CSA1017 - Data Structures and Algorithms 1 - Assignment

Name: Miguel Dingli
I.D Card: 49997M
Course Code: UBSCHICGCFT
Study Unit Code: CSA1017
Degree Programme: B.Sc. (Hons) in Computing Science

Contents

This page intentionally left blank.

Introduction to the assignment

As an introduction to the assignment, a few notes about the structure and contents of the assignment are given below.

- The IDE used to write the code is the Eclipse IDE.
- Every task documentation follows the same structure:
  - Introduction: contains the layout of the methods within the source files, a general overview about the implementations, and any assumptions made for the respective task;
  - Source file(s): includes all classes used in the implementation;
  - Testing: in most cases this is split into three parts consisting of valid/invalid/other data. In each each type, a screenshot shows outputs for some inputs and a table lists all tested data. Additional information is provided in the table when seen necessary.
- In the testing sections:
  - BLANK refers to simply pressing the ENTER key, while SPACE refers to an input consisting only of spaces.
  - *Actual outcome*s are not identical to the output produced by the program. They are meant to extract and show the relevant information from the output and exclude information that is either not crucial to the correctness of the outcome, or cannot be represented in the table due to factors such as size of output.
  - In screenshots, the input/s to the program are shown in cyan.
  - A screenshot may be placed after the table for a better fit within the document.
- Invalid inputs are not limited to the ones listed in the Testing sections since other types of invalid inputs that were not tested may exist.
- In many cases, error checking and/or error handling was not done within the method/s that will come up with the main result required for the task, but was most commonly done in the main method. This approach was taken with the assumption that the program will be used as a whole; if the functions were to be used separately (i.e. with a different main method), some error checking might be excluded as a result, and hence some inputs might cause the program to crash. This approach reduces the potential modularity of the program, and could have been improved by moving error checks into the methods.
- In source code listings, any lines without a line number indicate that the previous line was spread across two lines due to being too long.

Task 1 Roman Numerals

1a Introduction

The code for this task consists of two methods within a single source file. Below is a list showing the layout of the methods within the file.

- Question1_MainClass.java
  - public static void main(String args[])
  - private static String numToRom(int num)

The method 'numToRom' converts the decimal number parameter to a String in Roman numerals format and returns the result. The implementation uses a *greedy algorithm* where the largest values that fit in the decimal number are subtracted first and the smallest values are subtracted last. With each subtraction, the respective Roman number is added to the String which will hold the final Roman number.

1b Source file

Question1_MainClass.java

```java
import java.util.Scanner;
import java.util.InputMismatchException;

public class Question1_MainClass
{
    public static void main(String args[])
    {
        int input = 0; //stores input decimal number
        Scanner sc = new Scanner(System.in); //handles input

        /*Loop until input is negative (indicating a quit)*/
        while (input >= 0)
        {
            System.out.print("Insert decimal number (-ve to quit): ");
            try {
                input = sc.nextInt(); //user input of decimal number

                /*Check if input out of range; else calculate Roman num*/
                if(input == 0 || input > 1024)
                    System.out.println("Input out of range; insert a number from 1
     to 1024.\n");
                else if(input > 0)
                    System.out.println("Roman number: " + numToRom(input) + "\n");
            }
            catch(InputMismatchException ime) //detects non-integer inputs
            {
                System.out.println("Invalid input; insert an integer from 1 to
     1024.\n");
                sc.next(); //clear invalid input
            }
        }
        sc.close(); //close scanner
```

```java
        System.out.println("Program terminated."); //final output
    }

    private static String numToRom(int num)
    {
        String finalRom = ""; //stores resultant Roman numeral
        //All possible letter combinations
        final String ROM[] = {"I","IV","V","IX","X","XL","L","XC","C","CD","D","CM",
    "M"};
        //All corresponding decimal values
        final int DEC[] = {1,4,5,9,10,40,50,90,100,400,500,900,1000};

        /*Greedy algorithm; starting from last decimal value*/
        for(int i=DEC.length-1; i >= 0; i--)
        {
            /*Subtracting until num is smaller than current decimal*/
            while(num >= DEC[i])
            {
                num -= DEC[i]; //subtract decimal
                finalRom += ROM[i]; //add respective numeral to output
            }
        }
        return finalRom; //return resultant Roman numeral
    }
}
```

1c Testing

For this section, valid, invalid, and other test data were input to the program. For each type of test data, a screenshot showing some of the results and a table showing all tested results are provided. The expected outcomes in the valid values table were produced using a decimal to Roman numeral converter built into Google [1]. An example of a conversion is '123 to Roman', which returns CXXIII.

Valid Data

As valid data, many different integer inputs were tested and were all found to be correct. Refer to Figure 1 and Table 1.

Input	Expected Outcome	Actual Outcome	Description
1	I	I	Lower limit
4	IV	IV	Simple value
5	V	V	Simple value
9	IX	IX	Simple value
10	X	X	Simple value
40	XL	XL	Simple value
50	L	L	Simple value
90	XC	XC	Simple value
100	C	C	Simple value
400	CD	CD	Simple value

500	D	D	Simple value
900	CM	CM	Simple value
1000	M	M	Simple value
111	CXI	CXI	Complex value
333	CCCXXXIII	CCCXXXIII	Complex value
444	CDXLIV	CDXLIV	Complex value
555	DLV	DLV	Complex value
777	DCCLXXVII	DCCLXXVII	Complex value
999	CMXCIX	CMXCIX	Complex value
84	LXXXIV	LXXXIV	Complex value
105	CV	CV	Complex value
126	CXXVI	CXXVI	Complex value
147	CXLVII	CXLVII	Complex value
168	CLXVIII	CLXVIII	Complex value
189	CLXXXIX	CLXXXIX	Complex value
1024	MXXIV	MXXIV	Upper limit
1 2 3	Three valid results	Three valid results	Three inputs at once.

Table 1: Valid Data

```
Insert decimal number (-ve to quit): 1
Roman number: I

Insert decimal number (-ve to quit): 90
Roman number: XC

Insert decimal number (-ve to quit): 444
Roman number: CDXLIV

Insert decimal number (-ve to quit): 999
Roman number: CMXCIX

Insert decimal number (-ve to quit): 84
Roman number: LXXXIV

Insert decimal number (-ve to quit): 147
Roman number: CXLVII

Insert decimal number (-ve to quit): 1024
Roman number: MXXIV

Insert decimal number (-ve to quit): 1 2 3
Roman number: I

Insert decimal number (-ve to quit): Roman number: II

Insert decimal number (-ve to quit): Roman number: III

Insert decimal number (-ve to quit):
```

Figure 1: Program output for valid data

Invalid Data

As invalid test data, inputs which are out of range, floating-point number inputs, and inputs containing forbidden characters were all tested and were all handled by the program. Refer to Figure 2 and Table 2.

```
Insert decimal number (-ve to quit): 0
Input out of range; insert a number from 1 to 1024.

Insert decimal number (-ve to quit): 1025
Input out of range; insert a number from 1 to 1024.

Insert decimal number (-ve to quit): 12.34
Invalid input; insert an integer from 1 to 1024.

Insert decimal number (-ve to quit): abc
Invalid input; insert an integer from 1 to 1024.

Insert decimal number (-ve to quit): A B C
Invalid input; insert an integer from 1 to 1024.

Insert decimal number (-ve to quit): Invalid input; insert an integer from 1 to 1024.

Insert decimal number (-ve to quit): Invalid input; insert an integer from 1 to 1024.

Insert decimal number (-ve to quit): MXXIV
Invalid input; insert an integer from 1 to 1024.

Insert decimal number (-ve to quit):
```

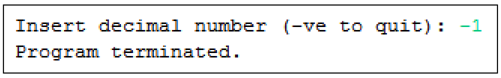
Figure 2: Program output for invalid data

Input	Expected Outcome	Actual Outcome	Description
0	Input out of range	Input out of range	Out of range value
1025	Input out of range	Input out of range	Out of range value
2000	Input out of range	Input out of range	Out of range value
12.34	Invalid Input	Invalid Input	Floating point value
abc	Invalid input	Invalid input	Illegal characters
A B C	Invalid input	Invalid input x3	Illegal characters
MXXIV	Invalid input	Invalid input	Roman number input

Table 2: Invalid Data

Other Data

As other data, spaces as inputs, a blank input, and a negative number indicating that the user wants to quit were all tested. Refer to Figure 3 and Table 3.



```
Insert decimal number (-ve to quit): -1
Program terminated.
```

Figure 3: Program output for other data

Input	Expected Outcome	Actual Outcome	Description
-1	Program quits	Program quit	Negative number
SPACE	No output	No output	Spaces
BLANK	No output	No output	Blank input

Table 3: Other Data

This page intentionally left blank.

Task 2 Evaluation of Expressions in RPN

2a Introduction

The code for this task consists of twelve methods within three source files. Below is a list showing the layout of the methods within the files.

- Question2_MainClass.java
  - public static void main(String args[])
  - private static void evaluateExpress(final String exp)
  - private static void errorInEvaluation(final int type)
- Stack.java
  - public void push(final double itemToPush)
  - public double pop() throws EmptyStackException
  - private void printStackContents(final ArrayList<Double> val)
- ValidationAndChecks.java
  - public boolean isExpressionValid(final String exp)
  - private boolean hasValidCharacters(final String exp)
  - private boolean containsOperands(final String exp)
  - private boolean hasCorrectStructure(final String exp)
  - public boolean isOperator(final char ch)
  - public boolean isNumPart(final int i, final String exp)

The 'evaluateExpress' method decomposes the postfix expression passed to it, performs the necessary operations, and outputs the result. The parameter 'exp' is assumed to have gone through the validation processes implemented in the ValidationAndChecks class which filters out as many potentially invalid inputs as possible, preventing the program from crashing during the evaluation process.

