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**

*"I am rarely happier than when spending an entire day programming
my computer to perform automatically a task that would otherwise take me
a good ten seconds to do by hand."*

— **Douglas Adams**

Problem 1   An n digit integer $(a_1 a_2 \ldots a_n)$, where each digit $a_i \in \{0, 1, \ldots, 9\}$, is said to have *unique digits* if no digits are repeated, i.e., there is no i, j such that $a_i = a_j$ $(i \neq j)$.

Verify whether an inputted number has *unique digits*.

Solution   The problem involves simply counting the number of occurrences of each digit in the given number and checking whether any of them exceed 1.

`main (number:Integer)`

1. Initialize an integer array `digits` of length 10, indexed with integers from `[0]` to `[9]` with all elements set to `0`.
2. If `number` exceeds `0`, proceed. Otherwise, jump to (3).
   (a) Store the last digit[1] of `number` in a temporary variable `d`.
   (b) Increment the integer at the `d` index of `digits`.
   (c) If `digits[d]` exceeds `1`, the number does not have *unique digits*. Display a suitable message, and **exit**.
   (d) Discard the last digit of `number` by performing an integer division by `10` and storing the result back in `number`.
   (e) Jump to (2).
3. The number has *unique digits*. Display a suitable message.
4. **Exit**

---

[1]The last digit of an integer n is simply $n \bmod 10$

Source Code

```java
public class Unique {
    public static void main (String[] args) {
        try {
            /* Parse the first command line argument as the number
               to check for unique digits */
            long number = Long.parseLong(args[0]);
            if (isUnique(number)) {
                System.out.println("Unique Number!");
            } else {
                System.out.println("Not a Unique Number!");
            }
        } catch (NumberFormatException | IndexOutOfBoundsException e) {
            /* Handle missing or incorrectly formatted arguments */
            System.out.println("Enter 1 argument (number[integer])!");
        }
    }

    public static boolean isUnique (long number) {
        /* Keep track of the number of occurrences of each digit */
        int[] count = new int[10];
        for (long n = Math.abs(number); n > 0; n /= 10) {
            /* Extract the last digit of the number */
            int digit = (int) n % 10;
            count[digit]++;
            if (count[digit] > 1){
                return false;
            }
        }
        return true;
    }
}
```

Variable Description

Unique::main(String[])		
long	number	The inputted number
Unique::isUnique(long)		
long	number	The number to check for uniqueness
int[]	count	The number of occurrences of each digit
long	n	Counter, temporarily stores the value of number
int	digit	The last digit in n

> *"Elegance is not a dispensable luxury but a factor that decides between success and failure."*
>
> — **Edsger W. Dijkstra**

Problem 2 A *partition* of a positive integer n is defined as a collection of other positive integers such that their sum is equal to n. Thus, if $(a_1, a_2, \ldots, a_k)$ is a partition of n,

$$n \; = \; a_1 + a_2 + \cdots + a_k \qquad\qquad (a_i \in \mathbb{Z}^+)$$

Display every *unique partition* of an inputted number.

Solution This problem can be solved elegantly using *recursion*[2]. Note that when partitioning a number n, we can calculate the partitions of $(n-1)$ and append 1 to each solution. Similarly, we can append 2 to partitions of $(n-2)$, 3 to partitions of $(n-3)$, and so on. By continuing in this fashion, all cases will be reduced to the single *base case*[3] of finding the partitions of 0, of which there are trivially none.[citation needed]

There is a slight flaw in this algorithm — partitions are often repeated. This can be overcome by imposing the restriction that each new term has to be of a lesser magnitude than the previous. In this way, repeated partitions will be automatically discarded.

```
main (target:Integer)
    1. Call partition(target, target, "").
    2. Exit
```

```
partition (target:Integer, previousTerm:Integer, suffix:String)
    1. If target is 0, display suffix and return.
    2. Initialize a counter i to 1.
    3. If i is less than or equal to both the target and previousTerm, proceed. Other-
       wise, jump to (4).
        (a) Call partition(target - i, i, suffix + " " + i).
        (b) Increment i by 1.
        (c) Jump to (3).
    4. Return
```

---

[2]Recursion occurs when a thing is defined in terms of itself or of its type.
[3]A base case is a case for which the answer is known and can be expressed without recursion.

3

Source Code

```
1  public class Partition {
2      public static void main (String[] args) {
3          try {
4              /* Parse the first command line argument as the target sum */
5              int target = Integer.parseInt(args[0]);
6              if (target < 1) {
7                  throw new NumberFormatException();
8              }
9              partition(target);
10         } catch (NumberFormatException | IndexOutOfBoundsException e) {
11             /* Handle missing or incorrectly formatted arguments */
12             System.out.println("Enter 1 argument (number[natural
                  number])!");
13         }
14     }
15
16     /* Wrapper method for displaying partitions of a number */
17     public static void partition (int target) {
18         partition(target, target, "");
19     }
20
21     /* Display the partitions of the target */
22     public static void partition (int target, int previousTerm, String suffix) {
23         /* Base case : '0' has no partitions */
24         if (target == 0)
25             System.out.println(suffix);
26         /* Recursively solve for partitions by diminishing the target,
27            adding that difference to the solution, and partitioning the
28            remaining sum */
29         for (int i = 1; i <= target && i <= previousTerm; i++)
30             partition(target - i, i, suffix + " " + i);
31     }
32 }
```

Variable Description

Partition::main(String[])		
int	target	The inputted number

Partition::partition(int)		
int	target	The number to be partitioned

Partition::partition(int, int, String)		
int	target	The number to be partitioned
int	previousTerm	The previous term in the partition sequence
String	suffix	Terms in the sequence calculated so far
int	i	Counter variable, stores the next term in the sequence

Problem 3 A *Caesar cipher* is a type of monoalphabetic substitution cipher in which each letter in the plaintext is replaced by a letter some fixed number of positions down the alphabet. The positions are circular, i.e., after reaching Z, the position wraps around to A. For example, following is some encrypted text, using a right shift of 5.

```
Plain:   ABCDEFGHIJKLMNOPQRSTUVWXYZ
Cipher:  FGHIJKLMNOPQRSTUVWXYZABCDE
```

Thus, after mapping the alphabet according to the scheme $A \mapsto 0, B \mapsto 1, \ldots, Z \mapsto 25$, we can define an encryption function E_n, in which a letter x is shifted rightwards by n as follows.

$$E_n(x) = (x + n) \mod 26$$

The corresponding decryption function D_n is simply

$$D_n(x) = (x - n) \mod 26$$

Implement a simple version of a *Caesar cipher*, encrypting capitalized plaintext by shifting it by a given value. Interpret positive shifts as rightwards, negative as leftwards.

Solution This problem can be solved simply by exploiting the fact that Unicode characters are already arranged in order, with successive alphabets encoded by consecutive numbers. In addition, the encryption function can be defined exactly as given in the question — characters can be converted to their corresponding codes, manipulated by addition of the `shift`, and converted back into alphabetic form.

main (shift:Integer, plainText:String)
1. Normalize `plainText` to uppercase.
2. Normalize `shift` by replacing it with `shift` mod 26.
3. Initialize an empty String `cipherText`.
4. Initialize a counter `i` to 0.
5. If `i` is less than the length of `plainText`, proceed. Otherwise, jump to (6).
    (a) Store the character in `plainText` at position `i` in a variable `plain`.
    (b) Initialize an empty character `crypt`.
    (c) If `plain` is not an alphabet, assign `plain` to `crypt` and jump to (5g).
    (d) Convert `plain` into a number, such that `A` is mapped to `0`, `B` to `1` and so on. Store this in a temporary variable `n`.

6

(e) Add `shift` to `n`, calculate its least residue modulo 26[4], and store the result in `n`.

(f) Convert `n` back into a character and store the result in `crypt`.

(g) Append `crypt` to `cipherText`.

(h) Increment `i` by `1` and jump to (5).

6. Display `cipherText`.

7. **Exit**

Source Code

```java
public class CaesarShift {
    public static void main (String[] args) {
        try {
            /* Parse the first command line argument as the shift */
            int shift = Integer.parseInt(args[0]) % 26;
            /* Parse the second command line argument as the text to
               encrypt */
            String plaintext = args[1].toUpperCase();
            String ciphertext = "";
            for (int i = 0; i < plaintext.length(); i++) {
                char plain = plaintext.charAt(i);
                char crypt = ' ';
                if ('A' <= plain && plain <= 'Z') {
                    /* Only shift letters of the alphabet */
                    crypt = numToChar(charToNum(plain) + shift);
                } else {
                    /* Keep special characters intact */
                    crypt = plain;
                }
                /* Append the encrypted character to the cipherText */
                ciphertext += crypt;
            }
            System.out.println(ciphertext);
        } catch (NumberFormatException | IndexOutOfBoundsException e) {
            /* Handle missing or incorrectly formatted arguments */
            System.out.println("Enter 2 arguments (shift[integer],
                plaintext[text])!");
        }
    }

    /* Map letters to numbers */
    public static int charToNum (char letter) {
```

---

[4]The set of integers $K = \{0, 1, 2, \ldots, n-1\}$ is called the least residue system modulo n. The number k such that $k \in K$ and $a \equiv k \pmod{n}$ is called the least residue of a modulo n.

```
31              return Character.toUpperCase(letter) - 'A';
32          }
33
34          /* Map numbers to letters */
35          public static char numToChar (int number) {
36              return (char) ('A' + Math.floorMod(number, 26));
37          }
38  }
```

Variable Description

CaesarShift::main(String[])		
int	shift	The inputted 'shift'
String	plainText	The text to encrypt
String	cipherText	The encrypted text
int	i	Counter variable, stores the position in plainText
char	plain	The character to encrypt
char	crypt	The encrypted form of plain
CaesarShift::charToNum(char)		
char	letter	The character to convert to an integer
CaesarShift::numToChar(int)		
int	number	The number to convert to a character

"There are 2 hard problems in computer science: cache invalidation, naming things, and off-by-1 errors."

— **Leon Bambrick**

Problem 4 A *palindrome* is a sequence of characters which reads the same backwards as well as forwards. For example, `madam`, `racecar` and `kayak` are words which are palindromes. Similarly, the sentence "`A man, a plan, a canal -- Panama!`" is also a palindrome.

Analyze a sentence of input and display all *words* which are palindromes. If the entire *sentence* is also a palindrome, display it as well.

(A word is an unbroken sequence of characters, separated from other words by whitespace. Ignore single letter words such as I *and* a*. Ignore punctuation, numeric digits, whitespace and case while analyzing the entire sentence.)*

Solution The main challenge here is intelligently dividing a *sentence* into its component *words*. Verifying whether a sequence of characters is a palindrome is fairly simple — extracting those characters from a string of alphabets, numbers, punctuation and whitespace is not.

The main idea behind isolating words from sentences is to define two *markers* — a `start` to keep track of the boundary between whitespace and letters, and an `end` to mark the boundary between letters and whitespace. In this way, the markers can inch their way along the sentence, isolating words in the process. Managing the order of condition checking and incrementing of counters does require some careful manoeuvring in order to avoid any *off-by-1 errors*[5] — any of which would inevitably result in incorrect, hence undesirable output.[citation needed]

`main ()`
1. Accept a string as input, store it in a variable `sentence`.
2. Call `checkWords(sentence)` and `checkSentence(sentence)`. Store the returned values in booleans.
   (a) If either of them is `true`, set a boolean `foundPalindrome` to `true`, otherwise set it to `false`.
3. Display a suitable message if `foundPalindrome` is `false`.
4. **Exit**

---

[5]An off-by-one error often occurs in computer programming when an iterative loop iterates one time too many or too few.

checkWords (sentence:String)

1. Initialize a boolean `foundPalindrome` to false.
2. Initialize two integer counters: `start` to `-1`, `end` to `0`.
3. If `end` is less than the length of `sentence`, proceed. Otherwise, jump to (4).
   (a) Increment `start` as long as the character at the `[start + 1]` position in `sentence` is whitespace.
   (b) Assign `end` to `start`.
   (c) Increment `end` as long as it does not exceed the length of `sentence` and the character at the `[end]` position in `sentence` is not whitespace.
   (d) Assign the string of characters between `start` and `end` from `sentence` (inclusive, exclusive) to a variable `word`.
   (e) Call `isPalindrome(word)`. If `word` is a palindrome:
       i. Set `foundPalindrome` to `true`.
       ii. Display `word`.
   (f) Assign `end - 1` to `start`.
   (g) Jump to (3)
4. **Return** `foundPalindrome`

checkSentence (sentence:String)

1. Call `isPalindrome(sentence)`. If `sentence` is a palindrome:
   (a) Display `word`.
   (b) **Return** `true`.
2. **Return** `false`.

isPalindrome (text:String)

1. Normalize `text` by converting it into uppercase and removing all non-alphabetic characters.
2. Let the length of `text` be labeled temporarily as `t`.
3. Initialize two integer counters: `i` to `0`, `j` to `l - 1`.
4. If `i` is less than `j`, proceed. Otherwise, jump to (5).
   (a) If the characters at positions `i` and `j` in `text` are not equal, **return** `false`.
   (b) Increment `i` by `1`.
   (c) Decrement `j` by `1`.
   (d) Jump to (4)
5. **Return** `true` only if `text` is longer than one character. Otherwise, **return** `false`.

Source Code

```java
import java.util.Scanner;

public class Palindrome {
    public static void main (String[] args) {
        System.out.print("Enter your sentence : ");
        String sentence = (new Scanner(System.in)).nextLine().trim();
        /* Keep track of whether palindromes have been found */
        boolean foundPalindrome = false;
        System.out.println("Palindromes : ");
        foundPalindrome |= checkWords(sentence);
        foundPalindrome |= checkSentence(sentence);
        if (!foundPalindrome) {
            System.out.println("(No palindromes found!)");
        }
    }

    /* Slice a sentence into words and check each individually */
    public static boolean checkWords (String sentence) {
        boolean foundPalindrome = false;
        int start = -1;
        int end = 0;
        while (end < sentence.length()) {
            while (Character.isWhitespace(sentence.charAt(++start)));
            end = start;
            while (end < sentence.length() &&
                !Character.isWhitespace(sentence.charAt(end++)));
            String word = sentence.substring(start, end).trim();
            if (isPalindrome(word)) {
                foundPalindrome = true;
                System.out.println(getAlphabets(word));
            }
            start = end - 1;
        }
        return foundPalindrome;
    }

    /* Check the sentence as a whole */
    public static boolean checkSentence (String sentence) {
        if (isPalindrome(sentence)) {
            System.out.println("The sentence '" + sentence + "' is a
                palindrome.");
            return true;
        }
        return false;
```

```java
43         }
44
45         /* Check whether a piece of text is identical forward as well as backwards */
46         public static boolean isPalindrome (String text) {
47                 String rawText = getAlphabets(text).toUpperCase();
48                 for (int i = 0, j = rawText.length() - 1; i < j; i++, j--) {
49                         if (rawText.charAt(i) != rawText.charAt(j)) {
50                                 return false;
51                         }
52                 }
53                 /* Make sure that the text is not just one letter */
54                 return (rawText.length() > 1);
55         }
56
57         /* Strip a piece of text of all characters except alphabetic ones */
58         public static String getAlphabets (String text) {
59                 String rawText = "";
60                 for (int i = 0; i < text.length(); i++) {
61                         if (Character.isAlphabetic(text.charAt(i))) {
62                                 rawText += text.charAt(i);
63                         }
64                 }
65                 return rawText;
66         }
67 }
```

Variable Description

Palindrome::main(String[])		
String	sentence	Stores the text to check for palindromes
boolean	foundPalindrome	Stores whether palindromes have been found
Palindrome::checkWords(String)		
String	sentence	Stores the sentence to divide into words
boolean	foundPalindrome	Stores whether palindromes have been found
int	start	Counter variable, stores the index of the start of a word
int	end	Counter variable, stores the index of the end of a word
String	word	Stores words in sentence, extracted between start and end
Palindrome::checkSentence(String)		
String	sentence	Stores the sentence to divide into words
Palindrome::isPalindrome(String)		
String	text	Stores the text to check
String	rawText	Stores only alphabets from text
int	i	Counter variable, stores the current index in text
Palindrome::getAlphabets(String)		
String	text	Stores the text to extract alphabets from
String	rawText	Stores only alphabets from text
int	i	Counter variable, stores the current index in text

Problem 5 A *prime number* (or a *prime*) is a natural number greater than 1 that has no positive divisors other than 1 and itself.

Display all primes upto a given limit, along with their number.

Solution This problem can be tackled in a multitude of ways.[citation needed] We could define a function for checking the primality of a given number, then iterate through all numbers in the required range. A common way of checking for primality is *trial division*. It consists of testing whether the number n is a multiple of any integer between 2 and $\sqrt{n}$. Although this works well enough for small numbers, repeating this consecutively for very large inputs is tedious and inefficient. Since the problem consists of identifying primes in a *range*, and not individually, we can make use of more efficient methods.

The *Sieve of Eratosthenes* is a simple, ancient algorithm for finding all prime numbers up to any given limit. It does so by iteratively marking as composite the multiples of each prime, starting with the first prime number, 2. As a result, when a prime p is found, none of its multiples will be tested further for primality — they are eliminated early on. In comparison, *trial division* has worse theoretical complexity than that of the *Sieve of Eratosthenes* in generating ranges of primes. When testing each prime, the optimal trial division algorithm uses all prime numbers not exceeding its square root, whereas the Sieve of Eratosthenes produces each composite only from its prime factors.

```
main (upperLimit:Integer)
```
1. Create a new `SieveOfEratosthenes`, pass it `upperLimit` and assign it to `sieve`.
2. Call `sieve->sievePrimes()`.
3. Display the indices which correspond to `true` in the boolean array `sieve->primes`.
4. **Exit**

```
SieveOfEratosthenes (upperLimit:Integer)
```
1. Initialize a boolean array `primes`, indexed with integers from `[0]` to `[upperLimit - 1]`, with all elements set to `true`.
2. Set `primes[0]` and `primes[1]` to `true`.
3. **Define** the function `SieveOfEratosthenes::sievePrimes()` and **return** the resultant object.

