Maxwell Maia

21236277

CT2109 Object Oriented Programming: Data Structures and Algorithms

Assignment 3: Double-base Palindromes and Complexity Analysis

Text

Description automatically generated

Problem Statement

A program that tells the user whether the input is a palindrome or not. A palindrome reads the same forwards as it does backwards. E.g. 1010101 and 7779777 are palindromes.

The program uses multiple different methods to determine whether the input is a palindrome or not. Each method returns a Boolean that is true when the input is determined to be a palindrome. We are going to use this to run automated tests to compare the efficiency of these methods. The amount of primitive operations and time taken will be recorded for each algorithm to check the palindromes from 0 to 1000000 (1 million) for both decimal and binary numbers. At the end of the execution of the program, the user will be able to see   
1. the time taken for each algorithm (decimal, binary and both added together),  
2. the amount of primitive operations each algorithm made (amount of operations of the decimal and binary checking added together)  
3. the amount of instances where both the decimal and binary equivalent of a number are palindromic

The step count method will be used for time complexity analysis.

The number of operations will be plotted against the size of the problem using Excel to allow a visual comparison of the time complexity between the different palindrome checking algorithms. This is equivalent to plotting a graph of the current total primitive operations against some amount of numbers checked. Data for this will be collected by displaying the current primitive operation counts every 50000 numbers checked.

Analysis and Design Notes

Method 1 – String Equals  
isPalindromeStringEquals(String input)  
For the length of the String, starting from the last index, append the charAt that location of the input a new String called reversed.

If input equals reversed, using the .equals method, then the input is a palindrome so return true; else the input is not a palindrome so return false.

Method 2 – Character By Character

isPalindromeCharacterByCharacter(String input)

Compare the first and the last characters

If they are equal then keep comparing

If at the end they are all equal then the input is a palindrome

If at any point the characters are not equal then it is not a palindrome.

If an input of length 1 is input then just return true.

Need to know how many times to loop when comparing

Figuring this out…

The example input is: 1221

Length: 4

Even length

4/2 = 2

Compare the first 2 with the last 2

Compare length/2 forward to backward

Loop duration is length/2

Now the example input is 12321

Length: 5

Odd length

(4-1)/2 = 2

Compare the first 2 with the last 2

Compare (length-1)/2 forward to backward

Leave the middle one alone because it is equal to itself

Method 3 – Stack Queue

isPalindromeStackQueue(String input)

Basic idea:

Place the input into both a stack and queue. If the input is a palindrome, the first character popped from the stack will be the same as the first character dequeued, and so on for the second and subsequent characters until the stack and queue are empty. This works because a Stack is LIFO “last in first out” and a queue is FIFO “first in first out.”

Pseudocode:

Create a stack and a queue

For each character in the input string, push that character to the stack and queue that character

While the stack is not empty, check that what you pop from the stack matches what you dequeue from the queue: if they match, do nothing, else: return false (then it is not a palindrome)

Return true; (will get to this if all characters popped and dequeued match each other)

Method 4 – Recursion Reverse

isPalindromeRecursionReverse(String input)

This method uses a utility method called **reverse()**:

String reverse(String input)

Example input string “abcde”

We want reverse() to return “edcba”

Using recursion:

Function call: input.length(): New call: reverse(input without the last character)

1: reverse(“abcde”) 5 bcde

2: reverse(“bcde”) 4 cde

3: reverse(“cde”) 3 de

4: reverse(“de”) 2 e

5: reverse(“e”) 1 [Base case reached, start returning values]

Function call: returns:

5: input = e e e

4: input = de e + input.substring(0, 1) ed

3: input = cde ed + input.substring(0, 1) edc

2: input = bcde edc + input.substring(0, 1) edcb

1: input = abcde edcb + input.substring(0, 1) edcba

Note function call 1 has “abcde” stored in the input variable. It didn’t send “a” to any reverse call. Function call 1 appends the “a” after all of the reverse calls return their values.

The returned value of function call 1 is “edcba” which is the reverse of the input “abcde”.

Back to method 4: isPalindromeRecursionReverse(String input)

Use the reverse function to get a reversed String of the input.

If the reverse String is equal to the input String, then the input is a palindrome.