A list of assumptions made for this task is found below.

1. Rounding the values to six decimal places in the stack contents output does not negatively impact the user.
2. Displaying the contents of the stack only after pushes to the stack during evaluation is enough.

2b Source files

Question2_MainClass.java

```
import java.util.EmptyStackException;
import java.util.Scanner;
import java.lang.NumberFormatException;

public class Question2_MainClass
{
    private static ValidationAndChecks valAndCh
        = new ValidationAndChecks();

    public static void main(String args[])
    {
        Scanner sc = new Scanner(System.in); //handles input
        String exp = ""; //stores user input expression

        /*
        User input and validation checks...
        Loop until input starts with a 'q' (indicating a quit)
        */
        mainLoop: do{
            /*Loop until expression is valid*/
            do{
                System.out.print("Insert a postfix expression (q to quit): ");
                exp = sc.nextLine(); //input postfix expression

                /*If expression starts with 'q', stop outer loop*/
                if(exp.startsWith("q"))
                    break mainLoop;
            }while(!valAndCh.isExpressionValid(exp));

            evaluateExpress(exp); //evaluate expression
        }while(true);

        sc.close(); //close scanner
        System.out.println("Program terminated."); //final output
    }

    private static void evaluateExpress(final String exp)
    {
        Stack opStack = new Stack(); //for operands
        int endOfNum = 0; //stores index of end of a number
        double op1 = 0.0, op2 = 0.0; //stored popped operands
        double answer = 0.0; //stores sub-answers

        /*Traverse expression's characters*/
        for(int i=0; i < exp.length(); i++)
        {
            /*
            If the start of a number was found, push.
            If an operators was found, pop two operands,
            perform the operation, and push the result.
            */
            if(valAndCh.isNumPart(i,exp)){
                /*
                Since each operand must be followed by a
                space, if a space is not found, then the
                expression is invalid.
                */
```

```java
                endOfNum = exp.indexOf(" ", i);

                /*If a space was not found after operand*/
                if (endOfNum == -1){
                    try{
                        /*Try to convert exp to a double and push*/
                        opStack.push(Double.parseDouble(exp));
                        break; //exit main loop
                    }catch(NumberFormatException e){
                        /*Expression was invalid after all*/
                        errorInEvaluation(1);
                        return; //dummy return
                    }
                }
                else{
                    try{
                        /*Push the extracted double*/
                        opStack.push( Double.parseDouble(
                                exp.substring(i,endOfNum)
                        ) );
                    }catch (NumberFormatException nfe){
                        errorInEvaluation(2); //operand was not a double
                        return; //dummy return
                    }
                }
                i = endOfNum; //index of end of operand
            }
            else if(valAndCh.isOperator(exp.charAt(i)))
            {
                /*Pop two operands; if stack is empty, error occurred*/
                try{
                    op2 = opStack.pop();
                    op1 = opStack.pop();
                }catch(EmptyStackException e){
                    errorInEvaluation(3);
                    return;
                }
                System.out.println("\nTwo operands popped; "
                    +"Operation performed: "+exp.charAt(i));

                /*Perform operation according to operator*/
                switch(exp.charAt(i)){
                case '+':
                    answer = op1 + op2;
                    break;
                case '-':
                    answer = op1 - op2;
                    break;
                case '*':
                    answer = op1 * op2;
                    break;
                case '/':
                    answer = op1 / op2;
                    break;
                case '^':
                    answer = Math.pow(op1, op2);
                    break;
                }
                opStack.push(answer); //Push result
            }
        }
        /*Print final result to 5 decimal places*/
```

```
        System.out.println("\nAnswer: "+opStack.pop()+"\n");
    }

    private static void errorInEvaluation(final int type)
    {
        /*Prints a custom error message based on 'type'*/
        System.out.println("Error of type "+type
            +" occurred in evaluation.\n");
    }
}
```

Stack.java

```
import java.util.ArrayList;
import java.util.EmptyStackException;

public class Stack
{
    /*Storage for the stack*/
    private ArrayList<Double> items = new ArrayList<Double>();

    public void push(final double itemToPush)
    {
        items.add(itemToPush); //add the item
        printStackContents(items); //print stack
        System.out.println("("+itemToPush+" pushed)");
    }

    public double pop() throws EmptyStackException
    {
        double poppedItem; //stores popped double

        /*If stack not empty, pop item*/
        if (items.size() > 0) {
            poppedItem = items.get(items.size()-1); //get last item
            items.remove(items.size() - 1); //remove last item from list

            /*If stack is not empty, print status*/
            return poppedItem; //return popped item
        }
        /*throw exception if stack is empty*/
        throw new EmptyStackException();
    }

    private void printStackContents(final ArrayList<Double> val)
    {
        int maxWidth=0; //stores maximum width of stack
        int length=0; //length of values to be output

        /*
        Calculate the maximum width of the output by rounding
        each operand in the stack to six decimal places
        */
        for(int i = 0; i < val.size(); i++){
            length = String.format("%.6f", val.get(i)).length();
            if (length > maxWidth)
                maxWidth = length;
        }

        /*Output stack contents*/
```

```java
        System.out.println("\nStack contents: ");
        for (int i=val.size()-1; i>=0; i--){
            /*minimum field width set to maxWidth and operands rounded*/
            System.out.printf(("|%"+maxWidth+".6f|\n"), val.get(i));
        }
        /*Output a base for the the stack*/
        for (int i = 0; i < maxWidth + 2; i++)
            System.out.print("-");
    }
}
```

ValidationAndChecks.java

```java
public class ValidationAndChecks
{
    public boolean isExpressionValid(final String exp)
    {
        /*Validation checks; if one not satisfied, expression is invalid*/
        if (exp.length() == 0) //Check if input is empty
            System.out.println("Invalid expression; empty input.\n");
        else if (!hasValidCharacters(exp)) //...for forbidden characters
            System.out.println("Illegal characters in expression.\n");
        else if (!containsOperands(exp)) //..for a lack of operands
            System.out.println("Invalid expression; lack of operands.\n");
        else if (!hasCorrectStructure(exp)) //..for an invalid structure
            System.out.println("Invalid expression; structural issue.\n");
        else
            return true; //Expression valid
        return false; //Expression invalid
    }

    private boolean hasValidCharacters(final String exp)
    {
        /*Traverse the expression's characters*/
        for(int i=0; i<exp.length(); i++)
        {
            /*
            If the character is not a part of an operand, an
            operator, or a space, then expression is invalid
            */
            if(!isNumPart(i,exp)
                && !isOperator(exp.charAt(i))
                && exp.charAt(i) != ' ')
                return false; //Invalid character detected
        }
        return true; //All characters are valid
    }

    private boolean containsOperands(final String exp)
    {
        /*Checks if expression contains any digit*/
        for (int i = 0; i < 10; i++) {
            if (exp.contains(""+i))
                return true; //Expression contains a digit
        }
        return false; //Expression does not contain digits
    }

    private boolean hasCorrectStructure(String exp)
    {
```

```
        exp += " ";

        /*Operand and operator counts*/
        int operandCount = 0, operatorCount = 0;

        /*Traverse the expression's characters*/
        for (int i = 0; i < exp.length(); i++)
        {
            /*If the start of an operand is found*/
            if (isNumPart(i,exp))
            {
                /*Skip remainder of operand*/
                do i++;
                while (i < exp.length() && isNumPart(i,exp));
                operandCount++; //increment operand count
            }
            else if (isOperator(exp.charAt(i))) //operator found
            {
                operatorCount++; //increment operator count
                i++; //Skip space after operator

                /*
                In postfix, there has to be as much operands
                as one plus the amount of operators
                */
                if(operatorCount + 1 > operandCount)
                    return false;
            }
        }

        /*
        In postfix notation, there has to be as much
        operands as one plus the amount of operators.
        */
        if (operatorCount + 1 == operandCount)
            return true;
        else
            return false;
    }

    public boolean isOperator(final char ch) {
        /*True if the character is one of the allowed operators*/
        return (ch == '+'    || ch == '-'
                || ch == '*' || ch == '/'
                || ch == '^');
    }

    public boolean isNumPart(final int i, final String exp) {
        /*
        True if the character is a digit, a decimal point
        or the character is a minus and the next character
        is a digit, indicating a negative number.
        */
        final char ch = exp.charAt(i);
        return(Character.isDigit(ch)
                || ch=='.'
                || (i+1<exp.length() && ch=='-'
                    && Character.isDigit(exp.charAt(i+1)))
        );
    }
}
```

2c Testing

For this section, valid, invalid, and other test data were input to the program. For each type of test data, a screenshot showing some of the results and a table showing all tested results are provided.

Valid Data

As valid data, many forms of input were tested to make full use of the potential of the calculator. The first test checks that the expression evaluator realizes that the input is only a single operand and simply returns it. The next ten inputs test all of the operators with negative numbers and specific special cases such as a division by zero and an exponent of zero. The remaining inputs use many operators. Refer to Figure 4 and Table 4.

