# Elementary cellular automata 基本细胞自动机

Eric Martin, CSE, UNSW

COMP9021 Principles of Programming, trimester 1, 2019

序列

An elementary cellular automaton (ECA) determines for each possible sequence of 3 consecutive pixels, say $a$, $b$ and $c$, each of which is either black or white (1 or 0), whether the pixel below $b$ should be black or white. That is 2 possible outcomes for each of the $2^3$ possible sequences of 3 pixels, hence there are $2^{2^3} = 256$ elementary cellular automata. The 256 ECA's can be put in one-to-one correspondence with the 256 natural numbers smaller than 256 based on the following coding scheme.

Let $E$ be a natural number smaller than 256. Let $\widehat{E} = e_7e_6e_5e_4e_3e_2e_1e_0$ be this number represented in base 2 as an 8 bit number. For all natural numbers $P$ smaller than 8, let $\widetilde{P} = p_2p_1p_0$ be this number represented in base 2 as a 3 bit number. Then $E$ encodes the ECA such that for all $P < 8$, the pixel below the middle pixel of $\widetilde{P}$ should be $e_P$. For instance:

黑色的方格是当前细胞，两边的灰色方格是它的邻居。由于状态集只有{0，1}两个状态，
也就是说方格只能有黑、白两种颜色，那么任意的一个方格加上它的两个邻居，这3个方格
的状态组合一共就有8种。这8种情况为下图示：
他们表示的状态分别是：111，110，101，100，011，010，001，000。也就是说对于状态数
2，邻居半径为1的所有一维细胞自动机的邻居和其自身的状态组合就这8种。

- $\widehat{0} = 00000000$, so 0 encodes the following ECA:

画方格，跟黄色的对应好

| 111 | 110 | 101 | 100 | 011 | 010 | 001 | 000 |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

- $\widehat{90} = 01011010$, so 90 encodes the following ECA:

b 是二进制　X 8进制　o 是16进制

| 111 | 110 | 101 | 100 | 011 | 010 | 001 | 000 |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 |

- $\widehat{255} = 11111111$, so 255 encodes the following ECA:

| 111 | 110 | 101 | 100 | 011 | 010 | 001 | 000 |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

We talk about "rule $E$" to refer to the ECA mapped to $E$ by this correspondence.
For a better visualisation, let us represent Rule 90 using black and white squares instead of 1's and 0's:

| ■■■ | ■■□ | ■□■ | ■□□ | □■■ | □■□ | □□■ | □□□ |
|---|---|---|---|---|---|---|---|
| □ | ■ | □ | ■ | ■ | □ | ■ | □ |

There are two standard ways to consider the workings of an ECA:
从一个随机的黑白像素序列开始，两边无限

- start with a random sequence of black and white pixels, infinite on both sides, or
- start with a unique black pixel and on both sides, an infinite sequence of white pixels.

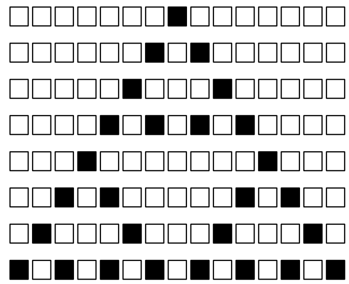
从一个唯一的黑色像素开始，两边都是无限的白色像素序列

The program `elementary_cellular_automata.py` creates a widget that has features for both workings; here we consider the second workings only. In any case, the conditions imposed by an ECA fully determine the infinite sequence of pixels $l_2$ below an infinite sequence of pixels $l_1$, and then fully determine the infinite sequence of pixels $l_3$ below $l_2$, and then fully determine the infinite sequence of pixels $l_4$

无穷序列；无限序列

在任何情况下，ECA施加的条件完全确定无限像素序列l1下方的无限像素序列l2，然后完全确定l2以下像素l3的无限序列，然
完全确定无限序列 L4

below $l_3$... For instance, with Rule 90, the first 8 sequences are as follows (all pixels that are not shown on both sides of all 8 lines are white):



椎体　更精确的

It is clear that the picture that results from this process is a cone. More precisely, working with rule $E$ and writing as above $\widehat{E} = e_7e_6e_5e_4e_3e_2e_1e_0$,

像素　那么圆锥周围的所有像素都是白色的

- if $e_0 = 0$ then all all pixels around the cone are white;
- if $e_0 = 1$ and $e_7 = 1$ then all all pixels around the cone are black (except for the first line of course);
- if $e_0 = 1$ and $e_7 = 0$ then successive lines around the cone alternate between all white and all black.

