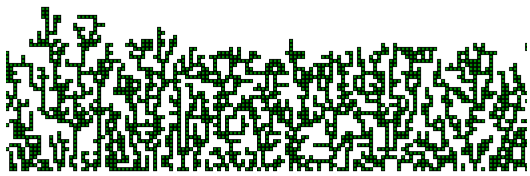
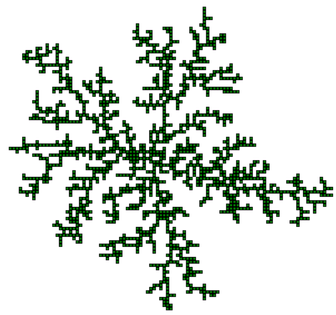
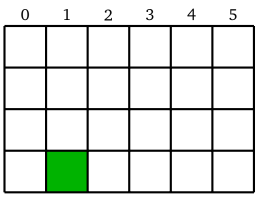
Random blocks

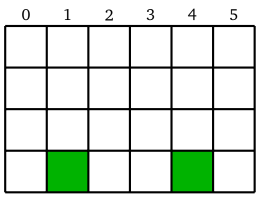
You will program two methods to build figures that look like algae or corals. Each figure is made up of small randomly thrown blocks that stick together.

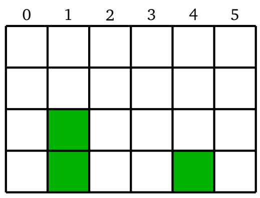
Lesson 1 (Fall of blocks)**.**
Square blocks are dropped into a grid, on the principle of the game "Connect 4": after choosing a column, a block falls from top to bottom. The blocks are placed on the bottom of the grid or on other blocks or next to other blocks. There is a big difference with the game "Connect 4", here the blocks are "sticky", i.e. a block stays stuck as soon as it meets a neighbor on the left or right.
Here is an example of how to throw blocks:



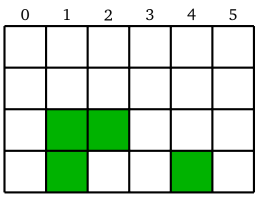
1. Block fallen from col. 1



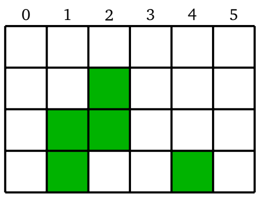
2. Block fallen from col. 4



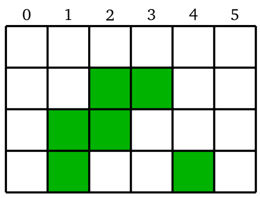
3. Block fallen from col. 1



4. Block fallen from col. 2

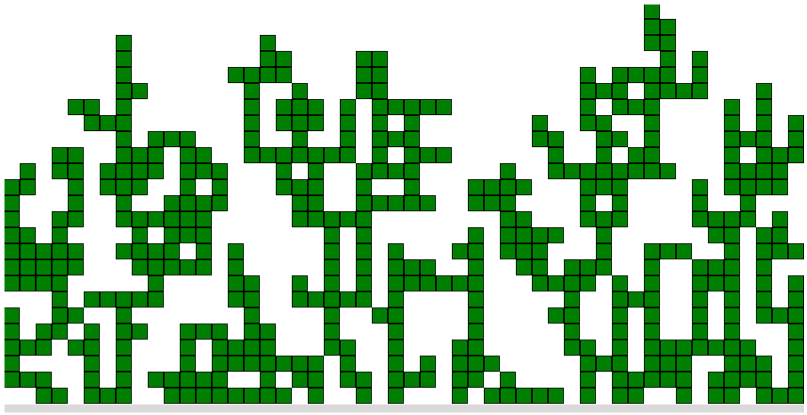


5. Block fallen from col. 2



6. Block fallen from col. 3

For example, in step 4, the block thrown in the column 2 does not go down to the bottom but remains hung on its neighbor, so it is permanently suspended.
The random throwing of hundreds of blocks on a large grid produces pretty geometric shapes resembling algae.



Activity 1 (Fall of blocks)**.**

Goal: program the drop of the blocks (without graphic display).

