**Faculty of Computing**

**SE-314: Software Construction**

**Class: BESE 13AB**

# Lab 12: Recursion-II

**CLO-03:** Design and develop solutions based on Software Construction principles.  
**CLO-04:** Use modern tools such as Eclipse, NetBeans etc. for software construction.

**Date: 09th Dec 2024**

**Time: 10:00 AM** **- 12:50 PM   
 02:30 PM – 04:50 PM**

***NAME : Hasnain Ali***

***CSM ID : 408546***

**Instructor: Dr. Mehvish Rashid  
Lab Engineer: Mr. Aftab Farooq**

**Introduction:**

# Lab 12: Recursion-II

Students will have hands-on experience on designing, testing, and implementing recursive problems. Given a scenario, you will write the specifications and implement it by dividing into base case and recursive step. You may design helper methods to simplify your implementations. Write unit tests that check for compliance with the specifications.

## Lab Tasks

**Task 1: Recursive Binary Search on a Sorted Array**

**Objective:** Students will implement a recursive version of the binary search algorithm to practice recursion in a practical application and understand its benefits for searching in sorted arrays.

**Instructions:**

1. Create a Java program that takes a sorted array of integers and a target value to search for.
2. Implement a recursive method *binarySearchRecursive* that returns the index of the target value if found, or -1 if the target is not in the array.
3. The method should divide the search range into halves and recursively search in the appropriate half of the array.
4. Ensure the base case terminates the recursion when the search range is empty.
5. Include error handling for cases where the array is null or empty.

Analyze the time complexity and compare it to iterative binary search.

Important: Do not forget to write the specifications and unit tests for the code.

**Code :**

package lab12;

import java.util.ArrayList;

import java.util.Arrays;

import java.util.List;

import java.util.Objects;

/\*\*

\* A utility class implementing recursive binary search algorithms

\* for both integer and string arrays.

\*/

public class RecursiveBinarySearch {

/\*\*

\* Recursive binary search for integer arrays.

\*

\* @param arr Sorted input array of integers

\* @param target Value to search for

\* @return Index of the target value, or -1 if not found

\* @throws IllegalArgumentException if input array is null

\*/

public static int binarySearchRecursive(int[] arr, int target) {

// Error handling for null or empty array

if (arr == null) {

throw new IllegalArgumentException("Input array cannot be null");

}

return binarySearchRecursive(arr, target, 0, arr.length - 1);

}

/\*\*

\* Recursive binary search for integer arrays with specified search range.

\*

\* @param arr Sorted input array of integers

\* @param target Value to search for

\* @param left Left boundary of search range

\* @param right Right boundary of search range

\* @return Index of the target value, or -1 if not found

\*/

private static int binarySearchRecursive(int[] arr, int target, int left, int right) {

// Base case: search range is empty

if (left > right) {

return -1;

}

// Calculate middle index

int mid = left + (right - left) / 2;

// Compare middle element with target

if (arr[mid] == target) {

return mid;

}

// Recursively search left or right half

if (target < arr[mid]) {

return binarySearchRecursive(arr, target, left, mid - 1);

} else {

return binarySearchRecursive(arr, target, mid + 1, right);

}

}

/\*\*

\* Recursive binary search for string arrays.

\*

\* @param arr Sorted input array of strings

\* @param target Value to search for

\* @return Index of the target value, or -1 if not found

\* @throws IllegalArgumentException if input array is null

\*/

public static int binarySearchRecursive(String[] arr, String target) {

// Error handling for null or empty array

if (arr == null) {

throw new IllegalArgumentException("Input array cannot be null");

}

return binarySearchRecursive(arr, target, 0, arr.length - 1);

}

/\*\*

\* Recursive binary search for string arrays with specified search range.

\*

\* @param arr Sorted input array of strings

\* @param target Value to search for

\* @param left Left boundary of search range

\* @param right Right boundary of search range

\* @return Index of the target value, or -1 if not found

\*/

private static int binarySearchRecursive(String[] arr, String target, int left, int right) {

// Base case: search range is empty

if (left > right) {

return -1;

}

// Calculate middle index

int mid = left + (right - left) / 2;

// Compare middle element with target

int compareResult = arr[mid].compareTo(target);

if (compareResult == 0) {

return mid;

}

// Recursively search left or right half

if (target.compareTo(arr[mid]) < 0) {

return binarySearchRecursive(arr, target, left, mid - 1);

} else {

return binarySearchRecursive(arr, target, mid + 1, right);

}

}

/\*\*

\* Recursive method to find all indices of a target value in an integer array.

