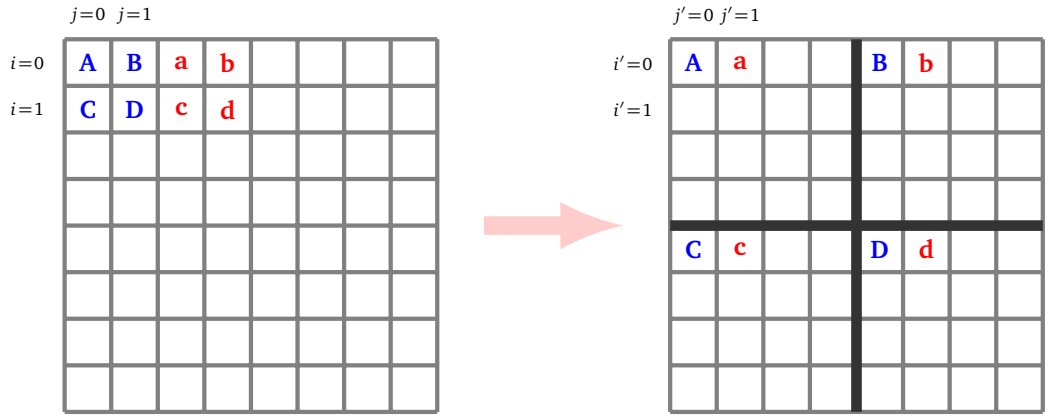
Dynamic images

We will distort images. By repeating these distortions, the images become blurred. But by a miracle after a certain number of repetitions the original image reappears!

Lesson 1 (Photo booth transformation)**.**
We start from an $n \times n$ array, with even n, each element of the table represents a pixel. The rows are indexed from $i = 0$ to $i = n-1$, the columns from $j = 0$ to $j = n-1$. From this image we calculate a new image by moving each pixel according to a transformation, called the ***photo booth transformation***.
We cut the original image into small 2×2 squares. Each small square is therefore composed of four pixels. Each of these pixels is sent to four different locations in the new image: the pixel at the top left remains in an area at the top left, the pixel at the top right of the small square is sent to an area at the top right of the new image,...



For example, the pixel in position $(1,1)$ (symbolized by the letter **D**) is sent to position $(4,4)$.

Let's explain this principle through formulas. For each pair (i,j), we calculate its image (i',j') using the photo booth transformation according to the following formulas:

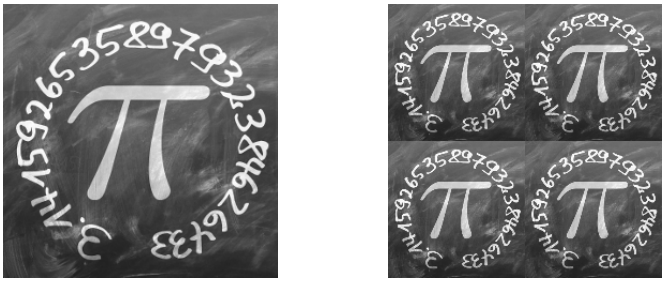
- If i and j are even: $(i',j') = (i//2, j//2)$.
- If i is even and j is odd: $(i',j') = (i//2, (n+j)//2)$.
- If i is odd and j is even: $(i',j') = ((n+i)//2, j//2)$.
- If i and j are odd: $(i',j') = ((n+i)//2, (n+j)//2)$.

Here is an example of a 4×4 array before (left) and after (right) the photo booth transformation.

