Preamble

Creation of a workspace for the exercise

Create a new directory `CSIE_340679`, from your operating system's file browser, or with the command `mkdir ~/CSIE_340679` in a terminal. This directory will be your *workspace*, it will contain all the course exercises.

Creation of a project dedicated to HW2

1. Launch *Visual Studio Code* (*VS Code* hereafter), for example by running the code command in the terminal, or by clicking on a shortcut; the `[Alt]` key makes the horizontal menu bar appear/hide.

2. In *VS Code* type the key combination `[Ctrl]+[Shift]+[P]` (or `[Alt]+[V]` then click on *Command Palette*) then type "`java:create`" in the bar that appears and select Create Java Project and the No build tools option .

3. A window opens and prompts you to choose your workspace , that is, the `CSIE_340679` directory that you created by following the instructions in the previous paragraph.

4. Enter the name of your project (`HW2`) in the text bar.

Source recovery

Download the src.zip, then click on "Extract", finally select your project directory (i.e. the HW2 directory which is inside the `CSIE_340679` directory) when the unzip tool asks you to and click on "Extract" again.

Enable assert in your Java virtual machine

In *Visual Studio Code*, type the key combination `[Ctrl]+[,]` (the second key is 'comma'), enter *vm args* in the search bar. The item `Java>Debug>settings: Vm Args` appears with a text bar in which you must write `-ea` (for enable asserts).

Documentation

The official (internet) documentation is here: `Java 11`.

Warning: All your code must be written in the `HW2.java` file

In order to minimize the risk of handling errors when uploading files, all the classes that you have to modify are grouped together in the same `HW2.java` file that you must upload after each question. (We remind you, however, that except in exceptional circumstances, we tend to write different classes in different files, both for the readability of the code and for the efficiency of compilation.)

Problem formulation

The following game is studied. We have a rectangular grid on which we can place and move *fruits*. Each square of the grid can only contain one fruit. There are several types of fruits. As soon as at least three fruits of the same type are adjacent in a row or in a column, they are eliminated and disappear from the grid. The player receives points according to the number of fruits thus eliminated.

Given the dimensions of the grid and the number of types of fruits, we seek to count the *stable* configurations, that is, the configurations where there are never three fruits of the same type adjacent in a row or in a column. These are the configurations where no fruit is eliminated.

The objective of the HW is to count the number of stable configurations in a 10×10 grid with two types of fruits.

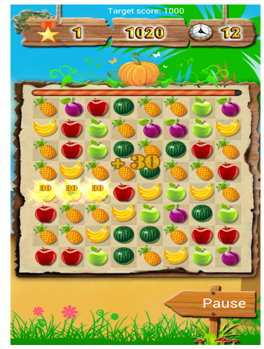

Figure 1: A tile-matching game

1 Game modeling

To count the grids, we will first represent the rows of the grid. The rows are represented by objects of the `Row` class. This class has a `private final int[] fruits` field which is an array of integers, each integer representing a type of fruit.

For simplicity, we assume that there are only two different types of fruits, represented by the two integers `0` and `1`.

In the `Row` class , you are provided with:

- two constructors `Row()` and `Row(int[] fruits)`;
- a `public String toString()` method that gives a nice representation of a line;
- a `public boolean equals(Object o)` method that compares two lines;
- a `public int hashCode()` method that we don't care about at the moment.

In the `Row` class , complete the methods:

- `Row extendedWith(int fruit)` so that it returns a new row constructed from `this` to which we add a new box at the end with a fruit of type `fruit`.
- `static LinkedList<Row> allStableRows(int width)` to generate and return the list of all *stable* rows of width `width` (i.e. that do not contain three consecutive identical fruits).
  Hint: to create all stable rows of width width , we iterate through all stable rows of width `width-1` and add a fruit `fruit` (`0` or `1`) to each row where at least one of the last two fruits is distinct from `fruit` .
- `boolean areStackable(Row r1, Row r2)` to return `true` if `r1` and `r2` are the same length as `this` and can be stacked above or below `this` (or in any other order) without there being three fruits of the same type in the same column.

Test your code by running `Test1.java`.

2 Naive enumeration

We will now proceed to a naive count of stable configurations, working in the `CountConfigurationsNaive` class .

In the `CountConfigurationsNaive` class, complete the recursive method `static long count(Row r1, Row r2, LinkedList<Row> rows, int height)` so that it determines the number of grids whose first rows are `r1` and `r2`, whose rows are `rows`, and whose height is `height`. The algorithm is as follows:

`count(r1, r2, rows, height)` =

- if `height` is less than or equal to 1, return 0;
- if `height` is 2, return 1;
- otherwise,
  - do the summation, for any row `r3` of `rows` that can be stacked on `r1` and `r2` , the result of `count(r2, r3, rows, height-1)`,
  - return this result.

In the `CountConfigurationsNaive` class , complete the `static long count(int n)` method so that it calculates the number of grids with `n` rows and `n` columns.

Test your code by running `Test2.java`.

