Paul Heather Palindromes OOP

**Problem Statement :**

The problem is to create 4 methods that check if an input decimal string is a palindrome (same backwards and forwards), then run them and check how long each method takes and how many primitive operations each method carries out. Graph the results on excel of how many operations are carried out every 50,000 numbered checked to visualize how efficient each method is when a problem grows.

**Analysis and Design Notes :**

4 methods need to be created for this:

Method 1:

This method reverses the whole input string and compares it to the original input string. To do this a new empty string needs to be created and take in the reversed version of the input string by looping backwards through the input and adding character by character to the new empty string. Then the new string is compared to the original string using a .equals() method because it is comparing 1 full string to another. If they are the same the method returns true, otherwise the method returns false.

Method 2:

This method compares the input string and reversed string character by character. A new empty string will once again need to be created and take in the reversed version of the input string by looping backwards through the input and adding character by character to the new empty string. Using 2 integers storing 0 and the length of the string minus 1, using a while loop to make sure the numbers don’t go past half way of the string and compare more characters then necessary, if the first character is equal to the last character check the second and second last characters and so on. If all characters are the same return true otherwise return false.

Method 3:

This method uses an arraystack and arrayqueue. The input string is added letter by letter to the arraystack and arrayqueue. The arraystack is then popped and the array queue is dequeued and the characters are compared. This essentially compares the first and last characters of the string input. This keeps happening until either the arraystack and arrayqueue are empty or the characters compared are not equal.

Method 4:

This method uses a recursive function to reverse the string that is input into the method. It then uses .equals() to compare the string input to the reversed string.

Reverse Recursive Method:

This String type method takes in a string. It then creates a substring keeps calling itself in its return statement until the string is null or only has 1 character.

Decimal to Binary utility method:

This will take in a decimal string , create an empty string, put the input string in the empty string. Then parse the new string to an integer and then change that integer to a binary string.

Each method will have an integer getting incremented every time an operation is carried out.

When each method is called in the public static void main I will use the System.currentTimeMillis() function to time how long each function takes.

**Code :**

\*\* ArrayStack, Stack, ArrayQueue and Queue implemented from canvas

import java.math.\*;  
public class Main {  
  
 // ints to count how many operations each method carries out  
 public static int *operationCount1* = 0;  
 public static int *operationCount2* = 0;  
 public static int *operationCount3* = 0;  
 public static int *operationCount4* = 0;  
  
 public Main(){  
  
 }  
  
 // method 1 (chacks if the whole string input is equal to the whole string flipped)  
 public static boolean isPalindromeFullCompare(String string) {  
 // empty string that will store flipped string  
 String newString = "";  
  
 // iterates through the input string backwards and adds each char to the empty string  
 for (int i = string.length(); i > 0; i--) {  
 newString += string.charAt(i - 1);  
 *operationCount1* += 2; // Increment by 2 operations  
 }  
  
 *operationCount1*++; // Increment operation  
  
 if (newString.equals(string)) {  
 *operationCount1*++; // Increment operation  
 return true;  
 } else {  
 *operationCount1*++; // Increment operation  
 return false;  
 }  
 }  
  
 // method 2 (checks if the strings first and last character is equal, if it is it checks second and scond last equal etc  
 public static boolean isPalindromeElmByElm(String string) {  
 int n = 0;  
 int m = string.length() - 1;  
  
 while (n < m) {  
 char firstChar = string.charAt(n);  
 char endChar = string.charAt(m);  
 *operationCount2* += 2; // Increment by 2 for each charAt operation  
  
 if (firstChar != endChar) {  
 return false;  
 }  
  
 n++;  
 m--;  
 *operationCount2* += 2; // Increment for each n++ and m-- operation  
 }  
  
 return true;  
 }  
  
 // method 3 ( uses stack and queue to compare strings )  
 public static boolean isPalindromeStackAndQueue(String string) {  
 // new stack and queue created  
 ArrayStack stringStack = new ArrayStack();  
 ArrayQueue stringQueue = new ArrayQueue();  
 //adds characters of string to queue and stack one by one  
 for (int i = 0; i < string.length(); i++) {  
 stringStack.push(string.charAt(i));  
 stringQueue.enqueue(string.charAt(i));  
 *operationCount3* += 2; // Increment by 2 operations  
 }  
  
 while (!stringStack.isEmpty()) {  
 // compares the character popped by stack(first letter of string) and character dequeued( last letter of string) and then compares the next elements if they are the same  
 if (stringStack.pop() != stringQueue.dequeue()) {  
 return false;  
 }  
 *operationCount3* += 2; // Increment by 2 for each pop and dequeue operation  
 }  
  
 return true;  
 }  
  
 // reverse method for method 4  
 // recursive method to flip string  
 public static String reverse(String str) {  
  
 if (str == null || str.length() <= 1) {  
 return str;  
 }  
 else {  
 // return calls itslef again and adds to str  
 return *reverse*(str.substring(1)) + str.charAt(0);  
 }  
 }  
  
