Rational Number Tree

Group 8

Anuj Shukla
Mesam Jafri
Aditya Ravi
Nathaniel Kebere

Code Specifications

Inputs:

- Line 1: N number of tests to follow
- Following N lines: either a rational number in the form of p/q or path representation L-R

Outputs:

For each line input, output:

- L-R string if the input is a rational number
- p/q rational number if the input line is a L-R string

Constraints:

$1 \leq p , q \leq 5{,}000{,}000{,}000{,}000{,}000$

$1 \leq$ length of L-R string ≤ 2000

Example
Input

```
5
3/5
2/5
8/5
LRL
RLRL
```

Output

```
LRL
LLR
RLRL
3/5
8/5
```

The Code

```java
/**
* SOFE 2715: Data Structures and Algorithms
* Final Group Project: Rational Number Tree (Binary Tree)
* Authors: Anuj Shukla, Syed Mesam Jafri, Nathaniel Kebere, Aditya Ravi
* Date: March 12, 2024
*/

import java.math.BigInteger;
import java.util.Scanner;

public class RationalNumberTree {

   public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        // Keep prompting the user until a valid integer is entered
        int N;
        while (true) {
            System.out.println("Enter the number of tests:");
            if (scanner.hasNextInt()) {
                N = scanner.nextInt();
                break;
            } else {
                System.out.println("Invalid input. Please enter an integer.");
                // Consume the invalid input to avoid an infinite loop
                scanner.nextLine();
            }
        }

        // Consume the newline character
        scanner.nextLine();

        long totalExecutionTime = 0; // Initialize total execution time

        // Timing measurement encapsulated around the loop
        for (int i = 0; i < N; i++) {
```

```java
System.out.println("Enter test #" + (i + 1) + " input:");
        String input = scanner.nextLine();
        long loopStartTime = System.nanoTime(); // Start measuring loop execution time
        if (input.contains("/")) {
            // Take in the rational number with a division operator, then convert to tree path
            String[] numbers = input.split("/");
            BigInteger p = new BigInteger(numbers[0]);
            BigInteger q = new BigInteger(numbers[1]);
            System.out.println(rationalToTreePath(p, q));
        } else {
            // Convert the path to a rational number and print the result
            System.out.println(treePathToRational(input));
        }
        long loopEndTime = System.nanoTime(); // Stop measuring loop execution time
        long loopExecutionTime = loopEndTime - loopStartTime; // Calculate loop execution time
        totalExecutionTime += loopExecutionTime; // Add current loop execution time to total
    }

    double seconds = (double) totalExecutionTime / 1_000_000_000.0; // Convert total execution time to seconds

    System.out.println("Total execution time: " + seconds + " seconds");

    scanner.close();
}

// Function to convert a rational number to a tree path
public static String rationalToTreePath(BigInteger p, BigInteger q) {
    // StringBuilder to construct the treePath
    StringBuilder treePath = new StringBuilder();
    BigInteger leftNumer = BigInteger.ZERO;
    BigInteger leftDenom = BigInteger.ONE;
    BigInteger rightNumer = BigInteger.ONE;
    BigInteger rightDenom = BigInteger.ZERO;

    // Continue until the target rational number is reached
    while (!p.equals(leftNumer.add(rightNumer)) || !q.equals(leftDenom.add(rightDenom))) {
        BigInteger mediantNumer = leftNumer.add(rightNumer);
        BigInteger mediantDenom = leftDenom.add(rightDenom);

        // Compare the target with the mediant of left and right
        if (p.multiply(mediantDenom).compareTo(q.multiply(mediantNumer)) > 0) {
```

```java
  treePath.append('R');
                leftNumer = mediantNumer;
                leftDenom = mediantDenom;
            } else {
                treePath.append('L');
                rightNumer = mediantNumer;
                rightDenom = mediantDenom;
            }
        }

        //Output the tree path as a string
        return treePath.toString();
    }

    // Function to convert a path in the tree to a rational number
    public static String treePathToRational(String treePath) {
        BigInteger leftNumer = BigInteger.ZERO;
        BigInteger leftDenom = BigInteger.ONE;
        BigInteger rightNumer = BigInteger.ONE;
        BigInteger rightDenom = BigInteger.ZERO;

        // Process each character in the path
        for (char c : treePath.toCharArray()) {
            if (c == 'L') {
                rightNumer = rightNumer.add(leftNumer);
                rightDenom = rightDenom.add(leftDenom);
            } else if (c == 'R') {
                leftNumer = leftNumer.add(rightNumer);
                leftDenom = leftDenom.add(rightDenom);
            } else {
                System.out.println("Invalid input. Please enter only 'L' or 'R'. Or a rational number in the form of p/q. Try again.");
                return "";
            }
        }

        // Return the rational number result in string format
        return leftNumer.add(rightNumer) +"/" + leftDenom.add(rightDenom);
    }
}
```

```
Enter the number of tests:
2/5
Invalid input. Please enter an integer.
Enter the number of tests:
5
Enter test #1 input:
2/5
LLR
Enter test #2 input:
1/2
L
Enter test #3 input:
3/7
LLRR
Enter test #4 input:
RLR
5/3
Enter test #5 input:
LLR
2/5
Total execution time: 0.008183 seconds
```

Time Complexity Analysis - Theoretically

1. Reading Input:

- Reading input typically has a time complexity of O(1) per character read.
- However, if you consider the average length of input strings, denoted as M, the time complexity for reading all inputs would be O(N * M), where N is the number of tests.

