Computer Project

(2017-2019)

Satvik Saha

Class: XII B

Roll number: 34

"Writing code a computer can understand is science. Writing code other programmers can understand is an art."

— **Jason Gorman**

"If Java had true garbage collection, most programs would delete themselves upon execution."

— **Robert Sewell**

Problem 17 The classical *Möbius function* $\mu(n)$ is an important function in number theory and combinatorics. For positive integers n, $\mu(n)$ is defined as the sum of the primitive n^{th} roots of unity. It attains the following values.

$\mu(1) = +1$
$\mu(n) = -1$ if n is a square-free positive integer with an odd number of prime factors.
$\mu(n) = \ \ 0$ if n has a squared prime factor.
$\mu(n) = +1$ if n is a square-free positive integer with an even number of prime factors.

Compute the $\mu(n)$ for positive integers n within a specified range.

Solution For any given $n \in \mathbb{N}$, all we have to do is search for factors by trial-division, and find their multiplicity. If this is greater than 1, we can stop here since we have found squared prime factors. Otherwise, we can reduce the problem by dividing out these factors from n and repeating. By trying factors in ascending order and then discarding them from n, we are guaranteed to hit only prime factors, and can thus skip primality checks.

`main (lo:Integer, hi:Integer)`

1. Assert that the integers in the range `[lo, hi)` are all positive.
2. For each `i` $\in \{$`lo`$, $`lo`$ + 1, \ldots,$ `hi`$ - 1\}$:
   (a) Call and display `mobius(i)`.
3. **Exit**

`mobius (n:Integer)`

1. If `n` is one, **return** `1`.
2. Initialize an integer variable `mob` to one.
3. For `i` $\in \{2, 3, \ldots, n\}$:
   (a) Initialize an integer `multiplicity` to zero.
   (b) While `i` divides `n`, assign `n / i` to `n` and increment `multiplicity`.
   (c) If `multiplicity` is one, flip the sign of `mob`.
   (d) If `multiplicity` is greater than one, **return** `0`.
4. **Return** `mob`

Source Code

```
public class Mobius {
	public static final String[] graph =
		{"*       ",
		 "   *    ",
		 "      *"};
	public static void main (String[] args) {
		try {
			int lo = Integer.parseInt(args[0]);
			int hi = Integer.parseInt(args[1]);
			if (lo < 1 || hi <= lo)
				throw new NumberFormatException();
			for (int i = lo; i < hi; i++) {
				int m = mobius(i);
				System.out.printf(" (%d)\t\t = %2d%24s\n", i, m, graph[m
					+ 1]);
			}
		} catch (NumberFormatException | IndexOutOfBoundsException e) {
			System.out.println("Enter 2 arguments (lower_limit[integer,
				>0], upper_limit[integer, >lower_limit])!");
		}
	}

	public static int mobius (int n) {
		if (n < 1)
			return 0;
		if (n == 1)
			return 1;
		int mob = 1;
		for (int i = 2; i <= n; i++) {
			int multiplicity = 0;
			while ((n % i) == 0) {
				n /= i;
				multiplicity++;
			}
			if (multiplicity == 1) {
				mob = -mob;
			} else if (multiplicity > 1) {
				return 0;
			}
		}
		return mob;
	}
}
```

Variable Description

`Mobius::main(String[])`		
`int`	`lo`	Lower bound of integers to evalute
`int`	`hi`	Upper bound of integers to evalute
`int`	`i`	Counter variable, stores the integer to be evaluated

