

## 1) Introduction:

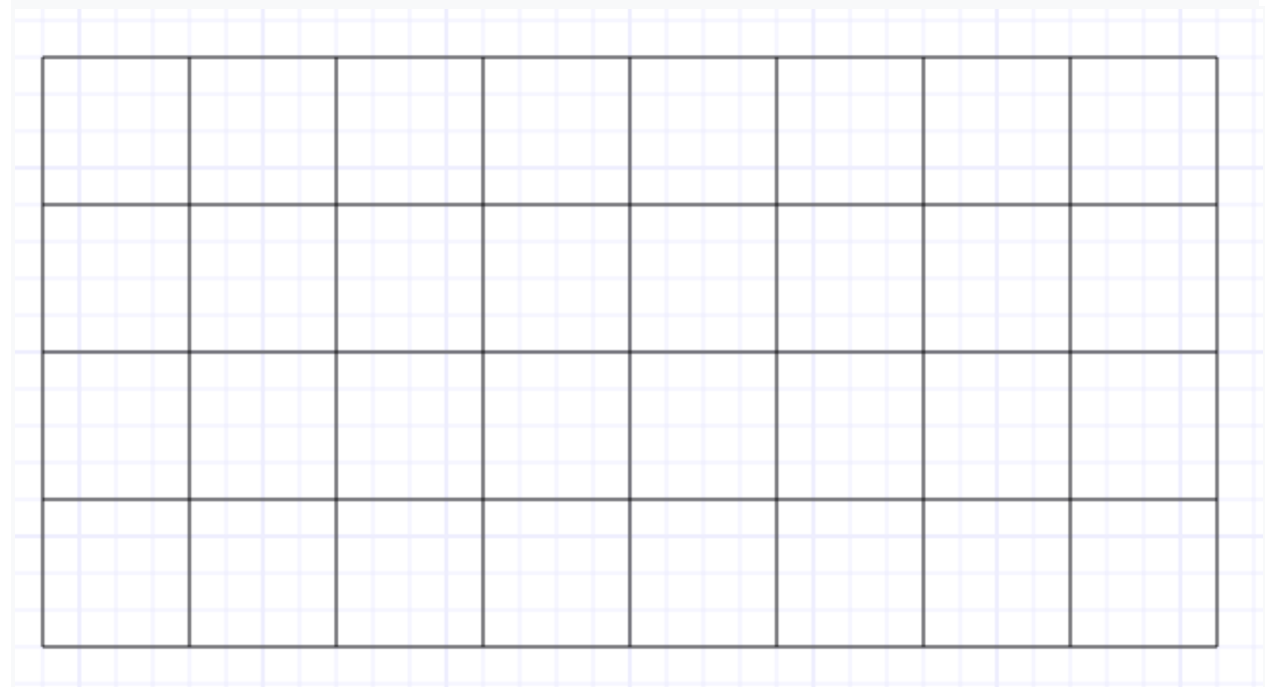
This practical work consists of developing a set of definitions of functions which make it possible to generate mazes and draw them using the turtle.

In this work, you have to imagine that you are the programmer of a library that allows you to generate labyrinths. You must write functions that will be used by other programmers to make specific drawings. (create a labyrinth of a certain size).

The algorithm for generating mazes is imposed on you and is described in the next section. Your task is to code this algorithm, and the necessary helper functions, in Python using Turtle.