\*

\* @param arr Sorted input array of integers

\* @param target Value to search for

\* @return List of all indices where target is found

\* @throws IllegalArgumentException if input array is null

\*/

public static List<Integer> binarySearchMultipleIndices(int[] arr, int target) {

if (arr == null) {

throw new IllegalArgumentException("Input array cannot be null");

}

List<Integer> indices = new ArrayList<>();

findAllIndices(arr, target, 0, arr.length - 1, indices);

return indices;

}

/\*\*

\* Helper method to recursively find all indices of a target value.

\*

\* @param arr Sorted input array of integers

\* @param target Value to search for

\* @param left Left boundary of search range

\* @param right Right boundary of search range

\* @param indices List to store found indices

\*/

private static void findAllIndices(int[] arr, int target, int left, int right, List<Integer> indices) {

// Base case: search range is empty

if (left > right) {

return;

}

// Calculate middle index

int mid = left + (right - left) / 2;

// If target found, add to indices and search surrounding areas

if (arr[mid] == target) {

indices.add(mid);

// Search left side for more occurrences

findAllIndices(arr, target, left, mid - 1, indices);

// Search right side for more occurrences

findAllIndices(arr, target, mid + 1, right, indices);

} else if (target < arr[mid]) {

// Recursively search left half

findAllIndices(arr, target, left, mid - 1, indices);

} else {

// Recursively search right half

findAllIndices(arr, target, mid + 1, right, indices);

}

}

/\*\*

\* Time complexity analysis:

\* - Recursive Binary Search: O(log n)

\* - Divides the search space in half with each recursive call

\* - Maximum number of recursive calls is log(n)

\*

\* Comparison with Iterative Binary Search:

\* - Time complexity is identical: O(log n)

\* - Recursive approach uses more memory due to call stack overhead

\* - Iterative approach is generally more space-efficient

\*/

// Unit tests demonstrating usage

public static void main(String[] args) {

// Integer array tests

int[] intArray = {1, 2, 3, 4, 5, 5, 5, 6, 7, 8, 9};

System.out.println("Array" + Arrays.toString(intArray));

System.out.println("Integer Search(give one occurence for 5): " + binarySearchRecursive(intArray, 5)); // Should return an index of 5

System.out.println("Multiple Indices: " + binarySearchMultipleIndices(intArray, 5)); // Should return [5, 4, 6]

// String array tests

String[] stringArray = {"apple", "banana", "cherry", "date", "elderberry"};

System.out.println("Array" + Arrays.toString(stringArray));

System.out.println("String Search(displays index for cherry): " + binarySearchRecursive(stringArray, "cherry")); // Should return 2

}

}

**Test:**

package lab12;

import lab12.RecursiveBinarySearch;

import org.junit.Test;

import static org.junit.Assert.\*;

import java.util.List;

public class RecursiveBinarySearchTest {

*@Test*

public void testBinarySearchRecursiveIntegerFound() {

int[] arr = {1, 2, 3, 4, 5, 5, 5, 6, 7, 8, 9};

*assertEquals*(5, RecursiveBinarySearch.*binarySearchRecursive*(arr, 5));

}

*@Test*

public void testBinarySearchRecursiveIntegerNotFound() {

int[] arr = {1, 2, 3, 4, 5, 6, 7, 8, 9};

*assertEquals*(-1, RecursiveBinarySearch.*binarySearchRecursive*(arr, 10));

}

*@Test*

public void testBinarySearchRecursiveStringFound() {

String[] arr = {"apple", "banana", "cherry", "date", "elderberry"};

*assertEquals*(2, RecursiveBinarySearch.*binarySearchRecursive*(arr, "cherry"));

}

*@Test*

public void testBinarySearchRecursiveStringNotFound() {

String[] arr = {"apple", "banana", "cherry", "date", "elderberry"};

*assertEquals*(-1, RecursiveBinarySearch.*binarySearchRecursive*(arr, "grape"));

}

*@Test*

public void testBinarySearchMultipleIndices() {

int[] arr = {1, 2, 3, 4, 5, 5, 5, 6, 7, 8, 9};

List<Integer> expected = List.*of*(5, 4, 6);

*assertEquals*(expected, RecursiveBinarySearch.*binarySearchMultipleIndices*(arr, 5));

