Assignment 2

COMP9021, Session 2, 2016

1 Aims

The main purpose of the assignment is to let you practice the following programming techniques:

- work with command line arguments, read from and write to files;
- perform operations on lists of lists, execute tests and repetitions;
- organise a program into well defined functions;
- design recursive procedures;
- gain a first acquaintance with backtracking.

2 General presentation

You will design and implement a program that will

- analyse the various characteristics of a *maze*, represented by a particular coding of its basic constituents into numbers stored in a file whose contents is read, and
- – either display those characteristics
  – or output some Latex code, to be saved in a file, from which a pictorial representation of the maze can be produced.

The representation of the maze is based on a coding with the four digits `0`, `1`, `2` and `3` such that

- `0` codes points that are connected to neither their right nor below neighbours
- `1` codes points that are connected to their right neighbours but not to their below ones:
- `2` codes points that are connected to their below neighbours but not to their right ones:
- `3` codes points that are connected to both their right and below neighbours: 

A point that is connected to none of their left, right, above and below neighbours represents a *pillar*: 

Analysing the maze will allow you to also represent:

- *cul-de-sacs*: 
- certain kinds of *paths*:

3 Examples

3.1 First example

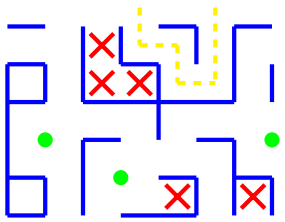
Given a file named `maze1.txt` whose contents is

```
1 0 2 2 1 2 3 0

3 2 2 1 2 0 2 2

3 0 1 1 3 1 0 0

2 0 3 0 0 1 2 0

3 2 2 0 1 2 3 2

1 0 0 1 1 0 0 0
```

your program when run as `python3 maze.py --file maze1.txt` should output

```
The maze has 12 gates.
The maze has 8 sets of walls that are all connected.
The maze has 2 inaccessible inner points.
The maze has 4 accessible areas.
The maze has 3 sets of accessible cul-de-sacs that are all connected.
The maze has a unique entry-exit path with no intersection not to cul-de-sacs.
```

and when run as `python3 maze.py -print --file maze1.txt` should produce some output saved in a file named `maze1.tex`, which can be given as argument to `pdflatex` to produce a file named `maze1.pdf` that views as follows.



3.2 Second example

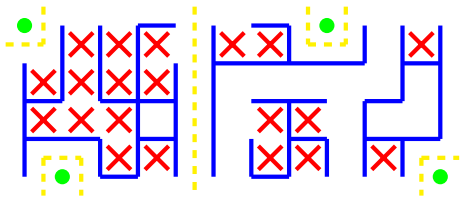
Given a file named `maze2.txt` whose contents is

```
022302120222
222223111032
301322130302
312322232330
001000100000
```

your program when run as `python3 maze.py --file maze2.txt` should output

```
The maze has 20 gates.
The maze has 4 sets of walls that are all connected.
The maze has 4 inaccessible inner points.
The maze has 13 accessible areas.
The maze has 11 sets of accessible cul-de-sacs that are all connected.
The maze has 5 entry-exit paths with no intersections not to cul-de-sacs.
```

and when run as `python3 maze.py -print --file maze2.txt` should produce some output saved in a file named `maze2.tex`, which can be given as argument to `pdflatex` to produce a file named `maze2.pdf` that views as follows.



3.3 Third example

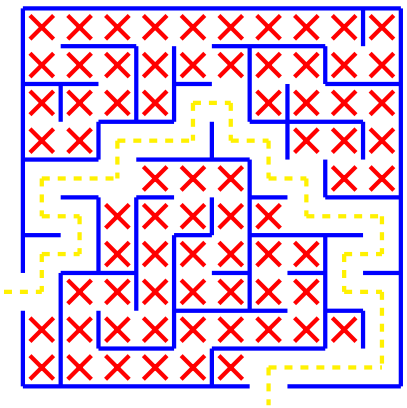
Given a file named `labyrinth1.txt` whose contents is

```
31111111132
21122131202
33023022112
20310213122
31011120202
21230230112
30223031302
03122121212
22203110322
22110311002
11111101110
```

your program when run as `python3 maze.py --file labyrinth1.txt` should output

```
The maze has 2 gates.
The maze has 2 sets of walls that are all connected.
The maze has no inaccessible inner point.
The maze has a unique accessible area.
The maze has 8 sets of accessible cul-de-sacs that are all connected.
The maze has a unique entry-exit path with no intersection not to cul-de-sacs.
```

and when run as `python3 maze.py -print --file labyrinth1.txt` should produce some output saved in a file named `labyrinth1.tex`, which can be given as argument to `pdflatex` to produce a file named `labyrinth1.pdf` that views as follows.