Decimal to binary String utility method:

public String decimalToBinary(String decimal)

Input = decimal as a String

Output = converted binary as a String

Convert the decimal String into an int - Using: Integer.parseInt(decimal)

Then convert the int to a binary String - Using: Integer.toBinaryString()

return Integer.toBinaryString(Integer.parseInt(decimal));

Main method

Firstly test that the decimal to binary method works

Begin palindrome checking  
Going to do 3 runs…. (to get an accurate reading of time taken)  
FOR 3 RUNS  
{  
I am going to loop through numbers 0 to 1000000  
Start of Looping 0 to 1000000  
On each iteration of the loop I am going to  
- output the current primitive operation counts every 50000 iterations (used for csv file for graph)  
- get a String for the current number (decimal)  
- get a String for the current number using my decimalToBinary method (binary)  
then for all 4 palindrome methods I am:  
- grabbing the current system time in milliseconds as the “start time”  
- performing the method  
- adding the time elapsed (current time - start time) to a variable that tracks the total time spent for a particular method using Binary or Decimal inputs  
(ie. in total there are 8 timers: 1 for binary, 1 for decimal for the 4 different methods)  
- Also note that within each method there are statements that increment a global primitive operation counter. There will be 4 counters in total, one for each method. The counter is, therefore, the sum of the primitive operations for palindrome checks for the binary and decimal numbers.  
- the variable (we will call is instancesPalindromicBothBases) that tracks the number of times that a number is palindromic in both binary and decimal, is incremented. This is done by storing the boolean result of one of the methods for both decimal and binary. If both the decimal and binary booleans are true, then increment the counter. Set the booleans to false for the next iteration.  
End of Looping 0 to 1000000

Reset counters for next run

}

Code

public class Palindrome

{

//Primitive operation counters for each of the palindrome detecting methods

//Variables for keeping track of time taken

private static int stringCount = 0;

private static long stringTime = 0;

private static long stringTimeBinary = 0;

private static int characterCount = 0;

private static long characterTime = 0;

private static long characterTimeBinary = 0;

private static int stackQueueCount = 0;

private static long stackQueueTime = 0;

private static long stackQueueTimeBinary = 0;

private static int recursiveCount = 0;

private static long recursiveTime = 0;

private static long recursiveTimeBinary = 0;

private static int instancesPalindromicBothBases = 0;

public static void main(String[] args) {

//Create a test object

Palindrome t = new Palindrome();

int decimal = 21;

System.out.println("Decimal to binary String test");

System.out.println("================================================");

System.out.println("Decimal: " + decimal);

System.out.println("Binary: " + t.decimalToBinary(String.valueOf(decimal)));

System.out.println("End of decimal to binary String test");

System.out.println("================================================");

String input = "";

String inputBinary = "";

//My test code for individual functions

// //Use method 1

// boolean stringEqualsPalindrome = t.isPalindromeStringEquals(input);

//

// //Use method 2

// boolean characterByCharacterPalindrome = t.isPalindromeCharacterByCharacter(input);

//

// //Use method 3

// boolean stackQueuePalindrome = t.isPalindromeStackQueue(input);

//

// //Use method 4

// boolean recursionReversePalindrome = t.isPalindromeRecursionReverse(input);

//

// if(characterByCharacterPalindrome)

// {

// System.out.println(input + ": is a palindrome");

// }

// else

// {

// System.out.println(input + " is not a palindrome");

// }

//Check the numbers 0-1000000 for palindromes in binary and decimal

//Do this three times to get an accurate reading of time taken

System.out.println("\n\nInitiating palindrome testing...");

for(int r = 1; r <= 3; r++)

{

System.out.println("\nRun " + r);

System.out.println("==================================================");

System.out.println("CSV data, paste into excel\n");

//Loop through every number from 1 including 1000000

long start = 0;

boolean isPalindromeDecimal = false;

boolean isPalindromeBinary = false;

for (int i = 0; i <= 1000000; i++) {

//Generate CSV

//Note: CSV data counts primitive operations for both the decimal and binary number together

//Every 50000th number, display the current operation counts

//Note: the last value (1000000) will not be included in this csv data

//because we are checking 1000001 numbers.

//1000001 numbers do not fit into a graph if we divide the graph into ranges of 50000

if (i % 50000 == 0) {

System.out.println(i + "," + stringCount + "," + characterCount + "," + stackQueueCount + "," + recursiveCount);

}

//Get the next number to be checked

input = String.valueOf(i);

//Get the binary of that number

inputBinary = t.decimalToBinary(input);

//Running the algorithms for the current number

//Use each algorithm on both the decimal and binary number

//Every time an algorithm is called, calculate the amount of time it takes to run it

//and add that time to a variable that is keeping track of its total time taken

//StringEquals method

//Decimal

start = System.currentTimeMillis();

isPalindromeDecimal = t.isPalindromeStringEquals(input);

stringTime += System.currentTimeMillis() - start;

//Binary

start = System.currentTimeMillis();

isPalindromeBinary = t.isPalindromeStringEquals(inputBinary);

stringTimeBinary += System.currentTimeMillis() - start;

//CharacterByCharacter method

//Decimal

start = System.currentTimeMillis();

isPalindromeDecimal = t.isPalindromeCharacterByCharacter(input);

characterTime += System.currentTimeMillis() - start;

//Binary

start = System.currentTimeMillis();

isPalindromeBinary = t.isPalindromeCharacterByCharacter(inputBinary);

characterTimeBinary += System.currentTimeMillis() - start;

//StackQueue method

//Decimal

start = System.currentTimeMillis();

isPalindromeDecimal = t.isPalindromeStackQueue(input);

stackQueueTime += System.currentTimeMillis() - start;

//Binary

start = System.currentTimeMillis();

isPalindromeBinary = t.isPalindromeStackQueue(inputBinary);

stackQueueTimeBinary += System.currentTimeMillis() - start;

//RecursionReverse method

//Decimal

start = System.currentTimeMillis();

isPalindromeDecimal = t.isPalindromeRecursionReverse(input);

recursiveTime += System.currentTimeMillis() - start;

//Binary

start = System.currentTimeMillis();

isPalindromeBinary = t.isPalindromeRecursionReverse(inputBinary);

recursiveTimeBinary += System.currentTimeMillis() - start;

//How many instances is a number palindromic in both bases?