Input	Expected Outcome	Actual Outcome
123456	123456	123456.0
1 6 +	7	7.0
2 5 -	-3	-3.0
2 -5 +	-3	-3.0
3 4 *	-12	12.0
-3 4 *	-12	-12.0
4 3 /	$1.\dot{3}$	1.3333333333333333
-4 -3 /	$1.\dot{3}$	1.3333333333333333
5 0 /	∞	Infinity
6 0 ∧	1	1.0
7 2 ∧	49	49.0
1 3 + 5 - 7 * 9 / 11 ∧	-0.06301019684	-0.06301019683965384
1 2 3 * - 4 5 * 6 / +	$-1.\dot{6}$	-1.6666666666666665
2 4 6 8 10 12 + - * / ∧	0.96753177852	0.9675317785238916
2 -4 6 -8 10 -12 + - * / ∧	1.08005973889	1.080059738892306
123.456 789.10 * 11.12 / 0.25 ∧ 1.75 ∧	53.0714572508	53.07145725082074
12.34 56.78 / 9101112 / 2 ∧ 1000 *	5.7115531e-13	5.702307395303041E-13

Table 4: Valid Data

```
Insert a postfix expression (q to quit): 12.34 56.78 / 9101112 / 2 ^ 1000 *

Stack contents:
|12.340000|
-----------(12.34 pushed)

Stack contents:
|56.780000|
|12.340000|
-----------(56.78 pushed)

Two operands popped; Operation performed: /

Stack contents:
|0.217330|
----------(0.2173300457907714 pushed)

Stack contents:
|9101112.000000|
|      0.217330|
----------------(9101112.0 pushed)

Two operands popped; Operation performed: /

Stack contents:
|0.000000|
----------(2.3879504591391843E-8 pushed)

Stack contents:
|2.000000|
|0.000000|
----------(2.0 pushed)

Two operands popped; Operation performed: ^

Stack contents:
|0.000000|
----------(5.702307395303041E-16 pushed)

Stack contents:
|1000.000000|
|   0.000000|
-------------(1000.0 pushed)

Two operands popped; Operation performed: *

Stack contents:
|0.000000|
----------(5.702307395303041E-13 pushed)

Answer: 5.702307395303041E-13

Insert a postfix expression (q to quit):
```

Figure 4: Program output for a valid expression

Invalid Data

As invalid data, many type of invalid inputs were tested, including inputs containing forbidden characters, inputs with missing or extra operands or operators, inputs with incorrect structure, and also inputs consisting only of spaces and a blank input. Each of these inputs were handled by the program and were not allowed to be passed to the expression evaluation function. Refer to Figure 5 and Table 5.

```
Insert a postfix expression (q to quit): abc
Illegal characters in expression.

Insert a postfix expression (q to quit): 1 2+
Invalid expression; structural issue.

Insert a postfix expression (q to quit): 1 2 + 3
Invalid expression; structural issue.

Insert a postfix expression (q to quit): + 1 2
Invalid expression; structural issue.

Insert a postfix expression (q to quit): + - *
Invalid expression; lack of operands.

Insert a postfix expression (q to quit):
Invalid expression; empty input.

Insert a postfix expression (q to quit):
```

Figure 5: Program output for invalid data

Input	Expected Outcome	Actual Outcome	Description
abc	Invalid input	Invalid input	Illegal characters
A B C	Invalid input	Invalid input	Illegal characters
1 2+	Invalid input	Invalid input	Missing space
1 2	Invalid input	Invalid input	Missing operator
1 2 + -	Invalid input	Invalid input	Invalid structure
1 2 + 3	Invalid input	Invalid input	Invalid structure
1 2 3 +	Invalid input	Invalid input	Invalid structure
+ 1 2	Invalid input	Invalid input	Invalid structure
1 2 + - 3	Invalid input	Invalid input	Invalid structure
+ - *	Invalid input	Invalid input	Lack of operands
SPACE	Invalid input	Invalid input	Lack of operands
BLANK	Invalid input	Invalid input	Blank expression

Table 5: Invalid Data

Other Data

As other data, the inputs ‘q’ and ‘quit’ were tested out to check that the program terminates when the input starts with the letter ‘q’. Refer to Figure 6 and Table 6.

```
Insert a postfix expression (q to quit): quit
Program terminated.
```

Figure 6: Program output for other data

Input	Expected Outcome	Actual Outcome	Description
q	Program quits	Program quit	Letter ’q’
quit	Program quits	Program quit	Word starting with ’q’

Table 6: Other Data

Task 3 Prime Numbers

3a Introduction

The code for this task consists of four methods within one source file. Below is a list showing the layout of the methods within the file.

- Question3_MainClass.java
  - public static void main(String args[])
  - private static boolean isPrime(final long n)
  - private static boolean isPrime_sieve(final int n)
  - private static int indOf(final int i)

The ‘isPrime’ method implements an algorithm that checks if a number is prime by going through the divisors of the number. A number of optimizations were used to make this check a much more efficient one and these are listed further on, in the next subsection.

The ‘isPrime_sieve’ is an implementation of the Sieve of Eratosthenes but with the added functionality that that method returns a boolean indicating whether the number is a prime or not. It is thus similar to the ‘isPrime’ method, but also prints a list of all the primes up to and including the number passed in and also a count indicating how many primes were listed.

For convenience, the method ‘indOf’ returns an adjusted index so that the index of the value ‘n’ is at index $indOf(n)$. This avoids having to leave the first two locations in the array unused for the indices to match with the value stored at the index.

For each input, the program checks whether the value is prime using both ‘isPrime’ and ‘isPrime_sieve’, compares the results to make sure that they match, and produces an output accordingly.

Optimizations

A list of optimizations implemented in the isPrime method is found below.

- If the parameter ‘n’ is 2, the program immediately knows that it is a prime and returns a ‘true’, bypassing further checks.
- If the parameter ‘n’ is either 1 or is even (exluding 2), the program immediately knows that it is not a prime and returns a ‘false’, bypassing further checks.
- The loop in the isPrime method loops only through odd potential divisors of ‘n’ since, by this point, it is certain that ‘n’ is not an even number and hence does not have even divisors.

- The loop in the isPrime method loops only up to at most $1 + \sqrt{n}$. For any divisor 'd' of 'n', there is a corresponding divisor 'f' such that $d \cdot f = n$, where $d \leq \sqrt{n}$ and $f \leq \sqrt{n}$. Since product is commutative, any divisors of 'n' beyond $1 + \sqrt{n}$ will be the list of values that had already been checked, but in reverse order. Hence, it is useless to check for divisors beyond $1 + \sqrt{n}$.
- The method immediately returns 'false' when the first divisor is found, indicating that 'n' is not a prime number. It does not check for more divisors.

3b Source file

Question3_MainClass.java

```java
import java.util.Scanner;
import java.util.InputMismatchException;

public class Question3_MainClass
{
    public static void main(String args[])
    {
        Scanner sc = new Scanner(System.in); //handles input

        int n=0; //stores number to check if prime
        boolean a, b; //stores result of the two functions

        /*Loop until input is negative*/
        do{
            try{
                System.out.print("Insert integer (0 or -ve to quit): ");
                n = sc.nextInt(); //number input
            }catch(InputMismatchException ime){
                System.out.println("Invalid input; not an integer.\n");
                sc.next(); //clear invalid input
                continue; //restart the loop
            }

            if(n <= 0) //If input is 0 or negative, break loop
                break;

            a = isPrime(n); //supports up to Long.MAX_VALUE
            b = isPrime_sieve(n); //supports up to VM limit of array size

            /*
            If results produced by isPrime and sieve do not match,
            an error has occurred. Otherwise, print final output.
            */
            if(a!=b){
                System.out.println("Error in one of the results.");
                break;
            }
            else{
                if(a) //if n is prime
                    System.out.println("By the two algorithms: " + n + " is a prime
    number.\n");
                else  //if n is not prime
```

```
                    System.out.println("By the two algorithms: " + n + " is not a prime number.\n");
                }
            }while(true);

            sc.close(); //close scanner
            System.out.println("Program terminated."); //final output
        }

        private static boolean isPrime(final long n)
        {
            if(n==2)
                return true; //2 is prime
            else if(n<=1 || n%2==0)
                return false; //values <= 1 and even numbers are not prime
            else{ //Check if n is prime
                /*Loop through odd values up to sqrt(n)+1*/
                for(long i=3; i<=1+Math.sqrt(n); i+=2)
                    if(n%i == 0) //if i is a divisor of n
                        return false; //value is not prime
                return true; //value is prime
            }
        }

        private static boolean isPrime_sieve(final int n)
        {
            int primesCount=0; //counts amount of primes found

            /*1 and smaller values are not primes*/
            if(n<=1){
                System.out.println("Sieve of Eratosthenes primes: none");
                return false;
            }

            /*Declaration and initialization of nums array*/
            int nums[] = new int[n-1]; //from 2 to n, so n-1 length
            for(int i=2; i<=n; i++)
                nums[indOf(i)] = i; //(i-2)th element corresponds to value i

            /*Cancelling out non-primes*/
            for(int i=2; i<=n; i++){
                /*if already cancelled, skip*/
                if(nums[indOf(i)] == -1)
                    continue;
                /*cancel all multiples of i {excluding i itself}*/
                for(int j=2; (i*j)<=n; j++)
                    nums[indOf(i*j)] = -1;
            }

            /*Output primes found*/
            System.out.print("Sieve of Eratosthenes primes: 2");
            primesCount=1; //First prime is 2
            for(int i=1; i<nums.length; i++){
                if(nums[i] != -1){
                    System.out.print("," + nums[i]);
                    primesCount++;
                }
            }
            System.out.println("\nAmount of primes: "+primesCount);

            /*If n was cancelled, then n is not prime*/
            if(nums[indOf(n)] == -1)
```

```
            return false; //n is not prime
        else
            return true; //n is prime
    }

    /*Returns the index of the value i (assuming i >= 2)*/
    private static int indOf(final int i){
        return i-2;
    }
}
```

3c Testing

For this section, valid, invalid, and other test data were input to the program. For each type of test data, a screenshot showing some of the results and a table showing all tested results are provided. The expected outcomes in the valid values table were produced using a primality test [2] and an nth prime calculator [3].