4 Detailed description

4.1 Input

The input is expected to consist of y_{dim} lines of x_{dim} members of $\{0, 1, 2, 3\}$, where x_{dim} and y_{dim} are at least equal to 2 and at most equal to 31 and 41, respectively, with possibly lines consisting of spaces only that will be ignored and with possibly spaces anywhere on the lines with digits. If n is the x^{th} digit of the y^{th} line with digits, with $0 \leq x < x_{dim}$ and $0 \leq y < y_{dim}$, then

- n is to be associated with a point situated $x \times 0.5$ cm to the right and $y \times 0.5$ cm below an origin,
- n is to be connected to the point 0.5 cm to its right just in case $n = 1$ or $n = 3$, and
- n is to be connected to the point 0.5 cm below itself just in case $n = 2$ or $n = 3$.

The last digit on every line with digits cannot be equal to 1 or 3, and the digits on the last line with digits cannot be equal to 2 or 3, which ensures that the input encodes a maze, that is, a grid of width $(x_{dim} - 1) \times 0.5$ cm and of height $(y_{dim} - 1) \times 0.5$ cm (hence of maximum width 15 cm and of maximum height 20 cm), with possibly gaps on the sides and inside. A point not connected to any of its neighbours is thought of as a pillar; a point connected to at least one of its neighbours is thought of as part of a wall.

We talk about *inner point* to refer to a point that lies $(x+0.5) \times 0.5$ cm to the right of and $(y+0.5) \times 0.5$ cm below the origin with $0 \leq x < x_{dim} - 1$ and $0 \leq y < y_{dim} - 1$.

4.2 Output

If not run as either `python3 maze.py --file filename.txt` or as `python3 maze.py -print --file filename.txt` (where `filename.txt` is the name of a file that stores the input) then the program should print out a single line that reads

```
I expect --file followed by filename and possibly -print as command line arguments.
```

and immediately exit. Otherwise, if the input is incorrect, that is, does not satisfy the conditions spelled out in the previous section, then the program should print out a single line that reads

```
Incorrect input.
```

and immediately exit.

4.2.1 When the program is run without -print as command-line argument

If the input is correct and the program is run as `python3 maze.py --file filename.txt` (where `filename.txt` is the name of a file that stores the input) then the program should output a first line that reads one of

```
The maze has no gate.
The maze has a single gate.
The maze has N gates.
```

with `N` an appropriate integer at least equal to 2, a second line that reads one of

```
The maze has no wall.
The maze has a single wall that are all connected.
The maze has N sets of walls that are all connected.
```

with `N` an appropriate integer at least equal to 2, a third line that reads one of

```
The maze has no inaccessible inner point.
The maze has a unique inaccessible inner point.
The maze has N inaccessible inner points.
```

with `N` an appropriate integer at least equal to 2, a fourth line that reads one of

```
The maze has no accessible area.
The maze has a unique accessible area.
The maze has N accessible areas.
```

with `N` an appropriate integer at least equal to 2, a fifth line that reads one of

```
The maze has no accessible cul-de-sac.
The maze has accessible cul-de-sacs that are all connected.
The maze has N sets of accessible cul-de-sacs that are all connected.
```

with `N` an appropriate integer at least equal to 2, and a sixth line that reads one of

```
The maze has no entry-exit path with no intersection not to cul-de-sacs.
The maze has a unique entry-exit path with no intersection not to cul-de-sacs.
The maze has N entry-exit paths with no intersections not to cul-de-sacs.
```

with `N` an appropriate integer at least equal to 2.