}

*@Test*

public void testBinarySearchRecursiveNullArrayInteger() {

try {

RecursiveBinarySearch.*binarySearchRecursive*((int[]) null, 3);

*fail*("Expected IllegalArgumentException");

} catch (IllegalArgumentException e) {

*assertEquals*("Input array cannot be null", e.getMessage());

}

}

*@Test*

public void testBinarySearchRecursiveNullArrayString() {

try {

RecursiveBinarySearch.*binarySearchRecursive*((String[]) null, "banana");

*fail*("Expected IllegalArgumentException");

} catch (IllegalArgumentException e) {

*assertEquals*("Input array cannot be null", e.getMessage());

}

}

*@Test*

public void testBinarySearchRecursiveEmptyArrayInteger() {

int[] arr = {};

*assertEquals*(-1, RecursiveBinarySearch.*binarySearchRecursive*(arr, 3));

}

*@Test*

public void testBinarySearchRecursiveEmptyArrayString() {

String[] arr = {};

*assertEquals*(-1, RecursiveBinarySearch.*binarySearchRecursive*(arr, "apple"));

}

*@Test*

public void testBinarySearchSingleElementFound() {

int[] arr = {10};

*assertEquals*(0, RecursiveBinarySearch.*binarySearchRecursive*(arr, 10));

}

*@Test*

public void testBinarySearchSingleElementNotFound() {

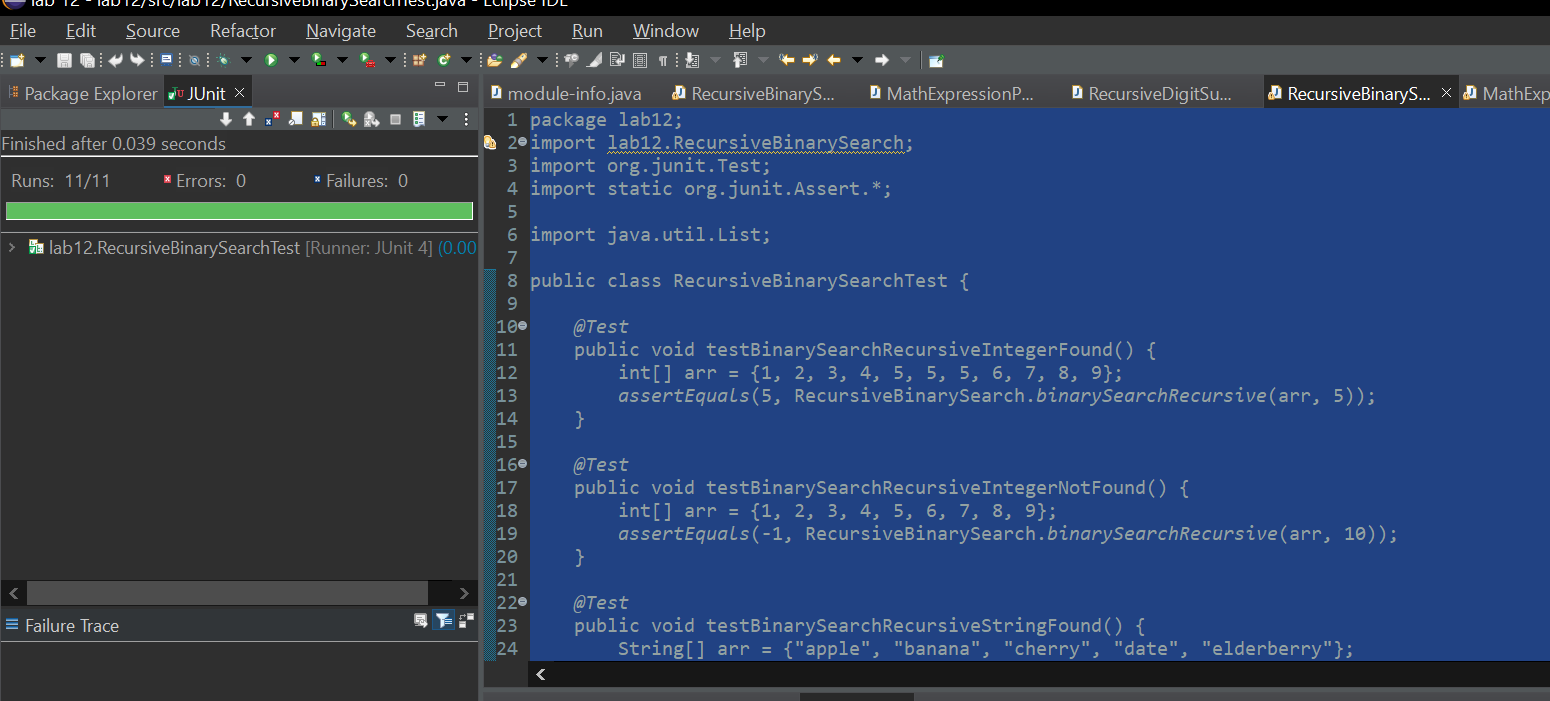
int[] arr = {10};