圆锥周围的连续线条在全白和全黑之间交替

Our aim is to write code to draw a similar kind of picture as the one above, for any ECA, encoded as an integer between 0 and 255 (the widget also accepts 8 consecutive 0's and 1's). To capture the encoded ECA, we first define a function, `decoded_rule()`, meant to take an integer whose value is a natural number $E$ smaller than 256 as argument and return a dictionary whose keys are triples of `0`'s and `1`'s with an associated value of `0` or `1` as determined by rule $E$. For instance, with rule 90, the dictionary would be `{(1, 1, 1): 0, (1, 1, 0): 1, (1, 0, 1): 0, (1, 0, 0): 1, (0, 1, 1): 1, (0, 1, 0): 0, (0, 0, 1): 1, (0, 0, 0): 0}`.

An integer can be represented as a string in any of bases 2, 8, 10 (the default), or 16, with two variants for base 16 to use either lowercase or uppercase letters for the "digits" 10 up to 15:

整数可以表示为以2、8、10(默认)或16为底的字符串，以16为底的字符串有两种变体，可以使用小写或大写字母表示从10到15的“数字”:

我们的目标是编写代码来绘制与上面类似的图像，对于任何ECA，它都被编码为0到255之间的整数(小部件还接受8个连续的0和1)。捕捉编码ECA,我们首先定义一个函数,decoded_rule(),为了把一个整数的值是一个自然数E小于256作为参数并返回一个字典的键是0和1的三元组的关联值0或1由规则E

二进制

```
In [1]: # b: binary
        # o: octal
        # x or X: hexadecimal
        f'90', f'{90:b}', f'{90:o}', f'{90:x}', f'{90:X}'
```

八进制　十六进制

```
Out[1]: ('90', '1011010', '132', '5a', '5A')
```

把90 用不同的进制表示

The formatting allows one to possibly pad either spaces or `0`'s to the left of the string to make sure the field width has a minimal value: 这种格式允许在字符串左侧填充空格或0，以确保字段宽度具有最小值:

字段宽度至少为3，如果需要，用空格填充

```
In [2]: # A field width of 3 at least, padding with spaces if needed
        f'{90:3}', f'{90:3b}', f'{90:3o}', f'{90:3x}', f'{90:3X}'
        # A field width of 3 at least, padding with 0's if needed
        f'{90:03}', f'{90:03b}', f'{90:03o}', f'{90:03x}', f'{90:03X}'
        # A field width of 8 at least, padding with spaces if needed
        f'{90:8}', f'{90:8b}', f'{90:8o}', f'{90:8x}', f'{90:8X}'
        # A field width of 8 at least, padding with 0's if needed
        f'{90:08}', f'{90:08b}', f'{90:08o}', f'{90:08x}', f'{90:08X}'
```

字段宽度至少为3，如果需要用0填充

```
Out[2]: (' 90', '1011010', '132', ' 5a', ' 5A')

Out[2]: ('090', '1011010', '132', '05a', '05A')
```

```
Out[2]: ('      90', ' 1011010', '     132', '      5a', '      5A')
```

```
Out[2]: ('00000090', '01011010', '00000132', '0000005a', '0000005A')
```

So the list of 8 bits that define a rule is easy to get by formatting the rule number in binary with a field width of 8 within a list comprehension: 因此，定义规则的8位列表很容易获得，方法是在理解列表的情况下，将规则编号格式化为字段宽度为8的二进制文件

```
In [3]: [int(d) for d in f'{90:08b}']
```

```
Out[3]: [1, 0, 1, 1, 0, 1, 0]
```

To generate the keys, we could use the same technique, first formatting all natural numbers smaller than 8 in binary with a field width of 3: 为了生成键，我们可以使用相同的技术，首先将所有小于8的自然数格式化为字段宽度为3的二进制数

```
In [4]: for i in range(8):
            print(f'{i:03b}')
```

```
000
001
010
011
100
101
110
111
```

从一个数中获取一串字符，然后从该字符串中获取一串数字，这不是最好的方法。注意，如果n是自然数，那么将n除以10的整数除法将n的小数表示形式中的所有数字都移位1，在这个过程中“失去”最右边的1，等于n模10

Getting a string of characters from a number, and then a list of digits from the string, is not the best approach. Note that if $n$ is a natural number, then integer division of $n$ by 10 shifts all digits in the decimal representation of $n$ by one, "losing" the rightmost one in the process, equal to $n$ modulo 10.