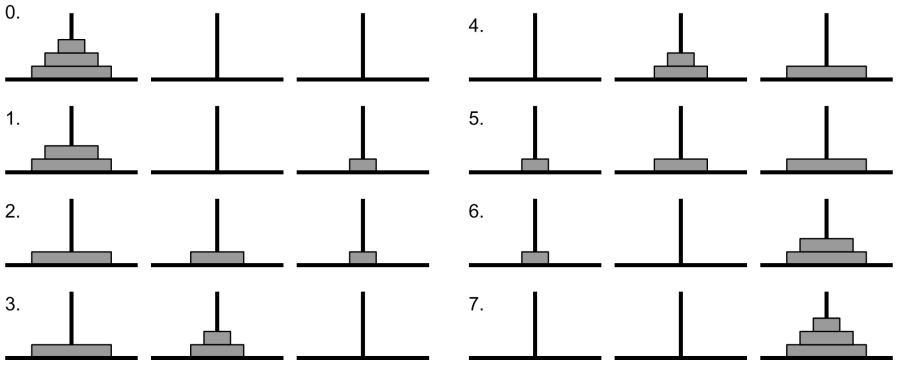
`Mobius::mobius(int)`		
`int`	`n`	The number where the mobius function is to be evaluated
`int`	`mob`	Sign of the value of the mobius function
`int`	`i`	Counter variable, stores the current factor to be tested
`int`	`multiplicity`	The power of `i` in the factorisation of `n`

"In order to understand recursion, one must first understand recursion."

— **Anonymous**

Problem 18 The *Tower of Hanoi* is a mathematical puzzle, consisting of three rods and a number of disks of different sizes which can slide onto any rod. The puzzle starts with all disks, in ascending order of size, on one rod. The objective of the puzzle is to move the entire stack to another rod, obeying the foolowing rules.

1. Only one disk can be moved at a time.
2. Each move consists of taking the upper disk from one stack and placing it on the top of another stack or empty rod.
3. No disk can be placed on a smaller disk.



Solution to the Towers of Hanoi with 3 disks.

Solve the *Tower of Hanoi* puzzle for an arbitrary number of disks, enumerating the required moves.

Solution The main insight here is that the problem involving n disks can be reduced to one with $n-1$ disks. Labelling the rods A, B and C, and the disks with numerals 1 through n (smallest to largest), our aim is to move the entire stack from A to C. If we can solve the problem with $n-1$ disks, all we have to do is to move the topmost $n-1$ disks from A to B, move the remaining disk on A to C, and again move the $n-1$ disks on B to C. The base case for this recursive solution is moving 1 disk, which is trivial.

Clearly, if the problem with n disks takes k_n number of moves, the problem with $n+1$ moves will take $k_n + 1 + k_n = 2k_n + 1$ moves. For the base case with one disk,

$k_1 = 1$. With this infromation, we see that the *Tower of Hanoi* with n disks can be solved in exactly $2^n - 1$ moves.

`main (disks:Integer)`

1. Call `solveHanoi(disks, "A", "C", "B")`.
2. **Exit**

`solveHanoi (disk:Integer, source:String, destination:String, spare:String)`

1. If `disk` is zero, **return**.
2. Call `solveHanoi(disk - 1, source, spare, destination)`.
3. Move disk number `disk` has to be moved from `source` to `destination`.
4. Call `solveHanoi(disk - 1, spare, destination, source)`.
5. **Return**

Source Code

```
1  public class TowersOfHanoi {
2        public static void main (String[] args) {
3              try {
4                     int disks = Integer.parseInt(args[0]);
5                     if (disks < 1)
6                           throw new NumberFormatException();
7                     solveHanoi(disks, "A", "C", "B");
8              } catch (NumberFormatException | IndexOutOfBoundsException e) {
9                     System.out.println("Enter 1 argument
                          (number_of_disks[integer])!");
10             }
11       }
12
13       public static void solveHanoi (int disk, String source, String destination,
             String spare) {
14             if (disk == 0)
15                    return;
16             solveHanoi(disk - 1, source, spare, destination);
17             System.out.printf("(%d) : %s -> %s%n", disk, source, destination);
18             solveHanoi(disk - 1, spare, destination, source);
19       }
20 }
```