*assertEquals*(-1, RecursiveBinarySearch.*binarySearchRecursive*(arr, 5));

}

}

**Screenshot:**

****

**Mandatory Enhancements:**

1. Modify the binary search to handle arrays of strings instead of integers.
2. Implement a variant that returns all indices of the target value if it appears multiple times in the array.

**Task 2: Recursive Parser for Mathematical Expressions**

**Objective:** Students will implement a recursive parser to evaluate simple mathematical expressions involving addition, subtraction, multiplication, and division.

**Instructions:**

1. Create a Java program that takes a string input representing a mathematical expression (e.g., "3 + 5 \* 2").
2. Implement a recursive function *evaluateExpression* that parses the expression and calculates its result.
3. Handle operator precedence (multiplication/division first, addition/subtraction second) and use recursion to break down the expression into simpler sub-expressions.
4. Follow good coding practices, including meaningful variable names, comments, and modular code.
5. Test with various expressions, including edge cases like parentheses and mixed operators.

**Mandatory Enhancements:**

1. Extend the parser to handle floating-point numbers.
2. Implement error handling for invalid expressions.

Important: Do not forget to write the specifications and unit tests for the code.

**Code:**

package lab12;

import java.util.regex.Matcher;

import java.util.regex.Pattern;

/\*\*

\* A recursive mathematical expression parser that evaluates

\* mathematical expressions with support for floating-point numbers,

\* basic arithmetic operations, and parentheses.

\*/

public class MathExpressionParser {

/\*\*

\* Evaluates a mathematical expression recursively.

\*

\* **@param** expression The mathematical expression to evaluate

\* **@return** The result of the expression

\* **@throws** IllegalArgumentException for invalid expressions

\*/

public static double evaluateExpression(String expression) {

// Remove all whitespace from the expression

expression = expression.replaceAll("\\s+", "");

// Validate if the expression contains only valid characters

if (!expression.matches("[0-9\\+\\-\\\*/\\.\\(\\)\\s]\*")) {

throw new IllegalArgumentException("Invalid expression: " + expression);

}

try {

return *parseExpression*(expression);

} catch (Exception e) {

throw new IllegalArgumentException("Invalid expression: " + expression);

}

}

/\*\*

\* Recursive method to parse and evaluate the expression.

\*

\* **@param** expression The expression to parse

\* **@return** The evaluated result

\*/

private static double parseExpression(String expression) {

// Handle parentheses first

while (expression.contains("(")) {

expression = *resolveParentheses*(expression);

}

// Then handle multiplication and division

expression = *resolveMultiplicationDivision*(expression);

// Finally, handle addition and subtraction

return *resolveAdditionSubtraction*(expression);

}

/\*\*

\* Resolves expressions within parentheses recursively.

\*

\* **@param** expression The expression containing parentheses

\* **@return** Expression with innermost parentheses resolved

\*/

private static String resolveParentheses(String expression) {

// Find the innermost parentheses

int lastOpenParen = expression.lastIndexOf('(');

int firstCloseParen = expression.indexOf(')', lastOpenParen);

if (lastOpenParen == -1 || firstCloseParen == -1) {

throw new IllegalArgumentException("Mismatched parentheses");

}

// Extract the sub-expression within parentheses

String subExpression = expression.substring(lastOpenParen + 1, firstCloseParen);

// Evaluate the sub-expression

double subResult = *parseExpression*(subExpression);

// Replace the entire parenthetical expression with its result

String replacementExpression = expression.substring(0, lastOpenParen) +

subResult +

expression.substring(firstCloseParen + 1);

return replacementExpression;

}

/\*\*

\* Resolves multiplication and division operations.

