Andrew Hayes
21321503
a.hayes18@nuigalway.ie

CT2109 Assignment 3

2024-06-05

Double-Base Palindromes & Complexity Analysis

1 Problem Statement

The overall purpose of this assignment is to perform some basic complexity analysis on four distinct algorithms, each of which aim to achieve the same result: determining whether a String is palindromic. More specifically, each algorithm will be fed the binary & decimal representations of each integer between 0_{10} and $1,000,000_{10}$, in String form. Each algorithm will then determine whether the String is palindromic, returning either `true` or `false`. Each method will have three counter variables to record the number of palindromes found: one for the numbers whose decimal form is palindromic, one for the numbers whose binary form is palindromic, and one for the numbers which are palindromic in both decimal & binary form.

The number of primitive operations will be recorded using a separate global variable for each method. Additionally, the real time taken by each method (in milliseconds) will be recorded. These measurements will form the basis of the complexity analysis performed. The number of primitive operations will be recorded for each method and plotted against the size of the problem (the number of values being checked for palindromicity, n). This graph should reflect the "order" of the problem, i.e. it should match the big-O representation derived from a time-complexity analysis of the algorithm. For example, an exponential curve on the graph would indicate that the algorithm is of $O(n^2)$ complexity, while a straight line through the origin would indicate that the algorithm is of $O(n)$ complexity.

2 Analysis & Design Notes

The first thing that we'll want to do in this program is declare the variables that we'll use to record the data for each method being tested. We will have seven arrays of size four, with one index for each method being tested. These arrays will be:

- `long[] operations` - To count the number of primitive operations for each method.
- `int[] decCount` - To count how many numbers that are palindromic in decimal form are found using each method.
- `int[] binCount` - To count how many numbers that are palindromic in binary form are found using each method.
- `int[] bothCount` - To count how many numbers that are palindromic in both decimal & binary form are found using each method.
- `long[] startTime` - To record the start time (in Unix epoch form) of the testing of each of the four methods.
- `long[] totalTime` - To record the total time (in milliseconds) by the testing of each of the four methods.
- `StringBuilder[] data` - This will be be used to create a String of CSV data for each method, which will be output to a `.csv` file at the end of testing. Doing this saves us from having to open, write to, & close a `.csv` file every time we want to record some data, which would be very inefficient.

The first thing that we'll do in the main method is initialise all the indices in the `StringBuilder[] data` array to have the appropriate headings from each column, one column for the number of primitive operations and one for the size of the problem that used that number of primitive operations, i.e. `"operations,size\n"`.

We'll also want to generate a two-dimensional array of all the Strings that we'll be testing for palindromicity, with one dimension for decimal Strings and one dimension for binary Strings, both going up to $1,000,000_{10}$. Generating this once in the beginning will save us from having to re-generate the same Strings four times, which would be very inefficient.

Each method that we will be testing will have its own class that implements the interface `PalindromeChecker`.

This interface will contain a single method with the signature `public boolean checkPalindrome(String str)`. The reason for doing this will be to follow OOP principles & the DRY principle so that we don't have unnecessary repetition of code. This will allow us to have one generic loop through each of the numbers from 0_{10} to $1,000,000_{10}$ instead of four separate ones. Each of the methods that we will be testing will be overridden implementations of `checkPalindrome`. We will have four classes that implement `PalindromeChecker`:

- `ReverseVSOriginal` - This class will contain "Method One" outlined in the assignment specification, which checks if a String is palindromic by comparing the String to a reversed copy of itself, hence the name.
- `IVersusIMinusN` - This will contain "Method Two" outlined in the assignment specification, which checks if a String is palindromic by looping through each character in the String using an iterator `i` and comparing the character at index `i` to the character at index `n-i`, where `n` is the last index in the String, i.e., comparing the first character to the last, the second character to the second-last, etc.
- `StackVSQueue` - This will contain "Method Three" outlined in the assignment specification, which checks if a String is palindromic using essentially the same technique as "Method Two" but instead of simply iterating through the String, each character of the String will be put onto both a Stack & a Queue, and then items will be removed from the Stack & Queue and compared to each other. The LIFO nature of the Stack and the FIFO nature of the Queue result in this comparison being first character versus last, second character versus second-last, and so on.
- `RecursiveReverse` - This will contain "Method Four" outlined in the assignment specification, which checks if a String is palindromic using essentially the same technique as "Method One" but using recursion to reverse the String instead of iteration.

