Computer Project

(2017-2019)

Satvik Saha

Class: XI B

Roll number: 24

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

— Jason Gorman

"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.

```
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
public class SquareRoot {
	public static void main (String[] args) {
		/* Parse the first command line argument as the number to square root */
		double number = Double.parseDouble(args[0]);
		/* Parse the second conmand line argument as the number of iterations.
		   Default to 100 */
		int maxIterations = (args.length > 1)? Integer.parseInt(args[1]) : 100;

		double root = squareRoot(number, maxIterations);
		double library_root = Math.sqrt(number);

		/* Display the calculated root, along with a comparison with the
		   library calculated value */
		System.out.printf("Calculated square root :  %f%n", root);
		System.out.printf("System library square root : %f%n", library_root);
		System.out.printf("Error : %f%n", (root - library_root));
	}

	public static double squareRoot (double n, int maxIterations) {
		/* Handle edge cases, ignore negative values */
		if (n < 0)
			return Double.NaN;
		if (n == 0)
			return 0.0;
		/* Start by guessing half of the number */
		double x = n / 2;
```

```
            for (int i = 0; i < maxIterations; i++) {
                x = 0.5 * (x + (n / x));
            }
            return x;
        }
}
```

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

```java
public class Fraction {

	/* Store the numerator and denominator */
	protected int numerator;
	protected int denominator;

	public Fraction (int numerator, int denominator) {
		/* Handle invalid fractions */
		if (denominator == 0)
			throw new ArithmeticException("Division by zero!");

		this.numerator = Math.abs(numerator);
		this.denominator = Math.abs(denominator);

		if (numerator != 0) {
			/* Reduce to lowest terms */
```

```java
            int g = gcd(this.numerator, this.denominator);
            this.numerator /= g;
            this.denominator /= g;
            /* Make sure that the sign is on the numerator */
            this.numerator *= Math.signum(numerator) *
                Math.signum(denominator);
        } else {
            /* Make sure all 'zero fractions' are the same */
            this.denominator = 1;
        }
    }

    public int getNumerator () {
        return this.numerator;
    }

    public int getDenominator () {
        return this.denominator;
    }

    /* Return a String representation of the Fraction for display */
    public String toString () {
        /* Format all fractions with denominator '1' as simple integers */
        if (this.denominator == 1)
            return this.numerator + "";
        return this.numerator + " / " + this.denominator;
    }

    /* Add 2 Fraction objects */
    public static Fraction addFractions (Fraction a, Fraction b) {
        int sumNumerator = (a.getNumerator() * b.getDenominator()) +
            (a.getDenominator() * b.getNumerator());
        int sumDenominator = a.getDenominator() * b.getDenominator();
        return new Fraction(sumNumerator, sumDenominator);
    }

    /* Calculate the greatest common divisor of integers, using Euclid's method
        recursively */
    private static int gcd (int p, int q) {
        return (p < q)? gcd(q, p) : ((p % q) == 0)? q : gcd(q, p % q);
    }
}
```

```java
import java.util.Scanner;

public class FractionAdder {
	public static void main (String[] args) {
		Scanner inp = new Scanner(System.in);
		try {
			/* Get the two fractions from user input */
			System.out.print("Enter the numerator and denominator [integer
			    integer] of the first fraction : ");
			Fraction a = new Fraction(inp.nextInt(), inp.nextInt());
			System.out.print("Enter the numerator and denominator [integer
			    integer] of the second fraction : ");
			Fraction b = new Fraction(inp.nextInt(), inp.nextInt());

			/* Calculate and display the sum of the fractions.
			   Here, we take advantage of the toString() method defined for
			       Fractions */
			Fraction sum = Fraction.addFractions(a, b);
			System.out.printf("%n(%s) + (%s) = (%s) %n", a, b, sum);
		} catch (ArithmeticException e) {
			System.out.println("Invalid fraction - division by zero!");
		}
	}
}
```

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`

"Dividing one number by another is mere computation; knowing what to divide by what is mathematics."