What do you notice? The program is correct, but with such a bad complexity, it does not finish in a reasonable time beyond `n`= 6. This is due to the fact that `count(r1, r2, rows, height)` is called many times for the same values `r1, r2, height`. To solve the problem, it is enough to store in a table the quadruplet `(r1, r2, height, count(r1, r2, rows, height))` the first time we calculate `count(r1, r2, rows, height)` (the value of rows never changes, so it is not necessary to store it). The following times, we will directly fetch the information (here the number of grids associated with `r1`, `r2` and `height`) from the table without recalculating the value. We could use the Java `HashMap` class to create such a table. However, in order to fully understand the principle of this data structure, we will start by creating a hash table ourselves.

3 Hash table

We will write a data structure that associates two lines `r1` and `r2` and a height `height` with a value of type `long`.

We use the `Quadruple` class to represent the elements of this table, that is, quadruplets `(r1, r2, height, result)` where `r1, r2` are rows of the `Row` class, `height` is an integer of type `int` and `result` is an integer of type `long`. The `Quadruple` class is given in the `HW2.java` file and there is nothing to modify in this class.

A hash table is nothing more than an array of linked lists of quadruplets, where the element with index i contains the quadruplets for which the hash value is i modulo the size of the array. We use the `LinkedList` class to represent the lists and we choose an arbitrary value (here 50000) for the size of the array. We will therefore work with a `HashTable` class containing:

- a `final static int M` constant corresponding to the number of "buckets" in the array, i.e. the number of boxes in the array, which we set to 50000 in what follows.
- a `Vector<LinkedList<Quadruple>> buckets` field which represents the array, that is to say the collection of buckets in which we will store the quadruplets according to their hash value.

3.1 Initialization

Complete the constructor of the `HashTable` class so that each element of the `buckets` array is properly initialized with a `new LinkedList<Quadruple>`.

Warning: a call to `new Vector<...>(M)` returns a new resizable array whose capacity is `M` but whose size is 0. It must then be filled, for example by using the `add` method . See the Vector class documentation.

3.2 Hash function

In the `HashTable` class , extend the `static int hashCode(Row r1, Row r2, int height)` method to compute an arbitrary hash value for the triple `(r1, r2, height)`. Any non-trivial arithmetic formula involving both `r1.hashCode()`, `r2.hashCode()`, and `height` will do. If the formula is too naive, the entries are poorly distributed across the buckets of the array, and memoization efficiency is reduced.

In the `HashTable` class , complete the `static int bucket(Row r1, Row r2, int height)` method so that it returns the value of `hashCode(r1, r2, height)` modulo `M`.

Warning: the result of the Java modulo operator (\%) on negative integers can itself be negative. We will therefore ensure here that the result is between `0` (inclusive) and `M` (exclusive).

3.3 Adding to the table

In the `HashTable` class , complete the `void add(Row r1, Row r2, int height, long result)` method so that it adds the quadruplet `(r1, r2, height, result)` to the bucket indicated by the bucket method. We will not try to check if the entry already exists; we will simply systematically add it to the list.

3.4 Searching the table

In the `HashTable` class , complete the `Long find(Row r1, Row r2, int height)` method so that it searches the table for a quadruplet of the form `(r1, r2, height, result)`. If such a quadruplet exists in the table, a `Long` with the value `result` is returned . Otherwise, `null` is returned.

Warning: it is necessary to use the `Long` class rather than the primitive `long` type to return result when a quadruplet of the form `(r1, r2, height, result)` is found. To convert an integer of primitive type to an object of the `Long` class , we can use `Long()` or `Long.valueOf()`. Also, we can return the `null` value to indicate that no such quadruplet is present in the table.

Test your code by running `Test3.java`.

4 Counting with memoization

We return to the initial combinatoric problem. We now work in the `CountConfigurationsHashTable` class , in which we have a `memo` field of type `HashTable` .

Taking inspiration from your `static long count(Row r1, Row r2, LinkedList<Row> rows, int height)` method of the `CountConfigurationsNaive` class , complete the `static long count(Row r1, Row r2, LinkedList<Row> rows, int height)` method of the `CountConfigurationsHashTable` class by using the `memo` field to remember the calculations already performed.

Taking inspiration from your `static long count(int n)` method of the `CountConfigurationsNaive` class , also complete the `static long count(int n)` method of the `CountConfigurationsHashTable` class so that it calculates the number of grids with `n` rows and `n` columns.

Test your code by running `Test4.java`, which finally calculates the number of 10×10 grids (which should take no more than a few seconds).

5 Using `HashMap`

Taking inspiration from your `CountConfigurationsHashTable` class, complete the `static long count(Row r1, Row r2, LinkedList<Row> rows, int height)` and `static long count(int n)` methods of the `CountConfigurationsHashMap` class, using the `HashMap` class from the Java standard library in place of your `HashTable` class .

Hint: since the keys are triplets `(Row, Row ,int)`, you will need to introduce a `Triple` class to represent them, in order to use the `HashMap<Triple, Long>` instance. This `Triple` class will need to override its `public boolean equals(Object o)`
(**Warning**: this method takes an `Object` as an argument and not a `Triple`) and `public int hashCode()` methods . For this last method, we can call the `static int hashCode(Row r1, Row r2, int height)` method of the `HashTable` class. Adding the `@Override` annotation before these two methods is a good idea.

Test your code by running `Test5.java`.

You have just calculated the diagonal entries of the sequence A203407 from the On-Line Encyclopedia of Integer Sequences. If you liked this problem, you will love Project Euler.