SieveOfEratosthenes::sievePrimes ()
1. Initialize an integer variable `prime` to 2.
2. If `prime` is less than the square root of `upperLimit`, proceed. Otherwise, **return**.

    (a) Initialize an integer variable `multiple` to the square of `prime`.
    (b) If `multiple` is less than `upperLimit`, proceed. Otherwise, jump to (2c).
        i. Set `primes[multiple]` to `false`.
        ii. Increment `multiple` by `prime`.
        iii. Jump to (2b)
    (c) Increment `prime` until `primes[prime]` is `true`.
    (d) Jump to (2).
3. **Return**

Source Code

```java
1  public class SieveOfEratosthenes {
2          private final int upperLimit;
3          private boolean[] primes;
4
5          /* Initialize the list of numbers using an upper limit */
6          public SieveOfEratosthenes (int upperLimit) {
7                  this.upperLimit = upperLimit;
8                  this.initPrimes();
9          }
10
11         public boolean[] getPrimes () {
12                 return primes;
13         }
14
15         /* Initialize all value to 'prime' by default */
16         public void initPrimes () {
17                 this.primes = new boolean[upperLimit];
18                 /* Mark known values as 'not prime' */
19                 primes[0] = false;
20                 primes[1] = false;
21                 for (int i = 2; i < upperLimit; i++)
22                         primes[i] = true;
23         }
24
25         /* Iteratively sieve the numbers to leave primes behind */
26         public void sievePrimes () {
27                 /* Start with the first prime */
28                 int prime = 2;
```

```
29                while ((prime * prime) < upperLimit) {
30                        /* Start with the first multiple not crossed off */
31                        int multiple = prime * prime;
32                        while (multiple < upperLimit) {
33                                /* Cross multiples of a prime off the list */
34                                primes[multiple] = false;
35                                multiple += prime;
36                        }
37                        /* Skip forward to the next prime */
38                        while (!primes[++prime]);
39                }
40        }
41  }
```

```
1   public class Primes {
2         public static void main (String[] args) {
3               try {
4                       /* Parse the first command line argument as the upper limit
5                          on primes to calculate */
6                       int upperLimit = Integer.parseInt(args[0]);
7                       if (upperLimit < 2) {
8                               throw new NumberFormatException();
9                       }
10                      SieveOfEratosthenes sieve = new
                            SieveOfEratosthenes(upperLimit);
11                      sieve.sievePrimes();
12                      showPrimes(sieve.getPrimes());
13              } catch (NumberFormatException | IndexOutOfBoundsException e) {
14                      /* Handle missing or incorrectly formatted arguments */
15                      System.out.println("Enter 1 argument (limit[integer, >1])!");
16                      System.out.println("(Primes will be displayed up to, not
                            including 'limit')");
17              }
18        }
19
20        /* Display all primes calculated */
21        public static void showPrimes (boolean[] primes) {
22              int primeCount = 0;
23              /* Format all number to the same width */
24              int maxLength = Integer.toString(primes.length).length();
25              for (int i = 0; i < primes.length; i++) {
26                      /* If 'i' is prime, primes[i] will be marked 'true' */
27                      if (primes[i]) {
28                              System.out.printf("%" + maxLength + "d ", i);
29                              primeCount++;
```

```
30                         }
31                 }
32                 System.out.println("\nTotal number of primes : " + primeCount);
33         }
34 }
```

Variable Description

SieveOfEratosthenes		
int	upperLimit	The number of integers to sieve
boolean[]	primes	Primes, with contents indicating the primality of the index
SieveOfEratosthenes::initPrimes()		
int	i	Counter variable
SieveOfEratosthenes::sievePrimes()		
int	prime	Counter variable, stores current primes found
int	multiple	Counter variable, stores the multiples of `prime`
Primes::main(String[])		
int	upperLimit	The highest integer to check for primality (exclusive)
SieveOf Eratosthenes	sieve	An object capable of sieving primes
Primes::showPrimes(boolean[])		
boolean[]	primes	Primes, with contents indicating the primality of the index
int	primeCount	The number of primes found
int	maxLength	The length of the longest number to display
int	i	Counter variable, stores the current integer to check for primality

"Any fool can use a computer. Many do."

— **Ted Nelson**

Problem 6   Design a simple interface for an examiner which can format and display marks scored by a group of students in a particular examination. Calculate the percentage scored by each candidate and display the list of students and percentages in an ASCII bar chart, arranged alphabetically.

Solution   This problem calls for a fairly straightforward flow of logic. The main goal is to present the user with a simple way of providing input, along with nicely formatted output.

```
main (upperLimit:Integer)
```
1. Input the maximum marks allotted for the examination as a floating point. Store it as `maxMarks`.
2. Input the total number of students whose marks are to be recorded as an integer. Store it as `numberOfStudents`.
3. Create a new `Marksheet`, pass it `maxMarks`, `numberOfStudents` and assign it to `sheet`.
4. Initialize an integer counter `i` to `0`;
5. If `i` is less than `numberOfStudents`, proceed. Otherwise, jump to (6).
    (a) Input a student's name as a string. Store it as `name`.
    (b) Input the student's marks as a floating point. Store it as `marks`.
    (c) Call `sheet->addMarks(name, marks)`.
    (d) Jump to (5).
6. Call `sheet->sortByName()`.
7. Call `sheet->displayChart()`.
8. Call `sheet->sortMaxScorers()`.
9. **Exit**


```
Marksheet (maxMarks:FloatingPoint, numberOfStudents:Integer)
```
1. Initialize a string array `names`, indexed with integers from `[0]` to `[numberOfStudents - 1]`.
2. Initialize a floating point array `marks`, indexed with integers from `[0]` to `[numberOfStudents - 1]`.
3. Initialize an integer counter `lastStudent` to `-1`.
4. **Define** the functions:

(a) `Marksheet::addMarks(name, score)`

(b) `Marksheet::sortByName()`

(c) `Marksheet::displayChart()`

(d) `Marksheet::displayMaxScorers()`

5. **Return** the resultant object.

`Marksheet::addMarks (name:String, score:FloatingPoint)`

1. Increment `lastStudent` by 1.
2. Set the `names[lastStudent]` to `name`.
3. Set the `marks[lastStudent]` to `score`.
4. **Return**

`Marksheet::sortByName ()`

1. Assign `lastStudent` to `right`.
2. If `right` exceeds 0, proceed. Otherwise, **return**.
    (a) Initialize an integer counter `i` to 1.
    (b) If `i` is less than or equal to `right`, proceed. Otherwise, jump to (2c).
        i. If `names[i-1]` comes lexicographically after `names[i]`:
            A. Swap the elements at `names[i-1]` and `names[i]`.
            B. Swap the elements at `marks[i-1]` and `marks[i]`.
        ii. Jump to (2b).
    (c) Jump to (2).

`Marksheet::displayChart ()`

1. For every string `name` in `names`:
    (a) Calculate the length of the bar in the chart as a fraction of the screen width. Store the calculated number of characters to display as `points`.
    (b) Display `name`, a string of suitable characters for the bar of length `points`, along with the percentage scored.
2. **Return**

`Marksheet::displayMaxScorers ()`

1. Calculate the maximum floating point in `marks` and store it as `maxScore`.
2. For every integer `i` between 0 and `numberOfStudents` (inclusive, exclusive) such that `marks[i]` is equal to the `maxScore`, display `names[i]`.
3. **Return**

Source Code

```java
public class Marksheet {
    public static final int SCREEN_WIDTH = 100;
    private final double maxMarks;
    private final int numberOfStudents;
    private int lastStudent;
    private String[] names;
    private double[] marks;

    /* Initialize some final data */
    public Marksheet (double maxMarks, int numberOfStudents) {
        this.maxMarks = maxMarks;
        this.numberOfStudents = numberOfStudents;
        this.names = new String[numberOfStudents];
        this.marks = new double[numberOfStudents];
        this.lastStudent = -1;
    }

    /* Add names and marks to the stack */
    public boolean addMarks (String name, double score) {
        try {
            names[++lastStudent] = name;
            marks[lastStudent] = score;
            return true;
        } catch (IndexOutOfBoundsException e) {
            return false;
        }
    }

    /* Display the names and percentages in a bar chart */
    public void displayChart () {
        System.out.println(Marksheet.multiplyString("-",
            Marksheet.SCREEN_WIDTH));
        for (int i = 0; i <= lastStudent; i++) {
            /* Calculate the fraction of marks earned */
            double fraction = marks[i] / maxMarks;
            String name = (names[i].length() < 16)
                            ? names[i]
                            : (names[i].substring(0,13) + "...");
            int points = (int) (fraction * (SCREEN_WIDTH - 34));
            /* Generate and pad the bar to display */
            String bar = multiplyString("*", points)
                + multiplyString(" ", SCREEN_WIDTH - 34 - points);
            System.out.printf("| %16s | %s | %6.2f %% |%n"
                                        , name
```

```java
44                                                       , bar
45                                                       , fraction * 100);
46                }
47                System.out.println(Marksheet.multiplyString("-",
                      Marksheet.SCREEN_WIDTH));
48        }
49
50        /* Display the name of students with the highest score */
51        public void displayMaxScorers () {
52                String maxScorers = "";
53                double maxScore = getMaxScore();
54                for (int i = 0; i <= lastStudent; i++) {
55                        if (marks[i] == maxScore) {
56                                maxScorers += ", " + names[i];
57                        }
58                }
59                System.out.println(maxScorers.substring(1)
60                                        + " scored the highest ("
61                                        + maxScore + "/"
62                                        + maxMarks + ")");
63        }
64
65        /* Sort the names and associated marks lexicographically */
66        public void sortByName () {
67                for (int right = lastStudent; right > 0; right--)
68                        for (int i = 1; i <= right; i++)
69                                if (names[i-1].compareToIgnoreCase(names[i]) > 0)
70                                        swapRecords(i, i - 1);
71
72        }
73
74        /* Get the value of the highest score */
75        public double getMaxScore () {
76                double max = Integer.MIN_VALUE;
77                for (int i = 0; i <= lastStudent; i++) {
78                        max = Math.max(max, marks[i]);
79                }
80                return max;
81        }
82
83        /* Utility function to swap student records */
84        private void swapRecords (int x, int y) {
85                String tempName = names[x];
86                double tempMark = marks[x];
87                names[x] = names[y];
88                marks[x] = marks[y];
```

```
89              names[y] = tempName;
90              marks[y] = tempMark;
91          }
92
93      /* Utility function for repeating strings */
94      public static String multiplyString (String s, int n) {
95              String out = "";
96              while (n --> 0)
97                      out += s;
98              return out;
99      }
100 }
```

```
1   import java.util.Scanner;
2   import java.util.InputMismatchException;
3
4   public class ScoreRecorder {
5       public static void main (String[] args) {
6               /* Create an object capable of managing input */
7               Scanner inp = new Scanner(System.in);
8               double maxMarks = 0.0;
9               int numberOfStudents = 0;
10              try {
11                      System.out.print("Enter the maximum marks allotted for each
                            student : ");
12                      maxMarks = inp.nextDouble();
13                      System.out.print("Enter the total number of students : ");
14                      numberOfStudents = inp.nextInt();
15                      /* Check for any erroneous data */
16                      if (maxMarks <= 0) {
17                              System.out.println("Maximum marks must be positive!");
18                              System.exit(0);
19                      }
20                      if (numberOfStudents <= 0) {
21                              System.out.println("Number of students must be
                                    positive!");
22                              System.exit(0);
23                      }
24                      /* Create an object capable of recording scoresheets */
25                      Marksheet sheet = new Marksheet(maxMarks, numberOfStudents);
26                      System.out.println("Enter " + numberOfStudents + " students'
                            names and marks : ");
27                      /* Accept student data */
28                      for (int i = 0; i < numberOfStudents; i++) {
29                              String name = "";
```

```
30                          while (!inp.hasNextDouble()) {
31                                  name += inp.next() + " ";
32                          }
33                          double marks = inp.nextDouble();
34                          if (marks <= 0 || marks > maxMarks) {
35                                  System.out.println("Marks must be within 0.0 and
                                        " + maxMarks + "!");
36                                  System.exit(0);
37                          }
38                          sheet.addMarks(name.trim(), marks);
39                  }
40                  /* Sort and display */
41                  sheet.sortByName();
42                  sheet.displayChart();
43                  sheet.displayMaxScorers();
44          } catch (InputMismatchException e) {
45                  /* Handle missing or incorrectly formatted arguments */
46                  System.out.println("Invalid Input!");
47                  System.exit(0);
48          }
49      }
50  }
```

Variable Description

Marksheet		
int	SCREEN_WIDTH	Number of characters to use in the display width
double	maxMarks	The maximum marks allotted for the examination
int	numberOf Students	The number of students whose marks are to be recorded
int	lastStudent	The index number of the last student added to the marksheet
String[]	names	The names of the students
double[]	marks	The marks of the students
Marksheet::addMarks(String, double)		
String	name	The name of the student to be added
double	score	The marks of the student to be added
Marksheet::displayChart()		
int	i	Counter variable
double	fraction	The fraction on marks scored over the maximum marks

String	name	Temporarily stores a formatted version of a student's name
int	points	The number of characters to display in the bar chart
String	bar	The bar in the chart, along with whitespace padding
Marksheet::displayMaxScorers()		
String	maxScorers	The list of highest scoring students
double	maxScore	The highest score
int	i	Counter variable
Marksheet::sortByName()		
int	right	Counter variable
int	i	Counter variable
Marksheet::getMaxScore()		
double	max	The maximum score in `marks`
int	i	Counter variable
Marksheet::swapRecords(int, int)		
int	x, y	The indices of the records to swap
String	tempName	Temporary storage of a name
double	tempMark	Temporary storage of a mark
Marksheet::multiplyString(String, int)		
String	s	The string to multiply
int	n	The number of times to multiply `s`
String	out	The string containing `n` copies of `s`
ScoreRecorder::main(String[])		
Scanner	inp	The input managing object
double	maxMarks	The maximum marks allotted for the examination
int	numberOf Students	The number of students whose marks are to be recorded
Marksheet	sheet	An object capable of managing student records
int	i	Counter variable
String	name	The name of the student to be added
double	marks	The marks of the student to be added

"To iterate is human, to recurse divine"

— **L. Peter Deutsch**

Problem 7   The *determinant* of a square matrix $A_{n,n}$ is defined recursively as follows.

$$det(A_{n,n}) = \begin{vmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{vmatrix} = \sum_{j=1}^{n} (-1)^{i+j} a_{i,j} \cdot det(M_{i,j})$$

where $M_{i,j}$ is defined as the minor of $A_{n,n}$, an $(n-1) \times (n-1)$ matrix formed by removing the ith row and jth column from $A_{n,n}$.

The determinant of a (2×2) matrix is simply given by

$$\begin{vmatrix} a & b \\ c & d \end{vmatrix} = ad - bc$$

For example, the determinant of a (3×3) matrix is given by the following expression.

$$\begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix} = a \begin{vmatrix} e & f \\ h & i \end{vmatrix} - b \begin{vmatrix} d & f \\ g & i \end{vmatrix} + c \begin{vmatrix} d & e \\ g & h \end{vmatrix}$$

$$= aei + bfg + cdh - ceg - bdi - afh$$

Calculate the *determinant* of an inputted $(n \times n)$ square matrix.

Solution   This problem offers the opportunity to showcase the power of recursive functions. Here, the complex task of calculating the determinant of a large matrix can be subdivided into multiple smaller tasks. In fact, each of these tasks is precisely the same as the larger one — the only difference is the size of the matrices. Eventually, the problem reduces to finding the determinants of multiple (2×2) matrices. The values thus obtained can be pieced together to form the final answer.

`main ()`
 1. Input the size (number of rows/columns) of the square matrix. Store it as `size`.
 2. Create a new `SquareMatrix`, pass it `size`, and assign it to `matrix`.
 3. For each `i` $\in \{1, 2, \ldots,$ `size`$\}$:

    (a) For each j ∈ {1, 2, . . . , `size`}:
         i. Input an integer as `n`.
         ii. Set the element at [`i, j`] of `matrix` to `n`.
4. Call `matrix->getDeterminant()` and display the returned value.
5. **Exit**

Matrix (rows:Integer, columns:Integer)

1. Initialize an integer array of integer arrays `elements`, indexed with integers from `[1]` to `[rows]`, with each contained integer array indexed with integers from `[1]` to `[columns]` .
2. **Return** the resultant object.

SquareMatrix (size:Integer)

1. **Define** the functions:
    (a) `SquareMatrix::getDeterminant()`
    (b) `SquareMatrix::getMinorMatrix(row, column)`
2. **Return** a `Matrix`, with both `rows` and `columns` set to `size`.

SquareMatrix::getDeterminant ()

1. If the `size` is 1, **return** the only element (`elements[1, 1]`).
2. If the `size` is 2, **return** (`elements[1, 1]`×`elements[2, 2]`) − (`elements[1, 2]`×`elements[2, 1]`).
3. Initialize an integer variable `determinant` to 0.
4. For each i ∈ {1, 2, . . . , `size`}:
    (a) Call `this->getMinorMatrix(i, i)->getDeterminant()`. Store the result in `d`.
    (b) Add ($(-1)^{i+1}$ × `matrix[1, i]` × `d`) to `determinant`.
5. **Return** `determinant`.