An array called `PalindromeCheckers[]` will be initialised to contain an instance of each of these four classes. This array will be iterated over (using iterator `j`) to test each method, which prevents code repetition. In said loop, firstly, the `startTime[j]` will be recorded using `System.getCurrentTimeMillis();`. Then, a loop will be entered that iterators over each number between 0_{10} to $1,000,000_{10}$. Both the decimal & binary Strings at index `i` of the `strings[]` array will be passed to `palindromeCheckers[j].checkPalindrome()` to be checked for palindromicity. Then, `decCount[j]`, `binCount[j]`, & `bothCount[j]` will be iterated or will remain unchanged, as appropriate.

The count of primitive operations for each method will be iterated as they are executed. I won't count the `return` statements at the end of methods or accessing a variable or array index as primitive operations, as these (especially the `return` statements) are computationally insignificant, and certainly aren't on the same level as something like creating a new variable in memory. Unfortunately, it's not really possible to say with accuracy what is & isn't a "primitive operation" in a language so high-level & abstract as Java, but I feel that this is a reasonable approximation. We want to record the count of primitive operations at regular intervals of 50,000 during the process, so we will append the operations count and the current `i` to `data[j]` if `i` is divisible by 50,000.

Once the loop using iterator `i` has ended, the total time taken will be recorded by subtracting the current time from the start time. The number of palindromes found in decimal, binary, and both decimal & binary will be printed out to the screen, along with the total time taken & the total number of primitive operations for that method. Finally, the data for that method will be written to a `.csv` file. This will be repeated for all four methods in the `palindromeCheckers` array.

3 Code

```java
import java.io.*;

public class NewPalindrome {
    public static long[] operations = new long[4];  // array to contain the global operations count for each method
    public static int[]  decCount   = new int[4];   // array to hold the count of decimal palindromes found using each method
    public static int[]  binCount   = new int[4];   // array to hold the count of binary palindromes found using each method
    public static int[]  bothCount  = new int[4];   // array to hold the count of numbers that are palindromes in both decimal & binary found using each method
    public static long[] startTime  = new long[4];  // array to hold the start time of each method's test loop
    public static long[] totalTime  = new long[4];  // array to hold the total time of each method's test loop