A syntactic digression is necessary to properly read the code fragment that follows. An identifier can start with an underscore, and it can even just consist of an underscore. It is good practice to use `_` in a statement of the form `for _ in range(n): ...` to indicate that the code loops `n` many times, as opposed to a statement of the form `for i in range(n): ...` where all values between 0 and `n` minus 1 are generated and assigned to `i`, which is then used in one way or another in the body of the loop. We make use of this convention to illustrate the previous observation:

为了正确地阅读后面的代码片段，语法上的离题是必要的。标识符可以从下划线开始，甚至可以只由下划线组成。
在range(n)的表达式中使用_是一种很好的做法:…表示代码循环n次，而不是i在(n)范围内的表达式:…0到n -之间的所有值呢
生成1并将其分配给i，然后在循环体中以某种方式使用i。我们利用这一公约来说明以前的意见:

```
In [5]: n = 21078
        print(n); print()
        for _ in range(7):
            n, d = divmod(n, 10)
            print(n, d)
```

```
21078

2107 8
210 7
21 0
2 1
```

python divmod() 函数把除数和余数运算结果结合起来，返回一个包含商和余数的元组(a // b, a % b)。
>>>divmod(7, 2)
(3, 1)
>>> divmod(8, 2)
(4, 0)
>>> divmod(1+2j,1+0.5j)
((1+0j), 1.5j)

```
python range() 函数可创建一个整数列表，一般用在 for 循环中
>>>range(10)        # 从 0 开始到 10
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(1, 11)     # 从 1 开始到 11
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> range(0, 30, 5)  # 步长为 5
[0, 5, 10, 15, 20, 25]
>>> range(0, 10, 3)  # 步长为 3
[0, 3, 6, 9]
>>> range(0, -10, -1) # 负数
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
>>> range(0)
[]
>>> range(1, 0)
[]
```

```
0 2
0 0
0 0
```

More generally, if $n$ and $k$ are natural numbers, then dividing $n$ by $10^k$ shifts all digits in the decimal representation of $n$ by $k$, “losing” the $k$ rightmost ones in the process, which make up the number $n$ modulo $10^k$:

更一般地说，如果n和k是自然数，那么将n除以10k会使n除以k的小数形式中的所有数字移位，在这个过程中会“丢失”最右边的k个数字，这些数字构成了n模
10 k

```
In [6]: n = 16503421078003459
        print(n); print()
        for _ in range(7):
            n, d = divmod(n, 1_000)
            print(n, d)
```

```
python range() 函数可创建一个整数列表，一般用在 for 循环中，range(start, stop[, step])
start: 计数从 start 开始。默认是从 0 开始。例如range (5) 等价于range (0， 5) ;
stop: 计数到 stop 结束，但不包括 stop。例如：range (0， 5)  是[0, 1, 2, 3, 4]没有5
step：步长，默认为1。例如：range (0， 5)  等价于 range(0, 5, 1)

>>>range(10)        # 从 0 开始到 10
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(1, 11)     # 从 1 开始到 11
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> range(0, 30, 5)  # 步长为 5
[0, 5, 10, 15, 20, 25]
>>> range(0, 10, 3)  # 步长为 3
[0, 3, 6, 9]
>>> range(0, -10, -1) # 负数
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
>>> range(0)
[]
>>> range(1, 0)
[]
```

```
16503421078003459

16503421078003 459
16503421078 3
16503421 78
16503 421
16 503
0 16
0 0
```

Similarly, if $n$ is a natural number, then integer division of $n$ by 2 shifts all digits in the binary representation of $n$ by one, “losing” the rightmost one in the process, equal to $n$ modulo 2:

同样，如果n是自然数，那么n除以2的整数除法将n的二进制表示形式中的所有数字都移位1，在这个过程中“失去”了最右边的1，等于n模2:

```
In [7]: n = 214
        print(f'{n:b}'); print()
        for _ in range(9):
            n, d = divmod(n, 2)
            print(f'{n:b} {d:b}')
```

```
11010110

1101011 0
110101 1
11010 1
1101 0
110 1
11 0
1 1
0 1
0 0
```

```
python divmod() 函数把除数和余数运算结果结合起来，返回一个包含商和余数的元组(a // b, a % b)。
>>>divmod(7, 2)
(3, 1)
>>> divmod(8, 2)
(4, 0)
>>> divmod(1+2j,1+0.5j)
((1+0j), 1.5j)
```

More generally, if $n$ and $k$ are natural numbers, then dividing $n$ by $2^k$ shifts all digits in the binary representation of $n$ by $k$, “losing” the $k$ rightmost ones in the process, which make up the number $n$ modulo $2^k$:

更一般地说，如果n和k是自然数，那么将n除以2k将把n的二进制表示形式中的所有数字都除以k，从而“失去”最右边的k个数字，这些数字构成了n模2k

```
In [8]: n = 2345678
        print(f'{n:b}'); print()
        for _ in range(9):
            n, d = divmod(n, 8)
            print(f'{n:b} {d:b}')
```

```
1000111100101011001110

1000111100101011001 110
1000111100101011 1
1000111100101 11
1000111100 101
1000111 100
1000 111
1 0
0 1
0 0
```

So the keys of the dictionary that `decoded_rule()` should return can be generated as follows:

因此，decoded_rule()应该返回的字典的键可以如下所示生成

```
In [9]: for p in range(8):
             p // 4, p // 2 % 2, p % 2
```

```
Out[9]: (0, 0, 0)

Out[9]: (0, 0, 1)

Out[9]: (0, 1, 0)

Out[9]: (0, 1, 1)

Out[9]: (1, 0, 0)

Out[9]: (1, 0, 1)

Out[9]: (1, 1, 0)

Out[9]: (1, 1, 1)
```

+ 加 - 两个对象相加 a + b 输出结果 30

- 减 - 得到负数或是一个数减去另一个数 a - b 输出结果 -10

* 乘 - 两个数相乘或是返回一个被重复若干次的字符串 a * b 输出结果 200

/ 除 - x除以y b / a 输出结果 2

% 取模 - 返回除法的余数 b % a 输出结果 0

** 幂 - 返回x的y次幂 a**b 为10的20次方， 输出结果 100000000000000000000

// 取整除 - 返回商的整数部分（向下取整）
>>> 9//2
4
>>> -9//2
-5

Putting it all together, with the help of a **dictionary comprehension**:

字典的理解

```
In [10]: def record_rule(E):
             values = [int(d) for d in f'{E:08b}']
             return {(p // 4, p // 2 % 2, p % 2): values[7 - p] for p in range(8)}

         # As Rule 90 is symmetric, had we written values[p]
         # instead of values[7 - p], we would not see the mistake.
         record_rule(90)
         record_rule(41)
```

```
Out[10]: {(0, 0, 0): 0,
          (0, 0, 1): 1,
          (0, 1, 0): 0,
          (0, 1, 1): 1,
          (1, 0, 0): 1,
          (1, 0, 1): 0,
          (1, 1, 0): 1,
          (1, 1, 1): 0}
```

```
Out[10]: {(0, 0, 0): 1,
          (0, 0, 1): 0,
          (0, 1, 0): 0,
          (0, 1, 1): 1,
          (1, 0, 0): 0,
          (1, 0, 1): 1,
          (1, 1, 0): 0,
          (1, 1, 1): 0}
```

正方形

Rather than displaying lines of `0`'s and `1`'s, it is preferable to take advantage of the Unicode character set and instead, display lines of white and black squares. The Unicode character set considerably extends the ASCII character set. A Unicode character has a code point, a natural number which when it is smaller than 128, is the ASCII code of an ASCII character. The `ord()` function returns the code point of the (string consisting of the unique) character provided as argument:

与其显示0和1的行，不如利用Unicode字符集来显示白色和黑色的方块行。Unicode字符集大大扩展了ASCII字符集。Unicode字符有一个码点，一个自然数，当它小于128时，它就是一个ASCII字符的ASCII码。The ord()函数的作用是:返回作为参数提供的(由唯一字符组成的)字符的代码点

```
In [11]: ord('+')
         ord('■')
         ord('😋')
```

```
Out[11]: 43
```

```
Out[11]: 11035
```

```
Out[11]: 128523
```

相反地

Conversely, the `chr()` function takes an integer whose value is a natural number $n$ as argument and returns the character with $n$ as code point:

chr()函数取一个自然数n为自变量的整数，返回n为码点的字符

```
In [12]: chr(43)
         chr(11035)
         chr(128523)
```

```
Out[12]: '+'
```

```
Out[12]: '■'
```

```
Out[12]: '😋'
```

整数

Code points are more often represented in base 16. More generally, integer literals can use either binary, octal, decimal, or hexadecimal representations:

二进制 八进制 十进制 十六进制 表示，代表

代码点通常以16为基数表示。更一般地，整数可以使用二进制、八进制、十进制或十六进制表示

```
In [13]: # 0b, 0o, and either 0x or 0X, are prefixes
         # for base 2, 8, and 16, respectively.
         # 43 in base 2, 8, 10, and 16
         0b101011, 0o53, 43, 0X2b
         # 11035 in base 2, 8, 19, and 16
         0b10101100011011, 0o25433, 11035, 0x2b1b
         # 128523 in base 2, 8, 19, and 16
         0b11111011000001011, 0o373013, 128523, 0x1F60B

Out[13]: (43, 43, 43, 43)

Out[13]: (11035, 11035, 11035, 11035)

Out[13]: (128523, 128523, 128523, 128523)
```

When written in base 16, code points are at most 8 hexadecimal digits long. A character whose code point has at least 5 hexadecimal digits has one Unicode string representation that starts with \U, followed by 8 hexadecimal digits (leading 0's are used when needed):

以16为基数编写时，代码点最多为8个十六进制数字长。编码点至少有5个十六进制数字的字符有一个Unicode字符串表示形式，以\U开头，后跟8个十六进制数字(需要时使用前导0)

```
In [14]: '\U0001f60b'

Out[14]: '😋'
```