SquareMatrix::getMinorMatrix (row:Integer, column:Integer)

1. Create a new `SquareMatrix`, pass it (`size - 1`), and assign it to `minor`.
2. Copy all elements from `this` to `minor`, except for those at position [`row, *`] or [`*, column`].
3. **Return** `minor`.

Source Code

```java
public class Matrix {
    protected final int rows;
    protected final int columns;
    protected int[][] elements;

    /* Initialize a matrix of a given order */
    public Matrix (int rows, int columns) {
        this.rows = rows;
        this.columns = columns;
        this.elements = new int[rows][columns];
    }

    public int getRows () {
        return this.rows;
    }

    public int getColumns () {
        return this.columns;
    }

    /* Set elements in the matrix using natural indices */
    public void setElementAt (int element, int row, int column) {
        if (row < 1 || row > rows || column < 1 || column > columns)
            return;
        elements[row-1][column-1] = element;
    }

    /* Get elements from the matrix using natural indices */
    public int getElementAt (int row, int column) {
        if (row < 1 || row > rows || column < 1 || column > columns)
            return Integer.MIN_VALUE;
        return elements[row-1][column-1];
    }
}
```

```java
public class SquareMatrix extends Matrix {
    protected int size;

    /* Initialize the matrix with the same number of rows and columns */
    public SquareMatrix (int size) {
        super(size, size);
        this.size = size;
    }

```

```java
10          public int getSize () {
11                  return this.size;
12          }
13
14          /* Recursively calculate the determinant of the matrix */
15          public int getDeterminant () {
16                  /* Base cases */
17                  if (this.size == 1)
18                          return getElementAt(1, 1);
19                  if (this.size == 2)
20                          return (getElementAt(1, 1) * getElementAt(2, 2))
21                                  - (getElementAt(1, 2) * getElementAt(2, 1));
22                  int determinant = 0;
23                  /* Accumulate the determinants of minors with alternating signs */
24                  for (int i = 1; i <= size; i++)
25                          determinant += ((int) Math.pow(-1, 1+i))
26                                          * getElementAt(1, i)
27                                          * getMinorMatrix(1, i).getDeterminant();
28                  return determinant;
29          }
30
31          /* Get the minor matrix by removing a row and a column */
32          public SquareMatrix getMinorMatrix (int row, int column) {
33                  /* Check bounds */
34                  if (row < 1 || row > size || column < 1 || column > size)
35                          return null;
36                  if (this.size <= 1)
37                          return new SquareMatrix(0);
38                  SquareMatrix minor = new SquareMatrix(this.size - 1);
39                  for (int i = 1, p = 1; p < size; i++, p++) {
40                          /* Skip 'row' */
41                          if (i == row)
42                                  i++;
43                          for (int j = 1, q = 1; q < size; j++, q++) {
44                                  /* Skip 'column' */
45                                  if (j == column)
46                                          j++;
47                                  /* Copy values into the new matrix */
48                                  minor.setElementAt(this.getElementAt(i, j), p, q);
49                          }
50                  }
51                  return minor;
52          }
53  }
```

```java
import java.util.Scanner;

public class Determinant {
    public static void main (String[] args) {
        /* Create an object for managing input */
        Scanner inp = new Scanner(System.in);
        try {
            System.out.print("Enter the size of the (size X size) square
                matrix : ");
            int size = inp.nextInt();
            /* Create a square matrix which has suitable methods for
                calculation */
            SquareMatrix matrix = new SquareMatrix(size);
            System.out.println("Enter " + (size * size) + " integers : ");
            for (int i = 1; i <= size; i++)
                for (int j = 1; j <= size; j++)
                    matrix.setElementAt(inp.nextInt(), i, j);
            System.out.println("\nThe determinant is : " +
                matrix.getDeterminant());
        } catch (Exception e) {
            /* Handle missing or incorrectly formatted arguments */
            System.out.println("Invalid Input!");
        }
    }

    /* Display the matrix in a neat format */
    public static void showMatrix (Matrix m) {
        for (int i = 1; i <= m.getRows(); i++) {
            for (int j = 1; j <= m.getColumns(); j++) {
                System.out.printf("%4d ", m.getElementAt(i, j));
            }
            System.out.println();
        }
    }
}
```

Variable Description

Matrix		
int	rows	Number of rows in the matrix
int	columns	Number of columns in the matrix
int[][]	elements	The array of integer arrays, storing the elements of the matrix
SquareMatrix		
int	size	Number of both rows and columns in the matrix
SquareMatrix::getDeterminant()		
int	determinant	The determinant of the SquareMatrix
int	i	Counter variable
SquareMatrix::getMinorMatrix(int, int)		
int	row	The row to remove from the matrix
int	column	The column to remove from the matrix
SquareMatrix	minor	The matrix obtained by removing row and column
int	i, j	Counter variables
Determinant::main(String[])		
Scanner	inp	The input managing object
int	size	Number of both rows and columns in the matrix
SquareMatrix	matrix	The matrix whose determinant is to be calculated
int	i, j	Counter variables
Determinant::showMatrix(Matrix)		
Matrix	m	The matrix to display
int	i, j	Counter variables

Problem 8 A *Knight's Tour* is a sequence of moves of a knight on a chessboard such that the *knight* visits every square only once. If the knight ends on a square that is one knight's move from the beginning square, the tour is *closed* forming a closed loop, otherwise it is *open.*

There are many ways of constructing such paths on an empty board. On an 8×8 board, there are no less than $26,534,728,821,064$ *directed*[6] *closed* tours. Below is one of them.
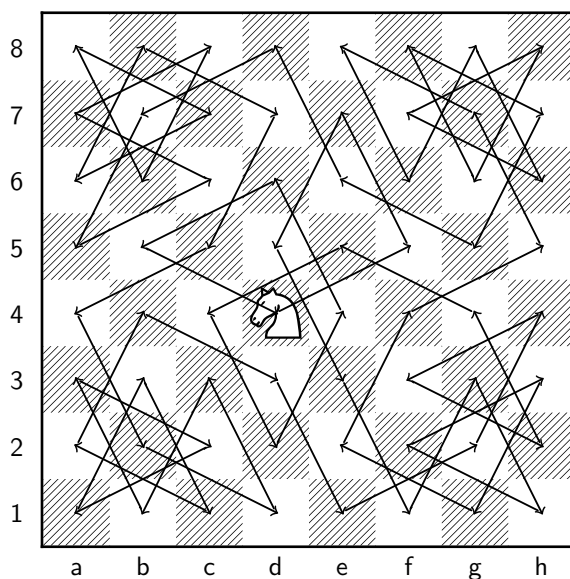

Construct a *Knight's Tour* (*open* or *closed*) on an $n \times n$ board, starting from a given square.

(Mark each square with the move number on which the knight landed on it. Mark the starting square 1*.)*

---

[6]Two tours along the same path that travel in opposite directions are counted separately, as are rotations and reflections.

Solution   A knight on a chessboard can move to a square that is two squares away horizontally and one square vertically, or two squares vertically and one square horizontally.
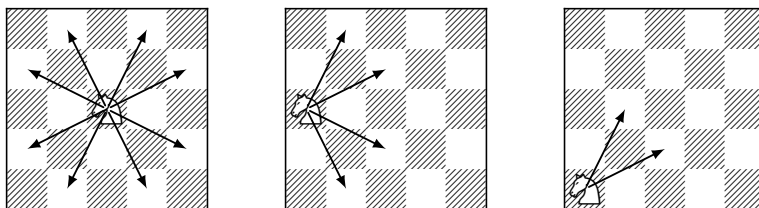
The mobility of a knight can make varies greatly with its position on the board — near the centre, it can jump to one of 8 squares while when in a corner, it can jump to only 2. On the other hand, the number of possible *sequences* of squares a knight can traverse grows extremely quickly. Although it may seem that a simple *brute force* search can quickly find one of *trillions* of solutions, there are approximately 4×10^{51} different paths to consider on an 8×8 board. For even larger boards, iterating through every possible path is clearly impractical.[citation needed]

This problem calls for implementing a *backtracking*[7] *algorithm*, coupled with some *heuristic*[8] to speed up the search. One such heuristic is *Warnsdorf's Rule*.

> The knight is moved so that it always proceeds to the square from which the knight will have the *fewest* onward moves.

This allows us to define a ranking algorithm for each possible path — the positions which result in the smallest number of further moves, or is furthest away from the board's centre will be investigated first. In case of a tie, we can either proceed without making any changes to the already existing positions, or introduce a random element. This has the effect of producing different results on successive executions, giving a variety of solutions.

One drawback of resolving ties randomly is that an early "wrong" choice in the position tree can force the calculation of every resulting path without reaching a solution, effectively reducing the algorithm to a brute force search. This is especially problematic

---

[7]Backtracking is a general algorithm for finding some or all solutions to some computational problems that incrementally builds candidates to the solutions, and abandons each partial candidate ("backtracks") as soon as it determines that the candidate cannot possibly be completed to a valid solution.

[8]A heuristic technique is any approach to problem solving that employs a practical method not guaranteed to be optimal or perfect, but sufficient for the immediate goals. Where finding an optimal solution is impossible or impractical, heuristic methods can be used to speed up the process of finding a satisfactory solution.

for large boards, where it may take hours to backtrack and reach a solution. Thus, the "randomness factor" should be adjusted according to the board size.

A high randomness can be useful for searching specifically for *closed tours*, as a randomness of 0 simply produces the same solution every time (which may or may not be closed). Below are some tours generated by the program.

The tendency of the path to remain close to the edges of the board, where the mobility of the knight is restricted, is clearly evident.

```
main (boardSize:Integer, initSquare:Position, randomness:FloatingPoint)
```
  1. Create a new `TourSolver`, pass it `boardSize`, `initSquare`, `randomness`, and assign it to `t`.
  2. Call `t->getSolution()`. Store the returned move stack as `solution`.
  3. Display the board obtained by calling `t->getBoard()` along with the moves in `solution`.
  4. **Exit**


```
TourSolver (size:Integer, initSquare:Position, randomness:FloatingPoint)
```
  1. Initialize an integer arrays of integer arrays indexed with integers from `[1]` to `[size]`, simulating a chessboard. Store it as `board`, which records the move numbers on which the knight lands on it.
  2. Initialize a `Position` stack `path`, along with methods to add and remove `Position`'s from it.
  3. Set an integer counter `numberOfMoves` to `0`, as part of the `path` stack.
  4. **Define** the functions:
     (a) `TourSolver::solve(p)`

33

(b) `TourSolver::getPossibleMoves(p)`
5. **Return** the resultant object.

`TourSolver::solve (p:Position)`
1. If the `path` stack is full, **return true**, indicating that the tour has been solved.
2. Call `this->getPossibleMoves(p)`. Store the returned list of possible legal moves as `moves`.
3. Sort `moves`, ranking each possible position according to *Warnsdorf's Rule*.
4. For every `move` in the list `moves`:
   (a) Push `move` onto the `path` stack and `board`.
   (b) If the call `this->solve(move)` returns **true**, **return true**. Otherwise, pop `move` from the `path` stack and `board` (*backtrack*).
5. If the list `moves` has been exhausted, **return false**, indicating that there are no solutions from the position `p` for that particular move stack.

`TourSolver::getPossibleMoves (p:Position)`
1. Initialize a list of moves `possibleMoves`.
2. For every possible square `move` a knight can jump to from `p` (on an empty board):
   (a) If `move` is currently a legal move, without falling outside the board or on a previously traversed square, add it to `possibleMoves`.
3. **Return** `possibleMoves`

Source Code

```
1   public class TourSolver {
2          private final int size;
3          private Position[] path;
4          private int numberOfMoves;
5          private int[][] board;
6          private int[][] degreesOfFreedom;
7          private Position initPosition;
8          private double tieBreakRandomness;
9
10         /* Store the list of possible changes in the 'x' and 'y' coordinates of
11            a knight on an empty board */
12         private static final int[][] KNIGHT_MOVES = {
13                 {-1, -2}, {-1, 2}, {1, -2}, {1, 2},
14                 {-2, -1}, {-2, 1}, {2, -1}, {2, 1}
15         };
16
17         /* Initialize the board and move stack */
18         public TourSolver (int size, Position initPosition, double randomness) {
19                 this.size = size;
20                 this.initPosition = initPosition;
21                 this.tieBreakRandomness = randomness / 2.0;
22                 this.path = new Position[size * size];
23                 this.numberOfMoves = 0;
24                 this.initBoard();
25                 this.initDegreesOfFreedom();
26         }
27
28         /* Reset the board */
29         public void resetSolution () {
30                 this.path = new Position[size * size];
31                 this.numberOfMoves = 0;
32                 this.initBoard();
33         }
34
35         /* Initialize a blank board */
36         private void initBoard () {
37                 board = new int[size][size];
38                 for (int i = 0; i < size; i++)
39                         for (int j = 0; j < size; j++)
40                                 board[i][j] = 0;
41         }
42
43         /* Calculate the mobility of a knight on each square */
44         private void initDegreesOfFreedom () {
```

```
45                   degreesOfFreedom = new int[size][size];
46               for (int i = 0; i < size; i++)
47                   for (int j = 0; j < size; j++)
48                       degreesOfFreedom[i][j] = getPossibleMovesCount(new
                             Position(i, j));
49           }
50
51           /* Push a move onto the move stack, add it to the board */
52           public boolean addMove (Position p) {
53               if (numberOfMoves < (size * size)) {
54                   path[numberOfMoves++] = p;
55                   board[p.getX()][p.getY()] = numberOfMoves;
56                   return true;
57               }
58               return false;
59           }
60
61           /* Pop a move from the move stack, remove it from the board */
62           public boolean removeMove () {
63               if (numberOfMoves > 0) {
64                   Position p = path[numberOfMoves - 1];
65                   /* Empty squares are marked '0' */
66                   board[p.getX()][p.getY()] = 0;
67                   path[--numberOfMoves] = null;
68                   return true;
69               }
70               return false;
71           }
72
73           public int[][] getBoard () {
74               return board;
75           }
76
77           /* Get the stack of moves comprising a knight's tour */
78           public Position[] getSolution () {
79               if (size < 5)
80                   return null;
81               addMove(initPosition);
82               if(solve(initPosition))
83                   return path;
84               return null;
85           }
86
87           /* Recursively solve a tour from a given position */
88           public boolean solve (Position p) {
89               /* If the move stack is full, the tour has been solved */
```

