Computer Project

(2017-2019)

Satvik Saha

Class: XII B

Roll number: 34

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

— Jason Gorman

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

— **Robert Sewell**

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

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

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

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

```
main (lo:Integer, hi:Integer)
  1. Assert that the integers in the range [lo, hi) are all positive.
  2. For each i ∈ {lo, lo + 1, ..., 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 ∈ {2, 3, ..., 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++;
```

```
            }
            if (multiplicity == 1) {
                /* Flip the sign */
                mob = -mob;
            } else if (multiplicity > 1) {
                /* Squared factor found */
                return 0;
            }
        }
        return mob;
    }
}
```

Variable Description

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

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

"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);
```

```
            return ((i >= 0) && (i <= top) && (elements[i] == n));
    }

    /* Checks whether the set is empty */
    public boolean isEmpty () {
            return top < 0;
    }

    /* Clears all elements from the set */
    public void clear () {
            /* Only the top index has to be updated, since values byond it
               cannot be accessed */
            this.top = -1;
    }

    /* Adds an element to the set. Returns 'false' if it is already
       present, or there isn't enough space. */
    public boolean add (int n) {
            if (getSize() >= getMaxSize())
                    return false;
            /* Find the breakpoint to shift elements */
            int i = indexOfEqualOrGreater(n);
            if ((i >= 0) && (i <= top) && (elements[i] == n))
                    return false;
            /* Shift elements greater than 'n' to make room for it */
            for (int j = top; j >= i; j--)
                    elements[j + 1] = elements[j];
            elements[i] = n;
            top++;
            return true;
    }

    /* Removes an element from the set. Returns 'false' if it isn't
       already present. */
    public boolean remove (int n) {
            if (isEmpty())
                    return false;
            /* Find the location of the element */
            int i = indexOfEqualOrGreater(n);
            if ((i < 0) || (i > top) || (elements[i] != n))
                    return false;
            /* Shift elements into the desired element, erasing it */
            for (int j = i; j < top; j++)
                    elements[j] = elements[j + 1];
            top--;
            return true;
```

```
        }

        /* Returns the union of two sets */
        public static Set union (Set a, Set b) {
                Set r = new Set(a.getSize() + b.getSize());
                /* The 'add' methods take care of duplicates */
                for (int n : a)
                        r.add(n);
                for (int n : b)
                        r.add(n);
                return r;
        }

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

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

        /* Finds the index of the element equal to or greater than
           the desired element via binary search */
        private int indexOfEqualOrGreater (int n) {
                int hi = top + 1;
                int lo = 0;
                while (lo < hi) {
                        int mid = (lo + hi) / 2;
                        if (n < elements[mid])
                                hi = mid;
                        else if (n > elements[mid])
                                lo = mid + 1;
                        else
                                return mid;
                }
                return hi;
        }

        /* Format the set elements as a list */
        @Override
        public String toString () {
                if (getSize() == 0)
                        return "[]";
                String s = "";
                for (Integer n : this)
                        s += n + " ";
                return "[" + String.join(", ", s.split("\\s+")) + "]";
        }

        /* Allow 'Set' to be iterable, providing easy access to elements
           without indexing */
        @Override
        public Iterator<Integer> iterator () {
                return new Iterator<Integer>() {
                        private int currentIndex = 0;

                        @Override
                        public boolean hasNext () {
                                return currentIndex <= top;
                        }

                        @Override
                        public Integer next () {
                                return elements[currentIndex++];
                        }