— **Jordan Ellenberg**

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

$$f\left(\frac{p}{q}\right) = \left\lfloor \frac{p}{q} \right\rfloor + \frac{1}{f\left(\frac{q}{p \bmod q}\right)}$$

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

```java
public class ContinuedFraction {
      public static void main (String[] args) {
            try {
                  /* Parse command line arguments as the numerator and
                     denominator
                    of the fraction */
                  int numerator = Integer.parseInt(args[0]);
                  int denominator = Integer.parseInt(args[1]);
                  System.out.println(getContinuedFraction(new
                     Fraction(numerator, denominator)));
            } catch (Exception e) {
                  System.out.println("Enter 2 arguments! ([numerator]
                     [denominator])");
            }
      }

      /* Return the String representation of the continued fraction */
      public static String getContinuedFraction (Fraction f) {
        String expansion = "[" + getContinuedFraction(f.getNumerator(),
            f.getDenominator());
        /* By convention, the first comma is replaced with a semicolon */
        return expansion.replaceFirst(",", ";");
    }

    /* Recursively calculate the continued fraction representation */
    public static String getContinuedFraction (int numerator, int denominator) {
        /* Base case : the fraction is now irreducible */
        if (denominator == 1)
            return numerator + "]";
        /* Pull out the integer part, invert the fraction and recurse */
        return (numerator / denominator) + ", " +
            getContinuedFraction(denominator, numerator % denominator);
    }
}
```

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

```
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 coeffient 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.

$$\begin{aligned} 963 + 369 &= 1332 \\ 1332 + 2331 &= 3663 \end{aligned}$$

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
class PalindromeGenerator {
	public static void main (String[] args) {
		/* Parse the first command line argument as the seed */
		long n = Long.parseLong(args[0]);
		generatePalindrome(n, 0);
	}

	public static void generatePalindrome (long n, int step) {
		long r = reverse(n);
		if (n == r) {
			/* Base case : palindrome reached */
			System.out.printf("%d is a palindrome (%d step%s)%n", n, step,
			    ((step == 1)? "" : "s"));
		} else {
			try {
				/* Use a library method to add. This will throw an
				   Exception in case of overflow, which would have
				   otherwise been ignored */
				long sum = Math.addExact(n, r);
				System.out.printf("%d + %d = %d%n", n, r, sum);
				/* Recurse via tail recursion, simply incrementing the
				    step value */
				generatePalindrome(sum, step + 1);
			} catch (ArithmeticException e) {
				/* Stop if the numbers become too big */
				System.out.printf("Long Overflow - Sum exceeded maximum
				    size at step %d%n", step);
			}
```

```java
        }
    }

    /* Reverse the integer supplied */
    public static long reverse (long n) {
        long r = 0;
        while (n > 0) {
            /* Pull out the last digit and accumulate it on another
               variable */
            r = (r * 10) + (n % 10);
            n /= 10;
        }
        return r;
    }
}
```

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 doesn'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

```java
public class Factorize {
	public static void main (String[] args) {
		/* Parse the first command line argument as the number to factorize */
		int number = Integer.parseInt(args[0]);
		/* Start from 2 */
		System.out.println(factorize(number, 2));
	}

	/* Return the String representation of the prime factorization of an integer
	    */
	public static String factorize (int n, int next) {
		/* Base case 1 : nothing to factorize */
		if (n == 1)
			return "";
		/* Base case 2 : reached a prime */
		if (next >= n)
			return next + "";
		/* Check for a factor */
		if ((n % next) == 0)
			return next + " " + factorize(n / next, next);
		/* Recurse by incrementing the next 'factor' to check */
		return factorize(n, next + 1);
	}
}
```

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++;

```

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

    /* Initialize the map/dictionary by reading the file on the second pass */
    private void initWordMap () throws IOException {
            wordMap = new String[numberOfLines][2];

            FileReader fileReader = new FileReader(filename);
            BufferedReader bufferedReader = new BufferedReader(fileReader);

            for (int i = 0; i < numberOfLines; i++) {
                    /* Split a line along whitespace */
                    String[] words = bufferedReader.readLine().split("\\s+");
                    if (words.length >= 2) {
                            wordMap[i][0] = words[0];
                            wordMap[i][1] = words[1];
                    } else {
                            /* Ignore empty lines */
                            wordMap[i][0] = wordMap[i][1] = "";
                    }
            }

            bufferedReader.close();
            fileReader.close();
    }

    /* Returns the codeword, given a plain word */
    public String getEncodedText (String word) {
            /* Iterate through all entries */
            for (int i = 0; i < numberOfLines; i++) {
                    if (wordMap[i][0].equalsIgnoreCase(word)) {
                            return wordMap[i][1];
                    }
            }
            /* Reflect the original back if not found in the codebook */
            return word;
    }
}
```

```java
import java.util.Scanner;
import java.io.IOException;
import java.io.FileNotFoundException;

public class TextEncoder {
	public static void main (String[] args) throws Exception {
		try {
			/* Parse the first command line argument as the path to the
			    codebook */
			CodeSubstituter cs = new CodeSubstituter(args[0]);

			/* Get a sentence to encode, and extract the individual words
			    */
			System.out.print("Enter a sentence to encode : ");
			String sentence = (new Scanner(System.in)).nextLine();
			String[] words = sentence.split("\\s+");

			System.out.print("Encoded sentence     : ");
			/* Iterate through each word, replacing it with the codeword
			    in the codebook */
			for (int i = 0; i < words.length; i++) {
				String encodedText =
				    cs.getEncodedText(words[i].toLowerCase().replaceAll("[^a-z]",
				    ""));
				System.out.print(encodedText + " ");
			}
			System.out.println();
		} catch (ArrayIndexOutOfBoundsException e) {
			System.out.println("Enter 1 argument ([codebook_filename])");
		} catch (FileNotFoundException e) {
			System.out.println("Codebook not found! Enter a valid
			    filename.");
		} catch (IOException e) {
			e.printStackTrace();
		}
	}
}
```