36

```
 90                if (numberOfMoves == (size * size))
 91                        return true;
 92                /* Get every legal move and rank them using Warnsdorf's Rule */
 93                Position[] possibleMoves = getPossibleMoves(p);
 94                if (possibleMoves[0] == null)
 95                        return false;
 96                sortMoves(possibleMoves);
 97                for (Position move : possibleMoves) {
 98                        if (move != null) {
 99                                /* Try a move */
100                                addMove(move);
101                                if (solve(move))
102                                        return true;
103                                /* Backtrack */
104                                removeMove();
105                        }
106                }
107                return false;
108        }
109
110        /* Sort a list of positions using Warnsdorf's Rule */
111        public void sortMoves (Position[] moves) {
112                int count = 0;
113                for (Position p : moves)
114                        if (p != null)
115                                count++;
116                for (int right = count; right > 0; right--)
117                        for (int i = 1; i < right; i++)
118                                if (compareMoves(moves[i-1], moves[i]) > 0)
119                                        swapMoves(i-1, i, moves);
120        }
121
122        /* Compare 2 moves using Warnsdorf's Rule */
123        public int compareMoves (Position a, Position b) {
124                /* Compare the mobilities of the knight */
125                int aCount = getPossibleMovesCount(a);
126                int bCount = getPossibleMovesCount(b);
127                if (aCount != bCount)
128                        return aCount - bCount;
129                /* Compare the mobilities of the knight on an empty board */
130                int aFree = degreesOfFreedom[a.getX()][a.getY()];
131                int bFree = degreesOfFreedom[b.getX()][b.getY()];
132                if (aFree != bFree)
133                        return aFree - bFree;
134                /* Resolve ties using a pre-decided element of randomness */
135                return (Math.random() < tieBreakRandomness)? 1 : -1;
```

```java
136            }
137
138            /* Utility function to swap moves in the list of possible moves */
139            private static void swapMoves (int x, int y, Position[] moves) {
140                    Position t = moves[x];
141                    moves[x] = moves[y];
142                    moves[y] = t;
143            }
144
145            /* Get the list of all possible, legal moves not touching a previously
146               traveled square from a given position */
147            public Position[] getPossibleMoves (Position start) {
148                    Position[] possibleMoves = new Position[KNIGHT_MOVES.length];
149                    int i = 0;
150                    for (int[] move : KNIGHT_MOVES) {
151                            /* Generate a new */
152                            int x = start.getX() + move[0];
153                            int y = start.getY() + move[1];
154                            /* Check the legality of that move */
155                            if (isWithinBoard(x, y) && board[x][y] == 0) {
156                                    possibleMoves[i++] = new Position(x, y);
157                            }
158                    }
159                    return possibleMoves;
160            }
161
162            /* Get the number of legal moves */
163            public int getPossibleMovesCount (Position start) {
164                    int i = 0;
165                    for (Position p : getPossibleMoves(start))
166                            if (p != null)
167                                    i++;
168                    return i;
169            }
170
171            /* Check whether a position lies within the board */
172            public boolean isWithinBoard (int x, int y) {
173                    return (x >= 0 && x < size && y >= 0 && y < size);
174            }
175    }
```

```
1   public class Position {
2        private final int x;
3        private final int y;
4
5        /* Initialize using the coordinates on the board */
6        public Position (int x, int y) {
7             this.x = x;
8             this.y = y;
9        }
10
11       /* Initialize using the position in algebraic notation */
12       public Position (String s) {
13            int x = 0;
14            int i = 0;
15            while (i < s.length() && Character.isAlphabetic(s.charAt(i))) {
16                 x = (x * 26) + Character.toLowerCase(s.charAt(i)) - 'a' + 1;
17                 i++;
18            }
19            int y = Integer.parseInt(s.substring(i));
20            this.x = x - 1;
21            this.y = y - 1;
22       }
23
24       public int getX () {
25            return x;
26       }
27
28       public int getY () {
29            return y;
30       }
31
32       public boolean equals (Position p) {
33            return (p != null)
34                 && (this.getX() == p.getX()) && (this.getY() == p.getY());
35       }
36
37       @Override
38       public String toString () {
39            return xToString(this.x) + (this.y + 1);
40       }
41
42       /* Convert a file number to its algebraic notation form */
43       public static String xToString (int n) {
44            int x = n + 1;
45            String letters = "";
46            while (x > 0) {
```

```
47                      letters = (char) ('a' + (--x % 26)) + letters;
48                      x /= 26;
49                  }
50                  return letters;
51              }
52  }


 1  public class KnightTour {
 2          public static void main (String[] args) {
 3                  try {
 4                          /* Parse the first command line argument as the size of the
                                board */
 5                          int boardSize = Integer.parseInt(args[0]);
 6                          if (boardSize <= 0)
 7                                  throw new NumberFormatException();
 8                          /* Parse the second command line argument as the starting
                                square
 9                            of the knight, written in algebraic notation */
10                          String initSquare = (args.length > 1)? args[1] : "a1";
11                          /* Parse the third command line argument as the degree of
12                            randomness to be used while resolving ties */
13                          double randomness = (args.length > 2)?
                                Double.parseDouble(args[2])
14                                                      : Math.pow(0.8, boardSize) * 2;
15                          /* Create an object capable of solving knight's tours */
16                          TourSolver t = new TourSolver(boardSize, new
                                Position(initSquare), randomness);
17                          Position[] solution = t.getSolution();
18                          if (solution != null) {
19                                  showBoard(t.getBoard());
20                                  showMoves(solution);
21                                  if (isClosed(solution))
22                                          System.out.println("\nThe tour is Closed!");
23                          } else {
24                                  System.out.println("No Knight's Tours found!");
25                          }
26                  } catch (Exception e) {
27                          /* Handle missing or incorrectly formatted arguments */
28                          System.out.print("Enter an integer (> 1) as the first
                                argument, ");
29                          System.out.println("and a well formed chessboard coordinate as
                                the second!");
30                          System.out.println("                              (size,
                                startSquare * , randomness * )");
31                          System.out.println();
```

```
32          System.out.println("(size      -> Solve a Tour on a (size x
                size) board)");
33          System.out.println("(startSquare * -> A square in algebraic
                chess notation of the form 'fr',");
34          System.out.println("                where f = the letter
                representing the file(column)");
35          System.out.println("                and  r = the number
                representing the rank(row).)");
36          System.out.println("(startSquare is set to 'a1' by default)");
37          System.out.println("(randomness * -> A number between 0(no
                randomness) and 1(even chances),");
38          System.out.println("                determining the randomness in
                ranking positions of");
39          System.out.println("                the same weightage while
                searching. A randomness of 0 will");
40          System.out.println("                produce the same tour every
                time, for a specific size and");
41          System.out.println("                startSquare. Keep extremely
                small values of randomness for");
42          System.out.println("                very large boards.)");
43          System.out.println("(randomness is set to 2 * (0.8)^boardSize
                by default)");
44          System.out.println();
45          System.out.println("                                          <
                * = optional arguments >");
46        }
47     }
48
49     /* Display the board, with each square marked with the move number on which
50        the knight landed on it */
51     public static void showBoard (int[][] board) {
52          String hLine = " " + multiplyString("+-----", board.length) + "+";
53          System.out.println(hLine);
54          for (int column = board.length - 1; column >= 0; column--) {
55               System.out.printf(" %2d ", column + 1);
56               for (int row = 0; row < board.length; row++) {
57                    System.out.printf("| %3d ", board[row][column]);
58               }
59               System.out.printf("|%n%s%n", hLine);
60          }
61          System.out.print(" ");
62          for (int i = 0; i < board.length; i++) {
63               System.out.printf(" %2s ", Position.xToString(i));
64          }
65          System.out.println();
66     }
```

```java
        /* Display the list of moves in the tour in algebraic notation */
        public static void showMoves (Position[] moves) {
                System.out.print("\nMoves : ");
                String movesOut = "";
                for (int i = 1; i < moves.length; i++) {
                        movesOut += (moves[i-1] + "-" + moves[i] + ", ");
                }
                System.out.println(movesOut.substring(0, movesOut.length() - 2));
        }

        /* Utility function for repeating strings */
        public static String multiplyString (String s, int n) {
                String result = "";
                while (n --> 0)
                        result += s;
                return result;
        }

        /* Check whether a tour is closed or not */
        public static boolean isClosed (Position[] path) {
                int l = path.length - 1;
                int dX = Math.abs(path[0].getX() - path[l].getX());
                int dY = Math.abs(path[0].getY() - path[l].getY());
                return (dX == 1 && dY == 2) || (dX == 2 && dY == 1);
        }
}
```

Variable Description

TourSolver		
int	size	Number of files/ranks in the chessboard
Position[]	path	Stack of moves which are part of the solved tour
int	numberOfMoves	Counter variable, number of moves made in the solved tour
int[][]	board	An integer array of integer arrays, representing a chessboard, with each square marked with the move number at which the knight lands on it
int[][]	degreesOf Freedom	An integer array of integer arrays, representing a chessboard, with each square marked with the number of possible knight moves from it (on an empty board)
Position	initPosition	The position on the board the knight starts from
double	tieBreak Randomness	The degree to which a move in the path is randomly decided
int[][]	KNIGHT_MOVES	List of legal changes in the x and y positions of a knight
TourSolver::initBoard()		
int	i, j	Counter variables
TourSolver::initDegreesOfFreedom()		
int	i, j	Counter variables
TourSolver::addMove(Position)		
Position	p	The new position to add to the path stack
TourSolver::removeMove()		
Position	p	The position popped from the path stack
TourSolver::solve()		
Position[]	possible Moves	List of possible moves that can be added to the path stack
Position	move	Current move to evaluate in the path
TourSolver::sortMoves(Position[])		
Position[]	moves	List of moves to rank using Warnsdorf's heuristic
int	count	Total number of moves in moves
int	right	Counter variable
int	i	Counter variable

TourSolver::compareMoves(Position, Position)		
Position	a, b	Positions/moves to compare using Warnsdorf's heuristic
int	aCount, bCount	Respective number of possible legal moves for a and b
int	aFree, bFree	Respective number of possible legal moves on an empty board for a and b
TourSolver::swapMoves(int, int, Position[])		
int	x, y	The indices of the moves to swap
Position[]	moves	Array of moves containing the moves to be swapped
TourSolver::getPossibleMoves(Position)		
Position	start	Position from where possible moves are to be generated
int	i	Counter variable
int[]	move	Pair of legal changes in the x and y positions of a knight
int	x, y	New x and y positions of the knight
TourSolver::getPossibleMovesCount(Position)		
Position	start	Position from where possible moves are to be generated
Position	p	Possible position
TourSolver::isWithinBoard(int, int)		
int	x, y	The x and y positions on the board to verify
Position		
int	x, y	The x and y coordinates on the board encoded by the Position
Position::this(String)		
String	s	Chess position written in algebraic notation
int	x, y	The x and y coordinates on the board
int	i	Counter variable
Position::xToString(int)		
int	n	File (x position) to convert to algebraic notation
String	letters	n expressed as a base 26 number, digits starting from (a)

int	x	Counter variable, temporarily stores the file to convert
KnightTour::main(String[])		
int	boardSize	Number of files/ranks in the chessboard
String	initSquare	The position on the board the knight starts from (algebraic notation)
double	randomness	The degree to which a move in the path is randomly decided
TourSolver	t	An object capable of generating *knight's tours*
Position[]	solution	The solved sequence of moves in the *knight's tour*
KnightTour::showBoard(int[][])		
int[][]	board	An integer array of integer arrays, representing a chessboard, with each square marked with the move number at which the knight lands on it
String	hline	Horizontal line drawn to represent board squares
int	row, column, i	Counter variables
KnightTour::showMoves(Position[])		
Position[]	moves	The sequence of moves to display
int	i	Counter variable
KnightTour::multiplyString(String, int)		
String	s	The string to multiply
int	n	The number of times to multiply s
String	out	The string containing n copies of s
KnightTour::isClosed(Position[])		
Position[]	path	The solved sequence of moves in the *knight's tour*
int	l	Index of last move in path
int	dX, dY	Differences in x and y coordinates of the knight between the first and last moves

"Curiosity begins as an act of tearing to pieces, or analysis."

— **Samuel Alexander**

Problem 9   Calculate the *square root* of a given positive number, using only *addition, subtraction, multiplication* and *division*.

Solution   The problem of finding the *square root* of a positive real number k is equivalent to finding a positive root of the function $f : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$

$$f(x) = x^2 - k$$

This problem can be solved using *Newton's method. Newton's method* is an iterative process for finding a root of a general function $f : \mathbb{R} \to \mathbb{R}$ by creating an initial guess, then improving upon it.

Let f' denote the derivative of the function f. Thus, the equation of the tangent to the curve $f(x)$, drawn through the point $(x_n, f(x_n))$ is given by the following equation.

$$y = f'(x_n)(x - x_n) + f(x_n)$$

The idea here is that the *x-intercept* of this tangent will be a better approximation to the root of the function f. Setting $y = 0$, solving for x and renaming it to x_{n+1} yields the following expression.

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

Plugging in the required function for this problem, we have

$$x_{n+1} = x_n - \frac{x_n^2 - k}{2x_n}$$

Simplifying, we arrive at our expression for the term x_{n+1} in our iterative process.

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{k}{x_n}\right)$$

This is the sort of simple expression we have been looking for, involving only one addition and two multiplications per iteration. As n becomes very large, the term x_n approaches the *square root* of k.

46

main (number:FloatingPoint, maxIterations:Integer)
1. Call squareRoot(number, maxIterations). Store the result in root.
2. Display root, along with the error from the value calculated by the library function Math->sqrt(number).
3. **Exit**

squareRoot (n:FloatingPoint, maxIterations:Integer)
1. Store the initial guess n / 2 in the variable x.
2. For maxIterations times:
   (a) Calculate 0.5 * (x + (n / x)). Store the result back in x.
3. **Return x**

Source Code

```java
1  public class SquareRoot {
2      public static void main (String[] args) {
3          /* Parse the first command line argument as the number to square root
               */
4          double number = Double.parseDouble(args[0]);
5          /* Parse the second conmand line argument as the number of iterations.
6             Default to 100 */
7          int maxIterations = (args.length > 1)? Integer.parseInt(args[1]) :
               100;
8
9          double root = squareRoot(number, maxIterations);
10         double library_root = Math.sqrt(number);
11
12         /* Display the calculated root, along with a comparison with the
13            library calculated value */
14         System.out.printf("Calculated square root : %f%n", root);
15         System.out.printf("System library square root : %f%n", library_root);
16         System.out.printf("Error : %f%n", (root - library_root));
17     }
18
19     public static double squareRoot (double n, int maxIterations) {
20         /* Handle edge cases, ignore negative values */
21         if (n < 0)
22             return Double.NaN;
23         if (n == 0)
24             return 0.0;
25         /* Start by guessing half of the number */
26         double x = n / 2;
```

```
27              for (int i = 0; i < maxIterations; i++) {
28                  x = 0.5 * (x + (n / x));
29              }
30              return x;
31          }
32      }
```

Variable Description

SquareRoot::main(String[])		
double	number	Stores the number whose square root is to be extracted
int	maxIterations	Stores the number of iterations for which Newton's method is to be applied
double	root	Stores the calculated square root of number
double	library_root	Stores the square root of number given by the Java library
SquareRoot::squareRoot(double, int)		
double	x	Stores the results of successive iterations of Newton's method
int	i	Counter variable

"Objects are abstractions of processing. Threads are abstractions of schedule."

— **James O. Coplien**

Problem 10   Let a *fraction* here be restricted to the ratio of two integers, m and n, where $n \neq 0$. Thus, a fraction $\frac{m}{n}$ is said to be reduced its *lowest terms* when m and n are relatively prime.

Implement this model of *fractions*, such that they are *immutable* and reduced to their *lowest terms* by default. Also implement a simple method for adding two *fractions*.

Solution   The problem of reducing a fraction $\frac{m}{n}$ to its lowest terms can be solved simply by dividing the numerator and the denominator by their *greatest common divisor*, i.e., $\gcd(m,n)$. This works as $\gcd(p,q) = 1$ if and only if p and q are relatively prime. Fraction addition can also be implemented using the following formula.

$$\frac{a}{b} + \frac{c}{d} = \frac{ad + bc}{bd}$$

The *greatest common divisor* of two integers can be calculated recursively using *Euclid's algorithm*.

$$\gcd(a, b) = \gcd(b, a \bmod b)$$

```
main ()
```
1. Create 2 `Fraction` objects `a` and `b` using data supplied by the user.
2. Call `Fraction->addFractions(a, b)`. Store the result in another `Fraction` object `sum`.
3. Display `a`, `b` and `sum`.
4. **Exit**


```
Fraction (numerator:Integer, denominator:Integer)
```
1. Set internal variables `numerator` and `denominator`, keeping them private.
2. Reduce the fraction to its lowest form.
   (a) Calculate the *greatest common divisor* of `numerator` and `denominator`, then divide each by the result.
   (b) Shift any negative sign in `denominator` to `numerator`.
3. **Define** the function `Fraction::addFractions(fraction1, fraction2)`, and **return** the resultant object.

```
Fraction::addFractions (fraction1:Fraction, fraction2:Fraction)
```
1. Calculate the `numerator` and `denominator` of the sum using the formula discussed above.
2. Create a new `Fraction` object using the calculated `numerator` and `denominator`, then **return** it.

Source Code

```
1  public class Fraction {
2
3          /* Store the numerator and denominator */
4          protected int numerator;
5          protected int denominator;
6
7          public Fraction (int numerator, int denominator) {
8                  /* Handle invalid fractions */
9                  if (denominator == 0)
10                         throw new ArithmeticException("Division by zero!");
11
12                 this.numerator = Math.abs(numerator);
13                 this.denominator = Math.abs(denominator);
14
15                 if (numerator != 0) {
16                         /* Reduce to lowest terms */
```

```
17                         int g = gcd(this.numerator, this.denominator);
18                         this.numerator /= g;
19                         this.denominator /= g;
20                         /* Make sure that the sign is on the numerator */
21                         this.numerator *= Math.signum(numerator) *
                               Math.signum(denominator);
22                 } else {
23                         /* Make sure all 'zero fractions' are the same */
24                         this.denominator = 1;
25                 }
26         }
27
28         public int getNumerator () {
29                 return this.numerator;
30         }
31
32         public int getDenominator () {
33                 return this.denominator;
34         }
35
36         /* Return a String representation of the Fraction for display */
37         public String toString () {
38                 /* Format all fractions with denominator '1' as simple integers */
39                 if (this.denominator == 1)
40                         return this.numerator + "";
41                 return this.numerator + " / " + this.denominator;
42         }
43
44         /* Add 2 Fraction objects */
45         public static Fraction addFractions (Fraction a, Fraction b) {
46                 int sumNumerator = (a.getNumerator() * b.getDenominator()) +
                       (a.getDenominator() * b.getNumerator());
47                 int sumDenominator = a.getDenominator() * b.getDenominator();
48                 return new Fraction(sumNumerator, sumDenominator);
49         }
50
51         /* Calculate the greatest common divisor of integers, using Euclid's method
               recursively */
52         private static int gcd (int p, int q) {
53                 return (p < q)? gcd(q, p) : ((p % q) == 0)? q : gcd(q, p % q);
54         }
55 }
```

```
1   import java.util.Scanner;
2
3   public class FractionAdder {
4        public static void main (String[] args) {
5              Scanner inp = new Scanner(System.in);
6              try {
7                    /* Get the two fractions from user input */
8                    System.out.print("Enter the numerator and denominator [integer
                         integer] of the first fraction : ");
9                    Fraction a = new Fraction(inp.nextInt(), inp.nextInt());
10                   System.out.print("Enter the numerator and denominator [integer
                         integer] of the second fraction : ");
11                   Fraction b = new Fraction(inp.nextInt(), inp.nextInt());
12
13                   /* Calculate and display the sum of the fractions.
14                      Here, we take advantage of the toString() method defined for
                          Fractions */
15                   Fraction sum = Fraction.addFractions(a, b);
16                   System.out.printf("%n(%s) + (%s) = (%s) %n", a, b, sum);
17             } catch (ArithmeticException e) {
18                   System.out.println("Invalid fraction - division by zero!");
19             }
20        }
21  }
```

Variable Description

Fraction		
int	numerator	Stores the numerator of the fraction
int	denominator	Stores the denominator of the fraction
Fraction(int, int)		
int	g	Stores the greatest common divisor of `numerator` and `denominator`
Fraction::addFractions(Fraction, Fraction)		
Fraction	a, b	The two fractions to be added
int	sumNumerator	The numerator of the sum
int	sumDenominator	The denominator of the sum
FractionAdder::main(String[])		
Scanner	inp	The input managing object
Fraction	a, b	The two fractions to be added
Fraction	sum	The sum of the fractions a and b

Problem 11   A rational number q can be broken down into a *simple continued fraction* in the form given below.

$$a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{\ddots + \cfrac{1}{a_n}}}}$$

This may be represented by the abbreviated notation $[a_0; a_1, a_2, \ldots, a_n]$. For example, $[0; 1, 1, 2, 1, 4, 2]$ is shorthand for the following.

$$\frac{42}{73} = 0 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{2 + \cfrac{1}{1 + \cfrac{1}{4 + \cfrac{1}{2}}}}}}$$

Calculate the *simple continued fraction* expression for a given, positive fraction.