 // method 4 ( uses a recursive method to flip the string and then compares them  
 public static boolean isPalindromeRecursiveReverse(String string) {  
  
 // calls recursive reverse method on input string  
 String reversedString = *reverse*(string);  
 *operationCount4*++; // Increment method  
  
 // checks if string returned from reverse method is the same as input string  
 if (string.equals(reversedString)) {  
 *operationCount4*++; // Increment operation  
 return true;  
 }  
  
 return false;  
 }  
  
 // utility method to change decimal number string to binary string  
 public static String decimalToBinary(String string){  
 // creates empty string to hold input string that will get changed to decimal  
 String newString = "";  
 for(int i = 0; i < string.length(); i++){  
 newString += string.charAt(i);  
 }  
 // stores the newString as a decimal  
 int decString = Integer.*parseInt*(newString);  
 // changes the decimal value to a binary string  
 String binString = Integer.*toBinaryString*(decString);  
  
 // returns the new binary string  
 return binString;  
 }  
  
  
 /////////////////////////////////////////////////////  
  
 public static void main(String[] args) {  
  
 int numPalindromesBoth = 0;  
 int numPalindromesDec = 0;  
 int numPalindromesBin = 0;  
  
  
 // call each method for first 1000000 numbers and check if they are a palindrome as decimal and binary  
 // each function call is timed and the number of operations carried out is   
 System.*out*.println("-----Method 1 stats: -----");  
 double startTime = System.*currentTimeMillis*();  
  
 for(int i = 0; i <= 1000000; i++){  
 String decimalString = String.*valueOf*(i);  
 String binaryString = Integer.*toBinaryString*(i);  
 if(*isPalindromeFullCompare*(decimalString) == true){  
 numPalindromesDec++;  
  
 }  
 if(*isPalindromeFullCompare*(binaryString) == true){  
 numPalindromesBin++;  
 }  
 if(*isPalindromeFullCompare*(decimalString) == true && *isPalindromeFullCompare*(binaryString) == true){  
 numPalindromesBoth++;  
 }  
  
 }  
 double endTime = System.*currentTimeMillis*();  
 System.*out*.println(numPalindromesDec+" palindromes found for decimal");  
 System.*out*.println(numPalindromesBin+" palindromes found for binary");  
 System.*out*.println(numPalindromesBoth+" palindromes found for both decimal and binary");  
 System.*out*.println(*operationCount1*+" operations");  
 double totalTime = (endTime - startTime)/1000;  
 System.*out*.println(totalTime+" seconds taken");  
  
 startTime = 0;  
 endTime = 0;  
 totalTime = 0;  
 startTime = System.*currentTimeMillis*();  
 numPalindromesBoth = 0;  
 numPalindromesDec = 0;  
 numPalindromesBin = 0;  
  
 System.*out*.println("-----Method 2 stats: -----");  
 for(int i = 0; i <= 1000000; i++){  
 String decimalString = String.*valueOf*(i);  
 String binaryString = Integer.*toBinaryString*(i);  
 if(*isPalindromeElmByElm*(decimalString) == true){  
 numPalindromesDec++;  
 }  
 if(*isPalindromeElmByElm*(binaryString) == true){  
 numPalindromesBin++;  
 }  
 if(*isPalindromeElmByElm*(decimalString) == true && *isPalindromeFullCompare*(binaryString) == true){  
 numPalindromesBoth++;  
 }  
  
 }  
  
 endTime = System.*currentTimeMillis*();  
 System.*out*.println(numPalindromesDec+" palindromes found for decimal");  
 System.*out*.println(numPalindromesBin+" palindromes found for binary");  
 System.*out*.println(numPalindromesBoth+" palindromes found for both decimal and binary");  
 System.*out*.println(*operationCount2*+" operations");  
 totalTime = (endTime - startTime)/1000;  
 System.*out*.println(totalTime+" seconds taken");  
  
 startTime = 0;  
 endTime = 0;  
 totalTime = 0;  
 startTime = System.*currentTimeMillis*();  
 numPalindromesBoth = 0;  
 numPalindromesDec = 0;  
 numPalindromesBin = 0;  
  
