DM550: INTRODUCTION TO PROGRAMMING
Exercise list (Autumn 2021)

Part II: Writing Classes

1 Client classes

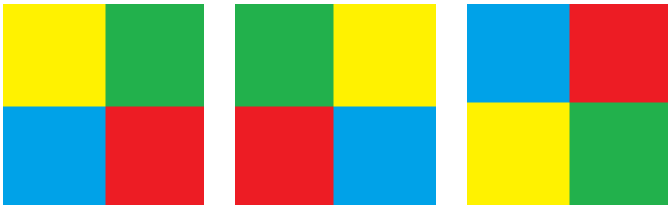
1. One of the simplest ways to represent images is as a bidimensional array of triples of integers, specifying for each dot (pixel) on the image its red, green and blue components. The class `Image` provides a simple interface for manipulating images in this format; it contains the following methods:

   - a constructor with two arguments that creates a black image with the specified dimensions;
   - a constructor with one argument that takes the name of a file containing an image and creates an object representing that image;
   - methods `int width()` and `int height()` that return the width and height of this image;
   - methods `int red(int x, int y)`, `int green(int x, int y)` and `int blue(int x, int y)` that return the red, green and blue components of pixel `(x,y)`, assuming that these coordinates represent a valid pixel;
   - a method `void setPixel(int x, int y, int red, int green, int blue)` that sets the red, green and blue values of the pixel with coordinates `(x,y)` to the given values (assuming that the coordinates represent a valid pixel and that the color values are in the range $[0, 255]$);
   - a method `void display()` that displays this image on the screen.

   Note that, by convention, pixels are counted from the top left corner of the picture.

   Your task is to develop a class `ImageUtils` implementing some of the usual image transformations available in most picture editors. In particular, you should implement the following methods.

   - Two methods `Image flipHorizontal(Image image)` and `Image flipVertical(Image image)` that return a new image obtained by flipping the original image horizontally or vertically.

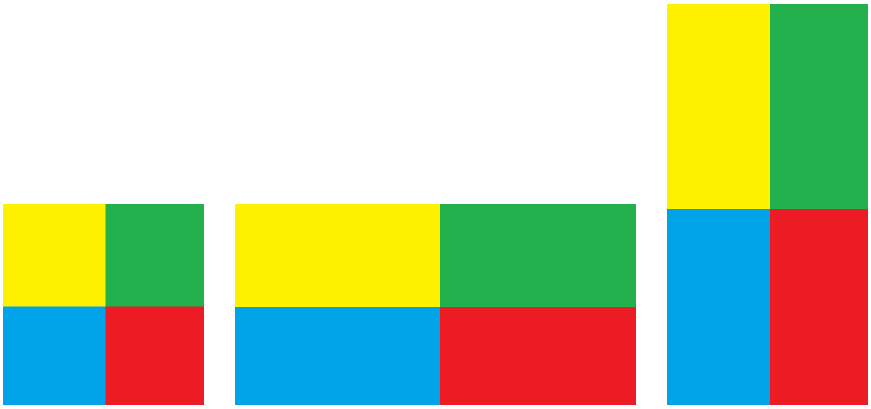
     original horizontal flip vertical flip

   - Three rotation methods `Image rotateLeft(Image image)`, `Image rotateHalf(Image image)` and `Image rotateRight(Image image)` that return a new image obtained by rotating the original image by perform a quarter turn, a half turn or a three-quarter turn counterclockwise.

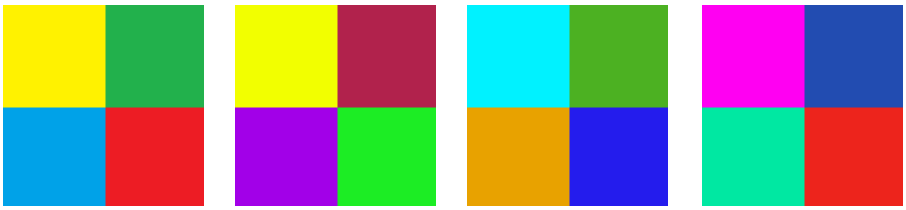
     original quarter turn half turn three-quarter turn