The workspace is modeled by an array of n rows and p columns. At the beginning the array only contains 0's; then the presence of a block is represented by 1.
Here is how to initialize the table:

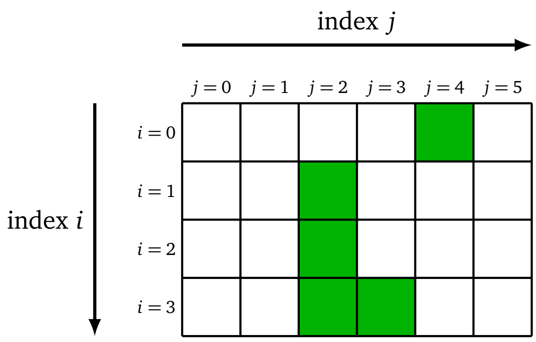
```
array = [[0 for j in range(p)] for i in range(n)]
```

The table is modified by instructions of the type:

```
array[i][j] = 1
```

Here is an example of a table (left) to represent the graphical situation on the right (the block at the top right is falling).

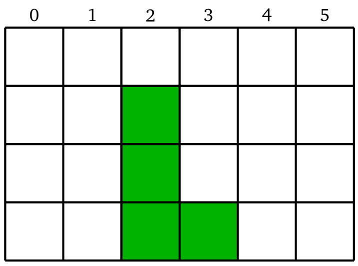
0	0	0	0	1	0
0	0	1	0	0	0
0	0	1	0	0	0
0	0	1	1	0	0



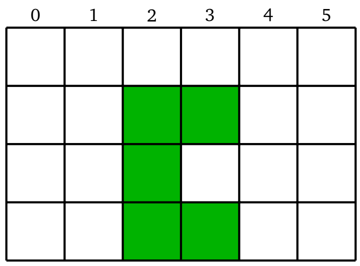
An array with 5 blocks ($n=4$, $p=6$)

1. Program a `can_fall(i,j)` function that determines if the block in position (i,j) can drop to the square below or not.
   Here are the cases in which the block *cannot* fall:
   - if the block is already on the last line,
   - if there is a block just below,
   - if there is a block just to the right or just to the left.
2. Program a `drop_one_block(j)` function which drops a block in the j column until it can no longer go down. This function modifies the array.
   For example, here is the table before (left) and after (right) dropping a block in the $j=3$ column.

$$\begin{array}{cccccc} 0&0&0&0&0&0\\ 0&0&1&0&0&0\\ 0&0&1&0&0&0\\ 0&0&1&1&0&0 \end{array} \qquad \begin{array}{cccccc} 0&0&0&0&0&0\\ 0&0&1&1&0&0\\ 0&0&1&0&0&0\\ 0&0&1&1&0&0 \end{array}$$



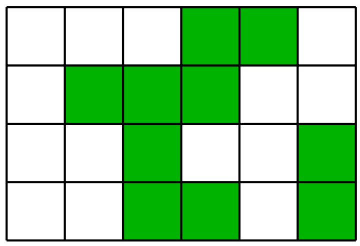
Before



After

3. Program a `drop_blocks(k)` function that launches k blocks one by one, each time choosing a random column (i.e. an integer j with $0 \leqslant j < p$).
   Here is an example of a table obtained after throwing 10 blocks:

$$\begin{array}{cccccc} 0&0&0&1&1&0\\ 0&1&1&1&0&0\\ 0&0&1&0&0&1\\ 0&0&1&1&0&1 \end{array}$$

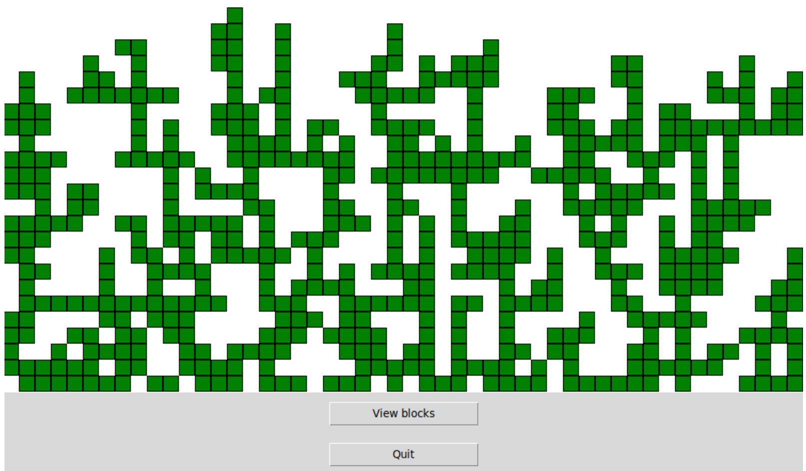


Throw of 10 blocks

Activity 2 (Falling blocks (continued)).

Goal: program the graphic display of the blocks.