A character whose code point has at most 4 hexadecimal digits has two Unicode string representations; one that that starts with \u followed by 4 hexadecimal digits, one that starts with \U followed by 8 hexadecimal digits (in both cases, leading 0's are used when needed):

编码点最多有4个十六进制数字的字符具有两个Unicode字符串表示形式;一个以\u开头，后面跟着4个十六进制数字，一个以\u开头，后面跟着8个十六进制数字(在这两种情况下，前面的0在需要时使用)

```
In [15]: '\u002B', '\U0000002B'
         '\u2b1b', '\U00002b1b'

Out[15]: ('+', '+')

Out[15]: ('⬛', '⬛')
```

回想一下，我们想要绘制由ECA的工作方式决定的直线l的一段，从一条两边都有一个黑色像素和无限多个白色像素的直线开始。现在我们定义一个函数display_line()，它可以满足这个目的。display_line()有三个参数

Recall that we want to draw a segment of a line $l$ determined by the workings of an ECA, starting with a line with a single black pixel and infinitely many white pixels on both sides. We now define a function, display_line(), that can fill this purpose. There are three arguments to display_line():

- The first argument is denoted by bit_sequence and meant to take as value a tuple that represents the pixels on that part of $l$ that intersects the cone determined by the workings of the ECA. For instance, with Rule 90, for the first four lines, bit_sequence takes the values (1,), (1, 0, 1), (1, 0, 0, 0, 1), (1, 0, 0, 0, 1) and (1, 0, 1, 0, 1, 0, 1), respectively.
- The second argument is denoted by end_bit and meant to take the value 0 or 1 and represent the pixel outside the cone on $l$. For instance, with Rule 90, it is 0.
- The third argument is denoted by nb_of_end_bits and meant to take as value a natural number, possibly equal to 0, that represents the number of times we want to display the pixel outside the cone on $l$, on each side.

display_line() makes use of an auxiliary function to display the pixel outside the cone; it calls it twice, one for each side of the cone. It also makes use of the fact that the unicode strings '\u2b1c' and '\u2b1b' depict white and black squares, respectively:

display_line () 有三个参数：
• 第一个参数由bit_sequence表示，意味着将一个元组作为值，该元组表示与ECA的工作所确定的锥体相交的l部分上的像素。例如，对于规则90，对于前四行，bit_sequence取值（1，），（1,0,1），（1,0,0,0,1），（1,0,0,0），1）和（1,0,1,0,1,0,1）。
• 第二个参数由end_bit表示，意味着取值0或1并表示l上锥体外的像素。例如，对于规则90，它为0。
• 第三个参数由nb_of_end_bits表示，并且意味着将自然数作为值，可能等于0，表示我们想要在外部显示像素的次数
锥体在l，在每一边。 display_line () 利用辅助函数显示锥体外的像素;它称之为两次，锥体的每一侧都有一次。它还利用了unicode字符串'\ u2b1c'和'\ u2b1b'分别描绘白色和黑色方块的事实：

```
In [16]: def display_end_squares(end_bit, nb_of_end_bits):
             print(end_bit * nb_of_end_bits, end = '')

         def display_line(bit_sequence, end_bit, nb_of_end_bits):
 正方形       squares = {0: '\u2b1c', 1: '\u2b1b'} 字典
             display_end_squares(squares[end_bit], nb_of_end_bits)
             print(''.join(squares[b] for b in bit_sequence), end = '')
             display_end_squares(squares[end_bit], nb_of_end_bits)
             print()

         display_line((1, 0, 1, 0, 1, 0, 1), 0, 4)
         display_line((1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1), 0, 0)
```

```
⬜⬜⬜⬜⬛⬜⬛⬜⬛⬜⬛⬜⬜⬜⬜
⬛⬜⬛⬜⬛⬜⬛⬜⬛⬜⬛⬜⬛⬜⬛
```

record_rule()和display_line(),来完成我们的任务所剩不多了的图段的前几行像素由ECA的运作从一条线组成的白色像素,除了一个黑色像素。函数display_ECA()接受两个参数。第一个用rule_nb表示，它的意思是取小于256的自然数作为值，这个自然数编码我们要处理的ECA。第二个是用大小表示的，它的意思是取要显示的白色像素的数量作为值

With `record_rule()` and `display_line()` in hand, not much is left to complete our task of drawing the segments of the first few lines of pixels as determined by the workings of an ECA starting with a line consisting of nothing but white pixels, with the exception of a single black pixel. The function `display_ECA()` takes two arguments. The first one is denoted by `rule_nb` and meant to take as value the natural number smaller than 256 that encodes the ECA we want to work with. The second one is denoted by `size` and meant to take as value the number of white pixels to display on both sides of the black pixel in the middle of the first line segment; hence the first line segment consists of `2 * size + 1` many pixels. The function `display_ECA()` will draw `size + 1` many line segments: that way, the last line segment will span from left to right boundaries of the cone, whereas the penultimate line segment will have one pixel outside the cone on both sides, the second last line segment will have two pixels outside the cone on both sides, etc.

