Object-Oriented Programming 2 – Graded Exercises, Series 2 (DOJ + KMS4)

Exercise 1 – Recursion

Task

Please translate the following (primitive) recursive definition (*specification*)

$$f(b,0) = 1 \quad (1)$$
$$f(b,\mathrm{succ}(0)) = b \quad (2)$$
$$f(b,\mathrm{succ}(\mathrm{succ}(n))) = b \cdot f(b,\mathrm{succ}(n)) \quad (3)$$

of the (total) function f into a 5-line instance Java-method (*implementation*) with a single explicit parameter and for $b \in \mathbb{N}$ with $\mathbb{N}$ modelled as our Java-class `NaturalNumber`, which we have already studied together in class (*code reuse*).

Instruction

To help yourself with your (main) task, start with writing a Java-method for both b and the second argument of f being of (primitive) Java-type `int`, and throw the necessary exception in there. Modify this program to obtain the required non-static Java-method. (So, you will implement two Java-methods in total.) Before you start, answer the following *helper questions*:

1. How many arguments does f have?
2. What are the parameter and return types of your Java-methods for f?
3. How many base cases does f have? Which one(s)?
4. How many recursive cases does f have? Which one(s)?
5. Is Equation 2 necessary for f to be totally defined? Why?
6. What is the *de*composition operation of the recursion?
7. What is the *re*composition operation of the recursion?
8. What does the function f compute?
9. What are the arguments of f usually called like?
10. Why must all your Java-programs for f terminate?

Please choose an appropriate name for the function f and for your Java-methods that implement f. Your resulting Java-methods for f over `int` and $\mathbb{N}$ should be similar. How could you optimise this suboptimal situation (*code factorization*)?

Emphasis

The pedagogic emphasis of this task is on

1. *correctness* (that your implementations of f meet the above specification for f) and
2. your ability to insert a small piece of new (necessary?) code into a bigger, already existing picture of code (our OO-number library in the making).

Exercise 2: Polynomial as Linked List or Map

2.1

Write a class `Polynomial` that stores a polynomial such as:

$$p(x) = 5x^{10} + 9x^7 - x - 10$$

as a `LinkedList` of terms. A term contains the coefficient and the power of `x`. For example:

```
(5, 10)->(9,7)->(-1,1)->(-10,0)
```

Supply convenient constructors and the following methods:

`public Polynomial add(Polynomial p)`: adds the polynomial `p` to this polynomial and returns the result as a new polynomial.

`public Polynomial multiply(Polynomial p)`: multiplies the polynomial `p` with this polynomial, and returns the result as a new polynomial.

`public String toString()`: builds and returns a string containing the polynomial. This string should be "nicely" formatted according to the usual representation of polynomials in algebra (i.e. powers in decreasing order and no "0" or "1" coefficients or powers). Use "^" to represent powers. For example, the above polynomial should be formatted as:

```
p(x) = 5x^10 + 9x^7 - x - 10
```

When building a polynomial, insert each new element at the correct position, so that the polynomial terms always remains sorted with powers in decreasing order. (Do NOT use sort algorithms).

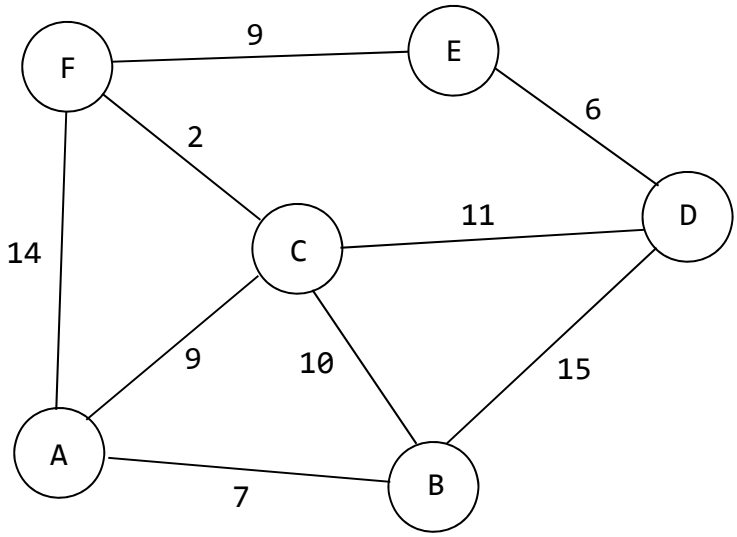
Supply a `Junit` test class to check your `add` and `multiply` methods and a `Main` class that allows to enter two polynomials and calculate their sum and their product.