//If this current number is palindromic in both bases, increment instancesPalindromicBothBases

//Note: the assignment of isPalindromeDecimal and isPalindromeBinary is repeated throughout

//the methods so that the time of one of the algorithms isn't more affected

if (isPalindromeDecimal && isPalindromeBinary) {

instancesPalindromicBothBases++;

}

}

System.out.println("\nResults for run " + r);

System.out.println("----------------------------------------");

System.out.println("Time taken for 0-1000000 (decimal only)");

System.out.println("----------------------------------------");

System.out.println("String Equals:\t\t\t" + stringTime + "ms");

System.out.println("Character by Character:\t" + characterTime + "ms");

System.out.println("Stack Queue:\t\t\t" + stackQueueTime + "ms");

System.out.println("Recursion Reverse:\t\t" + recursiveTime + "ms");

System.out.println("----------------------------------------");

System.out.println("Time taken for 0-1000000 (binary only)");

System.out.println("----------------------------------------");

System.out.println("String Equals:\t\t\t" + stringTimeBinary + "ms");

System.out.println("Character by Character:\t" + characterTimeBinary + "ms");

System.out.println("Stack Queue:\t\t\t" + stackQueueTimeBinary + "ms");

System.out.println("Recursion Reverse:\t\t" + recursiveTimeBinary + "ms");

System.out.println("----------------------------------------");

System.out.println("Time taken for 0-1000000 (total of decimal and binary)");

System.out.println("----------------------------------------");

System.out.println("String Equals:\t\t\t" + (stringTime + stringTimeBinary) + "ms");

System.out.println("Character by Character:\t" + (characterTime + characterTimeBinary) + "ms");

System.out.println("Stack Queue:\t\t\t" + (stackQueueTime + stackQueueTimeBinary) + "ms");

System.out.println("Recursion Reverse:\t\t" + (recursiveTime + recursiveTimeBinary) + "ms");

System.out.println("----------------------------------------");

System.out.println("Primitive operations for 0-1000000 (total of decimal and binary)");

System.out.println("----------------------------------------");

System.out.println("String Equals:\t\t\t" + stringCount + " primitive operations");

System.out.println("Character by Character:\t" + characterCount + " primitive operations");

System.out.println("Stack Queue:\t\t\t" + stackQueueCount + " primitive operations");

System.out.println("Recursion Reverse:\t\t" + recursiveCount + " primitive operations");

System.out.println("----------------------------------------");

System.out.println("Instances both bases palindromic for 0-1000000");

System.out.println("----------------------------------------");

System.out.println("Palindromic in both bases in: " + instancesPalindromicBothBases + " instances");

System.out.println("----------------------------------------");

System.out.println("End of run " + r);

System.out.println("==================================================\n");

//End of a run

//Reset all variables

stringTime = 0;

characterTime = 0;

stackQueueTime = 0;

recursiveTime = 0;

stringTimeBinary = 0;

characterTimeBinary = 0;

stackQueueTimeBinary = 0;

recursiveTimeBinary = 0;

stringCount = 0;

characterCount = 0;

stackQueueCount = 0;

recursiveCount = 0;

instancesPalindromicBothBases = 0;

}

//End of all runs

System.out.println("All runs complete");

}

public Palindrome()

{

}

public boolean isPalindromeStringEquals(String input)

{

String reversed = "";

stringCount++;

//Reverse the string using loop

//Count the initialization of i to the length() function

stringCount = stringCount + 2;

for(int i = input.length()-1; i >= 0; i--)

{

//Count loop condition and i--

stringCount = stringCount + 2;

stringCount = stringCount + 3;

reversed += input.charAt(i);

}

//The last evaluation of the conditional statement is false and the loop doesn't run

stringCount++;

//Examined String.java and StringLatin1.java to count primitive operations of equals

//Counted equals to have 7 + 1 + 5\*input.length() + 1 + 1 primitive operations

stringCount = stringCount + (10+(5\*input.length()));

if(input.equals(reversed))

{

stringCount++;

//The input is a palindrome

return true;

}

else

{

stringCount++;

//The input is not a palindrome

return false;

}

}

public boolean isPalindromeCharacterByCharacter(String input)

{

//An input of length 1 is a palindrome

characterCount = characterCount + 2;

if(input.length() == 1)

{

characterCount++;

return true;

}

//This method compares the first character to the last and keeps comparing, moving towards the centre

//If the input is odd, then we don't need to compare the middle item to anything

//121 is a palindrome

//The length is 3

//We would compare the first and last, 1 == 1.