Variable Description

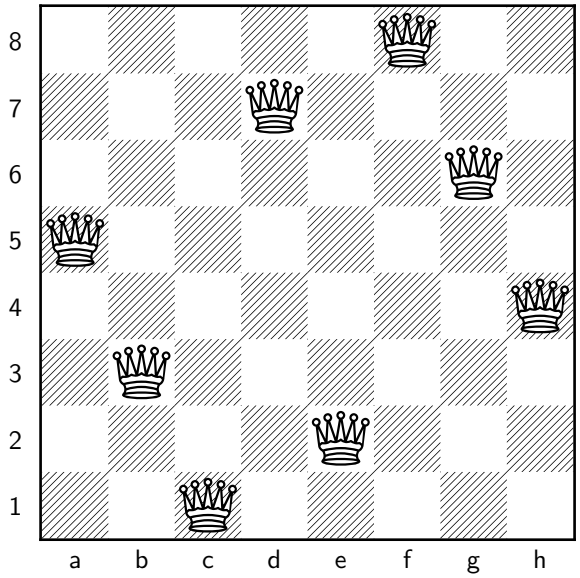
`TowersOfHanoi::main(String[])`		
`int`	`disks`	The number of disks in the problem

`TowersOfHanoi::solveHanoi(int, String, String, String)`		
`int`	`disk`	The current disk to be moved
`String`	`source`	The rod from which the stack is to be moved
`String`	`destination`	The rod to which the stack is to be moved
`String`	`spare`	The additional rod, where the remaining `n-1` disks are temporarily moved

"Chess is the gymnasium of the mind."

— Blaise Pascal

Problem 19 The *8 queens puzzle* involves placing 8 queens on an 8×8 chessboard such that no two queens threaten each other, i.e. no two queens share the same rank, file or diagonal. It was first published by the chess composer *Max Bezzel* in 1848. This puzzle has 92 solutions, including reflections and rotations. Below is one of them.



The *n queens puzzle* is an extension of this puzzle, involving n queens on an $n \times n$ chessboard. Count the total number of solutions for the *n queens puzzle*, including reflections and rotations.

Solution This problem can be solved with *recursion* and *backtracking*. Starting from the topmost row of the chessboard, we can place a queen and for each available choice, place a queen on the next row, and so on, recursively shrinking the chessboard to solve. Invalid solutions can thus be discarded as they are formed without brute-forcing every possible permutation of queens on the board.

Finally, by noting that exactly one queen must occupy each row, we can optimize the board by storing only the column numbers of queens on each row in an array, instead of simulating a full 2D board.

`main (size:Integer, drawSolutions:Boolean)`

1. Create an `NQueens` object by passing it `size` and `drawSolutions`. Call it `q`.
2. Call `q->countSolutions()` and display the result.
3. **Exit**

`NQueens (size:Integer, drawSolutions:Boolean)`

1. Copy `size` and `drawSolutions` into the object data.
2. Initialize an integer `numberOfSolutions` to zero.
3. Initialize an array of integers with length `size`. Call it `board`.
4. **Define** the functions:
   (a) `NQueens::countSolutions()`
   (b) `NQueens::solveNQueens(row)`
   (c) `NQueens::isThreatened(row)`
5. **Return** the resultant object.

`NQueens::countSolutions ()`

1. Call `this->solveNQueens(0)`.
2. **Return**

`NQueens::solveNQueens (row:Integer)`

1. If `row` is equal to `size`:
   (a) Increment `numberOfSolutions`.
   (b) If `drawSolutions` is set to `true`, display the current state of `board`.
   (c) **Return**
2. For each `i` $\in \{0, 1, \ldots,$ `size` $- 1\}$:
   (a) Place a queen at row `row`, column `i`, i.e. set `board[row]` to `i`.
   (b) Call `this->isThreatened(row)`. If this returns `false`, call `this->solveNQueens(row + 1)`.
3. **Return**

`NQueens::isThreatened (row:Integer)`

1. For each `i` $\in \{0, 1, \ldots,$ `size` $- 1\}$:
   (a) If there are two queens on the same column in rows `row` and `i`, or the columns in which those two queens are on are on the same diagonal, **return** `true`.
2. **Return** `false`