```

```java
    // array to hold all the String versions of the numbers so that they don't have to be generated for each method
    // 0th column will be decimal, 1st column will be binary
    public static String[][] strings = new String[1_000_001][2];

    // array of StringBuilder objects used to hold the csv data (size of problem, number of operations) for each method
    public static StringBuilder[] data = new StringBuilder[4];

    // array of the four classes that will be tested
    public static PalindromeChecker[] palindromeCheckers = {new ReverseVSOriginal(), new IVersusNMinusI(), new StackVSQueue(), new RecursiveReverse()};

    public static void main(String args[]) {
        // initialising the data array to StringBuilder objects
        for (int i = 0; i < 4; i++) {
            data[i] = new StringBuilder("operations,size\n");
        }

        // filling up the strings array
        for (int i = 0; i <= 1_000_000; i++) {
            strings[i][0] = Integer.toString(i, 10);        // converting i to a String base 10
            strings[i][1] = binary2string(strings[i][0]);   // converting the decimal String to a binary String

        }
        // looping through each PalindromeChecker object in the palindromeCheckers array
        for (int j = 0; j < 4; j++) {
            // getting start time
            startTime[j] = System.currentTimeMillis();  operations[j]++;

            // looping through the numbers 0 to 1,000,000 and checking if their binary & decimal representations are palindromic
            operations[j]++;
            for (int i = 0; i <= 1_000_000; i++) {
                // incrementing the operations count by 2, 1 for the loop condition check and 1 for incrementing i
                operations[j] += 2;

                // converting the number to a decimal or binary String and checking if is a palindrome
                boolean isDecPalindrome = palindromeCheckers[j].checkPalindrome(strings[i][0]); operations[j]++;
                boolean isBinPalindrome = palindromeCheckers[j].checkPalindrome(strings[i][1]); operations[j]++;

                // incrementing the appropriate counter if the number is a palindrome in that base
                decCount[j]   = isDecPalindrome ? decCount[j] + 1 : decCount[j];   operations[j] += 1 + 1;       // incremnting by 2, 1 for assignment, 1 for condition check
                binCount[j]   = isBinPalindrome ? binCount[j] + 1 : binCount[j];   operations[j] += 1 + 1;
                bothCount[j]  = isDecPalindrome && isBinPalindrome ? bothCount[j] + 1 : bothCount[j]; operations[j] += 1 + 1 +1;    // 2 condition checks and one assignment, so incrementing by 3

                // appending to the data StringBuilder at intervals of 50,000
                if (i % 50_000 == 0) {
                    data[j].append(operations[j] + "," + i + "\n");
                }
            }

            // calculating total time taken for method 1 and printing out the results
            totalTime[j] = System.currentTimeMillis() - startTime[j];  operations[j] += 1 + 1;   // incrementing by 2, 1 for getting current time and subtracting start time, 1 for assignment

            System.out.println("Number of decimal palindromes found using Method " + j + ": " + decCount[j]);
            System.out.println("Number of binary  palindromes found using Method " + j + ": " + binCount[j]);
            System.out.println("Number of palindromes in both decimal & binary found using Method " + j + ": " + bothCount[j]);
            System.out.println("Number of primitive operations taken in Method " + j + ": " + operations[j]);
            System.out.println("Time taken for Method " + j + ": " + totalTime[j] + " milliseconds");
            System.out.println();

            // outputting the data to separate csv files
            try {
                String filename = "method" + j + ".csv";
                File csv = new File(filename);

                // creating file if it doesn't already exist
                csv.createNewFile();

                FileWriter writer = new FileWriter(filename);
                writer.write(data[j].toString());
                writer.close();

            } catch (IOException e) {
                System.out.println("IO Error occurred");
                e.printStackTrace();
```

```java
                System.exit(1);
            }
        }
    }

    // utility method to convert a decimal String to its equivalent binary String
    public static String binary2string(String decimalStr) {
        return Integer.toString(Integer.parseInt(decimalStr), 2);  // parsing the String to an int and then parsing that int to a binary String
    }
}
```

NewPalindrome.java

```java
public interface PalindromeChecker {
    public boolean checkPalindrome(String str);
}
```

PalindromeChecker.java

```java
// class to implement Method 3
public class ReverseVSOriginal implements PalindromeChecker {
    // method 1 - reversed order String vs original String
    @Override
    public boolean checkPalindrome(String str) {
        String reversedStr = "";    NewPalindrome.operations[0]++;

        // looping through each character in the String, backwards
        // incrementing operations counter by 2, 1 for initialisating i, 1 for getting str.length()
        NewPalindrome.operations[0] += 1 + 1;
        for (int i = str.length(); i > 0; i--) {
            NewPalindrome.operations[0] += 1 + 1;           // for loop condition check & incrementing i

            reversedStr += str.charAt(i-1); NewPalindrome.operations[0] += 1 + 1;
        }

        // returning true if the Strings are equal, false if not
        NewPalindrome.operations[0] += str.length();    // the equals method must loop through each character of the String to check that they are equal so it is O(n)
        return str.equals(reversedStr);
    }
}
```

ReverseVSOriginal.java

```java
// class to implement Method 2
public class IVersusNMinusI implements PalindromeChecker {
    // method 2 - comparing each element at index i to the element at n - i where n is the last index
    @Override
    public boolean checkPalindrome(String str) {
        // looping through the first half of the String
        NewPalindrome.operations[1]++;
        for (int i = 0; i < Math.floor(str.length() / 2); i++) {
            NewPalindrome.operations[1] += 1 + 1 + 1 + 1;   // 1 for the getting str.length(), 1 for Math,floor, 1 for checking condition, 1 for incrementing

            // returning false if the digits don't match
            NewPalindrome.operations[1] += 1 + 1 + 1 + 1;   // 1 for str.charAt(i), 1 for ((str.lenght() -1) - 1), 1 for the other str.charAt(), 1 for checking the condition
            if (str.charAt(i) != str.charAt((str.length()-1) - i)) {
                return false;
            }
        }

        // returning true as default
        return true;
    }
}
```