//Don't have anything to compare the middle item to

//All comparisons were equal: therefore 121 is a palindrome

//Find the amount of times we need to compare characters

int i = 0;

int length = input.length();

characterCount = characterCount + 3;

if(length % 2 == 0)

{

characterCount = characterCount + 2;

i = length / 2;

}

else

{

characterCount = characterCount + 3;

i = (length - 1) / 2;

}

//Compare first and last letters

//if any of the comparisons are not equal, then the input is not a palindrome

characterCount++;

for(int j = 0; j<i; j++)

{

//Count loop condition and i++

characterCount = characterCount + 2;

//old code if needed: (input.charAt(j)+"").equals(input.charAt(length-j-1)+"")

characterCount = characterCount + 4;

if(input.charAt(j) == input.charAt(length-j-1))

{

//All good, the characters are equal, so far the input is a palindrome

}

else

{

characterCount++;

return false;

}

}

//The last evaluation of the conditional statement is false and the loop doesn't run

characterCount++;

characterCount++;

return true;

}

public boolean isPalindromeStackQueue(String input)

{

//Create a stack

Stack s = new ArrayStack(20);

//ArrayStack constructor has 2 time steps in it

stackQueueCount = stackQueueCount + 2;

//Create a queue

Queue q = new ArrayQueue(20);

//ArrayQueue constructor has 2 time steps in it

stackQueueCount = stackQueueCount + 2;

//Add each character to both stack and queue as it is read

//Count the initialization of i

stackQueueCount++;

for(int i = 0; i < input.length(); i++)

{

//Count loop condition and i++

stackQueueCount = stackQueueCount + 2;

//Push() takes 5 operations

//1 for if

//2 for isFull()

//1 for top++

//1 for assigning the element to the top index of the array

s.push(input.charAt(i));

//charAt(i) is 1 primitive operation

stackQueueCount = stackQueueCount + 6;

//Enqueue() takes 5 operations

//2 for isFull()

//1 for rear++

//2 for Q[rear] = n

q.enqueue(input.charAt(i));

//charAt(i) is 1 primitive operation

stackQueueCount = stackQueueCount + 6;

}

//The last evaluation of the conditional statement is false and the loop doesn't run

stackQueueCount++;

//For calculating dequeue()'s primitive operations

int r = (input.length() - 1);

//Compare characters from stack and queue

while(!s.isEmpty())

{

//1 primitive operation per isEmpty()

stackQueueCount++;

//Counting pop()'s primitive operations

// Object element; +1

// isEmpty() +1

// element = S[top]; +2

// S[top] = null; +2

// top--; +1

// return element; +1

// pop() takes 8 primitive operations

//Counting dequeue()'s primitive operations

// isEmpty() +2

// Object toReturn = Q[0]; +2

// int i = 1; +1

// while loop:

// r+1 "while condition"

// + r "i-1 calculation" + r "access array" + r "access array" + r "assignment" + r "i++ calculation")

// rear--; calculation +1

// That is: 6r + 7

// r is initially equal to the length of the string -1,

// because rear is equivalent to the amount of items in the ArrayQueue

// and r is decremented every iteration of this while loop,

// because an item is dequeued at every iteration

//Add the primitive operations of the dequeue() function for this iteration

if(r >= 1)

{

stackQueueCount = stackQueueCount + r \* 6 + 7;

}

r--;

//Add the primitive operations of the pop() function for this iteration

stackQueueCount = stackQueueCount + 8;

//primitive operation for if condition

stackQueueCount++;

//old code in case I need it again: if((""+s.pop()).equals(q.dequeue()+""))

if(s.pop() == q.dequeue())

{

//Character at the front matches character at the back

//Currently, still a palindrome

}

else

{

//Character at the front does not match character at the back

//input is not a palindrome

stackQueueCount++;

return false;

}

}

//Checked every letter

//input is definitely a palindrome

stackQueueCount++;

return true;

}

//Returns a reversed String of the input string

public String reverse(String input)

{

recursiveCount = recursiveCount + 2;

//Base case

if(input.length() == 1)

{

recursiveCount++;

return input;

}

recursiveCount = recursiveCount + 4;

//Reduction step is when the function is called by sending the string without the first letter

//Reduction: reverse(input.substring(1,input.length()))

return reverse(input.substring(1,input.length())) + input.substring(0, 1);

}

public boolean isPalindromeRecursionReverse(String input)

{

recursiveCount++;

//Get the reversed string using recursion

String reverse = reverse(input);

//Examined String.java and StringLatin1.java to count primitive operations of equals