Valid Data

As valid data, various integers were input to the program. Due to the lists of prime numbers being very long in some cases, the screenshots crop out long lists. However, the output also includes a count of primes which is assumed to be reliable enough to indicate that the list of prime numbers found is correct. Refer to Figure 7 and Table 7.

```
Insert integer (0 or -ve to quit): 1
Sieve of Eratosthenes primes: none
By the two algorithms: 1 is not a prime number.

Insert integer (0 or -ve to quit): 22
Sieve of Eratosthenes primes: 2,3,5,7,11,13,17,19
Amount of primes: 8
By the two algorithms: 22 is not a prime number.

Insert integer (0 or -ve to quit): 6666
Sieve of Eratosthenes primes: 2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,
Amount of primes: 859
By the two algorithms: 6666 is not a prime number.

Insert integer (0 or -ve to quit): 54321
Sieve of Eratosthenes primes: 2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,
Amount of primes: 5525
By the two algorithms: 54321 is not a prime number.

Insert integer (0 or -ve to quit): 2
Sieve of Eratosthenes primes: 2
Amount of primes: 1
By the two algorithms: 2 is a prime number.

Insert integer (0 or -ve to quit): 11
Sieve of Eratosthenes primes: 2,3,5,7,11
Amount of primes: 5
By the two algorithms: 11 is a prime number.

Insert integer (0 or -ve to quit): 257
Sieve of Eratosthenes primes: 2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,
Amount of primes: 55
By the two algorithms: 257 is a prime number.

Insert integer (0 or -ve to quit): 66889
Sieve of Eratosthenes primes: 2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,
Amount of primes: 6666
By the two algorithms: 66889 is a prime number.

Insert integer (0 or -ve to quit):
```

Figure 7: Program output for valid data

Input	Expected Outcome	Actual Outcome	Description
1	No primes; Input is not prime	No primes; Input is not prime	Smallest non-prime
22	8 primes; Input is not prime	8 primes; Input is not prime	Non-prime input
33	11 primes; Input is not prime	11 primes; Input is not prime	Non-prime input
6666	859 primes; Input is not prime	859 primes; Input is not prime	Non-prime input
7777	985 primes; Input is not prime	985 primes; Input is not prime	Non-prime input
54321	5525 primes; Input is not prime	5525 primes; Input is not prime	Non-prime input
2	1 prime; Input is prime	1 prime; Input is prime	Smallest prime
11	5 primes; Input is prime	5 primes; Input is prime	5th prime
257	55 primes; Input is prime	55 primes; Input is prime	55th prime
66889	6666 primes; Input is prime	6666 primes; Input is prime	6,666th prime
13651973	888888 primes; Input is prime	888888 primes; Input is prime	888,888th prime
15485857	999999 primes; Input is prime	999999 primes; Input is prime	999,999th prime
2038074739	99999999 primes; Input is prime	OutOfMemoryError	99,999,999th prime

Table 7: Valid Data

Invalid Data

As invalid data, a floating-point number input and inputs consisting of forbidden characters were tested. All of these were handled by the program. Refer to Figure 8 and Table 8.

```
Insert integer (0 or -ve to quit): 12.34
Invalid input; not an integer.

Insert integer (0 or -ve to quit): abc
Invalid input; not an integer.

Insert integer (0 or -ve to quit): A B C
Invalid input; not an integer.

Insert integer (0 or -ve to quit): Invalid input; not an integer.

Insert integer (0 or -ve to quit): Invalid input; not an integer.

Insert integer (0 or -ve to quit):
```

Figure 8: Program output for invalid data

Input	Expected Outcome	Actual Outcome	Description
12.34	Invalid input	Invalid input	Floating point value
abc	Invalid input	Invalid input	Illegal characters
A B C	Invalid input	Invalid input x3	Illegal characters

Table 8: Invalid Data

Other Data

As other data, a negative input and zero were tested to see if the program quits. Spaces as input and a blank input were also tested. The program acted as expected in every case. Refer to Figure 9 and Table 9.

```
Insert integer (0 or -ve to quit): 0
Program terminated.
```

```
Insert integer (0 or -ve to quit): -1
Program terminated.
```

Figure 9: Program output for other data

Input	Expected Outcome	Actual Outcome	Description
-1	Program quits	Program quit	Negative number
0	Program quits	Program quit	Zero
SPACE	No output	No output	Spaces
BLANK	No output	No output	Blank input

Table 9: Other Data

This page intentionally left blank.

Task 4 Optimized Shell Sort

4a Introduction

The code for this task consists of four methods within one source file. Below is a list showing the layout of the methods within the file.

- Question4_MainClass.java
  - public static void main(String args[])
  - private static void shellSort(int nums[])
  - private static void swap(int nums[], int i1, int i2)
  - private static boolean sorted(int nums[])

The 'shellSort' method implements a Shell Sort algorithm. This algorithm is optimized by using Hibbard's sequence of gaps which is discussed in the next subsection. The program sorts the numbers in the 'nums' array by comparing each pair of values separated by the gap 'g' and swapping them if they are out of order with respect to each other. The gap is reduced after each traversal of the array until it is equal to 1. At this point, the array is repeatedly traversed until no swaps take place in a particular traversal, indicating that the array of numbers is sorted.

For convenience, a 'sorted' method traverses the array after it is sorted to confirm that the array was successfully sorted. An output indicates this.

Optimizations

The Shell Sort algorithm was optimized by using Hibbard's sequence of gaps $(1, 3, 7, 15, 31, 63, ...)$, where the gap $g = 2^k - 1$ for an improved performance of $O\left(n^{3/2}\right)$. For the gap to be set to the previous term in the sequence for a smaller gap, it is divided by 2 using an integer division. The 0.5 fractional part due to the sequence terms begin odd is eliminated each time (due to the division being an *integer* division), which means that gap successfully follows Hibbard's sequence.

4b Source file

Question4_MainClass.java

```java
import java.util.Arrays;

public class Question4_MainClass
{
    public static void main(String args[])
    {
        int nums[] = new int[16384]; //numbers to be sorted

```

```java
        /*Fill array with random values*/
        for(int i=0; i<nums.length; i++)
            nums[i] = (int) (Math.random()*100000); //0 to 99999

        /*Outputs and sort*/
        System.out.println("Before shell sort: " + Arrays.toString(nums).substring
    (0, 120));
        shellSort(nums); //sort
        System.out.println("After shell sort:  " + Arrays.toString(nums).substring
    (0, 120));

        /*Checks if array was successfully sorted*/
        if(sorted(nums))
            System.out.println("Sort succeded.");
        else
            System.out.println("Sort failed.");
    }

    private static void shellSort(int nums[])
    {
        boolean swapped = true; //indicates that a swap occurred

        int g = 1;  //gap
        while(g < nums.length)
            g*=2;   //up to (2^k)
        g--;        //g is now (2^k)-1

        while(g!=1 || swapped){
            if(g >= 2) //if interval is not 1
                g/=2;  //divide interval by 2

            swapped = false; //no values swapped yet

            /*Traverse the array*/
            for(int i=0; i+g < nums.length; i++){
                if(nums[i] > nums[i+g]){
                    swap(nums, i, i+g); //swap values
                    swapped = true; //swap occurred
                }
            }
        }
    }

    private static void swap(int nums[], int i1, int i2)
    {
        /*Swap two values of nums[] found at i1 and i2*/
        int temp = nums[i1];
        nums[i1] = nums[i2];
        nums[i2] = temp;
    }

    private static boolean sorted(final int nums[])
    {
        /*Detect values that are out of order*/
        for(int i=0; i<nums.length-1; i++)
            if(nums[i] > nums[i+1])
                return false; //list is not sorted
        return true; //list is sorted
    }
}
```

4c Testing

Since the array of values is generated by the program and there is no input by the user, testing will focus on the range of random numbers generated. The array size is 16384. Only ranges expected to be valid will be tested.

Valid Random Number Ranges

As valid random number ranges, various ranges were tested. Due to the large amount of terms, screenshots crop out a part of the array output. However, the sort success indicator is assumed to be reliable enough to indicate that the sort was successful. Refer to Figure 10 and Table 10.

```
Before shell sort: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
After shell sort:  [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
Sort succeded.
```

(i)

```
Before shell sort: [51847, 58731, 91207, 20428, 88865, 41532, 67669, 70495, 33389,
After shell sort:  [4, 6, 13, 21, 24, 27, 33, 49, 52, 78, 80, 82, 86, 91, 106, 116,
Sort succeded.
```

(ii)

```
Before shell sort: [23837858, 69766642, 87044242, 18290811, 7508927, 42841783, 1421
After shell sort:  [468, 1083, 2010, 4935, 5871, 6702, 10543, 31271, 45667, 51878,
Sort succeded.
```

(iii)

```
Before shell sort: [-720550218, -551866851, -265544666, -137065771, -9857832, -2612
After shell sort:  [-999891612, -999847953, -999731828, -999587049, -999564651, -99
Sort succeded.
```

(iv)

Figure 10: Program output for (i) a 0 to 0 range, (ii) a 0 to 99,999 range, (iii) a 0 to 99,999,999 range, and (iv) a -999,999,999 to 0 range

Range	Expected Outcome	Actual Outcome	Description
0 to 0	Sort succeeded	Sort succeeded	Zero range
0 to 9	Sort succeeded	Sort succeeded	Small positive range
0 to 999	Sort succeeded	Sort succeeded	Small positive range
0 to 99999	Sort succeeded	Sort succeeded	Large positive range
0 to 9999999	Sort succeeded	Sort succeeded	Large positive range
0 to 999999999	Sort succeeded	Sort succeeded	Large positive range
100000 to 199999	Sort succeeded	Sort succeeded	Shifted positive range
-99999 to 0	Sort succeeded	Sort succeeded	Large negative range
-9999999 to 0	Sort succeeded	Sort succeeded	Large negative range
-999999999 to 0	Sort succeeded	Sort succeeded	Large negative range

Table 10: Valid Random Number Ranges

This page intentionally left blank.