Solution   We can thus solve this problem recursively by noting that the following holds.

$$\frac{p}{q} = \underbrace{\left\lfloor \frac{p}{q} \right\rfloor}_{\text{Integer part}} + \underbrace{\frac{p \bmod q}{q}}_{\text{Fractional part}}$$

Thus, by defining $f(\frac{p}{q})$ as the continued fraction representation of the fraction $\frac{p}{q}$, we can write

$$f\left(\frac{p}{q}\right) = \left\lfloor \frac{p}{q} \right\rfloor + f\left(\frac{q}{p \bmod q}\right)$$

Here, we are going to use the `Fraction` class defined in the solution to **Problem 10**, in order to take advantage of the reduced form and sign checks it carries out.

main (numerator:Integer, denominator:Integer)
1. Pack numerator and denominator into a Fraction object. Store it as f.
2. Call getContinuedFraction(f). Display the returned String.
3. **Exit**

getContinuedFraction (Fraction f)
1. Unpack numerator and denominator from f.
2. Call getContinuedFraction(numerator, denominator). Store the returned String in the variable expansion.
3. Replace the first comma (,) in expansion with a semicolon (;).
4. **Return** expansion

getContinuedFraction (numerator:Integer, denominator:Integer)
1. **If** denominator is 1, **return** numerator.
2. Calculate the integer part of numerator / denominator. Store it in x.
3. Call getContinuedFraction(denominator, numerator % denominator). Store the result in y.
4. **Return** x + y

Source Code

```
1  public class ContinuedFraction {
2      public static void main (String[] args) {
3          try {
4              /* Parse command line arguments as the numerator and
                  denominator
5                 of the fraction */
6              int numerator = Integer.parseInt(args[0]);
7              int denominator = Integer.parseInt(args[1]);
8              System.out.println(getContinuedFraction(new
                  Fraction(numerator, denominator)));
9          } catch (ArithmeticException e) {
10             System.out.println("Invalid fraction - division by zero!");
11         } catch (Exception e) {
12             System.out.println("Enter 2 arguments! ([numerator]
                  [denominator])");
13         }
14     }
15
16     /* Return the String representation of the continued fraction */
17     public static String getContinuedFraction (Fraction f) {
```

```
18                String expansion = "[" + getContinuedFraction(f.getNumerator(),
                      f.getDenominator());
19                /* By convention, the first comma is replaced with a semicolon */
20                return expansion.replaceFirst(",", ";");
21        }
22
23        /* Recursively calculate the continued fraction representation */
24        public static String getContinuedFraction (int numerator, int denominator) {
25                /* Base case : the fraction is now irreducible */
26                if (denominator == 1)
27                        return numerator + "]";
28                /* Pull out the integer part, invert the fraction and recurse */
29                return (numerator / denominator) + ", " +
                      getContinuedFraction(denominator, numerator % denominator);
30        }
31  }
```

Variable Description

ContinuedFraction::main(String[])		
int	numerator	Stores the numerator of the fraction to evaluate
int	denominator	Stores the denominator of the fraction to evaluate
ContinuedFraction::getContinuedFraction(Fraction)		
Fraction	f	Stores the fraction to evaluate
String	expansion	Stores the continued fraction representation of f

"Intelligence is the ability to avoid doing work, yet getting the work done."

— **Linus Torvalds**

Problem 12   The *binomial coefficient* [9] of two integers $n \geq k \geq 0$ is defined as follows.

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

Here, $n!$ is the *factorial* of n, defined as follows.

$$n! = 1 \times 2 \times 3 \times \cdots \times (n-2) \times (n-1) \times n$$

Compute the binomial coefficient for two given integers.

Solution   Note that we can rewrite the definition of the binomial by cancelling out common factors from the factorials.

$$\binom{n}{k} = \frac{n(n-1)(n-2)\cdots(n-(k-1))}{k(k-1)(k-2)\cdots 1}$$

Now that we have this definition, it is easy to see that we can separate the term $\frac{n}{k}$ and leave behind a smaller binomial coefficient. Thus, we arrive at the recursive formula

$$\binom{n}{k} = \frac{n}{k} \cdot \binom{n-1}{k-1}$$

Coupled with the observation that $\binom{n}{0} = 1$, we can solve this problem recursively.

We can introduce a small optimisation by observing that $\binom{n}{k} = \binom{n}{n-k}$. Thus, for $k > \frac{n}{2}$, we can replace k with $n-k$ to reduce the number of recursive calls.

---

[9]They are given this name as they describe the coefficients of the expansion of powers of a binomial, according to the *binomial theorem*.

$$(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^{n-k} y^k$$

main (n:Integer, k:Integer)
1. Call and display binomial(n, k).
2. **Exit**

binomial (n:Integer, k:Integer)
1. **If** k is zero, **return** 1.
2. **If** k exceeds half of n, call binomial(n, n - k).
3. **Return** binomial(n - 1, k - 1) * (n / k).

Source Code

```java
public class Binomial {
    public static void main (String[] args) {
        try {
            /* Parse the command line arguments as the terms in the
               binomial coefficient */
            long n = Long.parseLong(args[0]);
            long k = Long.parseLong(args[1]);
            System.out.println(binomial(n, k));

        } catch (NumberFormatException | IndexOutOfBoundsException e) {
            System.out.println("Enter 2 arguments! ([+integer]
                [+integer])");
        } catch (Exception e) {
            System.out.println("Invalid 'k'! (0 <= k <= n)");
        }
    }

    /* Recursively calculate the binomial coeffecient n choose k */
    public static long binomial (long n, long k) throws Exception {
        /* Invalid case */
        if (k > n)
            throw new Exception();
        /* Base case : n choose 0 is 1 */
        if (k == 0)
            return 1;
        /* Optimisation to reduce the number of recursive steps by reflecting
           k along the middle of n */
        if (k > (n / 2))
            return binomial(n, n - k);
        /* Recurse by unfolding the multiplication */
        return (n * binomial(n - 1, k - 1) / k);
    }
}
```

Variable Description

Binomial::main(String[])		
long	n, k	The arguments for calculating the binomial coefficient

Binomial::binomial(long, long)		
long	n, k	The arguments for calculating the binomial coefficient

"If people do not believe that mathematics is simple, it is only because they do not realize how complicated life is."

— **John von Neumann**

Problem 13 Palindromes can be generated in many ways. One of them involves picking a number, reversing the order of its digits and adding the result to the original. For example, we have

$$135 + 531 \;=\; 666$$

Not all numbers will yield a palindrome after one step. Instead, we can repeat the above process, using the sum obtained as as the new number to reverse.

$$963 + 369 \;=\; 1332$$
$$1332 + 2331 \;=\; 3663$$

This process is often called the *196-algorithm*. Some numbers seem never to yield a palindrome even after millions of iterations. These are called *Lychrel numbers*. The smallest of these in base 10 is conjectured to be the number 196, although none have been mathematically proven to exist.

Generate the steps and final palindrome of the *196-algorithm*, given a natural number as a *seed* [10].

Solution This problem can be solved without much complication. We can either create a loop, or use *tail recursion* [11] to roll up the process. The only problem here is that the numbers involved grow very large, very fast. Thus, care must be taken while dealing with such cases. Here, a library method for addition has been used to identify integer overflow.

---

[10]A *seed* is an initial number, from which subsequent numbers are generated.

[11]*Tail recursion* involves the use of *tail calls*. These are simply recursive function calls which appear as the last statement of the function body. Most programming languages can optimize tail recursion internally into a simple loop, thus avoiding the addition of stack frames on each recursive call.

```
main (number:Integer)
    1. Call generatePalindrome(number, 0).
    2. Exit

generatePalindrome (n:Integer, step:Integer)
    1. Reverse the digits in n. Store the result in r.
    2. If n is equal to r:
        (a) Display n as a palindrome, along with step.
        (b) Return
    3. Add n and r. Store the sum in the variable sum.
    4. Call generatePalindrome(sum, step + 1)
```

Source Code

```java
 1  class PalindromeGenerator {
 2      public static void main (String[] args) {
 3          /* Parse the first command line argument as the seed */
 4          long n = Long.parseLong(args[0]);
 5          generatePalindrome(n, 0);
 6      }
 7
 8      public static void generatePalindrome (long n, int step) {
 9          long r = reverse(n);
10          if (n == r) {
11              /* Base case : palindrome reached */
12              System.out.printf("%d is a palindrome (%d step%s)%n", n, step,
                     ((step == 1)? "" : "s"));
13          } else {
14              try {
15                  /* Use a library method to add. This will throw an
16                     Exception in case of overflow, which would have
17                     otherwise been ignored */
18                  long sum = Math.addExact(n, r);
19                  System.out.printf("%d + %d = %d%n", n, r, sum);
20                  /* Recurse via tail recursion, simply incrementing the
                        step value */
21                  generatePalindrome(sum, step + 1);
22              } catch (ArithmeticException e) {
23                  /* Stop if the numbers become too big */
24                  System.out.printf("Long Overflow - Sum exceeded maximum
                     size at step %d%n", step);
25              }
```

```
26                }
27            }
28
29            /* Reverse the integer supplied */
30            public static long reverse (long n) {
31                long r = 0;
32                while (n > 0) {
33                    /* Pull out the last digit and accumulate it on another
                           variable */
34                    r = (r * 10) + (n % 10);
35                    n /= 10;
36                }
37                return r;
38            }
39    }
```

Variable Description

PalindromeGenerator::main(String[])		
long	n	Stores the *seed* for the palindrome generation

PalindromeGenerator::generatePalindrome(long, int)		
long	n	Stores the current number to generate a palindrome from
long	r	Stores the reverse of n
int	step	Stores the step of the generation currently executing
long	sum	Stores the sum of n and r

PalindromeGenerator::reverse(long)		
long	r	Stores the reverse of n

"Over thinking leads to problems that don't even exist in the first place."

— **Jayson Engay**

Problem 14   Compute the *prime factorization* of a given natural number.

Solution   This solution is meant to showcase the drawbacks of using *recursion* in some problems.

Let $f(n)$ denote the expansion of the *prime factorization* of the natural number n. We *could* observe that if we can find naturals p and q such that $n = pq$, we can write

$$f(pq) \;=\; f(p) + f(q)$$

Using this, we can wrap up the iteration over the naturals into a recursive function.

The problem with this approach is that for moderately large numbers, the number of nested calls grows rapidly. For large enough numbers, the default memory allocated for the *call stack* by the *Java Virtual Machine* falls woefully short. As a result, it becomes necessary to manually set the size of the *thread stack size* by passing the `-Xss<size>` option to the *JVM* during program execution.

`main (number:Integer)`
1. Call and display `factorize(number, 2)`.
2. **Exit**

`factorize (n:Integer, next:Integer)`
1. **If n** is one, **return** an empty `String`.
2. **If next** exceeds, or is equal to, **n**, **return next**.
3. **If next** divides **n**:
    (a) Append **next** to the `String` returned by the call `factorize(n / next, next)`.
    (b) **Return** the above value.
4. **Return** `factorize(n, next + 1)`

Source Code

```
1  public class Factorize {
2      public static void main (String[] args) {
3          /* Parse the first command line argument as the number to factorize */
4          int number = Integer.parseInt(args[0]);
5          /* Start from 2 */
```

```
6                System.out.println(factorize(number, 2));
7          }
8
9          /* Return the String representation of the prime factorization of an integer
             */
10         public static String factorize (int n, int next) {
11               /* Base case 1 : nothing to factorize */
12               if (n == 1)
13                     return "";
14               /* Base case 2 : reached a prime */
15               if (next >= n)
16                     return next + "";
17               /* Check for a factor */
18               if ((n % next) == 0)
19                     return next + " " + factorize(n / next, next);
20               /* Recurse by incrementing the next 'factor' to check */
21               return factorize(n, next + 1);
22          }
23   }
```

Variable Description

Factorize::main(String[])		
int	number	Stores the number to be factorized
Factorize::main(String[])		
int	n	Stores the current number to be factorized
int	next	Stores the next number to check for divisibility

*"Meaning lies as much
in the mind of the reader
as in the Haiku."*

— **Douglas Hofstadter**

Problem 15 A *codebook* is a document which stores a *lookup table* for coding and decoding text – each word has a different word, phrase or string to replace it. Design a system which, when given a *codebook* written in plaintext, translates a given sentence into its encoded form.

Solution Solving this problem requires careful reading of the supplied codebook. Here, the following format is assumed.

```
word          codeword
next_word     other_codeword
.             .
.             .
```

Thus, this data can be transformed into an *array*, which can then be searched for strings appearing in the supplied input.

`main (codebook:String)`
1. Create a `CodeSubstituter` object, pass it the filename `codebook`, and assign it to `cs`.
2. Get a line of user input. Store it in `sentence`.
3. Split `sentence` along whitespace into the `String` array `words`.
4. For each `word` in `words`:
   (a) Call `cs->getEncodedText(word)`. Store the result in `encodedText`.
   (b) Display `encodedText`.
5. **Exit**

`CodeSubstituter (codebook:String)`
1. Open the file pointed to by `codebook`. Start from the beginning in read mode.
2. On the first pass through `codebook`, count the number of lines and store the result in `numberOfLines`.
3. Close, and reopen `codebook`. Start at the beginning.
4. Initialize a `2` column `String` array, with `numberOfLines` as the number of rows. Assign it to `wordMap`.

5. Start reading `codebook` again. For each line, stored in `line` and each row in `wordMap` :
    (a) Split `line` along whitespace.
    (b) Store the first half in the first column of `wordMap`, and the second half in the second column of the same.
6. Close the file `codebook`.
7. **Define** the function `CodeSubstituter::getEncodedText(word)` and **return** the resultant object.

`CodeSubstituter::getEncodedText (word:String)`
1. For each row in `wordMap`:
    (a) If the first column entry matches `word`, **return** the second column entry.
2. **Return** `word`

Source Code

```java
import java.io.IOException;
import java.io.FileReader;
import java.io.BufferedReader;

public class CodeSubstituter {
        protected String filename;

        protected int numberOfLines;
        protected String[][] wordMap;

        /* Create a codebook from a supplied file */
        public CodeSubstituter (String filename) throws IOException {
                this.filename = filename;
                countNumberOfLines();
                initWordMap();
        }

        /* Calculate the number of lines to store on the first pass */
        private void countNumberOfLines () throws IOException {
                FileReader fileReader = new FileReader(filename);
                BufferedReader bufferedReader = new BufferedReader(fileReader);

                numberOfLines = 0;
                /* Keep incrementing the accumulator while lines are available */
                while (bufferedReader.readLine() != null)
                        numberOfLines++;

```

```java
28              bufferedReader.close();
29              fileReader.close();
30          }
31
32          /* Initialize the map/dictionary by reading the file on the second pass */
33          private void initWordMap () throws IOException {
34              wordMap = new String[numberOfLines][2];
35
36              FileReader fileReader = new FileReader(filename);
37              BufferedReader bufferedReader = new BufferedReader(fileReader);
38
39              for (int i = 0; i < numberOfLines; i++) {
40                  /* Split a line along whitespace */
41                  String[] words = bufferedReader.readLine().split("\\s+");
42                  if (words.length >= 2) {
43                      wordMap[i][0] = words[0];
44                      wordMap[i][1] = words[1];
45                  } else {
46                      /* Ignore empty lines */
47                      wordMap[i][0] = wordMap[i][1] = "";
48                  }
49              }
50
51              bufferedReader.close();
52              fileReader.close();
53          }
54
55          /* Returns the codeword, given a plain word */
56          public String getEncodedText (String word) {
57              /* Iterate through all entries */
58              for (int i = 0; i < numberOfLines; i++) {
59                  if (wordMap[i][0].equalsIgnoreCase(word)) {
60                      return wordMap[i][1];
61                  }
62              }
63              /* Reflect the original back if not found in the codebook */
64              return word;
65          }
66  }
```

```java
1   import java.util.Scanner;
2   import java.io.IOException;
3   import java.io.FileNotFoundException;
4
5   public class TextEncoder {
6       public static void main (String[] args) throws Exception {
7           try {
8               /* Parse the first command line argument as the path to the
                   codebook */
9               CodeSubstituter cs = new CodeSubstituter(args[0]);
10
11              /* Get a sentence to encode, and extract the individual words
                   */
12              System.out.print("Enter a sentence to encode : ");
13              String sentence = (new Scanner(System.in)).nextLine();
14              String[] words = sentence.split("\\s+");
15
16              System.out.print("Encoded sentence    : ");
17              /* Iterate through each word, replacing it with the codeword
                   in the codebook */
18              for (int i = 0; i < words.length; i++) {
19                  String encodedText =
                        cs.getEncodedText(words[i].toLowerCase().replaceAll("[^a-z]",
                        ""));
20                  System.out.print(encodedText + " ");
21              }
22              System.out.println();
23          } catch (ArrayIndexOutOfBoundsException e) {
24              System.out.println("Enter 1 argument ([codebook_filename])");
25          } catch (FileNotFoundException e) {
26              System.out.println("Codebook not found! Enter a valid
                   filename.");
27          } catch (IOException e) {
28              e.printStackTrace();
29          }
30      }
31  }
```

Variable Description

CodeSubstituter		
String	filename	Stores the path of the file containing the codebook
int	numberOfLines	Stores the number of lines in the file `filename`
String[][]	wordMap	A table of plain words and their corresponding code-words

CodeSubstituter::countNumberOfLines()		
FileReader	fileReader	An object for reading character based files
Buffered Reader	bufferedReader	An object for buffering character streams

CodeSubstituter::initWordMap()		
FileReader	fileReader	An object for reading character based files
Buffered Reader	bufferedReader	An object for buffering character streams
String[]	words	Temporarily stores the parts of a line in the codebook

TextEncoder::main(String[])		
Code Substituter	cs	An object for accessing a codebook
String	sentence	Stores a line of user input to be encoded
String[]	words	Stores the list of words in `sentence`

"Hofstadter's Law: It always takes longer than you expect, even when you take into account Hofstadter's Law."