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

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

```java
                a.add((int) (Math.random() * 10));
        for (int i = 0; i < 10; i++)
                b.add((int) (Math.random() * 10));
        for (int i = 0; i < 10; i++)
                c.add((int) (Math.random() * 10));

        /* Demonstrate simple output formatting */
        System.out.printf("A [%2d] = %s\n", a.getSize(), a);
        System.out.printf("B [%2d] = %s\n", b.getSize(), b);
        System.out.printf("C [%2d] = %s\n", c.getSize(), c);
        System.out.println();

        /* Demonstrate set operations */
        System.out.printf("A union B [%2d] = %s\n",
                          Set.union(a, b).getSize(),
                          Set.union(a, b));
        System.out.printf("B union C [%2d] = %s\n",
                          Set.union(b, c).getSize(),
                          Set.union(b, c));
        System.out.printf("C union A [%2d] = %s\n",
                          Set.union(c, a).getSize(),
                          Set.union(c, a));
        System.out.printf("A union B union C [%2d] = %s\n",
                          Set.union(Set.union(a, b), c).getSize(),
                          Set.union(Set.union(a, b), c));
        System.out.println();
        System.out.printf("A intersection B [%2d] = %s\n",
                          Set.intersection(a, b).getSize(),
                          Set.intersection(a, b));
        System.out.printf("B intersection C [%2d] = %s\n",
                          Set.intersection(b, c).getSize(),
                          Set.intersection(b, c));
        System.out.printf("C intersection A [%2d] = %s\n",
                          Set.intersection(c, a).getSize(),
                          Set.intersection(c, a));
        System.out.printf("A intersection B intersection C [%2d] = %s\n",
                          Set.intersection(Set.intersection(a, b),
                              c).getSize(),
                          Set.intersection(Set.intersection(a, b), c));
        System.out.println();
        System.out.printf("A - B [%2d] = %s\n",
                          Set.difference(a, b).getSize(),
                          Set.difference(a, b));
        System.out.printf("B - C [%2d] = %s\n",
                          Set.difference(b, c).getSize(),
                          Set.difference(b, c));
        System.out.printf("C - A [%2d] = %s\n",
                    Set.difference(c, a).getSize(),
                    Set.difference(c, a));
    }
}
```

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 *scalars*. 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. $(a_i, b_i, k \in \mathbb{R})$

$$
\begin{aligned}
(a_1, a_2, \ldots, a_n) + (b_1, b_2, \ldots, b_n) &= (a_1 + b_1, a_2 + b_2, \ldots, a_n + b_n) && \text{(Addition)} \\
k\,(a_1, a_2, \ldots, a_n) &= (ka_1, ka_2, \ldots, ka_n) && \text{(Scalar Multiplication)} \\
(a_1, a_2, \ldots, a_n) \cdot (b_1, b_2, \ldots, b_n) &= a_1b_1 + a_2b_2 + \cdots + a_nb_n && \text{(Dot Product)} \\
\|(a_1, a_2, \ldots, a_n)\| &= \sqrt{a_1^2 + a_2^2 + \cdots + a_n^2} && \text{(Magnitude)}
\end{aligned}
$$

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 components, 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.
  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];
	}

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

	/* Wrapper methods which call static ones */

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

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

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

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

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

	/* Format vector components neatly */
	@Override
	public String toString () {
		String s = "(";
		for (double component : components)
			s += component + ", ";
		return s.replaceAll(", $", ")");
	}

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

```
            /* 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 the binary 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) {
```

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

```java
public class VectorDemo {
	public static void main (String[] args) {
		/* Simple 2D vector with magnitude sqrt(2) */
		Vector a = new Vector(1, 1);
		System.out.printf("Magnitude of %s is %f\n", a, a.getAbsoluteValue());

		/* Create 3 random 3D vectors */
		Vector b = new Vector(random(-10, 10), random(-10, 10), random(-10,
		    10));
		Vector c = new Vector(random(-10, 10), random(-10, 10), random(-10,
		    10));
		Vector d = new Vector(random(-10, 10), random(-10, 10), random(-10,
		    10));

		/* Demonstrate addition, dot products, angle measurement */
		System.out.printf("Sum of vectors %s, %s, %s is %s\n", b, c, d,
		    Vector.add(b, c, d));
		System.out.printf("Dot product of %s and %s is %d\n", b, c, (int)
		    Vector.dotProduct(b, c));
		System.out.printf("The angle between %s and %s is %f degrees\n", b, c,
		                                   Math.toDegrees(Vector.angleBetween(b,
		                                       c)));
	}

	/* Returns random integers in a specified range */
	public static int random (int lo, int hi) {
		return (int) (lo + ((hi - lo) * Math.random()));
	}
}
```