IVersusIMinusN.java

```java
// class to implement method 3
public class StackVSQueue implements PalindromeChecker {
    // method 3 - using a stack and a queue to do, essentially, what method 2 does (compare the first index to the last index, etc.)
    @Override
    public boolean checkPalindrome(String str) {
        ArrayStack stack = new ArrayStack();    NewPalindrome.operations[2] += 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1;
        ArrayQueue queue = new ArrayQueue();    NewPalindrome.operations[2] += 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1;

        // looping through each character in the String and adding the character to the stack & queue
        NewPalindrome.operations[2]++;
        for (int i = 0; i < str.length(); i++) {
            NewPalindrome.operations[2] += 1 + 1 + 1;
```

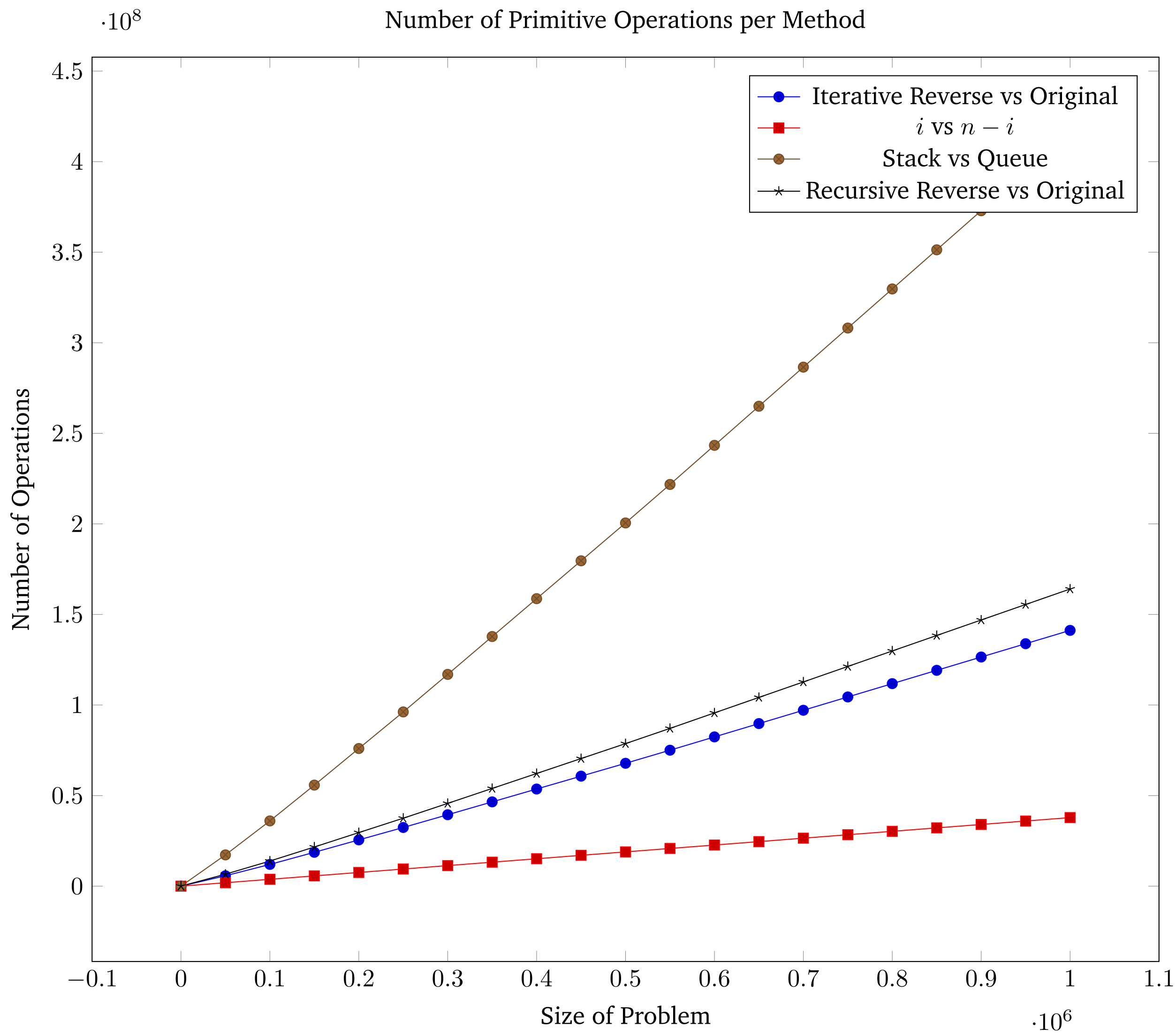
```java

            stack.push(str.charAt(i));          NewPalindrome.operations[2] += 1 + 1 + 1 + 1;
            queue.enqueue(str.charAt(i));       NewPalindrome.operations[2] += 1 + 1 + 1 + 1;
        }

        // looping through each character on the stack & queue and comparing them, returning false if they're different
        NewPalindrome.operations[2]++;
        for (int i = 0; i < str.length(); i++) {
            NewPalindrome.operations[2] += 1 + 1 + 1;

            NewPalindrome.operations[2] += 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1;
            if (!stack.pop().equals(queue.front())) {
                return false;
            }

            // the complexity of ArrayQueue.dequeue() is 3n+2, where n is the number of items in the queue when dequeue() is called.
            // we need to determine the number of items in the queue so that we can determine the number of primitive operations performed when queue.dequeue() is called.
            // to do this, we'll loop through the queue, dequeuing each object and enqueueing it in another ArrayQueue. once complete, we'll reassign the variable queue to point to the new ArrayQueue containing all the objects
            ArrayQueue newQueue = new ArrayQueue();     // not counting the operations for this as it's not part of the algorithm, it's part of the operations counting
            int n = 0;                                  // n is the number of items in the ArrayQueue when dequeue() is called

            while (!queue.isEmpty()) {
                newQueue.enqueue(queue.dequeue());
                n++;
            }

            queue = newQueue;                           // setting queue to point to the newQueue, which is just the state that queue would have been in if we didn't do this to calculate the primitive operations
            newQueue = null;                            // don't need the newQueue object reference anymore

            NewPalindrome.operations[2] += 3*n + 2;                     // complexity of dequeue is 3n+2
            queue.dequeue();
        }

        return true;
    }
}
```