\*

\* **@param** expression The expression to resolve

\* **@return** Expression with multiplication and division resolved

\*/

private static String resolveMultiplicationDivision(String expression) {

// Regex pattern to match multiplication and division (including negative and floating-point numbers)

Pattern pattern = Pattern.*compile*("([-]?\\d\*\\.?\\d+)[\\\*/]([-]?\\d\*\\.?\\d+)");

Matcher matcher = pattern.matcher(expression);

while (matcher.find()) {

double left = Double.*parseDouble*(matcher.group(1));

double right = Double.*parseDouble*(matcher.group(2));

double result;

if (matcher.group(0).contains("\*")) {

result = left \* right;

} else {

if (right == 0) {

throw new ArithmeticException("Division by zero");

}

result = left / right;

}

// Replace the operation with its result

expression = expression.replace(matcher.group(0), String.*valueOf*(result));

// Reset matcher

matcher = pattern.matcher(expression);

}

return expression;

}

/\*\*

\* Resolves addition and subtraction operations.

\*

\* **@param** expression The expression to resolve

\* **@return** Final numeric result

\*/

private static double resolveAdditionSubtraction(String expression) {

// Regex pattern to match addition and subtraction (including negative and floating-point numbers)

Pattern pattern = Pattern.*compile*("([-]?\\d\*\\.?\\d+)[+-]([-]?\\d\*\\.?\\d+)");

Matcher matcher = pattern.matcher(expression);

while (matcher.find()) {

double left = Double.*parseDouble*(matcher.group(1));

double right = Double.*parseDouble*(matcher.group(2));

double result;

if (matcher.group(0).contains("+")) {

result = left + right;

} else {

result = left - right;

}

// Replace the operation with its result

expression = expression.replace(matcher.group(0), String.*valueOf*(result));

// Reset matcher

matcher = pattern.matcher(expression);

}

return Double.*parseDouble*(expression);

}

/\*\*

\* Comprehensive test method to demonstrate the parser's capabilities.

\*

\* **@param** args Command-line arguments (not used)

\*/

public static void main(String[] args) {

// Test cases covering various scenarios

String[] testExpressions = {

"3 + 5 \* 2", // Basic precedence

"(3 + 5) \* 2", // Parentheses

"10 / 2 + 3 \* 4", // Mixed operations

"2.5 + 3.7 \* 2", // Floating-point numbers

"-5 + 3 \* -2", // Negative numbers

"(2 + 3) \* (4 - 1)" // Complex parentheses

};

for (String expr : testExpressions) {

try {

double result = *evaluateExpression*(expr);

System.***out***.println(expr + " = " + result);

} catch (IllegalArgumentException e) {

System.***out***.println("Error evaluating " + expr + ": " + e.getMessage());

}

}

// Error case tests

try {

*evaluateExpression*("1 / 0"); // Division by zero

} catch (Exception e) {

System.***out***.println("Caught expected error: " + e.getMessage());

}

try {

*evaluateExpression*("1 + (2 \* 3"); // Mismatched parentheses

} catch (Exception e) {

System.***out***.println("Caught expected error: " + e.getMessage());

}

}

}

**Test:**

package lab12;

import lab12.MathExpressionParser;

import org.junit.Test;

import static org.junit.Assert.\*;

/\*\*

\* Test class for MathExpressionParser.

\*/

public class MathExpressionParserTest {

// Test for simple addition

*@Test*

public void testAddition() {

*assertEquals*(8.0, MathExpressionParser.*evaluateExpression*("3 + 5"), 0.0001);

}

// Test for simple multiplication

*@Test*

public void testMultiplication() {

*assertEquals*(10.0, MathExpressionParser.*evaluateExpression*("5 \* 2"), 0.0001);

}

// Test for handling parentheses

*@Test*

public void testParentheses() {

*assertEquals*(16.0, MathExpressionParser.*evaluateExpression*("(3 + 5) \* 2"), 0.0001);

}

// Test for complex nested parentheses

*@Test*

public void testComplexParentheses() {

*assertEquals*(15.0, MathExpressionParser.*evaluateExpression*("(2 + 3) \* (4 - 1)"), 0.0001);

}

// Test for mismatched parentheses

*@Test*

public void testMismatchedParentheses() {

try {

MathExpressionParser.*evaluateExpression*("1 + (2 \* 3");

*fail*("Expected IllegalArgumentException for mismatched parentheses");

} catch (IllegalArgumentException e) {

*assertTrue*(e.getMessage().contains("Invalid expression"));