At any stage, `new_line` will denote a tuple representing the sequence of pixels that make up a given line segment spanning from left to right boundaries of the cone, and `end_bit` will represent the pixel outside the cone (always equal to 0 for Rule 90). In order to determine the next line segment from the current one, we add two copies of `end_bit` at the beginning of `new_line`, and two copies of `end_bit` at the end, making up `current_line`. So:

在任何阶段，new_line都将表示一个元组，表示构成从圆锥的左到右边界的给定线段的像素序列，end_bit将表示圆锥之外的像素(对于规则90，总是等于0)。为了确定当前线段的下一个线段，我们在new_line的开头添加两个end_bit副本，在末尾添加两个end_bit副本，组成current_line

current_line[1]、current_line[2]、current_line[3]对前两行求圆锥外像素值，对第三行求圆锥左边界像素值，并在下一行求圆锥左边界像素值

- `current_line[1]`, `current_line[2]` and `current_line[3]` evaluate to the pixel outside the cone for the first two, and the pixel on the left boundary of the cone for the third one, and determine the pixel on the left boundary of the cone on the next line.

current_line[-3]、current_line[-2]和current_line[-1]分别求出后两种情况下圆锥外的像素值和第一种情况下圆锥右边界上的像素值，并确定下一行圆锥右边界上的像素值

- `current_line[-3]`, `current_line[-2]` and `current_line[-1]` evaluate to the pixel outside the cone for the last two, and the pixel on the right boundary of the cone for the first one, and determine the pixel on the right boundary of the cone on the next line.
- As `end_bit` evaluates to the pixel outside the cone, `(end_bit, end_bit, end_bit)` determines the pixel outside the cone on the next line.

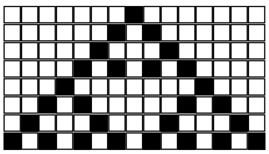
当end_bit计算到圆锥外部的像素时，(end_bit, end_bit, end_bit)决定了下一行圆锥外部的像素

```
In [17]: def display_ECA(rule_nb, size):
             bit_below = record_rule(rule_nb)
             new_line = [1]
             end_bit = 0
             display_line(new_line, end_bit, size)
             for n in range(size):
                 current_line = [end_bit] * 2 + new_line + [end_bit] * 2
```

```
        new_line = [bit_below[current_line[i],
                              current_line[i + 1],
                              current_line[i + 2]
                             ] for i in range(len(current_line) - 2)
                   ]
        end_bit = bit_below[end_bit, end_bit, end_bit]
        display_line(new_line, end_bit, size - n - 1)
```

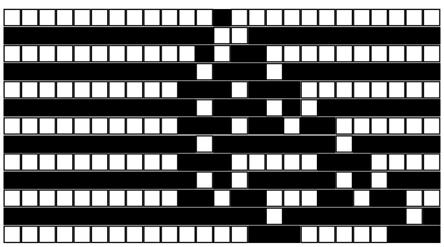
Rule 90 is an example where the outside of the cone consists of nothing but white pixels:

```
In [18]: display_ECA(90, 7)
```



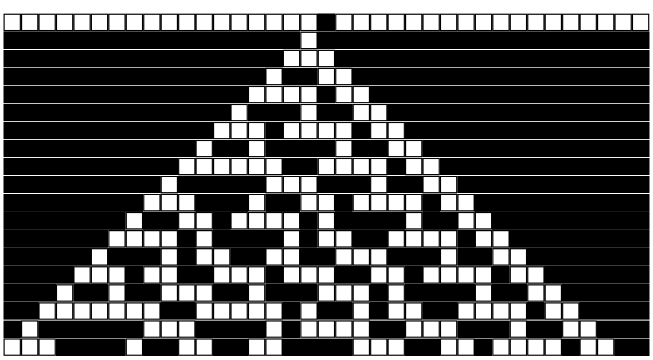
Rule 107 is an example where outside the cone, black and white half-infinite lines alternate:

```
In [19]: display_ECA(107, 12)
```



Rule 149 is an example where the outside of the cone consists of nothing but black pixels, except for the first line of course: 149规则是一个例子，圆锥体的外部只包含黑色像素，当然第一行除外

```
In [20]: display_ECA(149, 18)
```



左右对称 四分之一 对称性 镜像规则

Though there are 256 ECA's, only a quarter are really different due to symmetries. The *mirrored rule* of a rule exhibits vertical symmetry: given three pixels $p_0$, $p_1$ and $p_2$, the pixel imposed by a rule $E$ below the middle pixel of $p_0p_1p_2$ is the pixel imposed by the mirrored rule of rule $E$ below the middle pixel of $p_2p_1p_0$. Rule 90 exhibits vertical symmetry, hence it is its own mirrored rule.