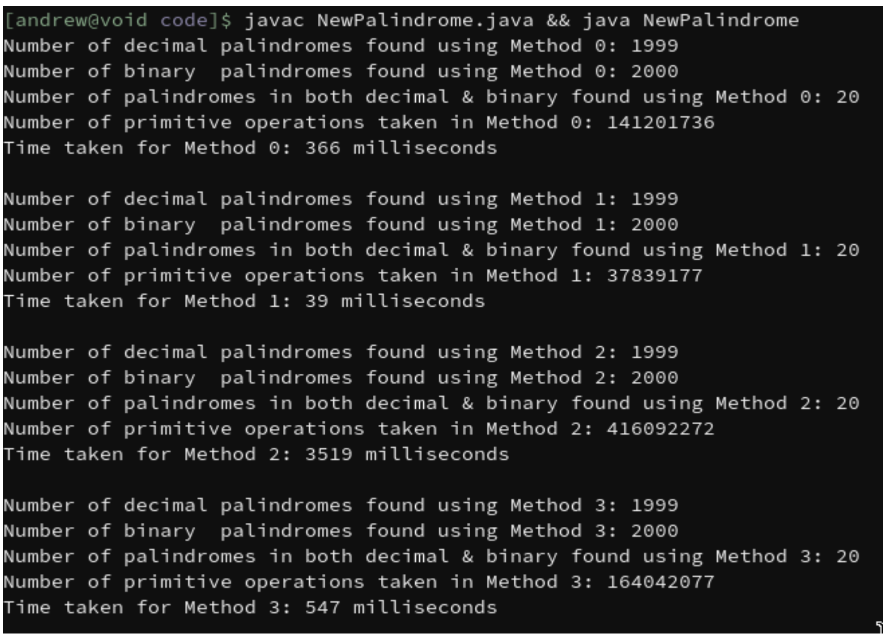
StackVSQueue.java

```java
// class to implement method 4
public class RecursiveReverse implements PalindromeChecker {
    // comparing the String reversed using recursion to the original String (essentially method 1 but with recursion)
    @Override
    public boolean checkPalindrome(String str) {
        // returning true if the original String is equal to the reversed String, false if not
        NewPalindrome.operations[3]++;
        return str.equals(reverse(str));
    }

    // method to reverse the characters in a String using recursion
    public static String reverse(String str) {
        // base case - returning an empty String if there is no character left in the String
        NewPalindrome.operations[3]++;
        if (str.length() == 0) {
            return "";
        }
        else {
            char firstChar = str.charAt(0);                 NewPalindrome.operations[3] += 1 + 1;
            String remainder = str.substring(1);            NewPalindrome.operations[3] += 1 + 1;  // selecting the rest of the String, excluding the 0th character

            // recursing with what's left of the String
            String reversedRemainder = reverse(remainder);  NewPalindrome.operations[3]++;

            // returning the reversed rest of String with the first character of the String appended
            return reversedRemainder + firstChar;
        }
    }
}
```