Source Code

```
public class NQueens {
	private final int size;
	private int[] board;
	private int numberOfSolutions;
	private final boolean drawSolutions;

	public NQueens (int size, boolean drawSolutions) {
		this.size = size;
		this.drawSolutions = drawSolutions;
		this.initBoard();
	}

	public int countSolutions () {
		solveNQueens(0);
		return numberOfSolutions;
	}

	private void initBoard () {
		this.board = new int[size];
		this.numberOfSolutions = 0;
		for (int i = 0; i < size; i++)
			board[i] = -1;
	}

	private boolean isThreatened (int row) {
		for (int i = 0; i < row; i++) {
			if ((board[row] == board[i])
			    || ((board[row] - board[i]) == (row - i))
			    || ((board[row] - board[i]) == (i - row))) {
				return true;
			}
		}
		return false;
	}

	private void solveNQueens (int row) {
		if (row == size) {
			numberOfSolutions++;
			if (drawSolutions) {
				drawBoard();
				System.out.println();
			}
			return;
		}
```

```
45                for (board[row] = 0; board[row] < size; board[row]++) {
46                      if (!isThreatened(row)) {
47                            solveNQueens(row + 1);
48                      }
49                }
50          }
51
52          public void drawBoard () {
53                for (int i = 0; i < size; i++) {
54                      for (int j = 0; j < size; j++) {
55                            System.out.print(((board[i] == j)? "Q" : "-") + " ");
56                      }
57                      System.out.println();
58                }
59          }
60
61          public static void main (String[] args) {
62                try {
63                      int size = Integer.parseInt(args[0]);
64                      boolean drawSolutions = (args.length > 1)?
                            Boolean.parseBoolean(args[1]) : false;
65                      if (size < 1)
66                            throw new NumberFormatException();
67
68                      NQueens q = new NQueens(size, drawSolutions);
69                      System.out.println(q.countSolutions());
70                } catch (NumberFormatException | IndexOutOfBoundsException e) {
71                      System.out.println("Enter at least 1 argument
                            (size_of_board[integer], <show_solutions>[true/false])!");
72                      System.out.println("(show_solutions defaults to false)");
73                }
74          }
75    }
```

Variable Description

NQueens		
int	size	The number of rows and columns in the chessboard
int[]	board	The list of positions of queens in columns, with their rows corresponding to their index.
int	numberOfSolutions	Counts the number of solutions found
boolean	drawSolutions	Stores whether to display solved boards or not

NQueens::isThreatened(int)		
int	row	The row of the queen to test
int	i	Counter variable, stores the row of the queen to test against

NQueens::solveNQueens(int)		
int	row	The current row on which a queen is to be placed

NQueens::drawBoard()		
int	i, j	Counter variables, store the row and column to be currently displayed

NQueens::main(String[])		
int	size	The number of rows and columns in the chessboard
boolean	drawSolutions	Stores whether to display solved boards or not
NQueens	q	Object capable of solving the *n queens* problem

"Computers are useless. They can only give you answers."

— Pablo Picasso

Problem 20 *Reverse Polish Notation (RPN)* or *postfix notation* is a mathematical notation for writing arithmetic expresssions in which operators follow their operands. Thus, as long as each operator has a fixed number of operands, the use of parentheses or rules of precedence are no longer required to write unambiguous expressions. For example, the expression `2 3 * 3 2 ^ 2 - *` evaluates to `42`.

Create a program capable of evaluating *RPN* expressions which use the following operators.

`+`	Addition
`-`	Subtraction
`*`	Multiplication
`/`	Division
`^`	Exponentiation

Solution The nature of *RPN* lends itself to a very simple implementation with a stack for pushing operands into as they appear in an expression. When an operator is encountered, the required number of operands are popped from the stack, the operation is carried out, and the result is popped back into the stack. This continued until the entire expression has been parsed, leaving only the evaluated result in the stack.

