat similar but we can see that method 4 is slightly more effective than method 1, method 1 has a big O complexity of O(n log n) and method 4 is somewhere just between O(n) and O(n log n).