2. Processing Input:

- Processing input involves checking whether the input is a rational number or a tree path.
- This operation has a time complexity of O(1) as it involves basic string operations and checks.

3. Calling Conversion Functions:

- The time complexity of rationalToTreePath() and treePathToRational() mainly depends on the size of the input rational numbers.
- Let's denote the average number of bits required to represent the input rational numbers as K.
- Both conversion functions involve a loop that iterates until reaching the target rational number, which takes O(K) iterations in the average case.
- Therefore, the average time complexity for calling these conversion functions would be O(K).

4. Printing Output:

- Printing output has a time complexity of O(1) per character printed.
- Considering the average length of output strings (which can be approximated as similar to the length of input strings), the time complexity for printing all outputs would be O(N * M).

Combining these factors, the overall average case time complexity of the program can be approximated as:

O(N * M) + O(K)

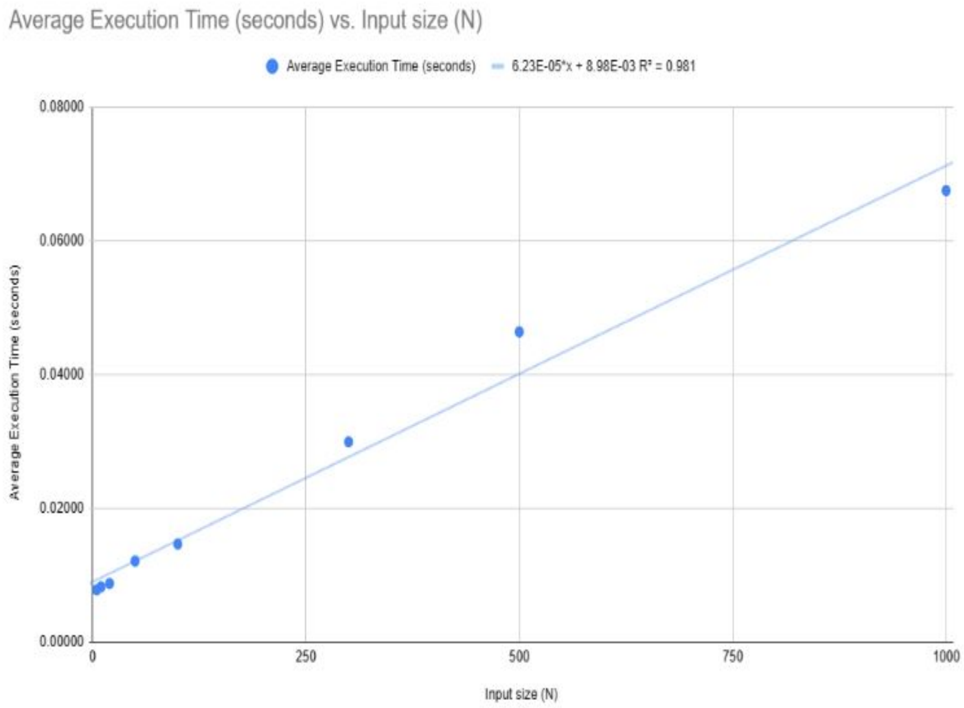
Where:
‘N’ is the number of tests, ‘M’ is the average length of the input/output strings, ‘K’ is the average number of bits required to represent the rational numbers

Performance Analysis with Varying Input Sizes

(Yes, we really ran a test that had 1000 inputs in it)

```
Enter test #1000 input:
LRLR
55/21
Total execution time: 0.0758784 seconds
```

Input size (N)	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Trial 10	Average Execution Time (seconds)
5	0.00740	0.00560	0.00690	0.00840	0.00932	0.01106	0.00749	0.00697	0.00780	0.00715	0.00781
10	0.00786	0.00831	0.00960	0.00671	0.00632	0.00848	0.00801	0.00889	0.00951	0.00855	0.00822
20	0.00717	0.00648	0.01051	0.01021	0.00838	0.01129	0.00967	0.00608	0.00709	0.01072	0.00876
50	0.01746	0.01204	0.01358	0.01016	0.01245	0.00747	0.01092	0.01209	0.01521	0.00972	0.01211
100	0.01722	0.01424	0.01184	0.01783	0.01390	0.01216	0.01572	0.01796	0.01210	0.01373	0.01467
300	0.02623	0.02727	0.02784	0.02896	0.03048	0.03234	0.02990	0.03645	0.03109	0.02876	0.02993
500	0.04996	0.03837	0.04937	0.04848	0.04598	0.04218	0.04962	0.04578	0.04888	0.04558	0.04642
1000	0.07588	0.07101	0.07023	0.07295	0.06912	0.05434	0.06459	0.06215	0.06515	0.06977	0.06752



Time Complexity Analysis - The Real Story

- Turns out, it becomes linear at higher input volumes
- The machine is efficiently able to handle the operations in 10^-4 seconds of magnitude
- M and K become irrelevant as N starts to become far more relevant to the runtime
- It all boils down to input size N => O(N) => Linear time complexity!
- As shown in the graph with the linear trend line and high correlation value

Project Takeaway and Findings

A theoretical time complexity and its practical application may differ in the form they appear in, but are fundamentally the same when you look deeper

Thank you! Questions?