— **Douglas Hofstadter**

Problem 16 Analyse the frequency of each letter in the English alphabet appearing in a given file. Store the results in a different file.

Solution All that has to be done here is reading the contents of a file, counting the occurrences of each character, then tabulating the results before writing them to another file. Here, the characters have also been sorted based on their frequencies.

main (fromFile:String, toFile:String)
1. Create a `CharacterCounter` object, pass it `fromFile`, and assign it to `cc`.
2. Call `cc->writeReportToFile(toFile)`.
3. **Exit**

CharacterCounter (fromFile:String)
1. Read all the lines from the file `fromFile` and store the resultant `String` in `fileData`.
2. Initialize a 26 row `Character` array `letters`, as well as a 26 row `Integer` array `letterCount`.
3. For each letter $c \in \{a, b, \ldots, z\}$:
    (a) Store `c` in an empty row in `letters`.
    (b) Count the number of occurrences of `c` in `fileData`. Store the result in the corresponding row in `letterCount`.
    (c) Move to a new row in `letters` and `letterCount`.
4. Store the sum of all entries in `letterCount` in the variable `totalLetters`.
5. Sort the entries in `letters` and `letterCount`, in descending order of the entries in `letterCount` using *bubble sort*.
6. **Define** the function `CharacterCounter::writeReportToFile(toFile)` and **return** the resultant object.

CharacterCounter::writeReportToFile (toFile:String)
1. Open the file pointed to by `toFile`. Start from the beginning in write mode.
2. Write all entries in `letters` and `letterCount`, formatted to include the ratio of the entry in `letterCount` to `totalLetters`.

69

3. Write `totalLetters` to `toFile`, along with any entry in `letters` whose corresponding entry in `letterCount` is zero.
4. Close the file `toFile`.
5. **Return**

Source Code

```java
import java.io.IOException;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.PrintWriter;

public class CharacterCounter {
        protected String filename;

        protected String fileData;
        protected char[] letters;
        protected int[] letterCount;
        protected int totalLetters;

        /* Create a table of letter counts in a given file */
        public CharacterCounter (String filename) throws IOException {
                this.filename = filename;
                this.fileData = "";
                this.letterCount = new int[26];
                this.letters = new char[26];
                this.totalLetters = 0;
                getFileData();
                countAllLetters();
                sortLetters();
        }

        /* Read all lines in the file and store them in a String */
        private void getFileData () throws IOException {
                FileReader fileReader = new FileReader(filename);
                BufferedReader bufferedReader = new BufferedReader(fileReader);

                String line = "";
                while ((line = bufferedReader.readLine()) != null)
                        fileData += line.toLowerCase();

                bufferedReader.close();
                fileReader.close();
```

```
39            }
40
41            /* Return the number of occurrences of a character in the file */
42            public int getCountOf (char c) {
43                    int count = 0;
44                    for (int i = 0; i < fileData.length(); i++) {
45                            if (fileData.charAt(i) == c) {
46                                    count++;
47                            }
48                    }
49                    return count;
50            }
51
52            /* Compile the counts of all letters in the file */
53            public void countAllLetters () {
54                    for (char c = 'a'; c <= 'z'; c++) {
55                            letters[c - 'a'] = c;
56                            letterCount[c - 'a'] = getCountOf(c);
57                            totalLetters += letterCount[c - 'a'];
58                    }
59            }
60
61            /* Sort the entries by frequency (bubble sort) */
62            private void sortLetters () {
63                    for (int right = 26; right > 0; right--)
64                            for (int i = 1; i < right; i++)
65                                    if (letterCount[i] > letterCount[i-1])
66                                            swap(i, i-1);
67            }
68
69            /* Utility swapping method */
70            private void swap (int i, int j) {
71                    char tmpChar = letters[i];
72                    int tmpCount = letterCount[i];
73                    letters[i] = letters[i-1];
74                    letterCount[i] = letterCount[i-1];
75                    letters[i-1] = tmpChar;
76                    letterCount[i-1] = tmpCount;
77            }
78
79            /* Create and write the final report to a file */
80            public void writeReportToFile (String toFilename) throws IOException {
81                    FileWriter fileWriter = new FileWriter(toFilename);
82                    BufferedWriter bufferedWriter = new BufferedWriter(fileWriter);
83                    PrintWriter printWriter = new PrintWriter(bufferedWriter);
84
```

```
 85                    /* Make sure the frequencies all fit, aligned in the same column */
 86                    int l = (totalLetters + "").length();
 87                    String unusedLetters = "";
 88                    for (int i = 0; i < 26; i++) {
 89                            /* Show the letter, frequency and percentage out of the total
                                  */
 90                            char c = letters[i];
 91                            int count = letterCount[i];
 92                            double percent = (count * 100.0) / totalLetters;
 93                            if (count > 0) {
 94                                    printWriter.printf("%c : %5.2f%% (%" + l + "d) %n",
 95                                                    c, percent, count);
 96                            } else {
 97                                    /* Separate unused letters */
 98                                    unusedLetters += c + " ";
 99                            }
100                    }
101                    printWriter.printf("Total letters : %d%n", totalLetters);
102                    if (unusedLetters.length() == 0)
103                            unusedLetters = "(none)";
104                    printWriter.printf("Unused letters : %s%n", unusedLetters);
105
106                    printWriter.close();
107                    bufferedWriter.close();
108                    fileWriter.close();
109            }
110 }
```

```
 1  import java.io.IOException;
 2  import java.io.FileNotFoundException;
 3
 4  public class AnalyseCharacterFrequency {
 5          public static void main (String[] args) {
 6                  try {
 7                          /* Parse the commnd line arguments as the file to analyse and
                               the
 8                            file to pipe the results into */
 9                          String fromFile = args[0];
10                          String toFile = args[1];
11
12                          /* Create and write the report */
13                          CharacterCounter cc = new CharacterCounter(fromFile);
14                          cc.writeReportToFile(toFile);
15                  } catch (ArrayIndexOutOfBoundsException e) {
16                          System.out.println("Enter 2 arguments! ([filename_from]
```

```
                        [filename_to])");
17              } catch (FileNotFoundException e) {
18                      System.out.println("Enter a valid filename!");
19              } catch (IOException e) {
20                      e.printStackTrace();
21              }
22          }
23  }
```

Variable Description

CharacterCounter		
String	filename	Stores the path of the file to analyse
String	fileData	Stores all character data from the file
char[]	letters	The list of all letters, in order of frequency
int[]	letterCount	The frequencies of each corresponding letter in `letters`
int	totalLetters	Stores the total number of letters in `fileData`
CharacterCounter::getFileData()		
FileReader	fileReader	An object for reading character based files
Buffered Reader	bufferedReader	An object for buffering character streams
String	line	Stores a line of text in the file
CharacterCounter::getCountOf(char)		
char	c	The character whose frequency is to be found in `fileData`
int	count	The frequency of `c` in `fileData`
CharacterCounter::countAllLetters()		
char	c	The character whose frequency is to be found
CharacterCounter::sortLetters()		
int	right, i	Counter variables
CharacterCounter::swap(int, int)		
int	i, j	Indices of `letters` and `letterCount` whose entries are to be swapped
CharacterCounter::writeReportToFile(String)		
String	toFilename	Stores the path of the file to write the report to
FileWriter	fileWriter	An object for writing character based files

Buffered Writer	`bufferedWriter`	An object for buffering character streams being written to a file
`PrintWriter`	`printWriter`	An object for writing data to an output stream
`int`	`l`	Stores the number of digits in `totalLetters`
`String`	`unusedLetters`	Stores the list of letters not present in `fileData`
`char`	`c`	Stores the current character being written
`int`	`count`	Stores the frequency of `c`
`double`	`percent`	Stores the percentage of `count` out of `totalLetters`
`AnalyseCharacterFrequency::main(String[])`		
`String`	`fromFile`	Stores the path of the file to analyse
`String`	`toFile`	Stores the path of the file to write the report to
Character Counter	`cc`	An object for analysing the frequencies of letters in files

"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$^{\text{th}}$ 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

```java
public class Mobius {
    /* Elements of a basic graph */
    public static final String[] graph =
            {"*        ",
             "    *    ",
             "        *"};
    public static void main (String[] args) {
        try {
            /* Parse the first command line argument as the lower limit */
            int lo = Integer.parseInt(args[0]);
            /* Parse the second commmand line argument as the upper limit
                */
            int hi = Integer.parseInt(args[1]);
            /* Incorrct input */
            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) {
            /* Handle missing or incorrectly formatted arguments */
            System.out.println("Enter 2 arguments (lower_limit[integer,
                >0], upper_limit[integer, >lower_limit])!");
        }
    }

    public static int mobius (int n) {
        /* Ignore negative numbers */
        if (n < 1)
            return 0;
        /* Trivial case */
        if (n == 1)
            return 1;
        /* Start with +1 */
        int mob = 1;
        for (int i = 2; i <= n; i++) {
            int multiplicity = 0;
            /* Count the number of times (i) appears */
            while ((n % i) == 0) {
                /* Reduce 'n' */
                n /= i;
                multiplicity++;
```

```
42                         }
43                         if (multiplicity == 1) {
44                                 /* Flip the sign */
45                                 mob = -mob;
46                         } else if (multiplicity > 1) {
47                                 /* Squared factor found */
48                                 return 0;
49                         }
50                 }
51             return mob;
52         }
53 }
```

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

"The mathematics is not there till we put it there."

— **Arthur Eddington**

Problem 18 A *set* is a collection of distinct objects. Implement a simple model of *sets*, capable of holding *integers*.

Solution This implementation uses *arrays* as the framework for storing elements. The set is sorted during insertion of elements, allowing for fast *binary searching*.

`Set (maxSize:Integer)`
1. Copy `maxSize` into the object data.
2. Initialize an array of integers `elements`, with length `maxSize`.
3. Initialize an integer `top` to `-1`.
4. **Define** the following functions:
   (a) `Set::updateMaxSize(newMaxSize)`
   (b) `Set::contains(n)`
   (c) `Set::add(n)`
   (d) `Set::remove(n)`
   (e) `Set::indexOfEqualOrGreater(n)`
5. **Return** the resultant object.

`Set::updateMaxSize (newMaxSize:Integer)`
1. Initialize an array of integers `temp`, with length `newMaxSize`.
2. Set `maxSize` to `newMaxSize`.
3. If the new size cannot accomodate the present elements of the set, discard them by setting `top` to `maxSize - 1`.
4. Copy all integers from indices `0` to `top` from `elements` to `temp`.
5. Set `elements` to `temp`.

`Set::contains (n:Integer)`
1. Call `this->indexOfEqualOrGreater(n)`. Call the returned value `i`.
2. If `i` is a valid index within the set, and the element at that index is equal to `n`, **return** true, otherwise **return** false.

`Set::add (n:Integer)`
1. Assert that the set is large enough to hold the new element.
2. If the set already contains `n`, **return** false.

3. Call `this->indexOfEqualOrGreater(n)`. Call the returned value `i`.
4. Shift all integers in `elements` from indices `i` to `top` one place to the right.
5. **Return true**

`Set::remove (n:Integer)`
1. Assert that the set is not empty.
2. If the set does not already contain `n`, **return false**.
3. Call `this->indexOfEqualOrGreater(n)`. Call the returned value `i`.
4. Shift all integers in `elemets` from indices `i + 1` to `top` one place to the left.
5. **Return true**

`Set::indexOfEqualOrGreater (n:Integer)`
1. Initialize an integer `hi` to `top + 1`.
2. Initialize an integer `lo` to `0`;
3. While `lo < hi`:
    (a) Set a temporary integer `mid` to `(lo + hi) / 2`.
    (b) If `n` is less than the element at mid, set `hi` to `mid`.
    (c) If `n` is greater than the element at mid, set `lo` to `mid + 1`.
    (d) If `n` is equal to the element at mid, **return mid**.
4. **Return hi**

`union (a:Set, b:Set)`
1. Create a new `Set`, capable of holding the combined elements of `a` and `b`. Call it `r`.
2. For each element `n` in `a`, call `r->add(n)`.
3. For each element `n` in `b`, call `r->add(n)`.
4. **Return r.**

`intersection (a:Set, b:Set)`
1. Create a new `Set`, with its `maxSize` equal to either of the sizes of `a` or `b`. Call it `r`.
2. For each element `n` in `a`, also contained in `n`, call `r->add(n)`.
3. **Return r**

`difference (a:Set, b:Set)`
1. Create a new `Set`, with its `maxSize` equal to either of the sizes of `a` or `b`. Call it `r`.
2. For each element `n` in `a`, not contained in `n`, call `r->add(n)`.

3. **Return r**

Source Code

```java
import java.util.Iterator;

public class Set implements Iterable<Integer> {
        protected int maxSize;

        /* Simple list setup */
        protected int[] elements;
        protected int top;

        /* Let the maximum capacity be specified during instantiation */
        public Set (int maxSize) {
                this.maxSize = maxSize;
                this.elements = new int[maxSize];
                this.top = -1;
        }

        /* Returns the number of elements in the set */
        public int getSize () {
                return top + 1;
        }

        /* Returns the maximum capacity of the set */
        public int getMaxSize () {
                return maxSize;
        }

        /* Expands or contracts the set as necessary, discards elements if
           they cannot be accomodated */
        public void updateMaxSize (int newMaxSize) {
                int[] temp = new int[newMaxSize];
                this.maxSize = newMaxSize;
                /* Make sure that the top index isn't out of bounds */
                this.top = Math.min(top, newMaxSize - 1);
                /* Copy data to the new list */
                for (int i = 0; i <= top; i++)
                        temp[i] = elements[i];
                this.elements = temp;
        }

        /* Checks whether an element is present in the set */
        public boolean contains (int n) {
                int i = indexOfEqualOrGreater(n);
```

```java
43              return ((i >= 0) && (i <= top) && (elements[i] == n));
44          }
45
46          /* Checks whether the set is empty */
47          public boolean isEmpty () {
48              return top < 0;
49          }
50
51          /* Clears all elements from the set */
52          public void clear () {
53              /* Only the top index has to be updated, since values byond it
54                 cannot be accessed */
55              this.top = -1;
56          }
57
58          /* Adds an element to the set. Returns 'false' if it is already
59             present, or there isn't enough space. */
60          public boolean add (int n) {
61              if (getSize() >= getMaxSize())
62                  return false;
63              /* Find the breakpoint to shift elements */
64              int i = indexOfEqualOrGreater(n);
65              if ((i >= 0) && (i <= top) && (elements[i] == n))
66                  return false;
67              /* Shift elements greater than 'n' to make room for it */
68              for (int j = top; j >= i; j--)
69                  elements[j + 1] = elements[j];
70              elements[i] = n;
71              top++;
72              return true;
73          }
74
75          /* Removes an element from the set. Returns 'false' if it isn't
76             already present. */
77          public boolean remove (int n) {
78              if (isEmpty())
79                  return false;
80              /* Find the location of the element */
81              int i = indexOfEqualOrGreater(n);
82              if ((i < 0) || (i > top) || (elements[i] != n))
83                  return false;
84              /* Shift elements into the desired element, erasing it */
85              for (int j = i; j < top; j++)
86                  elements[j] = elements[j + 1];
87              top--;
88              return true;
```

```
89          }

90

91          /* Returns the union of two sets */
92          public static Set union (Set a, Set b) {
93                  Set r = new Set(a.getSize() + b.getSize());
94                  /* The 'add' methods take care of duplicates */
95                  for (int n : a)
96                          r.add(n);
97                  for (int n : b)
98                          r.add(n);
99                  return r;
100         }

101

102         /* Returns the intersection of two sets */
103         public static Set intersection (Set a, Set b) {
104                 Set r = new Set(a.getSize());
105                 for (int n : a)
106                         if (b.contains(n))
107                                 r.add(n);
108                 return r;
109         }

110

111         /* Returns the difference of two sets */
112         public static Set difference (Set a, Set b) {
113                 Set r = new Set(a.getSize());
114                 for (int n : a)
115                         if (!b.contains(n))
116                                 r.add(n);
117                 return r;
118         }

119

120         /* Finds the index of the element equal to or greater than
121            the desired element via binary search */
122         private int indexOfEqualOrGreater (int n) {
123                 int hi = top + 1;
124                 int lo = 0;
125                 while (lo < hi) {
126                         int mid = (lo + hi) / 2;
127                         if (n < elements[mid])
128                                 hi = mid;
129                         else if (n > elements[mid])
130                                 lo = mid + 1;
131                         else
132                                 return mid;
133                 }
134                 return hi;
```

```
135          }
136
137          /* Format the set elements as a list */
138          @Override
139          public String toString () {
140                  if (getSize() == 0)
141                          return "[]";
142                  String s = "";
143                  for (Integer n : this)
144                          s += n + " ";
145                  return "[" + String.join(", ", s.split("\\s+")) + "]";
146          }
147
148          /* Allow 'Set' to be iterable, providing easy access to elements
149             without indexing */
150          @Override
151          public Iterator<Integer> iterator () {
152                  return new Iterator<Integer>() {
153                          private int currentIndex = 0;
154
155                          @Override
156                          public boolean hasNext () {
157                                  return currentIndex <= top;
158                          }
159
160                          @Override
161                          public Integer next () {
162                                  return elements[currentIndex++];
163                          }
164
165                          @Override
166                          public void remove () {
167                                  throw new UnsupportedOperationException();
168                          }
169                  };
170          }
171  }
```

```
1  public class SetDemo {
2        public static void main (String[] args) {
3                /* Create 3 sets with random elements */
4                Set a = new Set(10);
5                Set b = new Set(10);
6                Set c = new Set(10);
7                for (int i = 0; i < 10; i++)
```

```
8                          a.add((int) (Math.random() * 10));
9              for (int i = 0; i < 10; i++)
10                         b.add((int) (Math.random() * 10));
11             for (int i = 0; i < 10; i++)
12                         c.add((int) (Math.random() * 10));
13
14             /* Demonstrate simple output formatting */
15             System.out.printf("A [%2d] = %s\n", a.getSize(), a);
16             System.out.printf("B [%2d] = %s\n", b.getSize(), b);
17             System.out.printf("C [%2d] = %s\n", c.getSize(), c);
18             System.out.println();
19
20             /* Demonstrate set operations */
21             System.out.printf("A union B [%2d] = %s\n",
22                                  Set.union(a, b).getSize(),
23                                  Set.union(a, b));
24             System.out.printf("B union C [%2d] = %s\n",
25                                  Set.union(b, c).getSize(),
26                                  Set.union(b, c));
27             System.out.printf("C union A [%2d] = %s\n",
28                                  Set.union(c, a).getSize(),
29                                  Set.union(c, a));
30             System.out.printf("A union B union C [%2d] = %s\n",
31                                  Set.union(Set.union(a, b), c).getSize(),
32                                  Set.union(Set.union(a, b), c));
33             System.out.println();
34             System.out.printf("A intersection B [%2d] = %s\n",
35                                  Set.intersection(a, b).getSize(),
36                                  Set.intersection(a, b));
37             System.out.printf("B intersection C [%2d] = %s\n",
38                                  Set.intersection(b, c).getSize(),
39                                  Set.intersection(b, c));
40             System.out.printf("C intersection A [%2d] = %s\n",
41                                  Set.intersection(c, a).getSize(),
42                                  Set.intersection(c, a));
43             System.out.printf("A intersection B intersection C [%2d] = %s\n",
44                                  Set.intersection(Set.intersection(a, b),
45                                       c).getSize(),
                                     Set.intersection(Set.intersection(a, b), c));
46             System.out.println();
47             System.out.printf("A - B [%2d] = %s\n",
48                                  Set.difference(a, b).getSize(),
49                                  Set.difference(a, b));
50             System.out.printf("B - C [%2d] = %s\n",
51                                  Set.difference(b, c).getSize(),
52                                  Set.difference(b, c));
```

```
53              System.out.printf("C - A [%2d] = %s\n",
54                               Set.difference(c, a).getSize(),
55                               Set.difference(c, a));
56      }
57  }
```