Task 5 Square Root using the Newton-Raphson Method

5a Introduction

The code for this task consists of three methods within one source file. Below is a list showing the layout of the methods within the file.

- Question5_MainClass.java
  - public static void main(String args[])
  - private static double findSqrRt(final double num, final int acc)
  - private static double roundNum(final double num, final int acc)

The ‘findSqrRt’ method takes a floating-point value as a parameter and returns an approximation to the square root of the given number. The method implements the Newton-Raphson numerical method using an iterative approach. With each iteration, a closer approximation to the square root of the number is found. The formula used to produce such result is discussed in the next subsection titled accordingly.

The second method ‘roundNum’ is used to round the given number to the specified amount of decimal places using ‘DecimalFormat’. The ‘format’ string is set according to the accuracy required.

A list of assumptions made for this task is found below.

1. A maximum accuracy (the size of the fractional part) of eight digits is enough for the average user’s needs. This maximum was implemented to avoid cases where the algorithm cycles infinitely between two values when the two values reproduce each other in the Newton Raphson Method. The workings are rounded to a *maximum* of ten digits to preserve accuracy while avoiding the above mentioned issues.

Deriving the formula used to obtain the square root

To find an approximation to the square root $\sqrt{A}$ of a number ‘A’, the Newton-Raphson method was used. Provided an initial x_0, the method finds better approximations x_1, x_2, ..., $x_n + 1$ to a function’s roots by repeatedly applying the following formula [4]:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Since the Newton-Raphson method will be used to obtain the *square root* of a value, a suitable function needs to be formed. Consider the root $x = \sqrt{A}$. Squaring this function to get an equation in terms of A[1], we get

$x^2 = A$. It follows that $x^2 - A = 0$. But the roots of a function are found by obtaining the values of the domain for which the function is zero, i.e. $f(x) = 0$. Let $f(x) = x^2 - A$; hence, a function in terms of 'A' for which one of the roots is the square root of 'A' has been discovered.

The derivative $f'(x)$ of this function is $f'(x) = 2x$. Substituting $f(x)$ and $f'(x)$ in the formula discussed previously, we get $x_{n+1} = x_n - \frac{(x_n)^2 - A}{2x_n}$. This is the formula that was implemented in the 'findSqrRt' method.

5b Source file

Question5_MainClass.java

```
import java.util.Scanner;
import java.util.InputMismatchException;
import java.math.RoundingMode;
import java.text.DecimalFormat;

public class Question5_MainClass
{
    public static void main(String args[])
    {
        Scanner sc = new Scanner(System.in); //handles input
        double num = 0.0, sqrRt = 0.0; //user input and final answer
        int acc = 0; //accuracy (decimal places)

        /*Loop until an input is negative (indicating a quit)*/
        while(num>=0 && acc>=0)
        {
            try{
                System.out.print("Insert a number (-ve to quit): ");
                num = sc.nextDouble();  //input number
                if(num < 0)
                    break;
                System.out.print("Insert the accuracy (-ve to quit): ");
                acc = sc.nextInt();     //input accuracy
                if(acc < 0)
                    break;
            }catch(InputMismatchException ime){
                System.out.println("Invalid input.\n");
                sc.next(); //clear invalid input
                continue; //go to start of loop
            }

            /*Maximum fractional part size of 8 digits*/
            if(acc > 8)
                acc = 8;

            /*Find square root and convert to string*/
            sqrRt = findSqrRt(num, acc);

            /*If fractional part is zero, output an integer as answer*/
            if((int) sqrRt == sqrRt)
                System.out.println("Square root of " + num + " is " + (int)sqrRt + "
    \n");
            else
                System.out.println("Square root of " + num + " is " + sqrRt + "\n");
        }
        sc.close(); //close scanner
```

```
        System.out.println("Program terminated."); //final output
    }

    private static double findSqrRt(final double num, final int acc)
    {
        double sqrRt = num; //stores result
        double previous = 0.0; //stores previous value
        double latest = sqrRt; //stores latest value

        /*Square root of zero is zero*/
        if(num == 0.0)
            return 0.0;

        /*Loop until latest value is equal to previous value*/
        do{
            /*Generate next value by Newton-Raphson Method*/
            sqrRt -= (Math.pow(sqrRt,2) - num) / (2*sqrRt);
            /*Set previous to previous value*/
            previous = latest;
            /*Rounded to avoid infinite loops*/
            latest = roundNum(sqrRt,acc+2);
        }while (previous != latest);

        return roundNum(sqrRt,acc); //return rounded answer
    }

    private static double roundNum(final double num, final int acc)
    {
        /*Setting format of rounded number*/
        String format = "#.";
        for(int i=0; i<acc; i++)
            format += "#";
        DecimalFormat df = new DecimalFormat(format);
        df.setRoundingMode(RoundingMode.HALF_UP);

        /*Round number, convert to double, and return*/
        return Double.parseDouble(df.format(num));
    }
}
```

5c Testing

For this section, valid, invalid, and other test data were input to the program. For each type of test data, a screenshot showing some of the results and a table showing all tested results are provided. When the input shows two values, these correspond to the number (for which the square root will be found) and the accuracy (decimal places to round to). The expected outcomes in the valid values table were produced using an online square root calculator [5].

Valid Data

As valid data, various numbers and accuracies were tested. The test data ranges from small to larger values and includes floating-point numbers. The accuracy ranges from zero to eight, with the exception of a 20 accuracy shown in the table which the program changes to 8. In cases where the square root result is an integer, the accuracy is ignored and the output excludes a fractional part. Refer to Figure 11 and Table 11.

```
Insert a number (-ve to quit): 0
Insert the accuracy (-ve to quit): 3
Square root of 0.0 is 0

Insert a number (-ve to quit): 9
Insert the accuracy (-ve to quit): 5
Square root of 9.0 is 3

Insert a number (-ve to quit): 8888
Insert the accuracy (-ve to quit): 1
Square root of 8888.0 is 94.3

Insert a number (-ve to quit): 1234.5678
Insert the accuracy (-ve to quit): 7
Square root of 1234.5678 is 35.136417

Insert a number (-ve to quit): 987654321
Insert the accuracy (-ve to quit): 8
Square root of 9.87654321E8 is 31426.96805293

Insert a number (-ve to quit): 0.987654321
Insert the accuracy (-ve to quit): 8
Square root of 0.987654321 is 0.99380799

Insert a number (-ve to quit):
```

Figure 11: Program output for valid data

Input	Expected Outcome	Actual Outcome	Description
0 and 3	0	0	Smallest valid input
1 and 4	1	1	Accuracy ignored
9 and 5	3	3	Accuracy ignored
9 and 20	3	3	Accuracy ignored
7777 and 0	88	88	0 decimal places
8888 and 1	94.3	94.3	1 decimal places
9999 and 2	99.99	99.99	2 decimal places
321.654 and 6	17.934715	17.934715	6 decimal places
1234.5678 and 7	35.1364170	35.136417	Last zero eliminated
987654321 and 8	31426.96805293	31426.96805293	8 decimal places
0.987654321 and 8	0.99380799	0.99380799	8 decimal places

Table 11: Valid Data

Invalid Data

As invalid data, both floating-point values and forbidden characters were tested. Since the first input can be both an integer or a floating-point value, '9' was input for tests focusing on the second input. All of the invalid inputs were handled by the program and did not cause the program to crash. Refer to Figure 12 and Table 12.

```
Insert a number (-ve to quit): 9
Insert the accuracy (-ve to quit): 12.34
Invalid input.

Insert a number (-ve to quit): abc
Invalid input.

Insert a number (-ve to quit): 9
Insert the accuracy (-ve to quit): abc
Invalid input.

Insert a number (-ve to quit): A B C
Invalid input.

Insert a number (-ve to quit): Invalid input.

Insert a number (-ve to quit): Invalid input.

Insert a number (-ve to quit):
```

Figure 12: Program output for invalid data

Input	Expected Outcome	Actual Outcome	Description
9 and 12.34	Invalid input	Invalid input	Floating point value accuracy
9 and abc	Invalid input	Invalid input	Illegal characters
abc	Invalid input	Invalid input	Illegal characters
A B C	Invalid input	Invalid input x3	Illegal characters

Table 12: Invalid Data

Other Data

As other data, a negative first input and negative second input were tested. Spaces as input and a blank input were also tested. The program performed as expected for each input. Refer to Figure 13 and Table 13.

```
Insert a number (-ve to quit): -1
Program terminated.
```

```
Insert a number (-ve to quit): 9
Insert the accuracy (-ve to quit): -1
Program terminated.
```

Figure 13: Program output for other data

Input	Expected Outcome	Actual Outcome	Description
-1	Program quits	Program quit	Negative number
9 and -1	Program quits	Program quit	Negative accuracy
SPACE	No output	No output	Spaces
BLANK	No output	No output	Blank input

Table 13: Other Data

Task 6 Multiplication of Matrices

6a Introduction

The code for this task consists of two methods within one source file. Below is a list showing the layout of the methods within the file.

- Question6_MainClass.java
  - public static void main(String args[])
  - private static void printMatrix(double matrix[][], int fieldSize)

The product of the two matrices is performed within the 'main' method. The method used to come up with the resultant matrix is discussed in a subsection below titled accordingly. The 'printMatrix' outputs the values of a two-dimensional array as a matrix. It takes as an argument a minimum field size which is used to better present the values in the output.