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

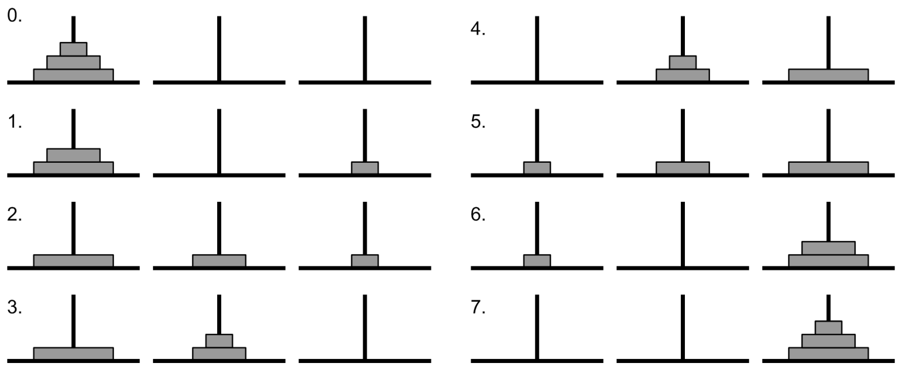
Vector::multiplyByScalar(double)		
`double`	`k`	The scalar to multiply the vector by

"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 following rules.

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



Solution to the Towers of Hanoi with 3 disks.

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

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

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

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

```
main (disks:Integer)
   1. Call solveHanoi(disks, "A", "C", "B").
   2. Exit

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

Source Code

```
public class TowersOfHanoi {
        public static void main (String[] args) {
                try {
                        /* Parse the first command line argument as the number of
                            disks */
                        int disks = Integer.parseInt(args[0]);
                        /* Make sure there is at least one disk */
                        if (disks < 1)
                                throw new NumberFormatException();
                        /* Initiate the recursive steps */
                        solveHanoi(disks, "A", "C", "B");
                } catch (NumberFormatException | IndexOutOfBoundsException e) {
                        /* Handle missing or incorrectly formatted arguments */
                        System.out.println("Enter 1 argument (number_of_disks[integer,
                            >0])!");
                }
        }

        /* Displays moves to solve the Towers of Hanoi problem with 3 pegs */
        public static void solveHanoi (int disk, String source, String destination,
            String spare) {
                /* Base case - nothing to do */
                if (disk == 0)
                        return;
                /* Move the stack of (n-1) disks to the spare peg */
                solveHanoi(disk - 1, source, spare, destination);
                /* Move the largest disk to the destination */
                System.out.printf("(%d) : %s -> %s%n", disk, source, destination);
                /* Move the stack of (n-1) disks back on top of the largest
                   disk, on the destination peg */
                solveHanoi(disk - 1, spare, destination, source);
        }
}
```

Variable Description

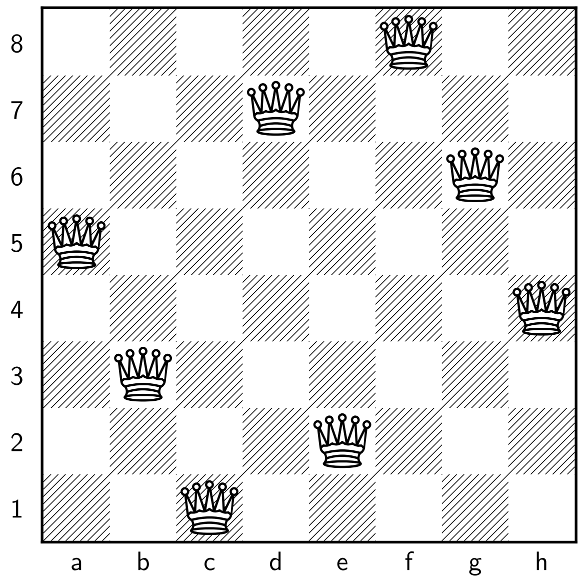
TowersOfHanoi::main(String[])		
int	disks	The number of disks in the problem

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.

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

Source Code

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

        /* Sets the size of the board and the number of queens */
        public NQueens (int size, boolean drawSolutions) {
                this.size = size;
                this.drawSolutions = drawSolutions;
                this.initBoard();
        }

        /* Returns the number of solutions to a board of given size */
        public int countSolutions () {
                solveNQueens(0);
                return numberOfSolutions;
        }

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

        /* Determines whether the queen on a specified row is threatened
           by a queen on a previous row */
        private boolean isThreatened (int row) {
                for (int i = 0; i < row; i++) {
                        if ((board[row] == board[i])
                            || ((board[row] - board[i]) == (row - i))
                            || ((board[row] - board[i]) == (i - row))) {
                                return true;
                        }
                }
                return false;
        }

        /* Recursively solves the n-queens problem */
        private void solveNQueens (int row) {
                if (row == size) {
                        /* Reached maximum recursion depth - found a solution */
```

```java
                numberOfSolutions++;
                if (drawSolutions) {
                    drawBoard();
                    System.out.println();
                }
                return;
            }
            /* Place queens on all possible columns on the row */
            for (board[row] = 0; board[row] < size; board[row]++) {
                if (!isThreatened(row)) {
                    /* Recurse if the board is valid so far */
                    solveNQueens(row + 1);
                }
            }
        }

        /* Displays the current configuration of the board */
        public void drawBoard () {
            for (int i = 0; i < size; i++) {
                for (int j = 0; j < size; j++) {
                    System.out.print(((board[i] == j)? "Q" : "-") + " ");
                }
                System.out.println();
            }
        }

        public static void main (String[] args) {
            try {
                /* Parse the first command line argument as the size of the
                   board */
                int size = Integer.parseInt(args[0]);
                /* Parse the second command line argument as a boolean,
                  indicating whether to draw the solved boards.
                  Defaults to not showing the solutions */
                boolean drawSolutions = (args.length > 1)?
                   Boolean.parseBoolean(args[1]) : false;
                /* Make sure the board exists */
                if (size < 1)
                    throw new NumberFormatException();
                /* Create a 'NQueens' object */
                NQueens q = new NQueens(size, drawSolutions);
                /* Display the number of solutions */
                System.out.println(q.countSolutions());
            } catch (NumberFormatException | IndexOutOfBoundsException e) {
                /* Handle missing or incorrectly formatted arguments */
```