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 codewords

`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.
```

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();
```

```java
	}

	/* Return the number of occurrences of a character in the file */
	public int getCountOf (char c) {
		int count = 0;
		for (int i = 0; i < fileData.length(); i++) {
			if (fileData.charAt(i) == c) {
				count++;
			}
		}
		return count;
	}

	/* Compile the counts of all letters in the file */
	public void countAllLetters () {
		for (char c = 'a'; c <= 'z'; c++) {
			letters[c - 'a'] = c;
			letterCount[c - 'a'] = getCountOf(c);
			totalLetters += letterCount[c - 'a'];
		}
	}

	/* Sort the entries by frequency (bubble sort) */
	private void sortLetters () {
		for (int right = 26; right > 0; right--)
			for (int i = 1; i < right; i++)
				if (letterCount[i] > letterCount[i-1])
					swap(i, i-1);
	}

	/* Utility swapping method */
	private void swap (int i, int j) {
		char tmpChar = letters[i];
		int tmpCount = letterCount[i];
		letters[i] = letters[i-1];
		letterCount[i] = letterCount[i-1];
		letters[i-1] = tmpChar;
		letterCount[i-1] = tmpCount;
	}

	/* Create and write the final report to a file */
	public void writeReportToFile (String toFilename) throws IOException {
		FileWriter fileWriter = new FileWriter(toFilename);
		BufferedWriter bufferedWriter = new BufferedWriter(fileWriter);
		PrintWriter printWriter = new PrintWriter(bufferedWriter);

```

```java
        /* Make sure the frequencies all fit, aligned in the same column */
        int l = (totalLetters + "").length();
        String unusedLetters = "";
        for (int i = 0; i < 26; i++) {
            /* Show the letter, frequency and percentage out of the total
               */
            char c = letters[i];
            int count = letterCount[i];
            double percent = (count * 100.0) / totalLetters;
            if (count > 0) {
                printWriter.printf("%c : %5.2f%% (%" + l + "d) %n",
                                   c, percent, count);
            } else {
                /* Separate unused letters */
                unusedLetters += c + " ";
            }
        }
        printWriter.printf("Total letters : %d%n", totalLetters);
        if (unusedLetters.length() == 0)
            unusedLetters = "(none)";
        printWriter.printf("Unused letters : %s%n", unusedLetters);

        printWriter.close();
        bufferedWriter.close();
        fileWriter.close();
    }
}
```

```java
import java.io.IOException;
import java.io.FileNotFoundException;

public class AnalyseCharacterFrequency {
    public static void main (String[] args) {
        try {
            /* Parse the commnd line arguments as the file to analyse and
               the
             file to pipe the results into */
            String fromFile = args[0];
            String toFile = args[1];

            /* Create and write the report */
            CharacterCounter cc = new CharacterCounter(fromFile);
            cc.writeReportToFile(toFile);
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Enter 2 arguments! ([filename_from]
                      [filename_to])");
            } catch (FileNotFoundException e) {
                  System.out.println("Enter a valid filename!");
            } catch (IOException e) {
                  e.printStackTrace();
            }
      }
}
```

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

`BufferedWriter`	`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
`CharacterCounter`	`cc`	An object for analysing the frequencies of letters in files

This project was compiled with XƎLaTeX.

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

Satvik Saha

`sahasatvik@gmail.com`
`https://sahasatvik.github.io`