}

}

// Test for empty expression

*@Test*

public void testEmptyExpression() {

try {

MathExpressionParser.*evaluateExpression*("");

*fail*("Expected IllegalArgumentException for empty expression");

} catch (IllegalArgumentException e) {

*assertTrue*(e.getMessage().contains("Invalid expression"));

}

}

// Test for invalid characters in the expression

*@Test*

public void testInvalidCharacter() {

try {

MathExpressionParser.*evaluateExpression*("3 + $5");

*fail*("Expected IllegalArgumentException for invalid character");

} catch (IllegalArgumentException e) {

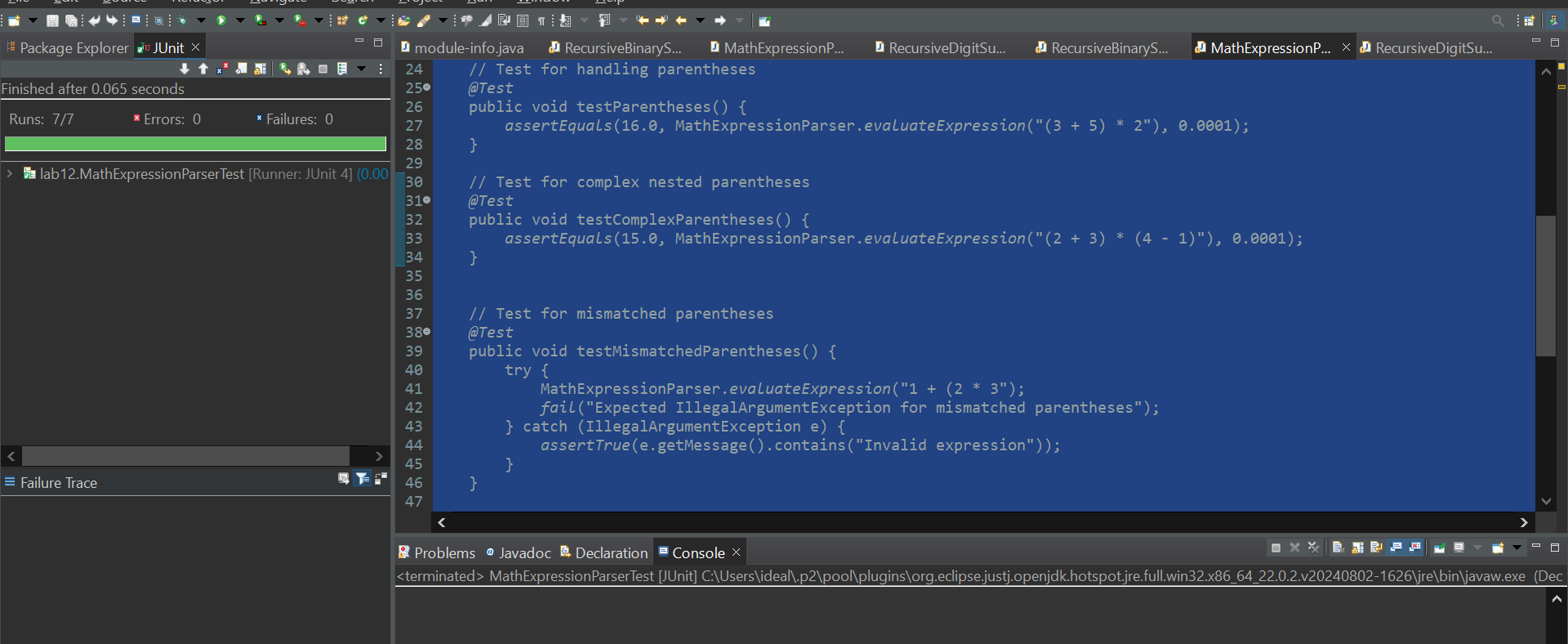
*assertTrue*(e.getMessage().contains("Invalid expression"));

}

}

}

**Screenshot:**

****

**Task 3: Recursive Sum of Digits**

**Objective:** Students will write a recursive function that calculates the sum of the digits of a given non-negative integer

**Instructions:**

1. Create a Java program that takes a non-negative integer as input.
2. Implement a recursive function *sumOfDigits* that computes the sum of its digits.
3. The base case should return 0 when the number is reduced to 0.
4. Test the program with various numbers, including edge cases like 0 and large integers.
5. Follow good coding practices, including meaningful variable names, comments, and modular code.
6. Test the program with various numbers, including edge cases like 0 and large integers.