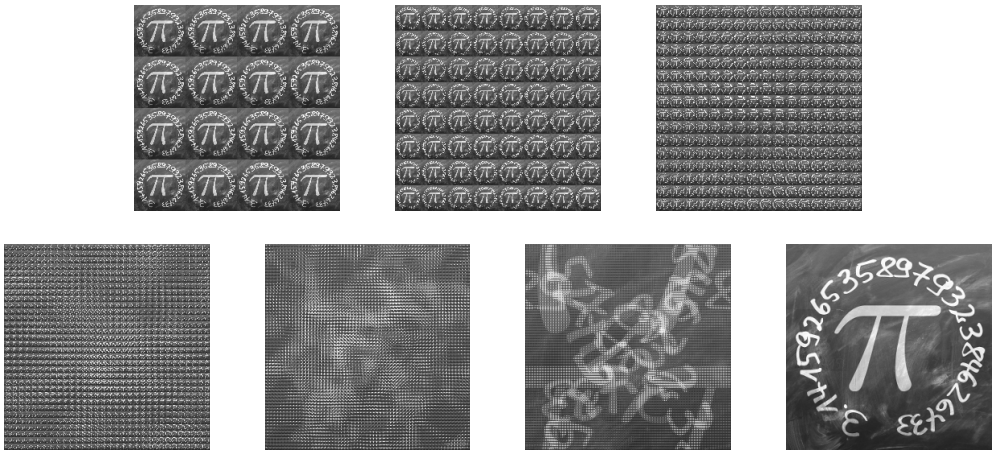
1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

1	3	2	4
9	11	10	12
5	7	6	8
13	15	14	16

Here is a 256×256 image and its first transformation:



Here is what happens if you repeat the photo booth transformation several times:



The image becomes more and more blurred, but after some number transformation, we fall back to the original image!

Activity 1 (Photo booth transformation)**.**

Goal: program the photo booth transformation that decomposes an image into sub-pictures. When this transformation is iterated, the image gradually disintegrates, then suddenly re-formes.

1. Program a `transformation(i,j,n)` function that uses the photo booth transformation formula and returns the new coordinates (i', j') of the pixel (i, j), in the image.
   For example, `transformation(1,1,8)` returns `(4,4)`.
2. Program a `photo_booth(array)` function that returns the table calculated by completing a transformation.
   For example, the array on the left is transformed into the array on the right.

$$\begin{array}{cccc} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{array} \qquad \begin{array}{cccc} 1 & 3 & 2 & 4 \\ 9 & 11 & 10 & 12 \\ 5 & 7 & 6 & 8 \\ 13 & 15 & 14 & 16 \end{array}$$

   Hints. You can initialize a new table with the command:

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

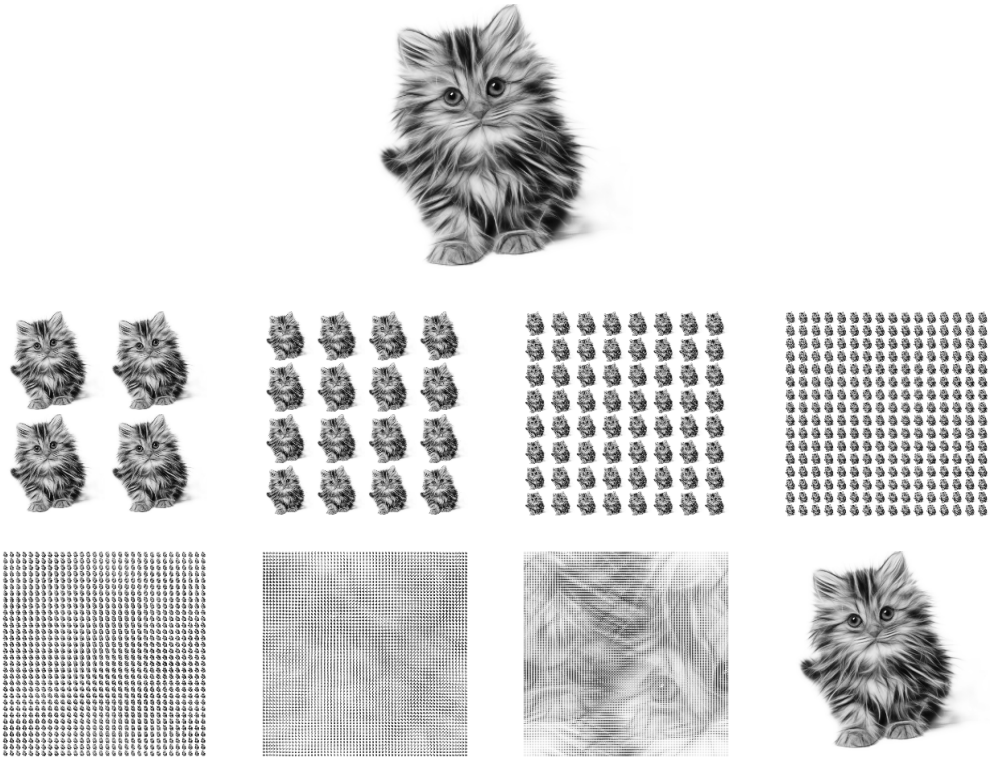
   Then fill it with commands of the type:

```
new_array[ii][jj] = array[i][j]
```