//Counted equals to have 7 + 1 + 5\*input.length() + 1 + 1 primitive operations

recursiveCount = recursiveCount + (10+(5\*input.length()));

//If the reverse string is equal to the input string, then the input is a palindrome

if(reverse.equals(input))

{

recursiveCount++;

//input is a palindrome

return true;

}

else

{

recursiveCount++;

//input is not a palindrome

return false;

}

}

public String decimalToBinary(String decimal)

{

//Convert the decimal String into an int

//Then convert the int to a binary String

//Return the binary String

return Integer.toBinaryString(Integer.parseInt(decimal));

}

}

Testing

Testing that each method correctly identifies palindromes

Method 1: isPalindromeStringEquals(String input)

Input = 0

Expected result: Is a palindrome

Actual result:

Input = 11

Expected result: Is a palindrome

Actual result:

Input = 101

Expected result: Is a palindrome

Actual result:

Input = 12

Expected result: Is not a palindrome

Actual result:

Input = 12366466321

Expected result: Is a palindrome

Actual result:

Method 2: isPalindromeCharacterByCharacter(String input)

Input = 0

Expected result: Is a palindrome

Actual result:

Input = 11

Expected result: Is a palindrome

Actual result:

Input = 101

Expected result: Is a palindrome

Actual result:

Input = 12

Expected result: Is not a palindrome

Actual result:

Input = 12366466321

Expected result: Is a palindrome

Actual result:

Method 3: isPalindromeStackQueue (String input)

Input = 0

Expected result: Is a palindrome

Actual result:

Input = 11

Expected result: Is a palindrome

Actual result:

Input = 101

Expected result: Is a palindrome

Actual result:

Input = 12

Expected result: Is not a palindrome

Actual result:

Input = 12366466321

Expected result: Is a palindrome

Actual result:

Method 4: isPalindromeRecursionReverse (String input)

Input = 0

Expected result: Is a palindrome

Actual result:

Input = 11

Expected result: Is a palindrome

Actual result:

Input = 101

Expected result: Is a palindrome

Actual result:

Input = 12

Expected result: Is not a palindrome

Actual result:

Input = 12366466321

Expected result: Is a palindrome

Actual result:

Utility method:

public String decimalToBinary(String decimal)

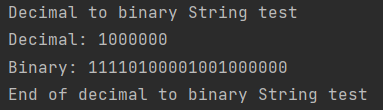
Input = decimal = “1000000”

Expected result: 1 million in binary

Using an online converter, 1 million in binary is:

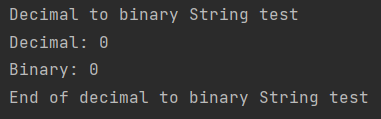
11110100001001000000

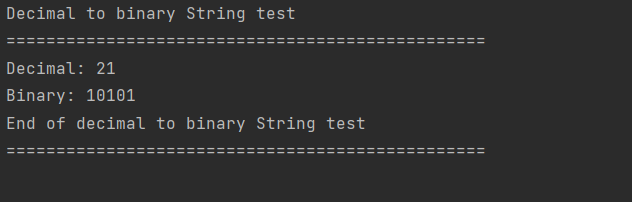
Actual result:   
11110100001001000000



Input = decimal = “0”