2.2

Make a new version of the exercise 2.1, but use a `Map` for the coefficients (the `Map` associates each power with a coefficient). To get a sorted polynomial, use a `TreeMap<>(Collections.reverseOrder())`.

Exercise 3 – Dijkstra's Algorithm

Consider the problem of finding the shortest paths between nodes in a graph, which may represent, for example, road networks.



For example, the shortest paths from A to E is 9 + 2 + 9 = 20 (going through C and F)

Write a program that stores in a convenient data structure the shortest paths of a graph from any given start node to all other nodes and outputs the length of these paths. Proceed as follows:

1. Use the following helper class `DistanceTo` that stores the distance to a node. The `Node` type should be an enumeration type. For example: `enum Node{A, B, C, D, E, F};`.

```
public class DistanceTo implements Comparable<DistanceTo> {
  private Node target;
  private int distance;

  public DistanceTo(Node node, int dist) { target = node, distance = dist; }
  public Node getTarget() { return target; }
  public int getDistance() { return distance; }
  public int compareTo(DistanceTo other) {
    return Integer.compare(distance, other.distance);
  }
}
```

2. Store the graph definition in a text file as shown below and upload that file in your program. Store the distances between each node and its neighbors (i.e. the distances between each node and the nodes that are directly connected with it) in an appropriate data structure. Of course, the node names must correspond to your `Node` type.

```
A->B 7 C 9 F 14
B->C 10 D 15
...
```

3. Now, select a starting node and implement the following algorithm (Dijkstra's algorithm):

```
Let "from" be the starting node.
Add DistanceTo(from, 0) to a queue "q".
Create a map "shortestKnownDistances" from node names to distances.
While "q" is not empty
   Get its smallest element.
   If the element target is not a key in shortestKnownDistances
      Let "d" be the distance to that target.
      Put(target, d) into shortestKnownDistances.
      For all nodes "n" that are neighbors of target
         Add DistanceTo(n, d + distance from target to n) to q.
```

When the algorithm is finished, *shortestKnownDistances* contains the shortest distances from the starting node to all reachable targets.

Exercise 4: Analyzing Movie Data

Download the file `movies.txt` and the class `Movie.java` from Moodle.

The file `movies.txt` contains data of approximately 23'000 movies taken from the database `freebase.com` and formatted by C. Horstmann as a text file. Each movie consists in fives lines that contain the movie name, year, director(s), producer(s) and actors, as follows:

```
Name: Five Easy Pieces
Year: 1970
Directed by: Bob Rafelson
Produced by: Bob Rafelson, Richard Wechsler, Harold Schneider
Actors: Jack Nicholson, Karen Black, Billy Green Bush, ... (more)
```

The class `Movie.java` allows you to load the `movies.txt` file and instantiate `Movie` objects with the `readMovie` method.

Use these files to implement a class that prints the following information about the movies data by making an extended use of Java streams:

1. The number of movies contained in the database.
2. The number of movies for which no director name is registered.
3. The list of the movies, whose title begins with the word “A” or “An”.
4. The number of movies, whose title starts with the same letter.
5. The number of movies where directors are also actors.
6. The movie that has the maximum number of actors (print the movie title, director name(s) and the number of actors).
7. The number of actor names in the database
    - If you count every single apparition of an actor name;
    - If you only count the *different* actor names.
8. The director who made the largest number of movies (print the director name and a list of the film titles). For movies with several directors, only consider the first one. Filter out the movies for which no director name is registered.
9. (A challenge) The hundred actors hat have played in the most movies.