   - A method `Image stretchHorizontal(Image image)` that returns a new image with the same height as the original one and double the width, where every pixel is duplicated horizontally, and a similar method `Image stretchVertical(Image image)` operating on the vertical direction.



original horizontal stretch vertical stretch

- Three methods `Image switchRedGreen(Image image)`, `Image switchRedBlue(Image image)` and `Image switchGreenBlue(Image image)` that return a new image where the two colors indicated have been switched.



original red/green switch red/blue switch green/blue switch

- A method `int[] histogram(Image image)` that returns an array indicating, respectively, how many pixels are predominantly red, predominantly green and predominantly blue in the given image. A pixel's dominant color is the one with highest value; if two or more colors share the highest value, that pixel is ignored.
- A method `int[] averageColor(Image image)` that computes the average color of the given image, given by the average value of the red, green and blue components for every pixel in the image.
- A method `Image resample(Image image)` that returns a new image obtained from the original as follows: for each pixel not on the border, its red component is replaced by the average of the red components of the nine pixels in a 3×3 rectangle centered on the current pixel. The same process is applied to the green and blue components.

2. Develop an utility class `ImageUtilsSE.java` similar to the previous one, but where all methods are `void` and change the original image, rather than returning a new one.

   (Note that this class only includes the methods that preserve the dimensions of the image, since class `Image` does not allow these to be modified.)

3. Rational numbers are represented in `java` as floating points. Although `java` includes several types for floating point numbers, in some cases it is important to work with exact fractions, without introducing rounding errors. One possible solution is to represent each fraction as a pair of integer numbers (numerator and denominator), with the restriction that the denominator be positive.

   The goal of this exercise is to develop a calculator that receives an expression typed in by the user and evaluates it, printing the result as an (exact) fraction. The program is structured in three classes: a class `Fraction` that provides objects representing fractions; a class `List` that provides lists of `String`s of variable length; and a class `Calculator` that is a client of the two other classes. You will develop the `Calculator` class following the contract for the two underlying classes, given below.

   Class `Fraction` provides:

   - a constructor with one argument, converting an integer number to a fraction;
   - methods
     - `Fraction add(Fraction f)`
     - `Fraction subtract(Fraction f)`
     - `Fraction multiply(Fraction f)`

– and `Fraction divide(Fraction f)`

returning the result of adding, subtracting, multiplying or dividing this fraction to/by fraction `f`, respectively;

- a method `String toString()` that returns a textual representation of this fraction.

Class `List` provides:

- a constructor with one argument `v` of type `String[]` that creates a new list containing the elements of `v` in the same order;
- a method `boolean isEmpty()` that returns whether this list is empty;
- a method `String head()` that returns the first string in this list;
- a method `void tail()` that removes the first string from this list;
- a method `String toString()` that returns a textual representation of this list.

The input is a valid arithmetic expression consisting of numbers, arithmetic operators and parentheses. Formally, we allow expressions generated by the following grammar.

$$e ::= m + e \mid m - e \mid m \qquad m ::= t \times m \mid t/m \mid t \qquad t ::= (e) \mid \texttt{int}$$

For simplicity, we assume that every token in the input is separated by spaces. That is, `2␣+␣3` is correct input, but `2+3` is not.

Your program should parse the input and compute the expression while parsing. The parser is developed in a straightforward way from the grammar of expressions, by writing one method for each case. For example, an expression e always start with a multiplication m, possibly followed by a + or − sign and another expression. This means that we first need to call the method for parsing multiplications (which returns a value); if, after this method returns, there are still unprocessed tokens, we check whether the next token is a + or a −, parse the remaining tokens as a (smaller) expression, and add or subtract the results. The remaining methods are similar.

Assume that only syntactically correct input is provided.

2 Provider classes

2.1 Time management

A team developing a project realized that they needed to be able to represent points in time.