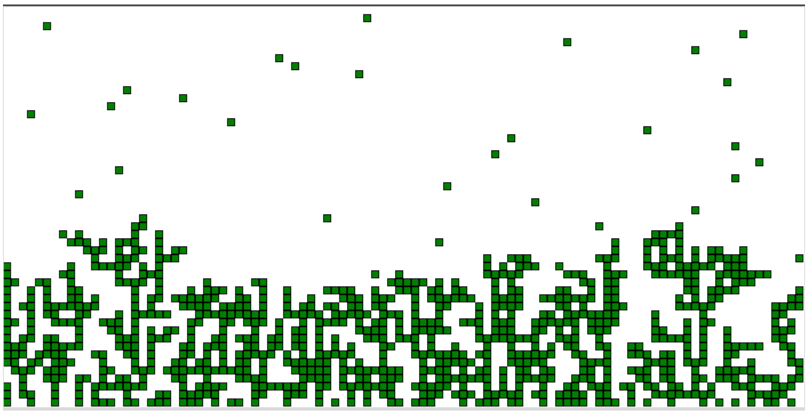
Static display. Program the graphical display of blocks from an array.



Hints.

- Use the module `tkinter`, see chapter "Statistics – Data visualization".
- You can add a button that launches a block (or several at once).

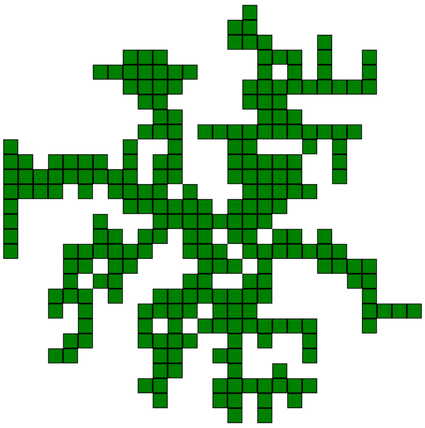
Dynamic display (optional and difficult). Program the display of falling blocks.



Hints.

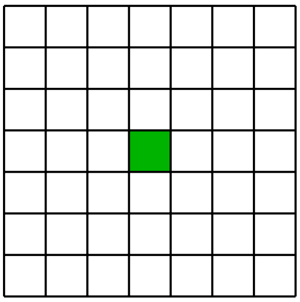
- It's much more complicated to program, but very nice to see!
- For moving the blocks, use the program "Moves with `tkinter`" at the end of this chapter.
- How to make a "block rain"? On a regular basis (for example every tenth of a second) all existing blocks are dropped by one square and a new one appears on the top line.

Lesson 2 (Brownian trees).
Here is a slightly different construction, much longer to calculate, but which also draws pretty figures called "brownian trees".

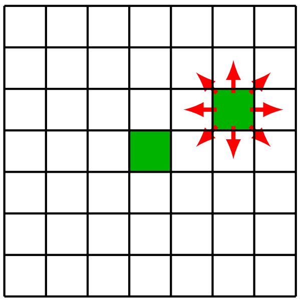


The principle is as follows:

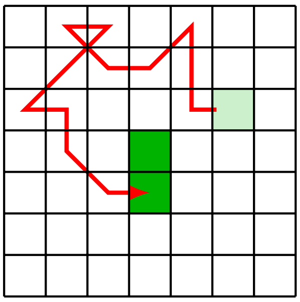
- We start from a grid (this time we have to imagine that it is drawn flat on a table). In its center, we place a first fixed block, the *seed*.
- A random new block is created on the grid. At each step, this block moves at random to one of the eight adjacent squares, we are talking about a *brownian movement*.
- As soon as this block touches another block from one side, it sticks to it and no longer moves.
- If the block leaves the grid, it disintegrates.
- Once the block has been glued or disintegrated, a new block is then restarted from a random point on the grid.



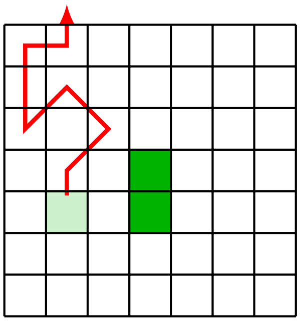
The seed



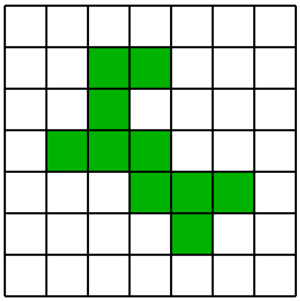
A new block and its 8 possible moves



Random move of the block



A block leaves the grid



10 blocks

Gradually, we obtain a kind of tree that looks like coral. The calculations are very long because many blocks come out of the grid or take a long time to fix (especially at the beginning). In addition, the blocks can only be thrown one by one.