A list of assumptions made for this task is found below.

1. Rounding the generated random values and the values to be output in the 'printMatrix' method to two decimal places for presentation purposes does not negatively impact the user.

Method used to compute the product of two matrices

The method involves finding the sum of the product of entries columns and rows. Generally, the k^{th} entry in the i^{th} row of the first matrix is multiplied to the k^{th} entry in the j^{th} column of the second matrix. 'k' ranges from 1 to the number of columns in the first matrix (or rows in the second matrix). The products are added to produce entry (i, j) for the resultant matrix.

6b Source file

Question6_MainClass.java

```java
import java.text.DecimalFormat;

public class Question6_MainClass
{
    final static int matrSize = 32; //Matrix size (matrSize x matrSize)

    public static void main(String args[])
    {
        /*Used for rounding values to 2 decimal places*/
        DecimalFormat decF = new DecimalFormat("0.00");

        /*Two matrices and the product matrix*/
        double matr1[][] = new double[matrSize][matrSize];
        double matr2[][] = new double[matrSize][matrSize];
```

```java
        double matr3[][] = new double[matrSize][matrSize];

        double sumOfProd; //stores intermediary products

        /*
        Fill the matrices with random values from 0 to 999
        rounded to two decimal places.
        */
        for(int i=0; i<matrSize; i++){
            for(int j=0; j<matrSize; j++){
                matr1[i][j] = Double.parseDouble(
                        decF.format(Math.random()*1000));
                matr2[i][j] = Double.parseDouble(
                        decF.format(Math.random()*1000));
            }
        }

        /*Print matrices with max field sizes 6*/
        System.out.println("Matrix 1: ");
        printMatrix(matr1, 6);
        System.out.println("Matrix 2: ");
        printMatrix(matr2, 6);

        /*The product calculation*/
        for(int i=0; i<matrSize; i++){
            for(int j=0; j<matrSize; j++){
                sumOfProd = 0.0;
                /*Traverse row k of matr1 and column k of
                matr2 and add products of terms traversed*/
                for(int k=0; k<matrSize; k++)
                    sumOfProd += matr1[i][k]*matr2[k][j];
                matr3[i][j] = sumOfProd;
            }
        }
        /*Print resultant matrix with minimum field size 11*/
        System.out.println("Resultant matrix: ");
        printMatrix(matr3, 11);
    }

    private static void printMatrix(double matrix[][], int fieldSize)
    {
        /*Format of values: a minimum field size and round to 2 d.p.*/
        String format = "%"+fieldSize+".2f ";

        /*Print top for matrix*/
        for(int i=0; i<((fieldSize+1)*matrSize)-1; i++)
            System.out.print("-");
        System.out.println();

        /*Print matrix values*/
        for(int i=0; i<matrSize; i++){
            for(int j=0; j<matrSize; j++)
                System.out.printf(format, matrix[i][j]);
            System.out.println(); //Skip a line for a new row
        }

        /*Print base for matrix*/
        for(int i=0; i<((fieldSize+1)*matrSize)-1; i++)
            System.out.print("-");
        System.out.println("\n");
    }
}
```

6c Testing

Since it is difficult to show by screenshots and workings that a product of a 32x32 matrix is correct, the testing will be done on a 3x3 matrix to confirm at least that the product algorithm is correct. An online matrix multiplication calculator [6] was later used to compare the results of a 32x32 matrix product. The way that the program outputs the matrices allows the matrices to be copied to the matrix product calculator on this website, allowing for comparison of results. Refer to Figure 14 and Table 14.

```
Matrix 1:
--------------------
 64.86 963.66 473.62
521.44 387.48 248.13
822.08 546.10 540.97
--------------------

Matrix 2:
--------------------
352.89 786.11 811.82
931.79 798.99 534.27
399.79 854.20 707.02
--------------------

Resultant matrix:
------------------------------------
 1110165.74  1225508.00   902368.09
  644260.84   931454.49   805767.23
 1015228.73  1544670.32  1341622.44
------------------------------------
```

Figure 14: Program output for a 3x3 matrix product

Size	Expected Outcome	Actual Outcome	Description
3x3	Product successful	Product successful	Reduced matrix size
32x32	Product successful	Product successful	Full matrix size

Table 14: Matrix sizes

The result in the screenshot can easily be partly confirmed by performing the necessary products for a few of the resultant matrix's entries:

- For entry (0,0):
  $(64.86 \times 352.89) + (963.66 \times 931.79) + (473.62 \times 399.79) = 1110165.7366$
- For entry (1,1):
  $(521.44 \times 786.11) + (387.48 \times 798.99) + (248.13 \times 854.20) = 931454.4896$
- For entry (2,2):
  $(822.08 \times 811.82) + (546.10 \times 534.27) + (540.97 \times 707.02) = 1341622.442$

This page intentionally left blank.

Task 7 Finding The Largest Number Recursively

7a Introduction

The code for this task consists of two methods within one source file. Below is a list showing the layout of the methods within the file.

- Question7_MainClass.java
  - public static void main(String args[])
  - private static int largest(int nums[], int upto)

The 'largest' method finds the largest value in the 'nums' array from the values found at the element at index 0 to the element at index 'upto'. When the method is initially invoked non-recursively, the second parameter is ideally the index of the last item so that the method checks all of the values for the largest value.

The value at index 'upto' is compared with the largest value found in the array up to index 'upto-1' recursively. In each recursive call, the largest of the two values (value at index 'upto' and largest value so far) is returned. When 'upto' is zero, the zeroth element need not be compared to any value, and so is the largest value, up to the zeroth element.

7b Source file

Question7_MainClass.java

```java
import java.util.Arrays;

public class Question7_MainClass
{
    public static void main(String args[])
    {
        /*Size of array from 10 to 20 items*/
        final int size = (int) (Math.random()*11)+10;
        final int nums[] = new int[size];

        /*Fill array with random integers from 0 to 9999*/
        for(int i=0; i<size; i++)
            nums[i] = (int) (Math.random()*10000);

        /*Print array and largest number*/
        System.out.println("Array of numbers: "
            + Arrays.toString(nums));
        System.out.println("Largest number: "
            + largest(nums, nums.length-1));
    }

    private static int largest(int nums[], int upto)
    {
        /*If list is empty, the largest cannot be found*/
        if(nums.length == 0){
            System.out.println("Error: array size is zero");
```

```
            return -1;
        }

        /*Largest up to index 0 is the first value, nums[0]*/
        if(upto == 0)
            return nums[0];
        else{
            /*Largest so far is the largest of the array tail*/
            int soFar = largest(nums, upto-1);

            /*
            If current value larger than largest value so far,
            then current value is new largest value so far.
            */
            if(nums[upto] > soFar)
                return nums[upto]; //current value is new largest
            else
                return soFar; //largest so far remained largest
        }
    }
}
```

7c Testing

Since there is no input by the user, the testing will focus on the range of random numbers and the size of the list of numbers. The default range of random numbers is 0 to 9999 while the default size of the list of numbers is between 10 and 20 items. These will be changed for each test case.

For valid results, only ranges of numbers and array sizes expected to be valid will be tested. There will be no invalid except for an array of size zero in the second set of test cases.

Various Random Number Ranges

For random number ranges, various positive and negative ranges were tested out, all of which produced the expected outcomes. Each test case in used the default random array size range of 10 to 20 values. Refer to Figure 15 and Table 15.

Number Range	Expected Outcome	Actual Outcome	Description
0 to 0	Algorithm succeeded	Algorithm succeeded	Zero range
0 to 9	Algorithm succeeded	Algorithm succeeded	Small positive range
0 to 999	Algorithm succeeded	Algorithm succeeded	Small positive range
0 to 99999	Algorithm succeeded	Algorithm succeeded	Large positive range
-9 to 0	Algorithm succeeded	Algorithm succeeded	Small negative range
-999 to 0	Algorithm succeeded	Algorithm succeeded	Small negative range
-99999 to 0	Algorithm succeeded	Algorithm succeeded	Large negative range

Table 15: Various Random Number Ranges

```
Array of numbers: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Largest number: 0
```

(i)

```
Array of numbers: [4, 1, 1, 3, 6, 2, 0, 7, 6, 0, 3, 3]
Largest number: 7
```

(ii)

```
Array of numbers: [528, 34, 215, 432, 615, 867, 791, 397, 822, 99, 636, 842, 306]
Largest number: 867
```

(iii)

```
Array of numbers: [-630, -772, -344, -873, -378, -551, -284, -514, -929, -424]
Largest number: -284
```

(iv)

Figure 15: Program output for (i) a 0 to 0 range, (ii) a 0 to 9 range, (iii) a 0 to 999 range, and (iv) a -999 to 0 range

Various List Sizes

For list size ranges, various ranges were tested out, all of which produced the expected outcomes. Every test case used the default random number range. Refer to Figure 16 and Table 16.

```
Array of numbers: []
Error: array size is zero
Largest number: -1
```

(i)

```
Array of numbers: [5882, 3442, 770, 6766, 4652, 7822, 7006]
Largest number: 7822
```

(ii)

```
Array of numbers: [9022, 5383, 5824, 8978, 1229, 4809, 2152, 3522, 9805, 8340]
Largest number: 9805
```

(iii)

Figure 16: Program output for (i) a 0 to 0 list size, (ii) a 1 to 10 list size, and (iii) a 10 to 20 list size

Size of list	Expected Outcome	Actual Outcome	Description
0 to 0	The error message	The error message	Zero size
1 to 10	Algorithm succeeded	Algorithm succeeded	Small size
1 to 100	Algorithm succeeded	Algorithm succeeded	Larger size
10 to 20	Algorithm succeeded	Algorithm succeeded	Small size
20 to 30	Algorithm succeeded	Algorithm succeeded	Small size

Table 16: Various List Sizes

Task 8 Computing Cosine and Sine By Series Expansions

8a Introduction

The code for this task consists of three methods within one source file. Below is a list showing the layout of the methods within the file.