1. Implement a class `TimeStamp` whose objects are points in time represented by hours, minutes and seconds. Decide which attributes this class should have and which getters and setters should be available for these attributes. The class should also provide the following methods:

   - constructors with 0, 1, 2 or 3 arguments, corresponding (respectively) to the hours, minutes and seconds of the required timestamp (assume the default value 0 for absent arguments);
   - a method `boolean valid(int hours, int minutes, int seconds)` that checks whether its given arguments can be passed along to the constructor;
   - methods `void skipSecond()`, `void skipMinute()` and `void skipHour()` that add one second, one minute and one hour, respectively, to this timestamp (assume that 23:59:59 is followed by 0:00:00);
   - a method `void skip(TimeStamp time)` that adds the amount of time described in `time` to this timestamp;
   - a method `boolean equals(Object other)` that determines whether this timestamp is the same as `other`;
   - a method `TimeStamp copy()` that returns a copy of this timestamp;
   - a method `String toString()` that returns a textual representation of this timestamp.

   Write a client to test this class.

2. As the project continued to grow, the team concluded that in some cases they needed to enrich timestamps with information about the date, represented as a year, month and day.

   Implement a class `Date` that represents a timestamp in a particular date, including information about the year, month, day and timestamp. Decide which attributes this class should have and which getters and setters should be available for these attributes. Exploit the class `TimeStamp` as much as possible. Your class should provide the same methods as class `TimeStamp` (except the construtors) plus the following ones:

   - a constructor with three arguments that creates an object corresponding to midnight on the given date;
   - a constructor with four arguments that creates an object corresponding to the given timestamp on the given date;
   - a method `boolean valid(int year, int month, int day)` that checks whether its arguments can be passed along to the constructor;
   - methods `void skipDay()`, `void skipMonth()` and `void skipYear()` that skip this date forward by one day, one month or one year, respectively;
   - methods `void skipTime(TimeStamp time)` skips this date forward by the indicated amount of time;
   - a method `int largestYear()` that returns the year of the date most in the future ever created or referenced;
   - a method `boolean equals(Object other)` that determines whether this date is the same as `other`;
   - a method `Date copy()` that returns a copy of this date;
   - a method `String toString()` that returns a textual representation of this date.

   Write a client to test your class.

2.2 Two-dimensional geometry

A point on a flat surface (such as a computer screen) is defined by two coordinates, also called its horizontal and vertical components.

1. Define a class `Point2D` whose objects represent two-dimensional points. Decide which attributes this class should have and which getters and setters should be available for these attributes. The class should also provide the following methods:

   - a constructor that creates a point given its coordinates;
   - a method `boolean isOrigin()` testing whether this point is the origin;
   - a method `void move(double deltaX, double deltaY)` that moves this point according to the vector (`deltaX`,`deltaY`);
   - a method `double distanceToOrigin()` that returns this point's distance to the origin;
   - a method `double distanceTo(Point2D point)` that returns this point's distance to `point`;
   - a method `int howManyOrigins()` that returns the number of objects currently pointing to the origin;
   - a method `boolean equals(Object other)` that checks whether this point is the same as `other`;
   - a method `Point2D copy()` that returns a copy of this point;
   - a method `String toString()` that returns a textual representation of this point.

   Consider the following client for `Point2D`. Describe the results of executing this code.

```
int x = 6, n = -1;
Point2D p = new Point2D(4, x);
if ( !p.isOrigin() )
  p.move(n, n*2);
System.out.println(p.distanceToOrigin());
```

   Now consider a second client for this class. Sketch the memory state after each instruction.

```
Point2D p1 = new Point2D(0, 0);
Point2D p2 = p1;

p1.move(1, 1);
System.out.println(p2.toString());

p2.move(3, 3);
System.out.println(p1.toString());
```

2. A polygon is a region on the plane limited by straight line segments (its sides).

   Implement a class `Polygon` whose objects represent polygons. A polygon is to be represented as a sequence of points (its vertices) such that there is a line between each two consecutive points, as well as between the first and the last. Exploit class `Point2D` as much as possible. Decide which attributes this class should have and which getters and setters should be available for these attributes. Each polygon should also have a unique identifier. The class should also provide the following methods:

   - a constructor that creates a polygon from an array of `Point2D` containing its vertices;
   - a method `double perimeter()` returning this polygon's perimeter;
   - a method `Point2D nearest()` that returns the vertex of this polygon that is closest to the origin;
   - a method `double longestSide()` returning the length of this polygon's longest side;
   - a method `void move(double deltaX, double deltaY)` that moves this polygon according to the vector (`deltaX`,`deltaY`);
   - a method `int verticesInQuadrant(int n)` counting how many of this polygon's vertices lie on the `n`-th quadrant;
   - a method `boolean isTriangle()` that determines whether this polygon is a triangle;
   - a method `boolean isRectangle()` that determines whether this polygon is a rectangle;
   - a method `int id()` returning this polygon's identifier;
   - a method `Polygon mostRecentTriangle()` returning the triangle most recently created;
   - a method `boolean equals(Object other)` that checks whether this polygon is equal to `other` (note that the polygon's vertices need not be given in the same order);
   - a method `Polygon copy()` that returns a copy of this polygon;
   - a method `String toString()` that returns a textual representation of this polygon.

3. A discrete representation of a function f is an array `graph` of `Point2D` such that $f(x) = y$ holds for each point (x, y) of the array. Furthermore, the elements of `graph` are sorted by their first coordinate.

   Implement a class `Function` whose objects are discrete representations of functions. Decide which attributes this class should have and which getters and setters should be available for these attributes. The class should also provide the following methods:

   - a constructor that receives an array as described and builds the corresponding function representation;
   - a method `boolean valid(Point2D[] graph)` that checks whether its argument can be passed along to the constructor;
   - a method `double max()` that finds the maximum value of this function, according to its discrete representation;
   - a method `boolean increasing()` that checks whether this function is increasing or not;
   - a method `double[] changeRate()` that returns an array with one less element than `graph` containing this function's average rate of change in each interval;
   - a method `boolean equals(Object other)` that checks whether this function has the same discrete representation as `other`;
   - a method `Function copy()` that returns a copy of this function;
   - a method `String toString()` that returns a textual representation of this function.

Write clients to test your classes.

2.3 Shopping carts

An online supermarket is building a `java` backoffice system, consisting of several interacting classes.

The class you have been selected to develop is the class whose objects represent shopping carts containing the products selected by a client. The products in the shopping cart are represented by objects of a class `Product` that you will not develop, and which provides (among others) the following methods.

- `double price()`: returns the price of this product.
- `boolean equals(Object other)`: returns `true` if this product is the same as `other`.
- `Product copy()`: returns a new product that is equal to this one.

Implement a class `ShoppingCart` whose objects represent shopping carts. Decide which attributes this class should have and which getters and setters should be available for these attributes. The class should also provide the following methods:

- a constructor with no arguments that returns an empty shopping cart;
- a method `void add(Product product)` that adds `product` to this shopping cart;
- a method `int numberOfItems()` returning the number of produts in this shopping cart;
- a method `boolean freeDelivery()` indicating whether this shopping cart is eligible for free delivery (only for carts with more than 50 items);
- a method `double totalPrice()` returning the cost of this shopping cart;
- methods `Product mostExpensive()` and `double highestPrice()` returning the most expensive `Product` in this shopping cart and its price;
- a method `int howMany(Product product)` returning the number of items equal to `product` in this shopping cart;
- a method `void removeMostExpensive()` removing all copies of the most expensive product in the shopping cart.

At a later stage, the supermarket will also be interested in obtaining statistics about its clients' shopping habits. Change the class `ShoppingCart` so that the following methods are also provided:

- a method `int howMany()` returning the total number of shopping carts created until this moment;
- a method `double mostExpensiveShoppingCart()` returning the total cost of the most expensive shopping cart ever seen;
- a method `ShoppingCart last()` returning a reference to the most recently created shopping cart.

Implementation tip. This is a good example of a class whose attribute can be implemented as an `ArrayList`.

Even though your class is meant to be used as part of a larger project, you still need to test it – write a client!

2.4 Tic-tac-toe

Tic-tac-toe is a game played on a 3×3 board. Two players alternate choosing a free cell and marking it with either **X** (first player) or **O** (second player). The first player to occupy the three cells in a line, columnn or diagonal wins the game.