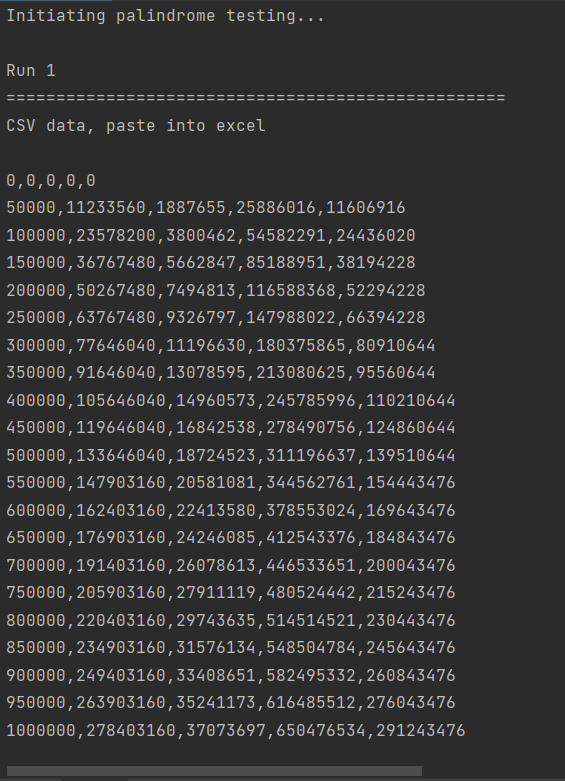
Expected result:   
0

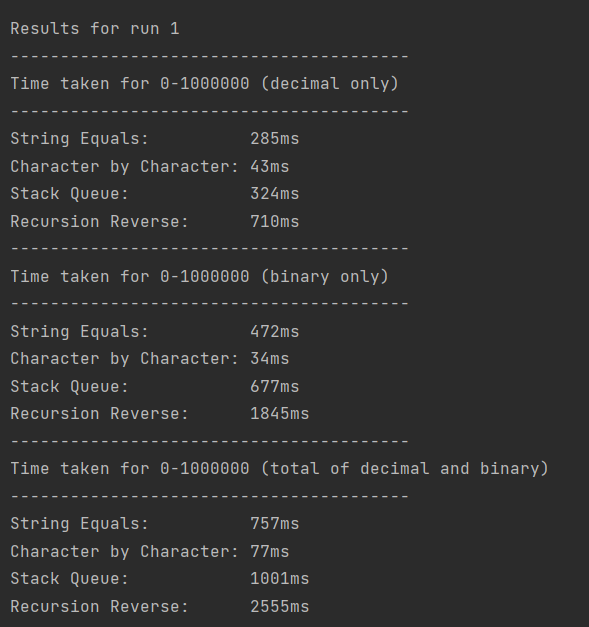
Actual result:   
0

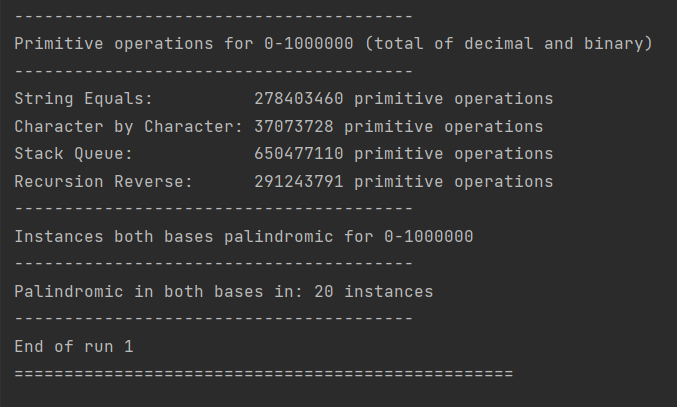


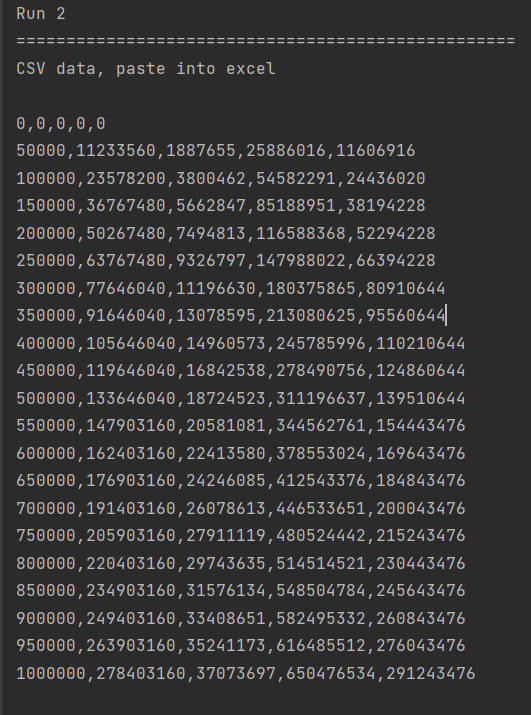
Tests successful

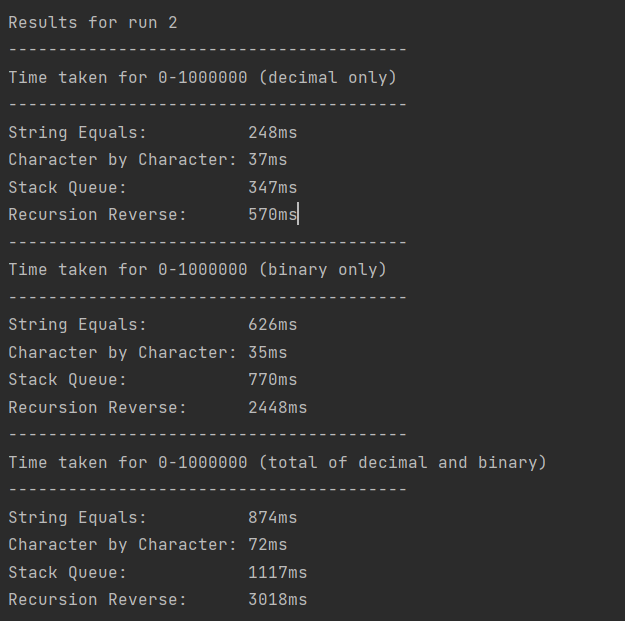
**Test the numbers 0-1000000 for palindromes**