- Question8_MainClass.java
  - public static void main(String args[])
  - private static String compCorS(double xRad, int n, String type)
  - private static BigDecimal bigDec(double i)

In the 'main' method, the user is asked for three inputs, one of which is the angle in degrees. This angle is reduced to its lowest form (i.e. $x\%360$), converted to radians, and passed to the 'compCorS' method along with the other two inputs. The 'compCorS' method computes the cosine or sine of an input 'xRad' by taking the first 'n' terms of the Maclaurin series for cosine and sine [7], respectively, which are defined as follows:

$$cos(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n)!} x^{2n} = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - ...$$
$$sin(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - ...$$

The calculation performed depends on the parameter 'type', which is expected to be either 'cos' or 'sin'. The initial numerator and denominator are common for both calculation types. In the main loop of each calculation type, the numerator and denominator are first set and are divided to produce a new term which is added to the result. Due to the denominator being made up of a factorial, it is set to the product of itself with specific values to produce a factorial effect.

A list of assumptions made for this task is found below.

1. Rounding the workings to eleven decimal places and the result to ten decimal places does not negatively impact the user.

8b Source file

Question8_MainClass.java

```java
import java.util.Scanner;
import java.util.InputMismatchException;
import java.math.BigDecimal;
import java.math.RoundingMode;

public class Question8_MainClass
{
    public static void main(String args[])
    {
        Scanner sc = new Scanner(System.in); //handles input
        double xDeg, xRad; //store input in degrees and radians
        int terms;   //stores input amount of terms
        String type; //shows if cos or sine will be computed

        /*Loop until input starts with 'q' (indicating a quit)*/
        do{
            try{
                System.out.print("Insert angle in degrees (q to quit): ");
                xDeg = sc.nextDouble(); //input angle in degrees
                System.out.print("Insert amount of terms (q to quit): ");
                terms = sc.nextInt(); //input amount of terms
            }catch(InputMismatchException ime){
                /*If input starts with 'q', then quit*/
                if(sc.next().startsWith("q"))
                    break;
                else{
                    System.out.println("Invalid input.\n");
                    continue; //go to start of loop
                }
            }

            if(terms < 1){ //if amount of terms less than 1, invalid
                System.out.println("Invalid amount of terms.\n");
                continue; //go to start of loop
            }

            System.out.print("Insert COS for cosine or SIN for sine: ");
            type = sc.next(); //input of calculation type
            /*If type is neither COS nor SIN*/
            if(!type.toLowerCase().equals("cos")
                && !type.toLowerCase().equals("sin")){
                System.out.println("Invalid calculation type.\n");
                continue; //go to start of loop
            }

            /*Convert xDeg to radians and store in xRad*/
            xRad = ((xDeg%360)/180)*Math.PI;

            /*Compute and output*/
            if(type.toLowerCase().equals("cos"))
                System.out.println("cos("+xDeg+") = " + compCorS(xRad,terms,"cos") + "\n");
            else if(type.toLowerCase().equals("sin"))
                System.out.println("sin("+xDeg+") = " + compCorS(xRad,terms,"sin") + "\n");
        }while(true);
        sc.close(); //close scanner
```

```java
        System.out.println("Program terminated."); //final output
    }

    private static String compCorS(double xRad, int n, String type)
    {
        BigDecimal result = null;     //stores result
        BigDecimal numer = bigDec(0); //stores numerator
        BigDecimal denom = bigDec(1); //stores denominator
        BigDecimal x = bigDec(xRad);  //stores input (in radians)

        /*Compute cosine*/
        if(type.equals("cos"))
        {
            result = bigDec(1); //first term is '1'
            for(int i=1; i<n; i++){
                /*Computing numerator of term*/
                numer = x.pow(2*i).multiply(bigDec((Math.pow(-1,i))));
                /*Computing denominator of term*/
                denom = denom.multiply(bigDec( ((2*i)-1)*(2*i)) );
                /*Dividing numerator by denominator and rounding to 11dp*/
                result = result.add(
                  numer.divide(denom,11,RoundingMode.HALF_UP));
            }
        }
        /*Compute sine*/
        else if(type.equals("sin"))
        {
            result = bigDec(xRad); //first term is 'x' (in radians)
            for(int i=1; i<n; i++){
                /*Computing numerator and denominator of term*/
                numer = x.pow((2*i)+1).multiply(bigDec(Math.pow(-1,i)));
                /*Computing denominator of term*/
                denom = denom.multiply(bigDec( (2*i)*((2*i)+1) ));
                /*Dividing numerator by denominator and rounding to 11dp*/
                result = result.add(
                  numer.divide(denom,11,RoundingMode.HALF_UP));
            }
        }

        /*If type is a valid type, finalize and return result*/
        if(type.equals("cos") || type.equals("sin")){
            /*Round result to 10 decimal places and return result*/
            result = result.setScale(10, RoundingMode.HALF_UP);
            return result.toString();
        }
        else{
         System.out.println("Error: Invalid calculation type.");
         return null;
        }
    }

    private static BigDecimal bigDec(final double i)
    {
        /*Returns a BigDecimal of value 'i'*/
        return new BigDecimal(i);
    }
}
```

8c Testing

For this section, valid, invalid, and other test data were input to the program. For each type of test data, a screenshot showing some of the results and a table showing all tested results are provided. When the input shows three values, these correspond to (a) the input for cosine or sine, (b) the number of terms ‘n’, and (c) the input indicating whether cosine or sine will be computed.

Valid Data

As valid data, various integer and floating-point value inputs were tested both with cosine and sine as the type of calculation and using 100 terms in many of the tests to produce very accurate results which can easily be verified. Test cases with terms set to 1 were meant to test the first term, which for cosine is 1, and for sine is the input angle itself (which had been converted to radians). Test cases with terms set to 3 were meant to test the first three terms. The expected outcomes were obtained using a calculator. The third input was also tested as uppercase and lowercase. Refer to Figure 17 and Table 17.

```
Insert angle in degrees (q to quit): 0
Insert amount of terms (q to quit): 100
Insert COS for cosine or SIN for sine: COS
cos(0.0) = 1.0000000000

Insert angle in degrees (q to quit): 180
Insert amount of terms (q to quit): 100
Insert COS for cosine or SIN for sine: sin
sin(180.0) = 0E-10

Insert angle in degrees (q to quit): -360
Insert amount of terms (q to quit): 100
Insert COS for cosine or SIN for sine: COS
cos(-360.0) = 1.0000000000

Insert angle in degrees (q to quit): 12.34
Insert amount of terms (q to quit): 100
Insert COS for cosine or SIN for sine: sin
sin(12.34) = 0.2137124408

Insert angle in degrees (q to quit): 1234
Insert amount of terms (q to quit): 1
Insert COS for cosine or SIN for sine: COS
cos(1234.0) = 1.0000000000

Insert angle in degrees (q to quit): 789.456
Insert amount of terms (q to quit): 3
Insert COS for cosine or SIN for sine: sin
sin(789.456) = 0.9371508731

Insert angle in degrees (q to quit):
```

Figure 17: Program output for valid data

Input (a,b,c)	Expected Outcome	Actual Outcome	Description
0, 100, COS	1	1.0000000000	Basic angle
0, 100, SIN	0	0E-10	Basic angle
90, 100, cos	0	0E-10	Basic angle
90, 100, sin	1	1.0000000000	Basic angle
180, 100, COS	-1	-1.0000000000	Basic angle
180, 100, SIN	0	0E-10	Basic angle
270, 100, cos	0	0E-10	Basic angle
270, 100, sin	-1	-1.0000000000	Basic angle
-360, 100, COS	1	1.0000000000	Basic negative angle
-360, 100, SIN	0	0E-10	Basic negative angle
12.34, 100, cos	0.9768966131	0.9768966131	Complex angle
12.34, 100, sin	0.2137124408	0.2137124408	Complex angle
-21.43, 100, COS	0.9308646392	0.9308646392	Complex negative angle
-21.43, 100, SIN	-0.365364234	-0.3653642340	Complex negative angle
1234, 1, cos	1	1.0000000000	Large angle, one term
1234, 1, sin	2.6878070481	2.6878070481	Large angle, one term
789.456, 3, COS	0.3552202796	0.3552202796	Complex angle, few terms
789.456, 3, SIN	0.9371508731	0.9371508731	Complex angle, few terms

Table 17: Valid Data

Invalid Data

As invalid data, forbidden characters, negative inputs, floating-point value inputs, and invalid calculation types were tested. Where the angle and/or amount of terms inputs were not being tested, valid value ‘100’ was used as a dummy value. Note that ‘cosine’ and ‘sine’ are invalid type calculations since the program is actually expecting ‘cos’, ‘COS’, ‘sin’, or ‘SIN’. The program handled all of the invalid inputs and did not crash. Refer to Figure 18 and Table 18.