**Mandatory Enhancements:**

1. Modify the function to handle negative numbers by converting them to positive before performing the sum.
2. Analyze the time complexity of the recursive algorithm for very large numbers.

Important: Do not forget to write the specifications and unit tests for the code.

**Code:**

package lab12;

import java.math.BigInteger;

/\*\*

\* A utility class for calculating the sum of digits using recursive methods.

\* Handles both positive and negative integers, and supports large number inputs.

\*/

public class RecursiveDigitSum {

/\*\*

\* Calculates the sum of digits recursively for an integer input.

\*

\* **@param** number Input number to calculate digit sum

\* **@return** Sum of all digits in the number

\*/

public static int sumOfDigits(int number) {

// Handle negative numbers by converting to positive

number = Math.*abs*(number);

// Base case: when number becomes 0

if (number == 0) {

return 0;

}

// Recursive case: sum last digit with sum of remaining digits

return (number % 10) + *sumOfDigits*(number / 10);

}

/\*\*

\* Overloaded method to handle BigInteger for extremely large numbers.

\*

\* **@param** number BigInteger input

\* **@return** Sum of all digits in the number

\* **@throws** IllegalArgumentException if input is null

\*/

public static int sumOfDigits(BigInteger number) {

if (number == null) {

throw new IllegalArgumentException("Input cannot be null");

}

// Handle edge case for zero

if (number.equals(BigInteger.***ZERO***)) {

return 0;

}

// Take absolute value to avoid any negative BigInteger (not common for sum of digits, but for completeness)

number = number.abs();

// Convert to string for easier digit manipulation

String numberStr = number.toString();

// Base case: only one digit left

if (numberStr.length() == 1) {

return Character.*getNumericValue*(numberStr.charAt(0));

}

// Recursive case: add the first digit and continue with the rest

return Character.*getNumericValue*(numberStr.charAt(0)) +

*sumOfDigits*(new BigInteger(numberStr.substring(1)));

}

/\*\*

\* Time Complexity Analysis:

\* - For int input: O(log n)

\* - Number of recursive calls is proportional to number of digits

\* - Each recursive call reduces the number by a factor of 10

\*

\* - For BigInteger: O(m), where m is number of digits

\* - Slightly less efficient due to string conversion

\*

\* Space Complexity:

\* - Recursive call stack depth is O(log n)

\* - Each recursive call uses constant extra space

\*/

/\*\*

\* Comprehensive test method demonstrating various input scenarios.

\*

\* **@param** args Command-line arguments (not used)

\*/

public static void main(String[] args) {

// Test cases for standard integer input

int[] testIntegers = {

0, // Edge case: zero

123, // Standard positive number

-456, // Negative number

9999, // Large number

1000000 // Very large number

};

System.***out***.println("Integer Input Tests:");

for (int num : testIntegers) {

System.***out***.printf("Sum of digits for %d: %d%n",

num, *sumOfDigits*(num));

}

// Test cases for BigInteger input

BigInteger[] testBigIntegers = {

BigInteger.***ZERO***,

new BigInteger("123456789"),

new BigInteger("999999999999999"),

new BigInteger("123456789012345678901234567890")

};

System.***out***.println("\nBigInteger Input Tests:");

for (BigInteger num : testBigIntegers) {

System.***out***.printf("Sum of digits for %s: %d%n",

num.toString(), *sumOfDigits*(num));

}

// Error handling test

try {

*sumOfDigits*((BigInteger) null);

} catch (IllegalArgumentException e) {

System.***out***.println("\nNull input test passed: " + e.getMessage());

}

}

}

**Test:**

package lab12;

import lab12.RecursiveDigitSum;

import org.junit.Test;

import java.math.BigInteger;

import static org.junit.Assert.*assertEquals*;

import static org.junit.Assert.*assertThrows*;

public class RecursiveDigitSumTest {

*@Test*

public void testNullInputBigInteger() {

// Test for null input in BigInteger method

*assertThrows*(IllegalArgumentException.class, () -> {

RecursiveDigitSum.*sumOfDigits*((BigInteger) null);

});

}

*@Test*

public void testSumOfDigitsInteger() {

// Regular positive integer

*assertEquals*(6, RecursiveDigitSum.*sumOfDigits*(123));

// Negative integer

*assertEquals*(15, RecursiveDigitSum.*sumOfDigits*(-456));

// Zero input

*assertEquals*(0, RecursiveDigitSum.*sumOfDigits*(0));

// Large integer

*assertEquals*(36, RecursiveDigitSum.*sumOfDigits*(9999));

// Very large integer

*assertEquals*(1, RecursiveDigitSum.*sumOfDigits*(1000000));

}

*@Test*

public void testNullInputInteger() {

// Test for null input handling, though this won't be invoked since primitives can't be null.