Variable Description

Set		
int	maxSize	The maximum number of elements the set can hold
int[]	elements	The collection of elements contained in the set
int	top	The index of the topmost element in elements
Set::Set(int)		
int	maxSize	The maximum number of elements the set can hold
Set::updateMaxSize(int)		
int	newMaxSize	The maximum number of elements the set is to hold
int[]	temp	The new copy of elements with the updated size
Set::add(int)		
int	n	The element to be added to the set
int	i	The index of the breakpoint from which elements have to be shifted
Set::remove(int)		
int	n	The element to be removed from the set
int	i	The index of the breakpoint from which elements have to be shifted
Set::indexOfEqualOrGreater(int)		
int	n	The element to be searched for
int	hi	The upper index where n can be
int	lo	The lower index where n can be
int	mid	The midpoint of hi and lo

"Mathematics is the art of giving the same name to different things."

— **Henri Poincaré**

Problem 19 A *vector space* is a collection of objects called *vectors*, which may be added together and multiplied (scaled) by *scalara*. One way of implementing a *vector* is to describe the space $\mathbb{R}^n$, i.e. all possible ordered tuples of n real numbers. For example, the vector $(1, 7, 0, 1)$ belongs to the vector space $\mathbb{R}^4$ – it is a four-dimensional vector.

Addition, scalar multiplication, the dot product and the magnitude of vectors is defined as follows. $(a_i, b_i, k \in \mathbb{R})$

$$
\begin{aligned}
(a_1, a_2, \ldots, a_n) + (b_1, b_2, \ldots, b_n) &= (a_1 + b_1, a_2 + b_2, \ldots, a_n + b_n) & \text{(Addition)} \\
k\,(a_1, a_2, \ldots, a_n) &= (ka_1, ka_2, \ldots, ka_n) & \text{(Scalar Multiplication)} \\
(a_1, a_2, \ldots, a_n) \cdot (b_1, b_2, \ldots, b_n) &= a_1 b_1 + a_2 b_2 + \cdots + a_n b_n & \text{(Dot Product)} \\
\|(a_1, a_2, \ldots, a_n)\| &= \sqrt{a_1^2 + a_2^2 + \cdots + a_n^2} & \text{(Magnitude)}
\end{aligned}
$$

Implement a simple model of *vectors* as defined above.

Solution
```
Vector (components:FloatingPoint[])
```
  1. Set a constant integer `dimension` to the length of `components`.
  2. Copy `components` into the object data as a constant.
  3. **Define** the functions:
     (a) `Vector::getComponent(index)`
     (b) `Vector::getAbsoluteValue()`
  4. **Return** the resultant object.

```
Vector::getComponent (index:Integer)
```
  1. **Return** `components[index - 1]`

```
Vector::getAbsoluteValue ()
```
  1. Initialize a floating point `abs` to zero.
  2. For each `component` in , add `component * component` to `abs`.
  3. **Return** the square root of `abs`.

```
add (a:Vector, b:Vector)
```
  1. Assert that `a` and `b` have the same `dimension`.

86

2. Create an array of floating points `sum`, with length equal to their common `dimension`.
3. For each i ∈ {1, 2, . . . , dimension}:
    (a) Set `sum[i-1]` to `a->getComponent(i) + b->getComponent(i)`.
4. Create a new `Vector`, pass it `sum` and **return** the resultant object.

multiplyByScalar (v:Vector, k:FloatingPoint)
1. Create an array of floating points `t`, with length equal to the `dimension` of v.
2. For each i ∈ {1, 2, . . . , dimension}:
    (a) Set `t[i-1]` to `v->getComponent(i) * k`.
3. Create a new `Vector`, pass it `t` and **return** the resultant object.

dotProduct (a:Vector, b:Vector)
1. Assert that `a` and `b` have the same `dimension`.
2. Initialize a floating point `dotProduct` to zero.
3. For each i ∈ {1, 2, . . . , dimension}:
    (a) Add `a->getComponent(i) * b->getComponent(i)` to `dotProduct` .
4. **Return** dotProduct

Source Code

```java
public class Vector {
    /* Vector data is immutable */
    protected final int dimension;
    protected final double[] components;

    /* Use varargs to create an arbitrary dimensional vector */
    public Vector (double ... components) {
        this.dimension = components.length;
        this.components = new double[dimension];
        for (int i = 0; i < dimension; i++)
            this.components[i] = components[i];
    }

    /* Returns the dimensionality of the vector */
    public int getDimension () {
        return this.dimension;
    }

    /* Returns the component at the specified index.
       This uses indexing starting at '1' per mathematical convention */
    public double getComponent (int index) {
        return this.components[index - 1];
```

```java
23              }

25              /* Returns the absolute value/magnitude of the vector */
26              public double getAbsoluteValue () {
27                      double abs = 0.0;
28                      for (int i = 0; i < dimension; i++)
29                              abs += (components[i] * components[i]);
30                      return Math.sqrt(abs);
31              }

33              /* Wrapper methods which call static ones */

35              public Vector multiplyByScalar (double k) {
36                      return Vector.multiplyByScalar(this, k);
37              }

39              public Vector add (Vector v) {
40                      return Vector.add(this, v);
41              }

43              public double dotProduct (Vector v) {
44                      return Vector.dotProduct(this, v);
45              }

47              public double angleBetween (Vector v) {
48                      return Vector.angleBetween(this, v);
49              }

51              public boolean equals (Vector v) {
52                      return Vector.equals(this, v);
53              }

55              /* Format vector components neatly */
56              @Override
57              public String toString () {
58                      String s = "(";
59                      for (double component : components)
60                              s += component + ", ";
61                      return s.replaceAll(", $", ")");
62              }

64              /* Checks for equality between two vectors */
65              public static boolean equals (Vector a, Vector b) {
66                      /* Dimensionalities must be equal */
67                      if (a.getDimension() != b.getDimension())
68                              return false;
```

```java
                   /* Corresponding components must be equal */
                   for (int i = 1; i <= a.getDimension(); i++)
                       if (a.getComponent(i) != b.getComponent(i))
                           return false;
                   return true;
           }

           /* Multiplies a vector by a scalar to return a vector */
           public static Vector multiplyByScalar (Vector v, double k) {
                   double[] t = new double[v.getDimension()];
                   for (int i = 0; i < t.length; i++)
                       t[i] = v.getComponent(i+1) * k;
                   return new Vector(t);
           }

           /* Adds two vectors to return a vector */
           public static Vector add (Vector a, Vector b) {
                   double[] sum = new double[a.getDimension()];
                   /* Add corresponding components */
                   for (int i = 0; i < sum.length; i++)
                       sum[i] = a.getComponent(i+1) + b.getComponent(i+1);
                   return new Vector(sum);
           }

           /* Adds multiple vectors to return a vector */
           public static Vector add (Vector ... vectors) {
                   Vector v = vectors[0];
                   /* Repeatedly use binary the addition method */
                   for (int i = 1; i < vectors.length; i++)
                       v = Vector.add(v, vectors[i]);
                   return v;
           }

           /* Returns the dot product of two vectors */
           public static double dotProduct (Vector a, Vector b) {
                   double dotProduct = 0.0;
                   /* Multiply corresponding components */
                   for (int i = 1; i <= a.getDimension(); i++)
                       dotProduct += a.getComponent(i) * b.getComponent(i);
                   return dotProduct;
           }

           /* Returns the angle between two vectors in radians.
              If 'u' and 'v' are vectors, with an angle 'A' between them,
                       u.v = |u||v| cos(A)  */
           public static double angleBetween (Vector a, Vector b) {
```

```
115          return Math.acos(Vector.dotProduct(a, b) / (a.getAbsoluteValue() *
                 b.getAbsoluteValue())));
116      }
117  }
```

```
1  public class VectorDemo {
2      public static void main (String[] args) {
3              /* Simple 2D vector with magnitude sqrt(2) */
4              Vector a = new Vector(1, 1);
5              System.out.printf("Magnitude of %s is %f\n", a, a.getAbsoluteValue());
6
7              /* Create 3 random 3D vectors */
8              Vector b = new Vector(random(-10, 10), random(-10, 10), random(-10,
                    10));
9              Vector c = new Vector(random(-10, 10), random(-10, 10), random(-10,
                    10));
10             Vector d = new Vector(random(-10, 10), random(-10, 10), random(-10,
                    10));
11
12             /* Demonstrate addition, dot products, angle measurement */
13             System.out.printf("Sum of vectors %s, %s, %s is %s\n", b, c, d,
                   Vector.add(b, c, d));
14             System.out.printf("Dot product of %s and %s is %d\n", b, c, (int)
                   Vector.dotProduct(b, c));
15             System.out.printf("The angle between %s and %s is %f degrees\n", b, c,
16                                        Math.toDegrees(Vector.angleBetween(b,
                                            c)));
17      }
18
19      /* Returns random integers in a specified range */
20      public static int random (int lo, int hi) {
21              return (int) (lo + ((hi - lo) * Math.random()));
22      }
23  }
```

Variable Description

Vector		
`int`	`dimension`	The dimension of the vector
`double[]`	`components`	The ordered list of components of the vector

Vector::Vector(double[])		
`double[]`	`components`	The ordered list of components of the vector

Vector::getComponent(int)		
`int`	`index`	The index of the component to be retrieved

Vector::getAbsoluteValue()		
`double`	`abs`	Stores the square of the magnitude of the vector
`int`	`i`	Counter variable, counts through components of the vector

Vector::multiplyByScalar(double)		
`double`	`k`	The scalar to multiply the vector by

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

— **Anonymous**

Problem 20 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.
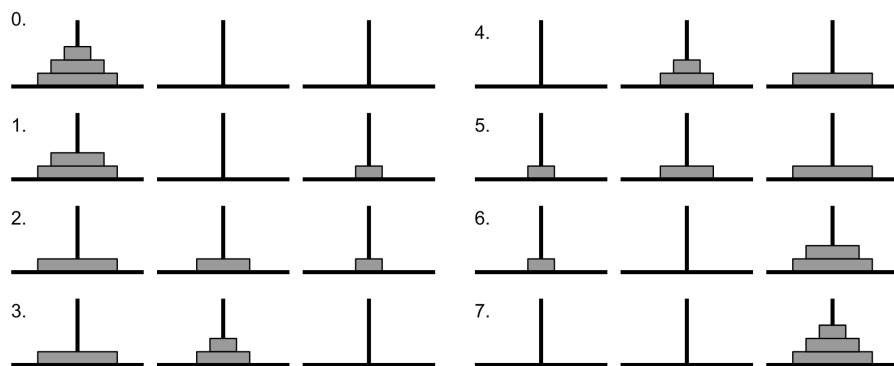
Solution to the Towers of Hanoi with 3 disks.

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,

92

$k_1 = 1$. 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

```java
1  public class TowersOfHanoi {
2      public static void main (String[] args) {
3          try {
4              /* Parse the first command line argument as the number of
                    diska */
5              int disks = Integer.parseInt(args[0]);
6              /* Make sure there is at least one disk */
7              if (disks < 1)
8                  throw new NumberFormatException();
9              /* Initiate the recursive steps */
10             solveHanoi(disks, "A", "C", "B");
11         } catch (NumberFormatException | IndexOutOfBoundsException e) {
12             /* Handle missing or incorrectly formatted arguments */
13             System.out.println("Enter 1 argument
                    (number_of_disks[integer])!");
14         }
15     }
16
17     /* Displays moves to solve the Towers of Hanoi problem with 3 pegs */
18     public static void solveHanoi (int disk, String source, String destination,
            String spare) {
19         /* Base case - nothing to do */
20         if (disk == 0)
21             return;
22         /* Move the stack of (n-1) disks to the spare peg */
23         solveHanoi(disk - 1, source, spare, destination);
24         /* Move the largest disk to the destination */
```

```
25              System.out.printf("(%d) : %s -> %s%n", disk, source, destination);
26              /* Move the stack of (n-1) disks back on top of the largest
27                 disk, on the destination peg */
28              solveHanoi(disk - 1, spare, destination, source);
29          }
30  }
```

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 21 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.
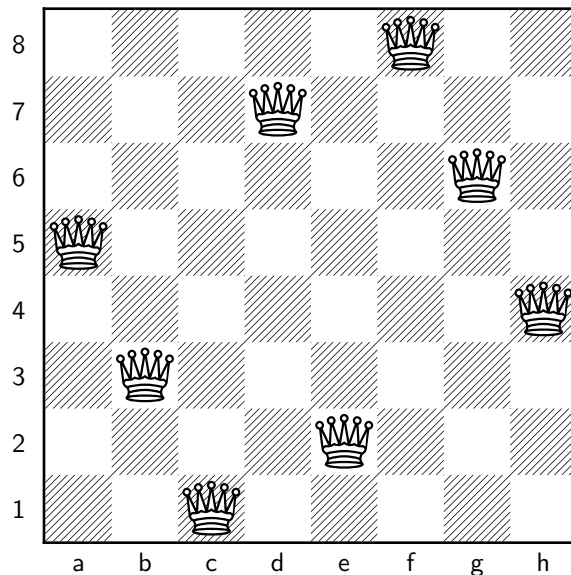
The *n queens puzzle* is an extension of this puzzle, involving n queens on an $n \times n$ chessboard. Count the total number of solutions for the *n queens puzzle*, including reflections and rotations.

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.