3. Program a `photo_booth_iterate(array,k)` function that returns the table calculated after k iterations of the photo booth transformation.

4. *To be finished after completing activity 2.*
   Program a `photo_booth_images(image_name,kmax)` function that calculates the images corresponding to the transformation, for all iterations from $k = 1$ to $k = k_{\max}$.
5. Experiment for different values of the size n, to see after how many iterations we return to the original image.

Here is the starting image of size 256×256 and the images obtained by iterations of the photo booth transformation for $k = 1$ up to $k = 8$. After 8 iterations we return to the initial image again.



Activity 2 (Conversion array/image)**.**

Goal: switch from an array to an image file and vice versa. The image is displayed in the "pgm" format which has been manipulated in the "Files" chapter.

1. **Array to image.**
   Program an `array_to_image(array,image_name)` function that writes an image from a grayscale table to a file in "pgm" format.

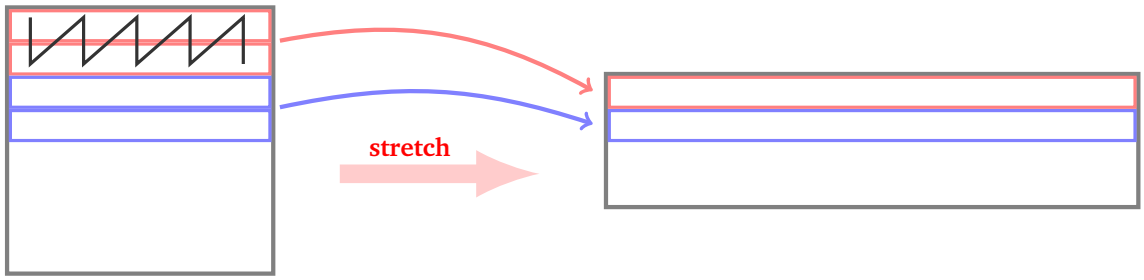
```
P2
5 5
255
128 192 128 192 128
224   0 228   0 224
228 228 228 228 228
224  64  64  64 224
192 192 192 192 192
```

   For example, with `array = [ [128,192,128,192,128], [224,...] ]`, the `array_to_image(array,"test")` command writes a `test.pgm` file (on the left) to be displayed as the image on the right.

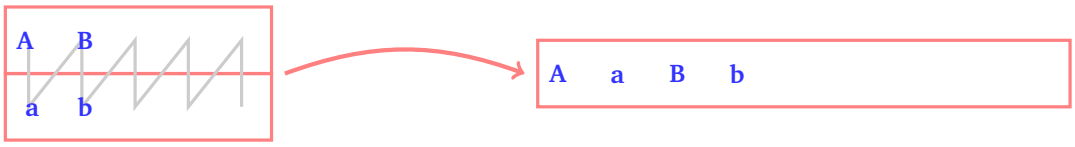
2. **Image to array.**
   Program an `image_to_array(image_name)` function which takes an image file in "pgm" format and returns an array of gray levels.

Lesson 2 (Baker's transformation)**.**
We start from an $n \times n$ array, with even n, where each element represents a pixel. We will apply two elementary transformations each time:

- **Stretching.** The principle is as follows: the first two lines (each with a length of n) produce a single line with a length of $2n$. We mix the values of each line by alternating an upper element and a lower element.