*assertThrows*(IllegalArgumentException.class, () -> {

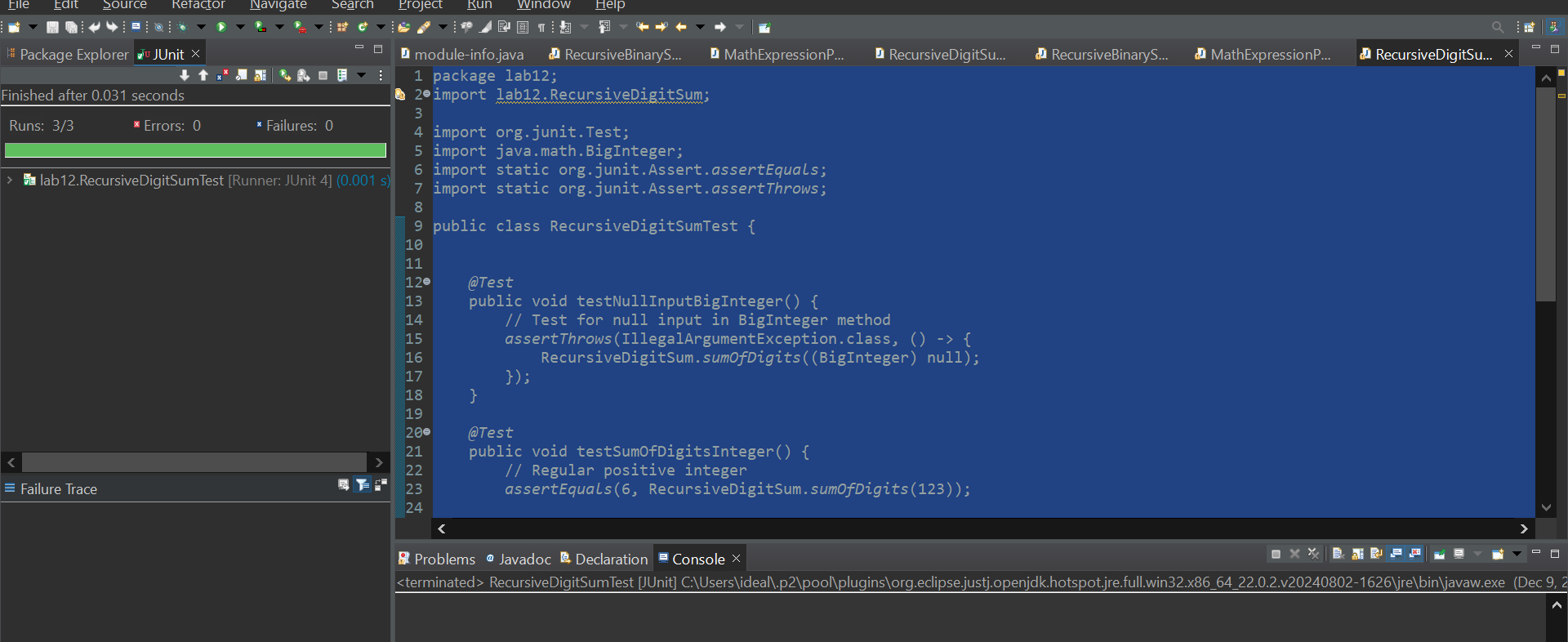
RecursiveDigitSum.*sumOfDigits*(null);

});

}

}

**Screenshot:**

****

**- Paste Code in word document**

**- Push Your Code on GitHub  
- Add Git Link in Document.**

**Compile a single word document by filling in the solution part and submit this Word file on LMS.**

**Solution**

### Deliverables:

Compile a single word document by filling in the solution part and submit this Word file on LMS.

In case of any problems with submissions on LMS, submit your Lab assignments by emailing it to [aftab.farooq@seecs.edu.pk.](mailto:aftab.farooq@seecs.edu.pk.)