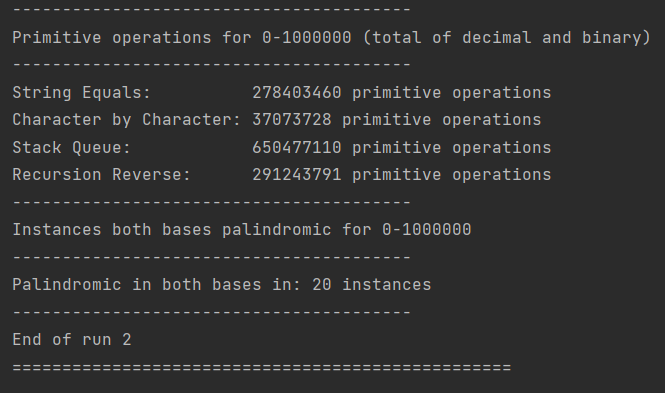


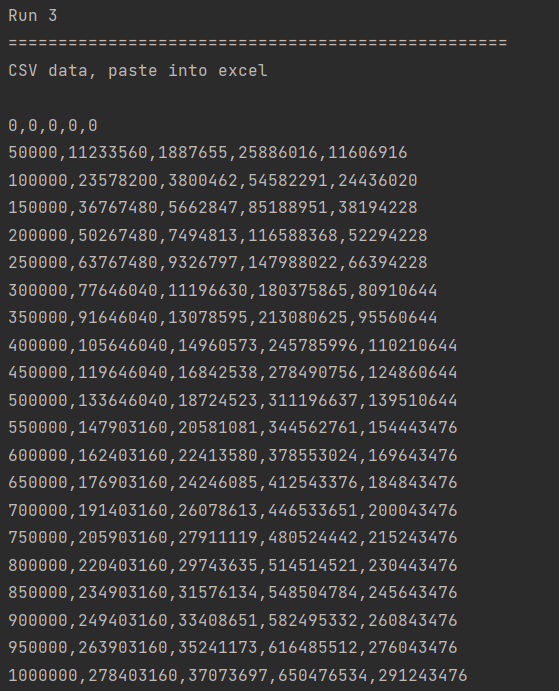


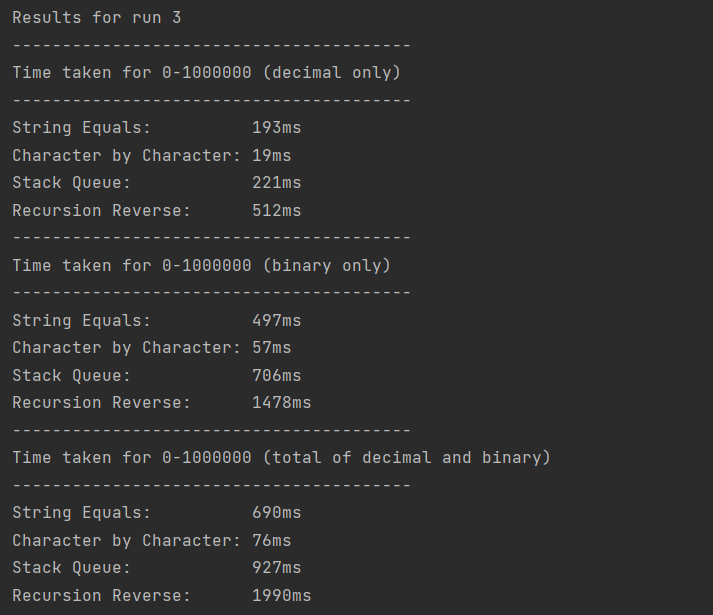


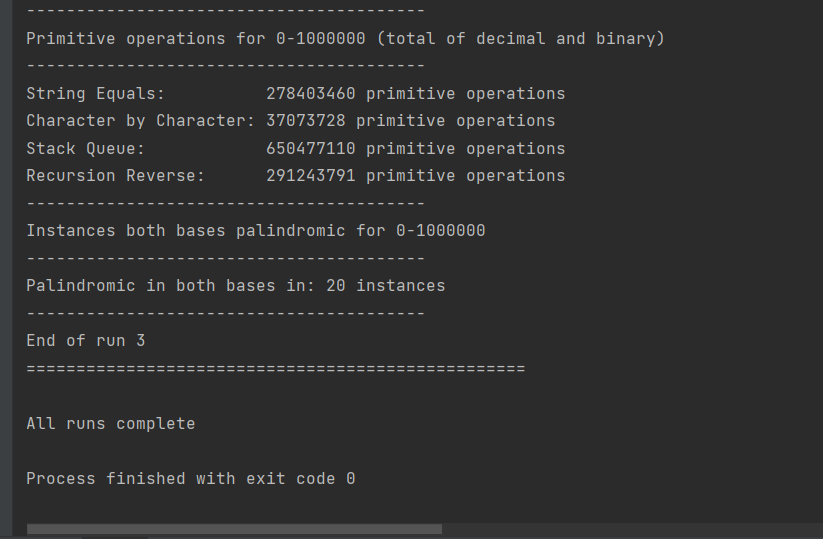




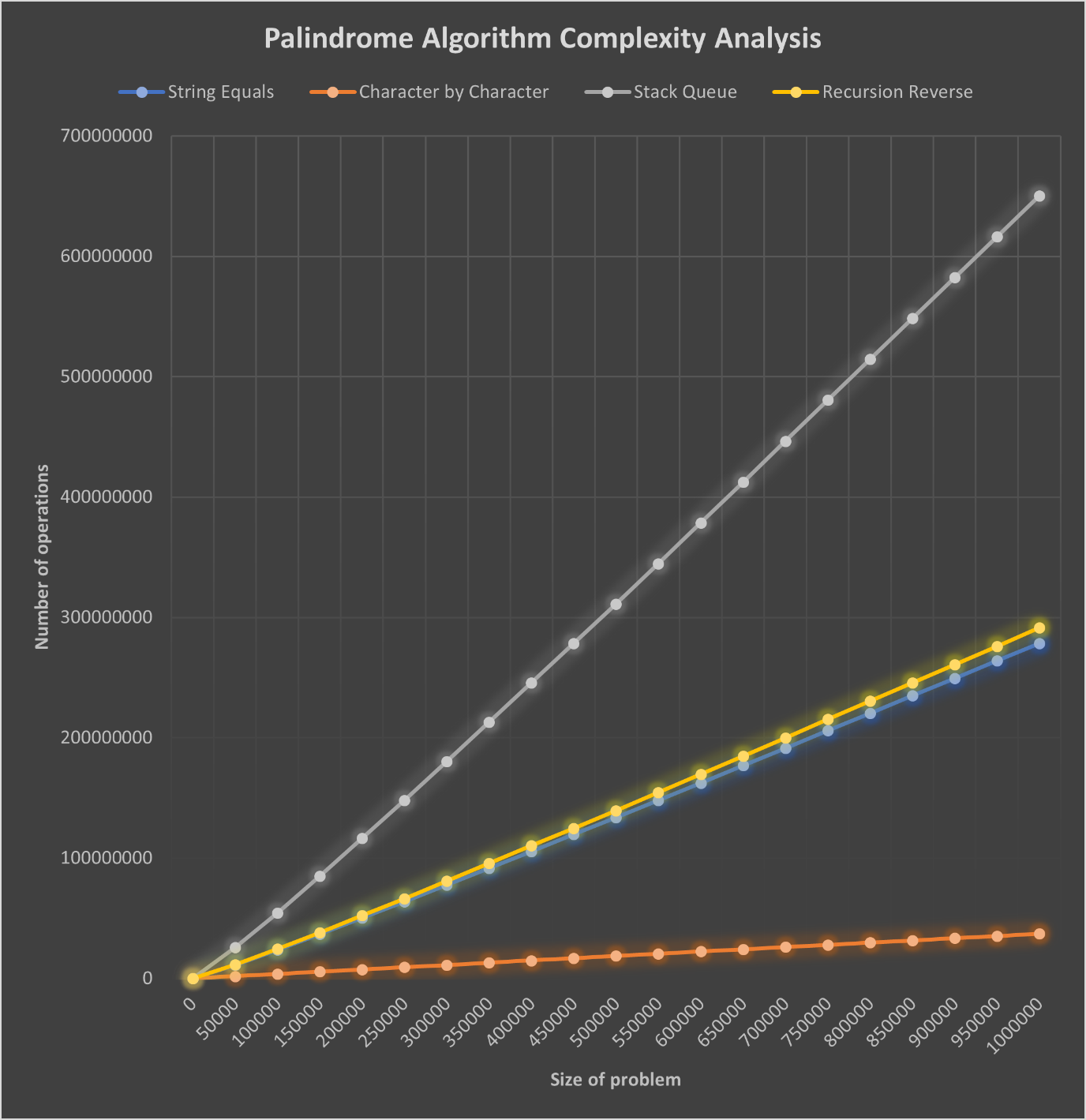








Complexity analysis



Analysis

**Every algorithm** seems to have a linear time complexity because the number of operations seems to increase linearly and proportional to the size of the problem.

The **Stack Queue** method makes the most primitive operations. This explains why it has the second longest time to complete.

The **Recursion Reverse** method has the longest time to complete but does not use the most primitive operations. This could be due to the fact that, in the primitive operation count, I have not included the count of the primitive operations that Java is making to manage functions in its function stack. If these were included too, the recursion reverse method may have been the steepest gradient on the graph. This method was expected to be the slowest because recursion is known for being an elegant to write yet inefficient solution.

The **String Equals** method is the 2nd fastest and this corresponds to the results showing that it is the second fastest.

The graph for the **Character by Characte**r method shows that it is the most efficient of these 4 algorithms. It makes the least amount of primitive operations compared to the other 3 methods. The time taken results show that is also the quickest method, consistently finishing with the lowest time taken.