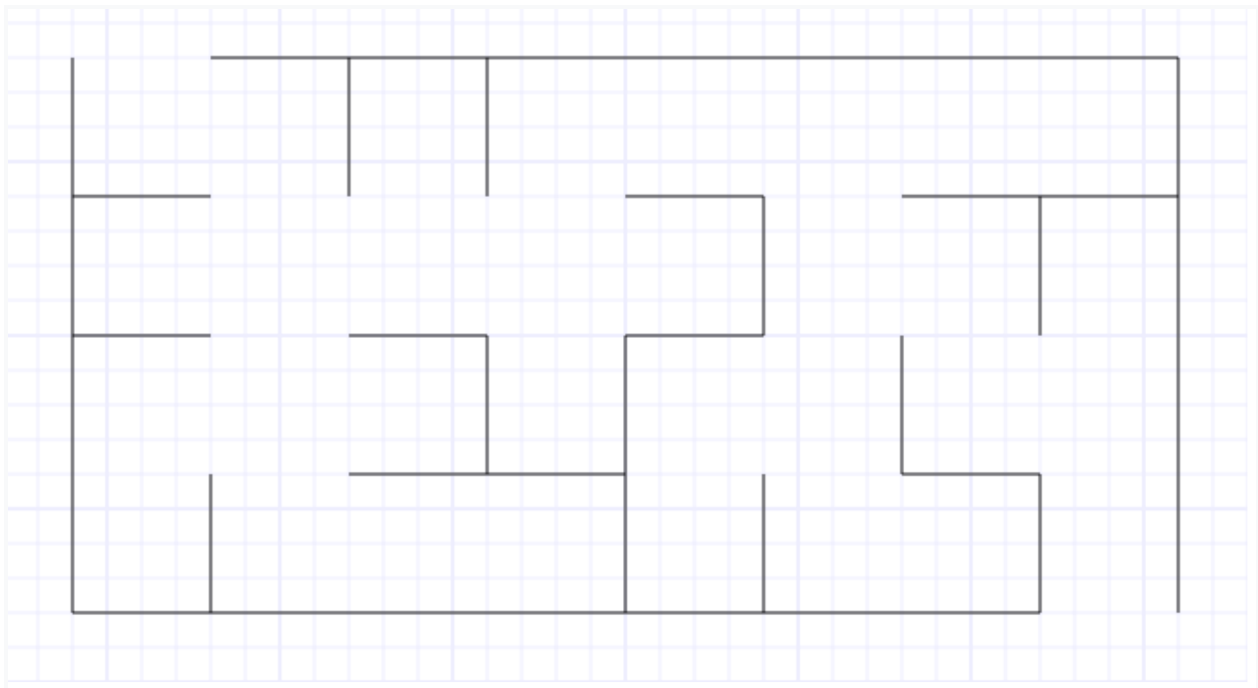
## 2) Labyrinths

The mazes you need to generate are based on a rectangular grid like this:



This grid is made up of square cells. In this example there are 8 columns and 4 rows of cells. For a given cell there are 4 walls that we will call the North, East, South, and West walls (to simplify \ N ", \ E", \ S ", \ O") of the cell. A labyrinth has a certain width in number of cells (NX), a certain height in number of cells (NY), and a certain "step"(the width and height of each square cell, in number of pixels). The labyrinth is therefore created by eliminating certain walls from the grid. The outer wall of the grid is drilled at the top left (the entrance to the labyrinth) and lower right (the exit from the labyrinth).

The creation of a labyrinth consists of determining which walls in each cell need to be removed. For each cell it is necessary to eliminate at least one of its 4 walls, and possibly all 4. Here is a labyrinth using the previous grid (i.e. with  $NX = 8$ ,  $NY = 4$ , step = 40):



The choice of the walls to eliminate cannot be done completely randomly because this could generate a drawing, which is not a labyrinth, that is to say a drawing which does not follow a path from the entrance to the exit. The algorithm you should use uses random numbers to create the maze but ensures that there always is exactly one path between entering and exiting the maze. In fact, the algorithm guarantees that there is always exactly one path between any cell and any other cell (what is formally called an underlying tree). Before explaining the algorithm we must first number each horizontal wall and each vertical wall and give a coordinate (x, y) to each cell as follows:

0	1	2	3	4	5	6	7	
0 (0,0)	1 (1,0)	2 (2,0)	3 (3,0)	4 (4,0)	5 (5,0)	6 (6,0)	7 (7,0)	8
8	9	10	11	12	13	14	15	
9 (0,1)	10 (1,1)	11 (2,1)	12 (3,1)	13 (4,1)	14 (5,1)	15 (6,1)	16 (7,1)	17
16	17	18	19	20	21	22	23	
18 (0,2)	19 (1,2)	20 (2,2)	21 (3,2)	22 (4,2)	23 (5,2)	24 (6,2)	25 (7,2)	26
24	25	26	27	28	29	30	31	
27 (0,3)	28 (1,3)	29 (2,3)	30 (3,3)	31 (4,3)	32 (5,3)	33 (6,3)	34 (7,3)	35
32	33	34	35	36	37	38	39	

It should be noted that the coordinate system (x, y) of the cells places (0,0) at the top left (x grows towards the right, and grows down there). Also notice that there are NX (NY + 1) horizontal walls and (NX + 1) NY vertical walls. So the horizontal walls are numbered from 0 to (NX (NY + 1)) 1 and the vertical walls from 0 to ((NX + 1) NY) 1. The relation between the (x, y) coordinate of a cell and the numbers of its walls N, E, S, O is the next one :

$$N = x + y \times NX$$

$$E = 1 + x + y \times (NX + 1)$$

$$S = x + (y + 1) \times NX$$

$$O = x + y \times (NX + 1)$$

Note that the wall number N can be used to uniquely identify a cell with only one number. For example in the grid above the cell in position (5,2) has the number 21, that is to say 5 + 2 NX.

The algorithm uses sets of numbers (in the mathematical sense of a set, i.e. a collection of numbers without duplication). We can define `wallsH` as the set of horizontal walls (and respectively `wallsV` as the set of vertical walls) which have not been removed by the maze creation algorithm. The information contained in these sets can be combined with the values of `NX` and `NY` to draw a labyrinth by tracing a horizontal line for each horizontal wall whose number is always in `wallsH` and a vertical line for each vertical wall whose number is always in `wallsV`.