Activity 3 (Brownian trees)**.**

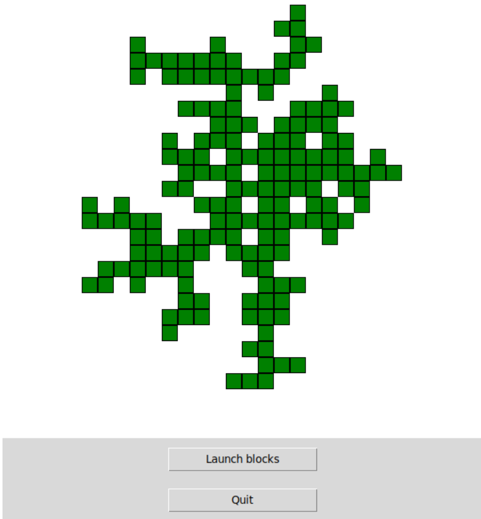
Goal: program the creation of a brownian tree.

Part 1.

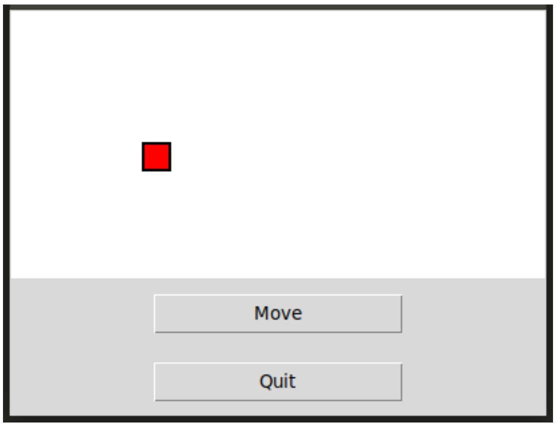
1. Model the workspace again with an array of n rows and p columns containing 0's or 1's. Initialize all values to 0, except 1 in the center of the table.
2. Program an `is_inside(i,j)` function which determines if the position (i, j) is in the grid (otherwise the block is coming out).
3. Program a `can_move(i,j)` function that determines if the block in position (i, j) can move (the function returns `True`) or if it is hung on (the function returns `False`).
4. Program a `launch_one_block()` function, without parameters, that simulates the creation of a block and its random movement, until it sticks or leaves the grid.
   Hints.
   - The block is created at a random position (i, j) of the grid.
   - As long as the block is in the grid and free to move:
     - you choose a horizontal move by randomly drawing an integer from $\{-1, 0, +1\}$, the same for a vertical move;
     - you move the block according to the combination of these two movements.
   - Then modify the table.
5. End with a `launch_blocks(k)` function that launches k blocks.

Second part.

Program the graphic display using `tkinter`. You can add a button that launches 10 blocks at once.



Lesson 3 (Moves with "tkinter").
Here is a program that moves a small square and bounces it off the edges of the window.



Here are the main points:

- A `rect` object is defined, it is a global variable, as well as its coordinates `x0`, `y0`.
- This object is (a little bit) moved by the `mymove()` function which shifts the rectangle by `(dx,dy)`.
- The key point is that this function will be executed again after a short period of time. The command:
  `canvas.after(50,mymove)`
  requests a new execution of the `mymove()` function after a short delay (here 50 milliseconds).
- The repetition of small shifts simulates movement.

```
from tkinter import *

the_width = 400
the_height = 200

root = Tk()
canvas = Canvas(root, width=the_width, height=the_height, background="white")
canvas.pack(fill="both", expand=True)
```

```
# Coordinates and speed
x0, y0 = 100,100
dx = +5  # Horizontal speed
dy = +2  # Vertical speed

# The rectangle to move
rect = canvas.create_rectangle(x0,y0,x0+20,y0+20,width=2,fill="red")

# Main function
def mymove():
    global x0, y0, dx, dy

    x0 = x0 + dx  # New abscissa
    y0 = y0 + dy  # New ordinate

    canvas.coords(rect,x0,y0,x0+20,y0+20)  # Move

    if x0 < 0 or x0 > the_width:
        dx = -dx  # Change of horizontal direction
    if y0 < 0 or y0 > the_height:
        dy = -dy  # Change of vertical direction

    canvas.after(50,mymove)  # Call after 50 milliseconds

    return

# Function for the button
def action_move():
    mymove()
    return

# Buttons
button_move = Button(root,text="Move", width=20, command=action_move)
button_move.pack(pady=10)

button_quit = Button(root,text="Quit", width=20, command=root.quit)
button_quit.pack(side=BOTTOM, pady=10)

root.mainloop()
```