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

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

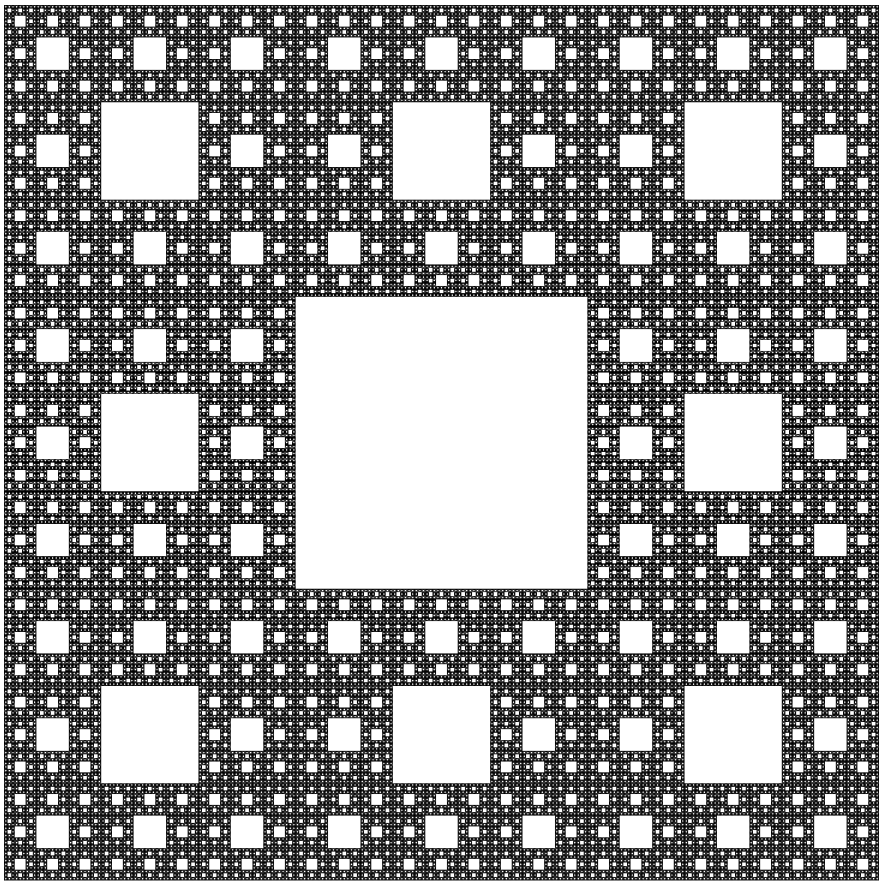
NQueens::main(String[])		
int	size	The number of rows and columns in the chessboard
boolean	drawSolutions	Stores whether to display solved boards or not
NQueens	q	Object capable of solving the *n queens* problem

“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 3-by-3 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 (i , j) ∈ $\{0, 1, \ldots, 3^n - 1\} \times \{0, 1, \ldots, 3^n - 1\}$:
     (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

