Java exercises

Exercise

Some "sense of ownership"

Introduction:

A rich owner wants to close his land. You must write a program to help calculate the number of meters required for fence.

Description:

The owner has a representation of his land in digitized form: the land is represented through a binary two-dimensional array. 1s are square plates that are part of his field and 0s are those that are not his land.
Thus, the task in this exercise to calculate the number of meters of fence required to surround the land represented in this format. For this, it is necessary to count the number of 1s constituting the perimeter of the field. Each 1 on the perimeter corresponds to 2.50 meters of fence for *each of its sides which are on the edge of the field*.

For example, the land

1
(all alone), which represents a tiny square terrain of 2.5 m on sides, requires 10 (= 4 times 2.5) meters of fence.

The land

0110
0110
needs 20 m of fence, and the land

0111110
0111110
0111110
40 m of fence.

The problem is a little bit more complex due to the fact that the field may contain ponds. The interior of the ponds is also represented by 0s but the edge of a pond should not be accounted in the perimeter of the field.

Example of a digitized terrain with ponds inside:

```
000000000000000000000000000001111111000000
000000000000000000000000000011111111111110
000000000000111111111111111111111111111111
001111111111111111111111111111111111111111
011111111111111111111111111111111111111111
111111111111111111111111111111111111111111
111111111111111111111111111111111111111111
111111111111111111111111111111111111111111
011111111111111001111111111111111111111111
011111111111111001111111111110001111111100
011111111111111001111111111100001111111100
001111111111111001111111111000011111111100
000111111111100011111111111000111111110000
000111111111000000111111111111111111110000
000111111110000000000111111111111111110000
000001111100000000000011111111111110000000
000001111111000000000111111111111100000000
000001111111100000001111111111111110000000
000111111111110000111111111111111111000000
000111111111111011111111111111111100000000
000011111111111101111111111111110000000000
000001111111111111011111111111100000000000
000001111111111111111111111110000000000000
000001111111111111111111111100000000000000
000000011111111111111111111100000000000000
000000111111111111111111111000000000000000
000000111111111111111111111000000000000000
000000111111000000000000000000000000000000
000000110000000000000000000000000000000000
```

which would correspond to the following land:

For simplicity, we assume:

- that the land is “in one piece”: there exists no land in the map that is dis- connected from each other;
- that there is no line that contains only 0s;
- that the outer perimeter of the field is “row convex”, meaning that for each line of the map of the land[1], the only 1s that belong to the outer perimeter are the first and the last ones on the line[2]; we cannot have a line like this: “0011110001111” where 0s in the middle would be 0 outside the field; this kind of 0s necessarily represent a pond.

All this is to ensure that a 0 is inside the field (as part of a pond) if, on its line (but not necessarily on its column! See the example above.), there is at least a 1 before and at least a 1 after.

The code to be produced

The provided code contains messages to display in particular situations described below. You are required to use these lines as they are.
The code you are going to write will first verify that the provided array (map) uniquely consists of 0s and 1s. If this is not the case, an error message strictly respecting the following format will be displayed (with line breaks):

```
Your terrain map does not have the correct format:
value '2' found in position [8] [7]
```

The program stops in this case its execution. Note that the array we give in the provided code will be replaced by other arrays to test your code.
If the array map representing the terrain is valid, your program will display the required number of meters of fence *strictly* respecting the display format given in the execution examples given below (including newline at end of message).

Indication:

- A simple way to proceed is to first “clear” the ponds (replacing the 0s of ponds by 1), then to proceed with the counting of 1s located on the perimeter.
- One point of the perimeter can have multiple 0s as neighbors (for example, a 0 above and another 0 on the left), meaning that each of such sides must be fenced. It must in that case be counted as many times for the fence (twice in the example).
- To force the output of the main method, if you think that the execution of the program has to stop at some point, you can use the return; command.