让我们定义一个函数mirrored_rule()，它的作用是获取一个规则作为参数并返回其镜像规则。给定E < 256，以2为底的E表示为8位数字e7e6e5e4e3e2e1e0，则E的镜像规则为e7e3e5e1e6e2e4e0，由对应关系反映

Let us define a function, `mirrored_rule()`, meant to get a rule as argument and return its mirrored rule. Given $E < 256$, and writing the representation of $E$ in base 2 as the 8 bit number $e_7e_6e_5e_4e_3e_2e_1e_0$, the mirrored rule of $E$ is then $e_7e_3e_5e_1e_6e_2e_4e_0$, as reflected by the correspondence between

$$111 \quad 110 \quad 101 \quad 100 \quad 011 \quad 010 \quad 001 \quad 000$$

and

$$111 \quad 011 \quad 101 \quad 001 \quad 110 \quad 010 \quad 100 \quad 000$$

`mirrored_rule()` could generate from its argument `E` the formatted string `f'{E:08b}`', say `s`, then create the new string `''.join((s[7], s[3], s[5], s[1], s[6], s[2], s[4], s[0]))`, and convert the latter into an integer. By default, `int()` converts a string that represents an integer in base 10, but it can also accept representations for other bases:

mirrored_rule()可以从它的参数E生成格式化字符串f'{E:08b} '，比如s，然后创建新字符串"。加入((s[7],[3],[5],[1],[6],[2],[4],[0])并将后者转换为一个整数。默认情况下，int()转换表示以10为基数的整数的字符串，但它也可以接受其他基数的表示

```
In [21]: # With 0 as second argument, interpret the base from the literal
         int('0b101011', 0), int('43', 0), int('0o53', 0), int('0X2b', 0)
         int('101011', 2), int('0b101011', 2)
         int('1121', 3)
         int('223', 4)
         int('133', 5)
         int('53', 8), int('0o53', 8),
         int('2b', 16), int('0X2b', 16)
         # 36 is the largest base
         int('17', 36)
         int('z', 36), int('Z', 36)
```

```
Out[21]: (43, 43, 43, 43, 43)

Out[21]: (43, 43)

Out[21]: 43

Out[21]: 43

Out[21]: 43

Out[21]: (43, 43)

Out[21]: (43, 43)

Out[21]: 43

Out[21]: (35, 35)
```

Let us still not "hardcode" the sequence of bits as `(s[7], s[3], s[5], s[1], s[6], s[2], s[4], s[0])`, but generate it. Let us first examine the `sorted()` function. By default, `sorted()` returns the list of members of its arguments in their default order: sort()以默认顺序返回其参数的成员列表

```
In [22]: sorted([2, -2, 1, -1, 0])
         # Lexicographic/lexical/dictionary/alphabetic order
         sorted({'a', 'b', 'ab', 'bb', 'abc', 'C'})
         sorted(((2, 1, 0), (0, 1, 2), (1, 2, 0), (1, 0, 2)))
```

```
Out[22]: [-2, -1, 0, 1, 2]

Out[22]: ['C', 'a', 'ab', 'abc', 'b', 'bb']

Out[22]: [(0, 1, 2), (1, 0, 2), (1, 2, 0), (2, 1, 0)]
```

`sorted()` accepts the `reverse` keyword argument:

```
In [23]: sorted([2, -2, 1, -1, 0], reverse = True)
         sorted({'a', 'b', 'ab', 'bb', 'abc', 'C'}, reverse = True)
         sorted(((2, 1, 0), (0, 1, 2), (1, 2, 0), (1, 0, 2)), reverse = True)

Out[23]: [2, 1, 0, -1, -2]

Out[23]: ['bb', 'b', 'abc', 'ab', 'a', 'C']

Out[23]: [(2, 1, 0), (1, 2, 0), (1, 0, 2), (0, 1, 2)]
```

`sorted()` also accepts the `key` argument, which should evaluate to a **callable**, e.g., a function. The function is called on all elements of the sequence to sort, and elements are sorted in the natural order of the values returned by the function:

sort()也接受key参数，该参数的值应该是可调用的，例如函数。对序列中的所有元素调用该函数进行排序，并按函数返回值的自然顺序对元素进行排序

```
In [24]: sorted([2, -2, 1, -1, 0], key = abs)
         sorted({'a', 'b', 'ab', 'bb', 'abc', 'C'}, key = str.lower)
         sorted({'a', 'b', 'ab', 'bb', 'abc', 'C'}, key = len)

Out[24]: [0, 1, -1, 2, -2]

Out[24]: ['a', 'ab', 'abc', 'b', 'bb', 'C']

Out[24]: ['C', 'a', 'b', 'bb', 'ab', 'abc']
```