The algorithm considers that the grid is an initially full space except for a randomly chosen cell, which is empty (this is the initial cavity). At each iteration of the algorithm a new cell will be chosen randomly among all the cells neighboring the cavity (but not forming part of the cavity) and one of the walls (i.e. horizontal or vertical) which separates it from the cavity will be removed randomly to form a larger cavity (i.e. from the whole `wallsH` if it is a horizontal wall or of the `wallsV` assembly if it is a vertical wall). This process is repeated until all the cells of the grid are part of the cavity.

The choice of the next cell to add to the cavity can be done simply by keeping at all times two other sets of numbers: `cellar` and `front`. These are sets that contain cell numbers. The `cellar` set is the set of cells that have been put into the cavity by the algorithm. The `front` set is the set of cells that are neighboring cells in the cavity (but not in the cavity). We can maintain and update these sets as new cells that are selected to add to the cavity. Indeed if we have add to the cavity the cells with coordinates  $(x, y)$  contained in the `front` set, we must add the neighboring cells with coordinates  $(x, y)$  horizontally and vertically (but not in the `cellar` assembly) a the `front` assembly and remove the cell  $(x, y)$  of the `front` set and add the cell  $(x, y)$  to the `cellar` set. Note that we must remove or add from these sets the number of cells.

The coding of this algorithm will be greatly simplified by the writing of set manipulation functions. These functions are described in the next section. Use them judiciously to code the algorithm labyrinth creation. We will use lists of numbers to represent sets. For example the table `[9,2,5]` represents the set which contains the three elements 2, 5 and 9.

### 3) Specifications:

Function `iota (n)`:

This function takes a non-negative integer `n` as parameter and returns an array of length `n` containing in order the integer values from 0 to `n - 1` inclusive. For example :

`iota(5) = [0,1,2,3,4]`

Function `contain(tab,x)`:

This function takes as parameters an array of numbers (`tab`) and a number `x` and returns a boolean indicating if `x` is contained in the array by traversing this array with a loop (not the right to use the keyword `in`).

For example :

`contain([9,2,5], 2) = True`

`contain([9,2,5], 7) = False`

Function `add(tab,x)`:

This function takes an array of numbers (`tab`) and a number `x` as parameters and returns a new array with the same content as `tab` except that `x` is appended to the end if it is not already contained in `tab`. For example :

`add([9,2,5], 2) = [9,2,5]`

`add([9,2,5], 7) = [9,2,5,7]`

Function `remove(tab,x)`:

This function takes an array of numbers (`tab`) and a number `x` as parameters and returns a new array with the same contents as `tab` except that `x` is removed from the array. For example :

`remove([9,2,5], 2) = [9,5]`

`remove([9,2,5], 7) = [9,2,5]`

Function `neighbors(x,y,nx,ny)`:

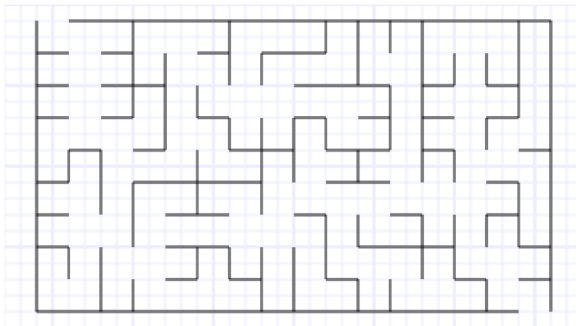
This function takes the coordinate (x, y) of a cell and the size of a grid (width = nx and height = ny) and returns an array containing the number of neighboring cells. For example :

`neighbors(7, 2, 8, 4) = [15,22,31]`

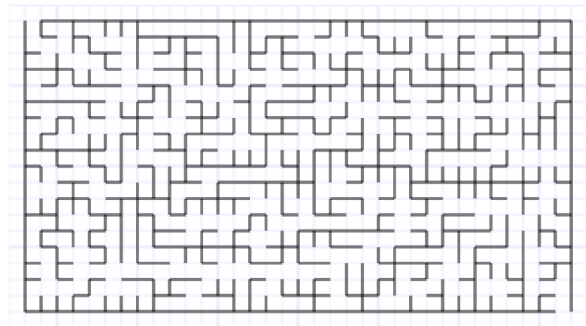
Function `laby(nx,ny, step):`

This procedure creates a random maze (width = nx and height = ny) and draws this maze in the center of the drawing window using a grid with cells of pixel pitch width and height. For example, here are two laby calls and the drawings obtained in each case:

`laby(16, 9, 20);`



`laby(34, 18, 10);`



20% bonus points for the implementation of the `labySol` procedure (nx, ny, pas) which draws a labyrinth exactly like the `laby` procedure but which in addition draws in the labyrinth, in red, the path traversed by Pledge's algorithm to exit the labyrinth from the top left entrance.