Here is how two lines mix into one:



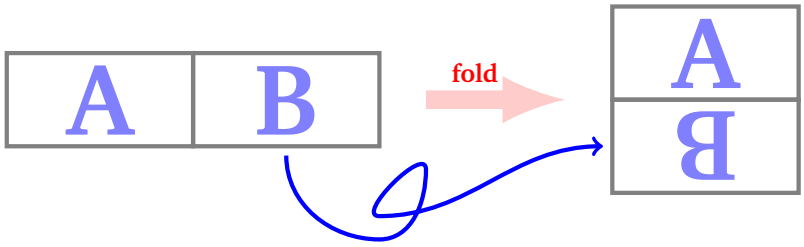
Formulas. An element at position (i, j) of the target array, corresponds to an element $(2i, j//2)$ (if j is even) or $(2i+1, j//2)$ (if j is odd) of the source array, with here $0 \leqslant i < \frac{n}{2}$ and $0 \leqslant j < 2n$.

Example. Here is a 4×4 array, and the stretched 2×8 array. The rows 0 and 1 on the left give the row 0 on the right. The rows 2 and 3 on the left give the row 1 on the right.

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

1	5	2	6	3	7	4	8
9	13	10	14	11	15	12	16

- **Fold.** The principle is as follows: the right part of a stretched array is turned upside down, then added under the left part. Starting from a $\frac{n}{2} \times 2n$ array you get an $n \times n$ array.



Formulas. For $0 \leqslant i < \frac{n}{2}$ and $0 \leqslant j < n$ the elements in position (i, j) of the array are kept in place. For $\frac{n}{2} \leqslant i < n$ and $0 \leqslant j < n$ an element of the array (i, j) corresponds to an element $\left(\frac{n}{2} - i - 1, 2n - 1 - j\right)$ of the source array.

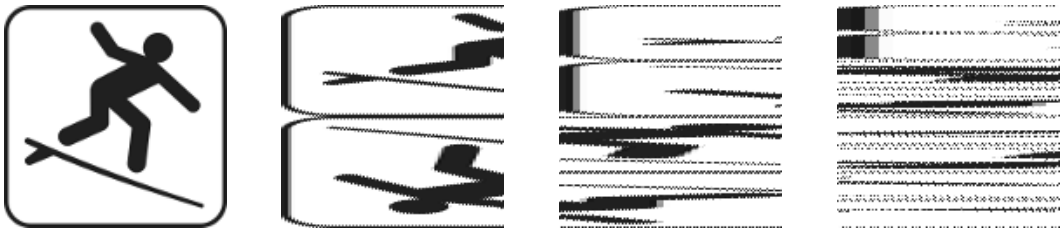
Example. From the stretched 2×8 array on the left, we obtain a folded 4×4 array on the right.

1	5	2	6	3	7	4	8
9	13	10	14	11	15	12	16

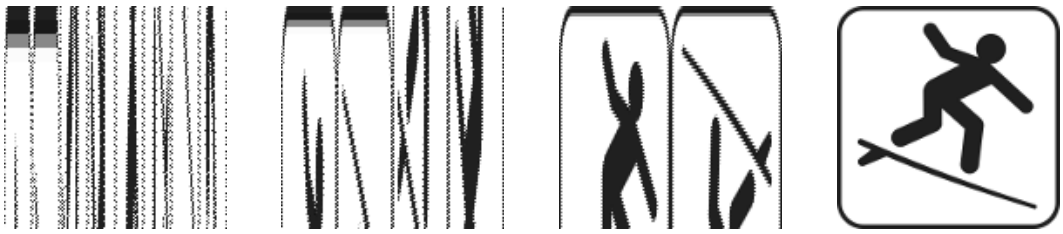
1	5	2	6
9	13	10	14
16	12	15	11
8	4	7	3

The ***baker transformation*** is the succession of stretching and folding, starting and ending with an $n \times n$ array.

Let's see an example of several baker transformations. On the left is the initial 128×128 image, then the result of $k = 1, 2, 3$ iterations.



Here are the images for $k = 12, 13, 14, 15$ iterations:



Activity 3 (Baker's transformation)**.**

Goal: program a new transformation that stretches and folds an image. Once again, the image becomes more and more distorted but, after a certain number of iterations, we return to the original image again.