We can also set `key` to an own defined function:

```
In [25]: def _2_0_1(s):
             return s[2], s[0], s[1]

         def _2_1_0(s):
             return s[2], s[1], s[0]

         sorted(((2, 1, 0), (0, 1, 2), (1, 2, 0), (1, 0, 2)), key = _2_0_1)
         sorted(((2, 1, 0), (0, 1, 2), (1, 2, 0), (1, 0, 2)), key = _2_1_0)

Out[25]: [(1, 2, 0), (2, 1, 0), (0, 1, 2), (1, 0, 2)]

Out[25]: [(2, 1, 0), (1, 2, 0), (1, 0, 2), (0, 1, 2)]
```

So we could generate the sequence (0, 4, 2, 6, 1, 5, 3, 7) as follows:

% 取模 - 返回除法的余数 b % a 输出结果 0
// 取整除 - 返回商的整数部分（向下取整）

```
In [26]: def three_two_one(p):
             return p % 2, p // 2 % 2, p % 4

         for p in sorted(range(8), key = three_two_one):
             p, f'{p:03b}'
```

字段宽度至少为3，如果需要用0填充

```
Out[26]: (0, '000')

Out[26]: (4, '100')

Out[26]: (2, '010')

Out[26]: (6, '110')

Out[26]: (1, '001')

Out[26]: (5, '101')

Out[26]: (3, '011')

Out[26]: (7, '111')
```

有一种更好的方法，使用lambda表达式。Lambda表达式提供了一种定义不需要命名的函数的简洁方法

There is a better way, using a **lambda expression**. Lambda expressions offer a concise way to define functions, that do not need to be named:

函数不带参数，所以返回一个常量

```
In [27]: # Functions taking no argument, so returning a constant
         f = lambda: 3; f()
         (lambda: (1, 2, 3))()
```

参数 常量

```
Out[27]: 3

Out[27]: (1, 2, 3)
```

函数有一个参数，第一个参数是恒等

```
In [28]: # Functions taking one argument, the first of which is identity
         f = lambda x: x; f(3)
         (lambda x: 2 * x + 1)(3)

Out[28]: 3

Out[28]: 7

In [29]: # Functions taking two arguments
         f = lambda x, y: 2 * (x + y); f(3, 7)
         (lambda x, y: x + y)([1, 2, 3], [4, 5, 6])

Out[29]: 20

Out[29]: [1, 2, 3, 4, 5, 6]
```

Putting everything together, we can define `mirrored_rule()` as follows:

```
In [30]: def mirrored_rule(E): 字段宽度至少为8，如果需要用0填充
             return int(''.join(f'{E:08b}'[i] for i in sorted(range(8),
                                          key = lambda i: (i % 2, i // 2 % 2, i // 4)
                                                                 )
                                   ), 2
                            )

         mirrored_rule(90)
         mirrored_rule(107)
         mirrored_rule(149)

Out[30]: 90

Out[30]: 121

Out[30]: 135
```

Another symmetry between ECA's emerges by exchanging all 0's to 1's and all 1's to 0's. This maps rules to their *complementaries*. For instance, the complementary of rule 90, represented as

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 111 | 110 | 101 | 100 | 011 | 010 | 001 | 000 |
| 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 |

is represented as

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
| 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 |

hence is the rule whose binary representation is 10100101 (10100101 read from right to left), hence is rule 165. Let us define a function, `complementary_rule()`, meant to get a rule as argument and return its complementary rule:因此，二进制表示为10100101(从右到左读取10100101)的规则就是规则
165. 让我们定义一个函数，complementary ary_rule()，它的意思是获取一个规则作为参数并返回它的互补规则

```
In [31]: def complementary_rule(E):
             return int(''.join({'0': '1', '1': '0'}[c]
                                 for c in reversed(f'{E:08b}')
                               ), 2
                        )

         complementary_rule(90)
         complementary_rule(107)
         complementary_rule(149)

Out[31]: 165

Out[31]: 41

Out[31]: 86
```

A rule can be its own mirror, or its own complementary, but it cannot be both. For most rules, the rule itself, and its mirror, and its complementary, are all different, exhibiting minimum symmetry:

规则可以是它自己的镜子，也可以是它自己的补充，但不能两者兼而有之。对于大多数规则，规则本身，它的镜像，和它的互补，都是不同的，表现出最小的对称性

```
In [32]: display_ECA(60, 15)
         print()
         display_ECA(mirrored_rule(60), 15)
         print()
         display_ECA(complementary_rule(60), 15)
         print()
         display_ECA(complementary_rule(mirrored_rule(60)), 15)
```