`main (expression:String)`

1. Call `evaluateRPNExpression(expression)` and display the returned value.
2. **Exit**

`evaluateRPNExpression (expression:String)`

1. Split `expression` along whitespace into an array of tokens. Call it `tokens`.
2. Create a stack of floating points large enough to hold all elements in `tokens`. Call it `operandStack`.
3. For each string `token` $\in$ `tokens`:
   (a) If `token` is a floating point:
      i. Push `token` onto `operandStack`.
      ii. Get the next `token` from `tokens`.
      iii. Jump back to (3a).
   (b) Pop an operand from `operandStack` and call it `rightOperand`.
   (c) Pop another operand from `operandStack` and call it `leftOperand`.

(d) Depending on which operator `token` represents, evaluate the operation with `token` as the operator and `leftOperand` and `rightOperand` as the respective operands. Call it `result`.

(e) Push `result` onto `operandStack`.

4. Pop and operand from `operandStack` and **return** it.

Source Code

```
1  import java.util.Scanner;
2
3  public class RPNCalculator {
4         private static double[] operandStack;
5         private static int top;
6
7         public static void main (String[] args) {
8                 System.out.printf("Reverse Polish Expression : ");
9                 String expression = (new Scanner(System.in)).nextLine();
10                double result = evaluateRPNExpression(expression);
11                System.out.printf("Evaluated Expression  :   %s %n",
                      Double.toString(result));
12        }
13
14        public static double evaluateRPNExpression (String expression) {
15                String[] tokens = expression.split("\\s+");
16                top = -1;
17                operandStack = new double[tokens.length];
18
19                for (String token : tokens) {
20                      if (isDouble(token)) {
21                            pushOperand(Double.parseDouble(token));
22                            continue;
23                      }
24
25                      double rightOperand = popOperand();
26                      double leftOperand = popOperand();
27                      double result = 0.0;
28                      switch (token.charAt(0)) {
29                            case '+' :    result = leftOperand + rightOperand;
30                                          break;
31                            case '-' :    result = leftOperand - rightOperand;
32                                          break;
33                            case '*' :    result = leftOperand * rightOperand;
34                                          break;
35                            case '/' :    result = leftOperand / rightOperand;
36                                          break;
```

```
37                              case '^' :    result = Math.pow(leftOperand,
                                    rightOperand);
38                                            break;
39                              default :     System.out.printf("Unknown operator
                                    (%s)!\n", token);
40                                            System.exit(0);
41                      }
42                      pushOperand(result);
43              }
44              return popOperand();
45      }
46
47      private static void pushOperand (double n) {
48              operandStack[++top] = n;
49      }
50
51      private static double popOperand () {
52              if (top < 0) {
53                      System.out.println("Insufficient operands!");
54                      System.exit(0);
55              }
56              return operandStack[top--];
57      }
58
59      private static boolean isDouble (String n) {
60              try {
61                       Double.parseDouble(n);
62                       return true;
63              } catch (NumberFormatException e) {}
64              return false;
65      }
66  }
```

Variable Description

RPNCalculator		
double[]	operandStack	The stack of operands in order of appearance.
int	top	The index of the topmost element of operandStack

RPNCalculator::main(String[])		
String	expression	The expression in RPN to be evaluated
double	result	The evaluated form of expression

RPNCalculator::evaluateRPNExpression(String)		
String	expression	The expression in RPN to be evaluated
String[]	tokens	The individual tokens in expression, separated by whitespace
String	token	An individual token from tokens
double	rightOperand	The operand to be taken on the right side of the operator
double	leftOperand	The operand to be taken on the left side of the operator
double	result	The result on evaluating the operator token on rightOperand and leftOperand

RPNCalculator::pushOperand(double)		
double	n	The operand to be pushed into operandStack

RPNCalculator::isDouble(String)		
String	n	The string to be tested on whether it is a floating point or not

This project was compiled with XeLaTeX.

All files involved in the making of this project can be found at
https://github.com/sahasatvik/Computer-Project/tree/master/ISC

Satvik Saha
sahasatvik@gmail.com
https://sahasatvik.github.io