Implement a class `TicTacToe` to manipulate a tic-tac-toe board. Decide which attributes this class should have and which getters and setters should be available for these attributes. The class should also provide the following methods:

- a constructor that creates an initial board (all cells are empty, and the next player is the one playing **X**);
- a method `void play(int i,int j)` that emulates the next player's move on cell (`i`,`j`), in case this is free;
- a method `char next()` returning the letter of the player whose turn it is;
- a method `boolean free(int i,int j)` that checks whether cell (`i`,`j`) is free;
- a method `char value(int i,int j)` that returns the letter in cell (`i`,`j`), in case it is occupied, and `-` otherwise;

- a method `boolean wins(char c)` that checks whether the player playing `c` has won the game;
- a method `String toString()` that returns a textual representation of the current board.

Write a client for this class allowing two players to play interactively. This client should check that all invocations of methods from the class `TicTacToe` are correct.

Enrichen your client with the possibility of playing against the computer. Make sure the computer never loses.

2.5 Storing secrets

An application requires a class whose objects represent safes. Each safe can store a single secret, with type `String`. In order to limit access to the secret, the safe also has a lock, which is an array of integers. Before being allowed to access the secret, a client needs to introduce the numbers in the lock one by one; any wrong number silently resets the safe's state to fully locked.

Write a class `Safe` implementing this functionalities. Decide which attributes this class should have and which getters and setters should be available for these attributes. The class should also provide the following methods:

- a constructor with only one argument, corresponding to the combination on the lock;
- a method `boolean locked()` indicating whether the safe is locked;
- a method `void store(String secret)` that stores `secret` in the safe, in case it is unlocked;
- a method `String contents()` that returns the secret, if the safe is open;
- a method `void insert(int n)` that inserts value `n` in the combination lock;
- a method `void reset()` that forgets any numbers already introduced in the combination.

Write a client to test your class.

2.6 Classifying fish

A fish-processing plant works daily with different kinds of fish. Fish come directly from the harbour, but the different species often arrive mixed. The plant wants to design software to categorize fish using state-of-the-art equipment that photographs fish passing on a moving belt and measures their width and length.

The data characterizing an object to be classified is called a *pattern*. In the case of fish, the pattern includes their width and length. The method used for classifying fish is the *nearest neighbour method*: a set of fish is analysed and its pattern and species are recorded. This information (the *reference set*) is then used to classify other fish automatically. For each fish, the steps are as follows.

1. Measure its dimensions.
2. Find the fish in the reference set with the most similar dimensions (the closest neighbour). To measure similarity, we compute the Euclidean distance between the dimensions of the fish we are classifying and those of all fish in the reference set.
3. Assign the species of the nearest neighbour to the fish we are classifying.

The class we want to develop should cover the general case, where the dimension of the pattern and the number of possible species is unspecified. The reference set must always be given.

Develop a class `Classifier` providing the following methods:

- a constructor with two arguments:
  1. a two-dimensional array with elements of type `double`, corresponding to the reference set;
  2. an array of integers listing the species of each element in the previous array – both arrays must have the same number of entries;
- a method `int classify(double[] pattern)` that receives a pattern and returns the species computed from the nearest neighbour method;
- a method `int[] classifySet(double[][] set)` that receives a set of patterns (in the same format as the constructor) and returns the corresponding array of classifications;
- a method `double error(double[][] set, int[] classifications)` receiving a set of patterns and an array of classifications, classifies each pattern and compares with the given classification, returning the percentage of errors.

2.7 Matrices, revisited

Implement a class `Matrix` whose objects represent matrices of `double`s. Decide which attributes this class should have and which getters and setters should be available for these attributes. The class should also provide the following methods:

- a constructor with a single argument `int n` creating a new identity matrix of size `n`×`n`;
- a construtor with two arguments `int m` and `int n` creating a new zero matrix of size `m`×`n`;
- a constructor with one argument `double[] v` creating a matrix with the elements of `v` along the diagonal and 0 elsewhere;
- a method `void multiply(int k)` that multiplies all entries in this matrix by `k`;
- a method `Matrix transpose()` that returns the transpose of this matrix;
- a method `double add()` that returns the sum of all elements in this matrix;
- a method `void addTo(Matrix m)` that adds `m` to this matrix, in case their dimensions are the same;
- a method `double average()` returning the average of all elements in this matrix;
- a method `boolean lessThan(Matrix m)` that checks whether each element in this matrix is less than the element in the same position of `m`;
- methods `boolean upperTriangular()`, `boolean lowerTriangular()` and `boolean diagonal()` indicating whether this matrix is upper triangular, lower triangular or diagonal, respectively;
- a method `Matrix multiplyBy(Matrix m)` that returns the product of this matrix with `m`, in case their dimensions are compatible;
- a method `int[] gauss(int[] v)` returning a vector `w` such that this matrix multiplied by `w` yields `v`, computed by Gaussian elimination (this method should throw an exception if there is not exactly one solution);
- a method `double determinant()` returning the determinant of this matrix;
- a method `boolean equals(Object other)` checking whether this matrix is the same as `other`;
- a method `Matrix copy()` that returns a copy of this matrix;
- a method `String toString()` returning a textual representation of this matrix.

Write a client to test your class.

3 Implementing recursive datatypes

1. Write a generic interface `MyCollection<E>` that defines the following operations.
   - `void add(E e)`: ensures that this collection contains `e`.
   - `void clear()`: removes all elements in this collection.
   - `boolean contains(E e)`: checks whether this collection contains `e`.
   - `boolean isEmpty()`: checks whether this collection is empty.
   - `int size()`: returns the number of elements in this collection.
2. A *queue* is a datatype for collections where elements must be removed in the exact order in which they were added (like the queue in the supermarket).

   Develop a class `Queue<E>` implementing queues of elements of type `E`. Your implementation should implement interfaces `MyCollection<E>` (above) and `Iterable<E>`, and provide the following additional method.
   - `E next()`: returns the next element in the queue and removes it from the queue.

   Your class should also include the standard methods `toString`, `equals` and `copy`.

3. A *stack* is a datatype for collections where elements must be removed in the *opposite* order in which they were added (like a stack of dishes in a restaurant).

   Develop a class `Stack<E>` implementing stacks of elements of type `E`. Your implementation should implement interfaces `MyCollection<E>` (above) and `Iterable<E>`, and provide the following additional methods.

   - `E top()`: returns the top element of the stack.
   - `void pop()`: removes the top element from the stack. from the queue.

   Your class should also include the standard methods `toString`, `equals` and `copy`.

4. A *double linked list* is a list implementation where elements are stored in nodes and each node has a pointer to both the previous and next item, and the list itself has pointers to the first and last elements of the list.

   Develop a class `DLList<E>` implementing double linked lists of elements of type `E`. Your implementation should implement interfaces `MyCollection<E>` (above) and `Iterable<E>`, and provide the following additional methods.

   - `void add(int i, E e)`: adds element `e` at position `i` in the list, shifting all subsequent elements.
   - `E get(int i)`: returns the `i`-th element of the list.
   - `void remove(int i)`: removes the `i`-th element of the list.

   Your class should also include the standard methods `toString`, `equals` and `copy`.

5. A *binary tree* is a datatype for collections where elements are stored in nodes, but each node now has a pointer to two other nodes (typically called *left* and *right children*). The topmost node (which is not a child of any node) is called the *root*, and a node whose children are both `null` is called a *leaf*.

   Develop a class `BinaryTree<E>` implementing binary trees of elements of type `E`. Your implementation should implement interfaces `MyCollection<E>` (above) and `Iterable<E>`, and provide the following additional methods.

   - `E root()`: returns the element in the root note of the tree.
   - `BinaryTree<E> left()` and `BinaryTree<E> right()`: return the subtrees with the left (respectively, right) child of the current root note as root.
   - `int height()`: returns the length of the longest path from the root to a leaf node.
   - `void mirror(BinaryTree t)`: transforms `t` into its mirror image (every node's left child becomes its right child, and vice-versa).

   Classically, there are three ways to iterate through the elements of a tree.

   - *pre-order traversal*, where we analyse the element in each node before moving to the left and right subtrees;
   - *in-order traversal*, where we first analyse the left subtree of each node, then the element in the node, and then the right subtree;
   - *post-order traversal*, where we analyse both left and right subtrees before processing the element in the node.

   Implement these three variants of iterators.

   Your class should also include the standard methods `toString`, `equals` and `copy`.