1. Program a `baker_stretch(array)` function that returns the new array obtained by "stretching" the input table.
2. Program a `baker_fold(array)` function that returns the table obtained by "folding" the input table.
3. Program a `baker_iterate(array,k)` function that returns the table calculated after k iterations of baker's transformation.
   For example, the original 4×4 table is on the left, its image after a transformation ($k = 1$) and its image after a second transformation ($k = 2$) are also shown.

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

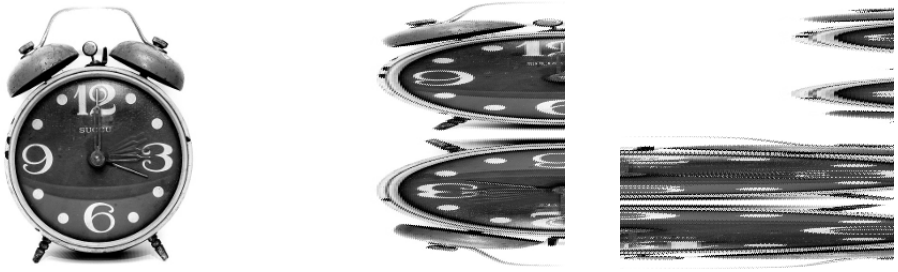
1	5	2	6
9	13	10	14
16	12	15	11
8	4	7	3

1	9	5	13
16	8	12	4
3	11	7	15
14	6	10	2

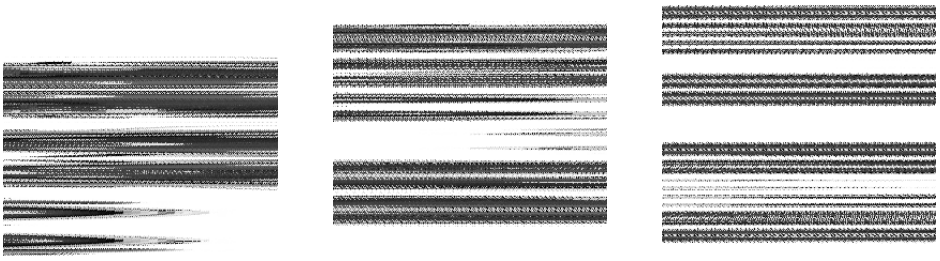
4. Program a `baker_images(image_name,kmax)` function that calculates the images corresponding to baker's transformation, with iterations ranging from $k = 1$ to $k = k_{\max}$.
5. Experiment with different values of the size n to see after how many iterations we get back to the original image.

Caution! It sometimes takes many iterations to get back to the original image. For example when $n = 4$, we return to the starting image after $k = 5$ iterations; when $n = 256$ it takes $k = 17$. Conjecture a return value in the case where n is a power of 2. However, for $n = 10$, you need $k = 56\,920$ iterations!

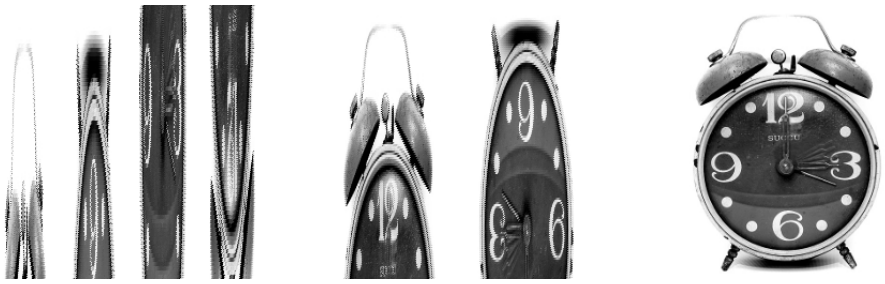
Here is an example with an image of size 256×256, first the initial image, then one transformation ($k = 1$) and a second iteration ($k = 2$).



$k = 3, 4, 5$:



$k = 15, 16, 17$:



For $k = 17$ you get back to the original image!

This chapter is based on the article “Blurred images, recovered images” by Jean-Paul Delahaye and Philippe Mathieu (Pour la Science, 1997).