1 However, we do not make this assumption for columns! See the example above.
2 But there can be only one 1 when the first and the last 1s are the same: “000010000”

Checking the row-convexity

Finally, we ask you to add, between checking that the card does contain only 0s and 1s and calculating the length of the fence, an additional check that the card is properly “row convex”.
Note that this part, more difficult, is completely independent from the rest and you can completely calculate the length of the fence and have some points for this exercise, without having made this last part.
The requested verification will have to reject any map in which the 0s belonging to the exterior the private field are present between 1s of the same line, such as in this map:

The algorithm that we propose to detect these erroneous cards is:

1. find all zones of 0s (called “connected components”);

for example, different zones of 0s in the following image are represented by different colors:

2. find among these zones, those outside the private field (the others being the ponds);
3. browse the picture line by line to see if a zone of 0 outside the field is between two 1s.

Note that you can combine this last step with your “clear” the ponds step (described in the previous section).

In the first stage (finding the connected components), with programming knowledge at this stage of the course we propose to do the following:

1. declare two dynamic arrays of integers; these will be used to store the co- ordinates of points on the map during processing, one for ordinates (i.e., y-axis) i, one for the abscissas (i.e., x-axis) j;

2. declare an integer variable and initialize it with the value 1; we will use this variable to count and label the different zones of 0. For simplicity, let's call it "component"; it will be increased by 1 for each new zone;
3. loop over all the positions of the map; if the value at the current position is 0:
    - Increment the zone counter variable (component) by 1;
    - Insert the ordinate i and the abscissa j of the current position to your dynamic arrays;
    - As these dynamic arrays are not empty:
        - retrieve and delete the values of the first element of each array (one abscissa and one ordinate);
        - if the value of the map (carte) at this newly retrieved position is 0:

∗ assign the value of the zone counter variable (component) to this position of the map;
*for each of the neighbors of the recovered position (adjacent NORTH, SOUTH, EAST and WEST, if they exist): if the value of the neighbor is 0, add its coordinates (abscissa and ordinate) to your dynamic arrays.

If you print the previous map after this step, you should get:

```
22222222222222222222222222211111133333
22222222222222222222222222111111111113
22222222222111111111111111111111111111
22111111111111111111111111111111111111
21111111111111111141111111111111111111
11111111111111111144111111111111111111
11111111111111111144111111111111111111
11111111111111111144111111111111111111
51111111111111661111441111111111111111
51111111111111661111144111117771111144
51111111111111661111114111177771111144
55111111111111661111114441777711111144
55511111111116661111111141777111111114444
55511111111166666611111141111111111114444
55511111116666666666111141111111114444
55555111116666666666611141111111144444444
55555511111666666666111141111111144444444
55555111111166666661111141111111144444444
55511111111116666611114444111111111444444
55511111111111161111141111111111144444444
55551111111111118111141111111114444444444
55555111111111111911141111111444444444444
55555111111111111111141111144444444444444
55555111111111111111141111444444444444444
55555551111111111111111411144444444444444
55555511111111111111111411144444444444444
55555511111111111111111411144444444444444
55555511111444444444444444444444444444444
55555511444444444444444444444444444444444
```

For the second step, you just have to traverse the border of the map and store in a dynamic array all values encountered which are not 1. Although this is not optimal, it does not matter here to repeat several times the same value in this array.

For the third and final step, just browse each line and look for any value that is not 1 between the first and the last 1 of this line. If such a value is present in the arrays constructed in the previous step (array of components of the boundary), then this means that the map is erroneous. The program must then display an error message *strictly* respecting the following format (with line breaks):

```
Your terrain map does not have the correct format:
Incoming outer edge found in position [4] [18]
```

Execution Example:

With the map given in the code and plotted above:

```
You need 385.0 meters of fence for your land.
```

With a map containing a 2 in position i=8, j=7:

```
Your terrain map does not have the correct format:
value '2' found in position [8] [7]
```

With the following plot:
01110
01010
01110
you should get:

```
You need 30.0 meters of fence for your land.
```

And with this one:
111
001
111
you should get:

You need 40.0 meters of fence for your land.

With a map not being “row convex” such as that shown above, you should get:

```
Your terrain map does not have the correct format:
Incoming outer edge found in position [4] [18]
```