6. A *map* is a datatype for collections where elements are stored together with a key. Typically, the key is a primitive type (e.g. `int`), while the stored elements have a complex type where comparisons are expensive.

   Develop a class `Map<K,E>` implementing maps of elements of type `E` with keys of type `K`. Your implementation should implement interface `Iterable<E>` (but not `MyCollection<E>` or `MyCollection<K>`), and provide the following additional methods.

   - `void add(K key,E e)`: adds to this map an element `e` assigned to key `key`, removing any other element eventually assigned to `key`.
   - `void clear()`: removes all elements in this map.
   - `E find(K key)`: return the element in this map assigned to key `key`, if one exists.
   - `boolean isEmpty()`: checks whether this map is empty.

- `void remove(K key)`: removes the element assigned to key `key`, if one exists.
- `int size()`: returns the number of elements in this map.
- `K[] keys()`: returns an array with all the keys in this map.
- `E[] values()`: returns an array with all the elements in this map.
- `Iterator<K> keyIterator()`: returns an iterator over the keys in this map.

4 Recursive programming

1. Write a recursive method `int sumUpTo(int n)` that computes the sum of the natural numbers smaller than `n`.
2. Write a recursive method `int sumBetween(int m, int n)` that computes the sum of the natural numbers larger than `m` and smaller than `n`.
3. Write a recursive method `int sumEven(int n)` that computes the sum of all even numbers smaller than `n`.
4. Write a recursive method `int factorial(int n)` that returns `n`!.
5. Write a recursive method `int doubleFactorial(int n)` that returns `n`!!.
6. The sequence of Fibonacci numbers f_n is defined by $f(0) = f(1) = 1$ and $f(n+2) = f(n) + f(n+1)$. Write a recursive method `int fibonacci(int n)` that returns the `n`-th Fibonacci number.
7. Write a recursive method `int logarithm(int n)` that returns the integer base-2 logarithm of `n`.
8. Write a recursive method `int gcd(int m, int n)` that computes the greatest common divisor of `m` and `n` using Euclides' algorithm:
$$\begin{cases} \gcd(m,m) = m \\ \gcd(m,n) = \gcd(m, n-m) & m < n \\ \gcd(m,n) = \gcd(m-n, m) & m > n \end{cases}$$
9. Write a recursive method `int firstDigitInBase(int n, int k)` that returns the first digit of the representation of `n` in base `k`.
10. Given a number `n`, we repeatedly apply the following recipe: if our current value is even, we divide it by 2; if it is odd, we multiply it by 3 and add 1. Write a recursive method `int downToOne(int n)` that finds out how many steps it takes to get to 1.

The next exercises use the datatypes defined in the previous exercises.

11. Write a recursive method `double sum(Stack<Double> s)` that computes the sum of all values in `s`.
12. Write a recursive method `int zeros(Stack<Integer> s)` that returns the number of zeros in `s`.
13. Write a recursive method `int count(Stack<E> s, E e)` that counts the number of occurrences of `n` in `s`.
14. Write a recursive method `DLList<Integer> squares(int n)` that returns a list with the squares of all natural numbers from 1 to `n`.
15. Write a recursive method `double max(DLList<Double> v)` that returns the largest element in `v`.
16. Write a recursive method `Stack<T> reverse(Stack<T> s)` that returns a new stack containing the elements of `s` in reverse order. Would your solution also work for `Queue`s?
17. Write a recursive method `DLList<Integer>[] compare(DLList<Integer> v, int n)` that returns an array containing: as first element, a list containing the elements of `v` larger than `n`; as second element, a list containing the elements of `v` equal to `n`; and, as third element, a list containing the elements of `v` smaller than `n`.
18. Write a recursive method `Stack<Integer> sortedJoin(Stack<Integer> q1, Stack<Integer> q2)` that takes two ordered stacks `s1` and `s2` as input and returns an ordered stack containing all elements from either `s1` or `s2`.
19. Write a recursive method `Queue<T> shuffle(Queue<T> q1, Queue<T> q2)` that takes two queues `q1` and `q2` and constructs a new queue by taking alternately one element from each of `q1` and `q2`. Would your solution also work for `Stack`s?