- A gate is any pair of consecutive points on one of the four sides of the maze that are not connected.
- An inaccessible inner point is an inner point that cannot be reached from any gate.
- An accessible area is a maximal set of inner points that can all be accessed from the same gate (so the number of accessible inner points is at most equal to the number of gates).
- A set of accessible cul-de-sacs that are all connected is a maximal set S of connected inner points that can all be accessed from the same gate g and such that for all points p in S, if p has been accessed from g for the first time, then either p is in a dead end or moving on without ever getting back leads into a dead end.
- An entry-exit path with no intersections not to cul-de-sacs is a maximal set S of connected inner points that go from a gate to another (necessarily different) gate and such that for all points p in S, there is only one way to move on from p without getting back and without entering a cul-de-sac.

Pay attention to the expected format, including spaces. Note that your program should output no blank line. For a given test, the output of your program will be compared with the expected output; your program will pass the test if and only if both outputs are absolutely identical, character for character, including spaces. For the provided examples, the expected outputs are available in files that end in `_output.txt`. To check that the output of your program on those examples is correct, you can redirect it to a file and compare the contents of that file with the contents of the appropriate `_output.txt` file using the `diff` command. If `diff` silently exits then your program passes the test; otherwise it fails it. For instance, run

```
python3 maze.py --file maze1.txt >maze1_my_output.txt
```

and then

```
diff maze1_my_output.txt maze1_output.txt
```

to check whether your program succeeds on the first provided example.

4.3 When the program is run with -print as command-line argument

If the input is correct and the program is run as If the input is correct and the program is run as `python3 maze.py -print --file` *`filename`*`.txt` (where *`filename`*`.txt` is the name of a file that stores the input) then the program should output some lines saved in a file named *`filename`*`.tex`, that can be given as an argument to `pdflatex` to produce a file named *`filename`*`.pdf` that depicts the maze. The provided examples will show you what *`filename`*`.tex` should contain.

- Walls are drawn in blue. There is a command for every longest segment that is part of a wall. Horizontal segments are drawn starting with the topmost leftmost segment and finishing with the bottommost rightmost segment. Then vertical segments are drawn starting with the topmost leftmost segment and finishing with the bottommost rightmost segment.
- Pillars are drawn as green circles.
- Inner points in accessible cul-de-sacs are drawn as red crosses.
- The paths with no intersection not to cul-de-sacs are drawn as dashed yellow lines. There is a command for every longest segment on such a path. Horizontal segments are drawn starting with the topmost leftmost segment and finishing with the bottommost rightmost segment, with those segments that end at a gate sticking out by 0.25 cm. Then vertical segments are drawn starting with the topmost leftmost segment and finishing with the bottommost rightmost segment, with those segments that end at a gate sticking out by 0.25 cm.

Pay attention to the expected format, including spaces and blank lines. Lines that start with `%` are comments; there are 4 such lines, that have to be present even when there is no item to be displayed of the kind described by the comment. The output of your program redirected to a file will be compared with the expected output saved in a file (of a different name of course) using the `diff` command. For your program to pass the associated test, `diff` should silently exit, which requires that the contents of both files be absolutely identical, character for character, including spaces and blank lines. Check your program on the provided examples using the associated `.tex` files. For instance, run

```
python3 maze.py -print --file maze1.txt
```

and then

```
diff maze1.tex maze1_expected.tex
```

to check whether your program succeeds on the first provided example.

5 Submission and assessment

5.1 Submission

Your program will be stored in a file named `maze.py`. After you have developed and tested your program, upload your files using WebCMS. Assignments can be submitted more than once: the last version is marked. Your assignment is due by October 2, 11:59pm.

5.2 Assessment

Up to 8 marks will reward correctness of solutions by automatically testing your program on some tests, all different to the provided examples. Read carefully the part on program output to maximise your chances of not failing some tests for stupid reasons. For each test, the automarking script will let your program run for 30 seconds. Still you should not take advantage of this and strive for a solution that gives an immediate output for any input.

Up to 2 marks will reward good comments, good choice of names for identifiers and functions, readability of code, simplicity of statements, compactness of functions. This will be determined manually.

Late assignments will be penalised: the mark for a late submission will be the minimum of the awarded mark and 10 minus the number of full and partial days that have elapsed from the due date.

The outputs of your programs should be **exactly** as indicated.

5.3 Reminder on plagiarism policy

You are permitted, indeed encouraged, to discuss ways to solve the assignment with other people. Such discussions must be in terms of algorithms, not code. But you must implement the solution on your own. Submissions are routinely scanned for similarities that occur when students copy and modify other people's work, or work very closely together on a single implementation. Severe penalties apply to a submission that is not the original work of the person submitting it.