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

```
                System.out.println();
            }
        } catch (NumberFormatException | IndexOutOfBoundsException e) {
            /* Handle missing or incorrectly formatted arguments */
            System.out.println("Enter 1 argument
                (order_of_carpet[integer])!");
        }
    }

    /* Determines whether a point is in the carpet */
    public static boolean isInSierpinskiCarpet (int x, int y) {
        /* Blocks are in the carpet if they are on the edge */
        if (x == 0 || y == 0)
            return true;
        /* Blocks at the centres of 3-by-3 squares on any level are
           not in the carpet */
        if (((x % 3) == 1) && ((y % 3) == 1))
            return false;
        /* Recurse to the next, larger level */
        return isInSierpinskiCarpet(x / 3, y / 3);
    }
}
```

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	Counter variables, 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 $\in$ tokens:
     (a) If token is a floating point:
           i. Push token onto operandStack.
          ii. Get the next token from tokens.
         iii. Jump back to (3a).
     (b) Pop an operand from operandStack and call it rightOperand.
     (c) Pop another operand from operandStack and call it leftOperand.
```

   (d) Depending on which operator `token` represents, evaluate the operation with `token` as the operator and `leftOperand` and `rightOperand` as the respective operands. Call it `result`.
   (e) Push `result` onto `operandStack`.
4. Pop and operand from `operandStack` and **return** it.

Source Code

```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();
```

```
                /* Determine the operator encountered and calculate the
                    appropriate result */
                double result = 0.0;
                switch (token.charAt(0)) {
                        case '+' :    result = leftOperand + rightOperand;
                                      break;
                        case '-' :    result = leftOperand - rightOperand;
                                      break;
                        case '*' :    result = leftOperand * rightOperand;
                                      break;
                        case '/' :    result = leftOperand / rightOperand;
                                      break;
                        case '^' :    result = Math.pow(leftOperand,
                            rightOperand);
                                      break;
                        default :     System.out.printf("Unknown operator
                            (%s)!\n", token);
                                      System.exit(0);
                }
                /* Push the result onto the stack */
                pushOperand(result);
        }
        /* Return the last item in the stack */
        return popOperand();
}

/* Pushes an operand onto the stack */
private static void pushOperand (double n) {
        operandStack[++top] = n;
}

/* Pops an operand from the stack. Exits on failure. */
private static double popOperand () {
        if (top < 0) {
                System.out.println("Insufficient operands!");
                System.exit(0);
        }
        return operandStack[top--];
}

/* Determines whether a token is a number */
private static boolean isDouble (String n) {
        try {
                 Double.parseDouble(n);
                 return true;
        } catch (NumberFormatException e) {}
```

```
                return false;
        }
}
```

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

```java
public class Node<T> {
	/* Item data is immutable */
	protected final T item;

	/* References to other nodes */
	protected Node<T> left;
	protected Node<T> right;

	/* Set the data item */
```

```
        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;
        }
}
```

```
import java.util.Iterator;

/* Use generics to allow arbitrary data typed queues, with 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())
                return null;
            Node<T> lastNode = TAIL.right;
            Node.<T>link(TAIL, lastNode.right);
            return lastNode.getItem();
        }

        /* Returns the data item at the tail without removing it */
        public T peek () {
            return TAIL.right.getItem();
        }

        /* Clears the queue */
        public void clear () {
            /* Garbage collection takes care of orphaned nodes */
            Node.<T>link(TAIL, HEAD);
        }

        /* Checks if the queue is empty */
        public boolean isEmpty () {
            return TAIL.right == HEAD;
        }

        /* Returns the size of the queue */
        public int size () {
            int n = 0;
            /* Start at the tail */
            Node<T> current = TAIL;
            /* Iterate through all nodes until the head */
            while ((current = current.right) != HEAD)
                n++;
            return n;
        }

        /* Format the elements of the queue neatly */
        @Override
        public String toString () {
            String[] elements = new String[this.size()];
            Node<T> current = TAIL;
            int n = 0;
            while ((current = current.right) != HEAD)
                elements[n++] = current.toString();
            return "[" + String.join(", ", elements) + "]";
        }

        /* Allow the elements of the queue to be iterated over simply */
        @Override
```

```
	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();
			}
		};
	}
}
```

```
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());
```

```java
                System.out.printf("Queue[%2d] : %s\n", q.size(), q);

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

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 XeLaTeX.

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

Satvik Saha

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