Input	Expected Outcome	Actual Outcome	Description
abc	Invalid Input	Invalid Input	Non-number first input
100, abc	Invalid Input	Invalid Input	Non-number second input
100, -1	Invalid Input	Invalid Input	Invalid amount of terms
100, 12.34	Invalid Input	Invalid Input	Invalid amount of terms
100, 100, cosine	Invalid Input	Invalid Input	Invalid type of calculation
100, 100, sine	Invalid Input	Invalid Input	Invalid type of calculation
100, 100, -1	Invalid Input	Invalid Input	Invalid type of calculation
A B C	Invalid Input	Invalid Input x3	Multiple non-number inputs

Table 18: Invalid Data

```
Insert angle in degrees (q to quit): abc
Invalid input.

Insert angle in degrees (q to quit): 100
Insert amount of terms (q to quit): abc
Invalid input.

Insert angle in degrees (q to quit): 100
Insert amount of terms (q to quit): -1
Invalid amount of terms.

Insert angle in degrees (q to quit): 100
Insert amount of terms (q to quit): 12.34
Invalid input.

Insert angle in degrees (q to quit): 100
Insert amount of terms (q to quit): 100
Insert COS for cosine or SIN for sine: cosine
Invalid calculation type.

Insert angle in degrees (q to quit): A B C
Invalid input.

Insert angle in degrees (q to quit): Invalid input.

Insert angle in degrees (q to quit): Invalid input.

Insert angle in degrees (q to quit):
```

Figure 18: Program output for invalid data

Other Data

As other data, the inputs ‘q’ and ‘quit’ were tested out to check that the program terminates when the input starts with the letter ‘q’. Spaces as an input and a blank input were also tested. Refer to Figure 19 and Table 19.

```
Insert angle in degrees (q to quit): q
Program terminated.
```

```
Insert angle in degrees (q to quit): quit
Program terminated.
```

Figure 19: Program output for other data

Input	Expected Outcome	Actual Outcome	Description
q	Program quits	Program quit	Letter ’q’
quit	Program quits	Program quit	Word starting with ’q’
SPACE	No output	No output	Spaces
BLANK	No output	No output	Blank input

Table 19: Other Data

Task 9 Sum of Fibonacci Sequence Terms

9a Introduction

The code for this task consists of two methods within one source file. Below is a list showing the layout of the methods within the file.

- Question9_MainClass.java
  - public static void main(String args[])
  - private static String fibSum(final int n)

The 'fibSum' method computes the sum of the first 'n' numbers of the Fibonacci sequence. By considering the fact that a Fibonacci term is the sum of the two previous Fibonacci terms, it may be shown that the sum of the first 'n' Fibonacci numbers is equal to the (n+2)'nd term in the Fibonacci sequence, minus one [8]:

$$\sum_{i=1}^{n} F_i = F_{n+2} - 1$$

The 'fibSum' makes use of this equation to implement a much more efficient method to find the sum of the first 'n' numbers. It was assumed that the Fibonacci Sequence starts from '1' as the first term as follows: '1, 1, 2, 3, 5,...' and not from '0' like so: '0, 1, 1, 2, 3, 5,...'.

9b Source file

Question9_MainClass.java

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

public class Question9_MainClass
{
    public static void main(String args[])
    {
        Scanner sc = new Scanner(System.in); //handles input
        int terms=1; //stores input amount of terms

        /*Loop until terms is zero or negative*/
        do{
            try{
                /*Input of amount of terms*/
                System.out.print("Insert a number (0 or -ve to quit): ");
                terms = sc.nextInt();
            }catch(InputMismatchException ime){
                System.out.println("Invalid input.\n");
                sc.next(); //clear invalid input
                continue; //go to start of loop
            }

```

```
            /*If terms > 0, calculate and output sum of terms*/
            if(terms > 0)
                System.out.println("Sum is: "+fibSum(terms)+"\n");
        }while(terms > 0);

        System.out.println("Program terminated."); //final output
        sc.close(); //close scanner
    }

    private static String fibSum(final int n)
    {
        BigInteger fib0 = BigInteger.ZERO; //zeroth term
        BigInteger fib1 = BigInteger.ONE;  //first term
        BigInteger temp; //temporarily stores value of fib1

        /*Compute up to the (n+2)nd term*/
        for(int i=2; i <= n+2; i++){
            temp = fib1;           //store current fib1
            fib1 = fib1.add(fib0); //calculate next term
            fib0 = temp;           //set fib0 to previous value of fib1
            /*
            fib1 is now the ith     Fibonacci sequence term.
            fib0 is now the (i-1)th Fibonacci sequence term.
            */
        }
        /*Subtract 1 from the (n+2)nd term and return in string form*/
        return fib1.subtract(BigInteger.ONE).toString();
    }
}
```

9c Testing

For this section, valid, invalid, and other test data were input to the program. For each type of test data, a screenshot showing some of the results and a table showing all tested results are provided. The expected outcomes in the valid values table were produced using an online Fibonacci calculator [9].

Valid Data

As valid data, integers of various sizes were tested. Due to the large length of the outputs, many of the outcomes had to be split across rows in the table of test data. Refer to Figure 20 and Table 20.

```
Insert a number (0 or -ve to quit): 1
Sum is: 1

Insert a number (0 or -ve to quit): 3
Sum is: 4

Insert a number (0 or -ve to quit): 22
Sum is: 46367

Insert a number (0 or -ve to quit): 111
Sum is: 184551825793033096366332

Insert a number (0 or -ve to quit): 333
Sum is: 4585371016945309254695820765383405232040578837489482816555438621189664

Insert a number (0 or -ve to quit): 1000
Sum is: 113796925398360272257523782552224175572745930353730513145086634176691092
53614598547014612933464186690278367304232208862586339605288869009696957717369637
0562180400527049497109023054114771394568040040412172632375

Insert a number (0 or -ve to quit): 1500
Sum is: 354773075861193433583230405734583243362729730700963205617431108366332244
14869683457397230159251404953822275198639462583960142202354300086972011354285518
39037349968103851458646395904786416432161218107155817249192173666320029265280982
13617099770855840584949758791422553405448102377189832152788363582013186190348470
00

Insert a number (0 or -ve to quit):
```

Figure 20: Program output for valid data

Input	Expected Outcome	Actual Outcome
1	1	1
2	2	2
3	4	4
11	232	232
22	46367	46367
33	9227464	9227464
111	184551825793033096366332	184551825793033096366332
222	290901803555033622569101 ...11038089984964854261892	290901803555033622569101 ...11038089984964854261892
333	458537101694530925469582 ...07653834052320405788374 ...894828165555438621189664	458537101694530925469582 ...07653834052320405788374 ...894828165555438621189664

1000	11379692539836027225752 ...3782552224175572745930 3 ...53730513145086634176691 ...09253614598547014612933 ...46418669027836730423220 ...88625863396052888690096 ...96957717369637056218040 ...05270494971090230541147 ...71394568040040412172632 ...375	11379692539836027225752 ...37825522241755727459303 ...53730513145086634176691 ...09253614598547014612933 ...46418669027836730423220 ...88625863396052888690096 ...96957717369637056218040 ...05270494971090230541147 ...71394568040040412172632 ...375

Table 20: Valid Data

Invalid Data

As invalid data, floating-point value input and inputs consisting of illegal characters were tested. All invalid inputs tested were handled by the program, preventing a crash. Refer to Figure 21 and Table 21.

```
Insert a number (0 or -ve to quit): 12.34
Invalid input.

Insert a number (0 or -ve to quit): abc
Invalid input.

Insert a number (0 or -ve to quit): A B C
Invalid input.

Insert a number (0 or -ve to quit): Invalid input.

Insert a number (0 or -ve to quit): Invalid input.

Insert a number (0 or -ve to quit):
```

Figure 21: Program output for invalid data

Input	Expected Outcome	Actual Outcome	Description
12.34	Invalid input	Invalid input	Floating point value
abc	Invalid input	Invalid input	Illegal characters
A B C	Invalid input	Invalid input x3	Illegal characters

Table 21: Invalid Data

Other Data

As other data, spaces as inputs, a blank input, and a negative number and zero (both indicating that the user wants to quit) were all tested. Refer to Figure 22 and Table 22.

```
Insert a number (0 or -ve to quit): -1
Program terminated.
```

```
Insert a number (0 or -ve to quit): 0
Program terminated.
```

Figure 22: Program output for other data

Input	Expected Outcome	Actual Outcome	Description
-1	Program quits	Program quit	Negative number
0	Program quits	Program quit	Zero
SPACE	No output	No output	Spaces
BLANK	No output	No output	Blank input

Table 22: Other Data

Statement of Completion

The list below indicates which questions were attempted and completed:

Question 1: This question was successfully completed.

Question 2: This question was successfully completed.

Question 3: This question was successfully completed.

Question 4: This question was successfully completed.

Question 5: This question was successfully completed.

Question 6: This question was successfully completed.

Question 7: This question was successfully completed.

Question 8: This question was successfully completed.

Question 9: This question was successfully completed.

Signature

Date

Reference List

[1] Google. Available: www.google.com

[2] A Primality Test. Available:
https://primes.utm.edu/curios/includes/primetest.php

[3] The nth Prime Page. Available:
https://primes.utm.edu/nthprime/index.php

[4] E. W. Weisstein. Newton's Method. Available:
http://mathworld.wolfram.com/NewtonsMethod.html

[5] Square Root Calculator. Available:
http://www.math.com/students/calculators/source/square-root.htm

[6] Online Matrix Multiplication. Available:
http://www.bluebit.gr/matrix-calculator/matrix_multiplication.aspx

[7] E. W. Weisstein. Maclaurin Series. Available:
http://mathworld.wolfram.com/MaclaurinSeries.html

[8] Combinatorial identities. Available:
https://en.wikipedia.org/wiki/Fibonacci_number#Combinatorial_identities

[9] R. Knott. Fibonacci and Lucas Number Calculator 1.3. Available:
http://www.maths.surrey.ac.uk/hosted-sites/R.Knott/Fibonacci/fibCalcX.html