95

```
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 $i \in \{0, 1, \ldots, \texttt{size} - 1\}$:
   (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 $i \in \{0, 1, \ldots, \texttt{size} - 1\}$:
   (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

```java
1   public class NQueens {
2           private final int size;
3           private int[] board;
4           private int numberOfSolutions;
5           private final boolean drawSolutions;
6
7           /* Sets the size of the board and the number of queens */
8           public NQueens (int size, boolean drawSolutions) {
9                   this.size = size;
10                  this.drawSolutions = drawSolutions;
11                  this.initBoard();
12          }
13
14          /* Returns the number of solutions to a board of given size */
15          public int countSolutions () {
16                  solveNQueens(0);
17                  return numberOfSolutions;
18          }
19
20          /* Initializes the board */
21          private void initBoard () {
22                  this.board = new int[size];
23                  this.numberOfSolutions = 0;
24                  for (int i = 0; i < size; i++)
25                          board[i] = -1;
26          }
27
28          /* Determines whether the queen on a specified row is threatened
29             by a queen on a previous row */
30          private boolean isThreatened (int row) {
31                  for (int i = 0; i < row; i++) {
32                          if ((board[row] == board[i])
33                                  || ((board[row] - board[i]) == (row - i))
34                                  || ((board[row] - board[i]) == (i - row))) {
35                                  return true;
36                          }
37                  }
38                  return false;
39          }
40
41          /* Recursively solves the n-queens problem */
42          private void solveNQueens (int row) {
43                  if (row == size) {
44                          /* Reached maximum recursion depth - found a solution */
```

97

```
45                        numberOfSolutions++;
46                        if (drawSolutions) {
47                              drawBoard();
48                              System.out.println();
49                        }
50                        return;
51                  }
52                  /* Place queens on all possible columns on the row */
53                  for (board[row] = 0; board[row] < size; board[row]++) {
54                        if (!isThreatened(row)) {
55                              /* Recurse if the board is valid so far */
56                              solveNQueens(row + 1);
57                        }
58                  }
59            }
60
61            /* Displays the current configuration of the board */
62            public void drawBoard () {
63                  for (int i = 0; i < size; i++) {
64                        for (int j = 0; j < size; j++) {
65                              System.out.print(((board[i] == j)? "Q" : "-") + " ");
66                        }
67                        System.out.println();
68                  }
69            }
70
71            public static void main (String[] args) {
72                  try {
73                        /* Parse the first command line argument as the size of the
                              board */
74                        int size = Integer.parseInt(args[0]);
75                        /* Parse the second command line argument as a boolean,
                              indicating
76                          whether to draw the solved boards. Defaults to not showing
                              the solutions */
77                        boolean drawSolutions = (args.length > 1)?
                              Boolean.parseBoolean(args[1]) : false;
78                        /* Make sure the board exists */
79                        if (size < 1)
80                              throw new NumberFormatException();
81                        /* Create a 'NQueens' object */
82                        NQueens q = new NQueens(size, drawSolutions);
83                        /* Display the number of solutions */
84                        System.out.println(q.countSolutions());
85                  } catch (NumberFormatException | IndexOutOfBoundsException e) {
86                        /* Handle missing or incorrectly formatted arguments */
```

```
87                          System.out.println("Enter at least 1 argument
                                (size_of_board[integer], <show_solutions>[true/false])!");
88                          System.out.println("(show_solutions defaults to false)");
89                  }
90          }
91  }
```

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

"In the mind's eye, a fractal is a way of seeing infinity."

— **James Gleick**

Problem 22 The *Sierpinski Carpet* is a plane fractal. It can be produced iteratively by taking a solid square, dividing it into 9 congruent squares in a 3-by-3 grid, removing the centre square, and recursively applying the same procedure on each of the remaining squares *ad infinitum*.

Display the *Sierpinski Carpet* to a specified number of iterations.
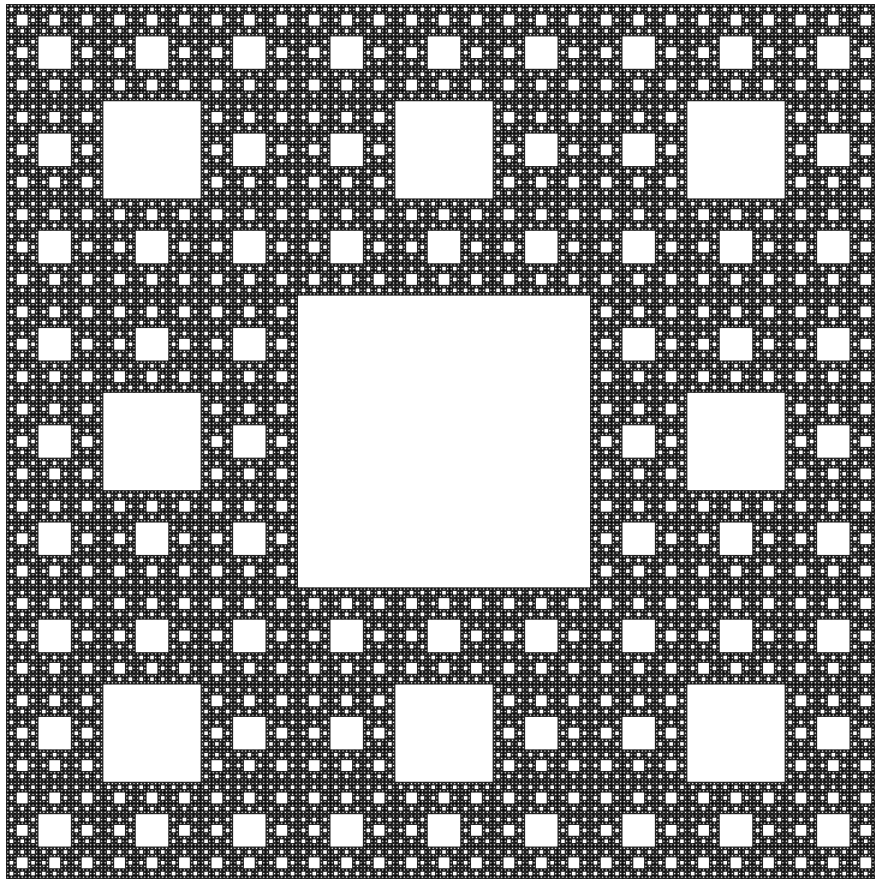
The Sierpinski Carpet

Solution In an ASCII terminal, we can only display a rough representation of the `Sierpinski Carpet`, a few levels deep. A level n carpet will have a width and height of 3^n. Within this grid, every character lies either in the centre of a 9-by-9 square, in which case it is not in the carpet, or it lies on the edge, in which case it is in the carpet. If neither can be determined, we can scale up the search square to the next level, and repeat recursively.

Here, points in the carpet are drawn, while points not in the the carpet are left as whitespace.

`main (level:Integer)`
1. For each pair $(\text{i}, \text{j}) \in \{0, 1, \ldots, 3^n - 1\} \times \{0, 1, \ldots, 3^n - 1\}$:
   (a) Call `isInSierpinskiCarpet(i, j)`. If it returns `true`, display a solid block at (`i`, `j`), otherwise, leave a blank space there.
2. **Exit**

`isInSierpinskiCarpet (x:Integer, y:Integer)`
1. If either of `x` or `y` is zero, the point (`x`, `y`) is on the edge of a square of some level. **Return `true`**.
2. If both `x` and `y` leave a remainder of one on division by 3, the point(`x`, `y`) is at the centre of a square of some level. **Return `false`**.
3. Call `isInSierpinskiCarpet(x / 3, y / 3)`, and **return** the returned value.

Source Code

```
1  public class SierpinskiCarpet {
2      public static void main (String[] args) {
3          try {
4              /* Parse the first command line argument as the level of
                  detail of the carpet */
5              int level = Integer.parseInt(args[0]);
6              /* Make sure that the level is positive */
7              if (level < 0)
8                  throw new NumberFormatException();
9              /* Iterate over every 'point' in the carpet */
10             for (int i = 0; i < Math.pow(3, level); i++) {
11                 for (int j = 0; j < Math.pow(3, level); j++) {
12                     /* Display a full block for points 'in' the
                          carpet */
13                     System.out.print(isInSierpinskiCarpet(i, j)?
                          "\u2588\u2588" : " ");
14                 }
```

```
15                                System.out.println();
16                            }
17                    } catch (NumberFormatException | IndexOutOfBoundsException e) {
18                            /* Handle missing or incorrectly formatted arguments */
19                            System.out.println("Enter 1 argument
                                (order_of_carpet[integer])!");
20                    }
21            }
22
23            /* Determines whether a point is in the carpet */
24            public static boolean isInSierpinskiCarpet (int x, int y) {
25                    /* Blocks are in the carpet if they are on the edge */
26                    if (x == 0 || y == 0)
27                            return true;
28                    /* Blocks at the centres of 3-by-3 squares on any level are
29                       not in the carpet */
30                    if (((x % 3) == 1) && ((y % 3) == 1))
31                            return false;
32                    /* Recurse to the next, larger level */
33                    return isInSierpinskiCarpet(x / 3, y / 3);
34            }
35 }
```

Variable Description

SierpinskiCarpet::main(String[])		
int	level	The depth to which to render the carpet
int	i, j	Counter variables, represent a point on the screen to be displayed
SierpinskiCarpet::isInSierpinskiCarpet(int, int)		
int	x, y	Couter variable, represent the point in question

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

— Pablo Picasso

Problem 23 *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` ∈ `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`.

103

     (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

```java
import java.util.Scanner;

public class RPNCalculator {
        /* Simple stack setup */
        private static double[] operandStack;
        private static int top;

        public static void main (String[] args) {
                /* Prompt an RPN expression from the terminal */
                System.out.printf("Reverse Polish Expression : ");
                String expression = (new Scanner(System.in)).nextLine();
                /* Evaluate the expression and display the result */
                double result = evaluateRPNExpression(expression);
                System.out.printf("Evaluated Expression  :    %s %n",
                    Double.toString(result));
        }

        /* Evaluates expression in RPN */
        public static double evaluateRPNExpression (String expression) {
                /* Split the expression into tokens */
                String[] tokens = expression.split("\\s+");
                /* Initialize the stack with an appropriately large capacity */
                top = -1;
                operandStack = new double[tokens.length];

                /* Iterate through all tokens in the expression */
                for (String token : tokens) {
                        /* Push operands into the stack and continue */
                        if (isDouble(token)) {
                                pushOperand(Double.parseDouble(token));
                                continue;
                        }

                        /* Pop operands from the stack */
                        double rightOperand = popOperand();
                        double leftOperand = popOperand();
```

```
36                          /* Determine the operator encountered and calculate the
                               appropriate result */
37                          double result = 0.0;
38                          switch (token.charAt(0)) {
39                                  case '+' :    result = leftOperand + rightOperand;
40                                                break;
41                                  case '-' :    result = leftOperand - rightOperand;
42                                                break;
43                                  case '*' :    result = leftOperand * rightOperand;
44                                                break;
45                                  case '/' :    result = leftOperand / rightOperand;
46                                                break;
47                                  case '^' :    result = Math.pow(leftOperand,
                                        rightOperand);
48                                                break;
49                                  default :     System.out.printf("Unknown operator
                                        (%s)!\n", token);
50                                                System.exit(0);
51                          }
52                          /* Push the result onto the stack */
53                          pushOperand(result);
54                  }
55              /* Return the last item in the stack */
56              return popOperand();
57      }
58
59      /* Pushes an operand onto the stack */
60      private static void pushOperand (double n) {
61              operandStack[++top] = n;
62      }
63
64      /* Pops an operand from the stack. Exits on failure. */
65      private static double popOperand () {
66              if (top < 0) {
67                      System.out.println("Insufficient operands!");
68                      System.exit(0);
69              }
70              return operandStack[top--];
71      }
72
73      /* Determines whether a token is a number */
74      private static boolean isDouble (String n) {
75              try {
76                      Double.parseDouble(n);
77                      return true;
78              } catch (NumberFormatException e) {}
```

```
79            return false;
80        }
81 }
```

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

*"Computer Science is no more about computers than astronomy is
about telescopes"*

— **Edsger W. Dijkstra**

Problem 24 A *queue* is a linear data structure which allows storage and retrieval of elements in accordance with the *First In First Out (FIFO)* principle. Thus, elements exit a *queue* in the same order they entered it.

Implement a *queue* capable of holding an arbitrary number of elements of a specified type.

Solution The use of *linked lists*[12] is apropriate here. *Generics* ensure that once a queue is declared with a data type, only elements of that data type can be added to it, as opposed to merely storing `Objects`.

`Node<T> (item:T)`
1. Copy `item` as an object variable.
2. Declare two variables `left` and `right`, both of type `Node<T>`.
3. **Return** the resultant object.

`link (left:Node<T>, right:Node<T>)`
1. Set `left->right` to `right`.
2. Set `right->left` to `left`.

`LinkedQueue<T> ()`
1. Declare two constants `HEAD` and `TAIL`, both of type `Node<T>` with arbitrary data items.
2. Link `TAIL` and `HEAD`.
3. **Define** the functions:
    (a) `LinkedQueue<T>::enqueue(item)`
    (b) `LinkedQueue<T>::dequeue()`
    (c) `LinkedQueue<T>::peek()`
    (d) `LinkedQueue<T>::clear()`
    (e) `LinkedQueue<T>::isEmpty()`
    (f) `LinkedQueue<T>::size()`

---

[12]A linked list is a linear data structure where each element is a separate object, or *node*. Each *node* contains both *data* and *addresses* of the surrounding nodes.

4. **Return** the resultant object.

LinkedQueue<T>::enqueue (item:T)
1. Create a new Node<T>, pass it item, and call it newNode.
2. Link HEAD->left and newNode.
3. Link newNode and HEAD.

LinkedQueue<T>::dequeue ()
1. If the queue is empty, return null.
2. Temporarily store the node TAIL->right as lastNode.
3. Link TAIL and lastNode->right.
4. **Return** the item contained in lastNode.

LinkedQueue<T>::peek ()
1. **Return** the item in the node TAIL->right.

LinkedQueue<T>::clear ()
1. Link TAIL and HEAD.

LinkedQueue<T>::isEmpty ()
1. If the TAIL->right is HEAD, **return true**, otherwise **return false**.

LinkedQueue<T>::size ()
1. Initialize an integer n to zero.
2. Set a variable current to TAIL.
3. While current->right is not HEAD, set current to current->right and increment n.
4. **Return** n.

Source Code

```
1  public class Node<T> {
2      /* Item data is immutable */
3      protected final T item;
4
5      /* References to other nodes */
6      protected Node<T> left;
7      protected Node<T> right;
8
9      /* Set the data item */
```

```java
        public Node (T item) {
                this.item = item;
        }

        /* Get the data item */
        public T getItem () {
                return item;
        }

        /* Use the data item's 'toString()' method */
        @Override
        public String toString () {
                return item.toString();
        }

        /* Doubly link two nodes */
        public static <T> void link (Node<T> left, Node<T> right) {
                left.right = right;
                right.left = left;
        }
}
```

```java
import java.util.Iterator;

/* Use generics to allow arbitrary data typed queues, eith type checking
   enforced at compile-time */
public class LinkedQueue<T> implements Iterable<T> {
        /* Special nodes surrounding data nodes */
        private final Node<T> HEAD = new Node<T>(null);
        private final Node<T> TAIL = new Node<T>(null);

        public LinkedQueue () {
                Node.<T>link(TAIL, HEAD);
        }

        /* Enqueues a data item of generic type into the head */
        public void enqueue (T item) {
                Node<T> newNode = new Node<T>(item);
                Node.<T>link(HEAD.left, newNode);
                Node.<T>link(newNode, HEAD);
        }

        /* Dequeues a data item from the tail */
        public T dequeue () {
                if (this.isEmpty())
```

```
24                    return null;
25            Node<T> lastNode = TAIL.right;
26            Node.<T>link(TAIL, lastNode.right);
27            return lastNode.getItem();
28        }
29
30        /* Returns the data item at the tail without removing it */
31        public T peek () {
32            return TAIL.right.getItem();
33        }
34
35        /* Clears the queue */
36        public void clear () {
37            /* Garbage collection takes care of orphaned nodes */
38            Node.<T>link(TAIL, HEAD);
39        }
40
41        /* Checks if the queue is empty */
42        public boolean isEmpty () {
43            return TAIL.right == HEAD;
44        }
45
46        /* Returns the size of the queue */
47        public int size () {
48            int n = 0;
49            /* Start at the tail */
50            Node<T> current = TAIL;
51            /* Iterate through all nodes until the head */
52            while ((current = current.right) != HEAD)
53                n++;
54            return n;
55        }
56
57        /* Format the elements of the queue neatly */
58        @Override
59        public String toString () {
60            String[] elements = new String[this.size()];
61            Node<T> current = TAIL;
62            int n = 0;
63            while ((current = current.right) != HEAD)
64                elements[n++] = current.toString();
65            return "[" + String.join(", ", elements) + "]";
66        }
67
68        /* Allow the elements of the queue to be iterated over simply */
69        @Override
```

```java
        public Iterator<T> iterator () {
            return new Iterator<T>() {
                private Node<T> current = TAIL.right;

                @Override
                public boolean hasNext () {
                    return current != HEAD;
                }

                @Override
                public T next () {
                    T item = current.getItem();
                    current = current.right;
                    return item;
                }

                @Override
                public void remove () {
                    throw new UnsupportedOperationException();
                }
            };
        }
}
```

```java
public class QueueDemo {
    public static void main (String[] args) {
        /* Create an integer queue */
        LinkedQueue<Integer> q = new LinkedQueue<Integer>();

        /* Enqueue random numbers to the queue */
        for (int i = 0; i < (10 + (int) (10 * Math.random())); i++) {
            Integer n = (int) (100 * Math.random());
            System.out.printf("Enqueuing : %s\n", n);
            q.enqueue(n);
        }
        /* Demonstrate simple output formatting */
        System.out.printf("Queue[%2d] : %s\n", q.size(), q);

        /* Demonstrate peeking */
        System.out.printf("Number about to be dequeued : %s\n", q.peek());

        /* Demonstrate the FIFO principle in effect */
        System.out.println("(Dequeuing 10 numbers)");
        for (int i = 0; i < 10; i++)
            System.out.printf("Dequeuing : %s\n", q.dequeue());
```

```
22              System.out.printf("Queue[%2d] : %s\n", q.size(), q);
23
24              /* Demonstrate iteration until empty */
25              System.out.println("(Dequeueing until empty)");
26              while (!q.isEmpty())
27                      System.out.printf("Dequeuing : %s\n", q.dequeue());
28              System.out.printf("Queue[%2d] : %s\n", q.size(), q);
29      }
30 }
```

Variable Description

Node<T>		
T	item	The data stored in the node
Node<T>	left	Reference to the node to the left of this
Node<T>	right	Reference to the node to the right of this
LinkedQueue<T>		
Node<T>	HEAD	Special node, marks the point of entry of new data
Node<T>	TAIL	Special node, marks the point of exit of data
LinkedQueue<T>::enqueue(T)		
T	item	The data to be enqueued
LinkedQueue<T>::dequeue()		
Node<T>	lastNode	The node containing the data to be dequeued
LinkedQueue<T>::size()		
int	n	Stores the number of elements in the queue
LinkedQueue<T>::toString()		
String[]	elements	Temporary array, stores the string representations of the data items in the queue
int	n	Counter variable

This project was compiled with X_ƎL^AT_EX.

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