RecursiveReverse.java

4 Testing





```
[andrew@void code]$ javac NewPalindrome.java && java NewPalindrome
Number of decimal palindromes found using Method 0: 1999
Number of binary  palindromes found using Method 0: 2000
Number of palindromes in both decimal & binary found using Method 0: 20
Number of primitive operations taken in Method 0: 141201736
Time taken for Method 0: 366 milliseconds

Number of decimal palindromes found using Method 1: 1999
Number of binary  palindromes found using Method 1: 2000
Number of palindromes in both decimal & binary found using Method 1: 20
Number of primitive operations taken in Method 1: 37839177
Time taken for Method 1: 39 milliseconds

Number of decimal palindromes found using Method 2: 1999
Number of binary  palindromes found using Method 2: 2000
Number of palindromes in both decimal & binary found using Method 2: 20
Number of primitive operations taken in Method 2: 416092272
Time taken for Method 2: 3519 milliseconds

Number of decimal palindromes found using Method 3: 1999
Number of binary  palindromes found using Method 3: 2000
Number of palindromes in both decimal & binary found using Method 3: 20
Number of primitive operations taken in Method 3: 164042077
Time taken for Method 3: 547 milliseconds
```

Figure 1: Output of the Main Method

The output from the program shows that all the methods agreed on the number of palindromes found in each category, which shows us that they did indeed work as intended.

We can see from the graph that i versus $n - i$ or `IVersusNMinusI` method was the most efficient, as it used the fewest primitive operations out of the four methods (37,839,177) and the complexity grew relatively slowly as the size of the problem increased, demonstrated by the fact that its curve has quite a gentle slope. This is reflected in the testing times as it was by far the fastest method (referred to in the screenshot as "Method 1", as indexing was done from 0), taking only 39 milliseconds to complete. This makes sense, as the method consisted of just one loop and some basic operations, without using any fancy data structures. Furthermore, this method quickly detects non-palindromic Strings, and returns `false`, saving computation.

The next most efficient method was the iterative reverse versus original or `ReverseVSOriginal` method. Despite being the next most efficient method, it took almost ten times the time that `IVersusNMinusI` took to complete, taking 366 milliseconds, which, all things considered, is still quite fast. The number of primitive operations and the rate of growth of this method were also accordingly higher than the previous method.

The third most efficient method was the recursive reverse versus original or `RecursiveReverse` method. This makes sense, as it uses a very similar approach to the second most efficient method, but instead did it recursively. Despite this solution appearing somewhat more elegant from a lines of code perspective, it was in practice less efficient than its iterative counterpart, both in terms of memory (as the recursive function calls had to remain in memory until all the sub-calls were complete) and in terms of operations, taking approximately 20 million more primitive operations to complete the same task as the iterative approach. It was also significantly slower than its iterative counterpart, taking around 200 milliseconds more to complete the task. We can also tell from looking at the graph that at low problem sizes that this approach is comparable & rather similar in terms of efficiency to the iterative approach, but they quickly diverge, with the recursive approach having a significantly steeper slope. We can extrapolate from this that this method would be even less efficient at very large problem sizes, as its rate of growth is quite large.

By far the least efficient method was the Stack versus Queue or `StackVSQueue` method. It took by far the greatest number of primitive operations, and the rate of growth was ridiculously large, rapidly diverging from the other four techniques. Its rate of growth is so large that it would likely quickly become unusable for any significantly large problem. This is reinforced by the fact that it took 3,519 milliseconds to complete the task, being the only method that took more than one second to do so, and taking almost 100 times what the best-performing method took.