 System.*out*.println("-----Method 3 stats: -----");  
 for(int i = 0; i <= 1000000; i++){  
 String decimalString = String.*valueOf*(i);  
 String binaryString = Integer.*toBinaryString*(i);  
 if(*isPalindromeStackAndQueue*(decimalString) == true){  
 numPalindromesDec++;  
 }  
 if(*isPalindromeStackAndQueue*(binaryString) == true){  
 numPalindromesBin++;  
 }  
 if(*isPalindromeStackAndQueue*(decimalString) == true && *isPalindromeFullCompare*(binaryString) == true){  
 numPalindromesBoth++;  
 }  
  
 }  
  
 endTime = System.*currentTimeMillis*();  
 System.*out*.println(numPalindromesDec+" palindromes found for decimal");  
 System.*out*.println(numPalindromesBin+" palindromes found for binary");  
 System.*out*.println(numPalindromesBoth+" palindromes found for both decimal and binary");  
 System.*out*.println(*operationCount3*+" operations");  
 totalTime = (endTime - startTime)/1000;  
 System.*out*.println(totalTime+" seconds taken");  
  
 startTime = 0;  
 endTime = 0;  
 totalTime = 0;  
 startTime = System.*currentTimeMillis*();  
 numPalindromesBoth = 0;  
 numPalindromesDec = 0;  
 numPalindromesBin = 0;  
  
 System.*out*.println("-----Method 4 stats: -----");  
 for(int i = 0; i <= 1000000; i++){  
 String decimalString = String.*valueOf*(i);  
 String binaryString = Integer.*toBinaryString*(i);  
 if(*isPalindromeRecursiveReverse*(decimalString) == true){  
 numPalindromesDec++;  
 }  
 if(*isPalindromeRecursiveReverse*(binaryString) == true){  
 numPalindromesBin++;  
 }  
 if(*isPalindromeRecursiveReverse*(decimalString) == true && *isPalindromeFullCompare*(binaryString) == true){  
 numPalindromesBoth++;  
 }  
  
 }  
  
 endTime = System.*currentTimeMillis*();  
 System.*out*.println(numPalindromesDec+" palindromes found for decimal");  
 System.*out*.println(numPalindromesBin+" palindromes found for binary");  
 System.*out*.println(numPalindromesBoth+" palindromes found for both decimal and binary");  
 System.*out*.println(*operationCount4*+" operations");  
 totalTime = (endTime - startTime)/1000;  
 System.*out*.println(totalTime+" seconds taken");  
 }  
}

**Testing :**

I called all methods at the same time, I printed the number of decimal palindromes each method found, the number of binary palindromes each method found, and the decimals that are a palindrome in decimal and binary format. I timed each function by using the System.currentTimeInMillis() function. I called the time function at the start of the method call and stored it in a variable, then called it at the end of the function call and subtracted the start time from that. This gave the time in milliseconds which I divided by 1000 to give seconds. I repeated this for each method. For each method call I also printed out the amount of operations each method takes to complete the check of however many numbers it is asked to check.

**A screenshot of a computer program

Description automatically generated**

This graph displays all for methods on the same graph. The x-axis is the amount of numbers being checked for palindromes. It increases by 50,000 at a time. The y-axis is the number of primitive operations that the method carries out to check the number of palindromes.

To get the amount of operations carried out for numbers increasing by 50,000 I changed the for loops for each 4 of my methods in my code manually each time and input the number of operations into the excel for each method each time. Looking back I should have used a loop that would have incremented the number of numbers being checked by 50000 each iteration of the loop and that would have saved my a lot of time.

From the graph you can see that method 4 is clearly the most efficient method because as the amount of numbers being checked increases, its number of operations increases at the slowest rate. This is the method that uses the recursive reverse method for flipping its input string.

The least efficient method is method 1. This makes sense as this method flips the whole string and compares the two strings. This method cant return false early into the method call so therefore it will carry out the most operations.

Method 2 is the second most efficient method as it checks character by character through each string so if the strings first and last characters for example are not the same the method returns false relatively early compared to other methods.

Method 3 is the second least efficient method as it creates a full arraystack and arrayqueue no matter if the first and last characters are not the same, the string is added to an arraystack and arrayqueue.

**A graph on a white background

Description automatically generated**