Elementary cellular automata

Eric Martin, CSE, UNSW

COMP9021 Principles of Programming, session 2, 2018

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:

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

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:

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

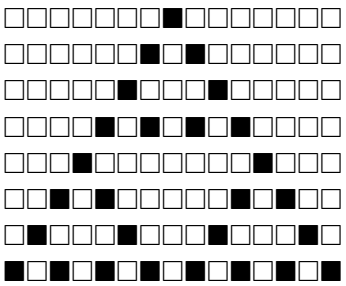
We talk about "rule E" to refer to the ECA mapped to E by this correspondence.
For a better visualisation, let us represent Rule 90 using black and white squares instead of 1's and 0's:

■■■	■■□	■□■	■□□	□■■	□■□	□□■	□□□
□	■	□	■	■	□	■	□

There are two standard ways to consider the workings of an ECA:

- 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.

The widget 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 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):



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

- 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.

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 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:

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

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

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:

```
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}'

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:

```
In [3]: [int(d) for d in f'{90:0b}']
```

```
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:

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

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

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 the value of **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:

```
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
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:

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

```
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:

```
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
```

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:

```
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
100011110010101 1
10001111001 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:

```
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)
```

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

```
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}
```

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 character provided as argument; in this context, "character" means "string consisting of a unique character":

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

Out[11]: 43

Out[11]: 11035

Out[11]: 128523
```

Conversely, the **chr()** function takes a natural number n as argument and returns the character with n as code point:

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

Out[12]: '+'

Out[12]: '■'

Out[12]: '😊'
```

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

```
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):

```
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):

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

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

Out[15]: ('■', '■')
```

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 two arguments to **display_line()**:

- The first argument, **bit_sequence**, is meant to represent the pixels on that part of l that intersects the cone determined by the workings of the ECA, preceded with the value of the pixel outside the cone on that line. For instance, with Rule 90:
  - The sequence of pixels that intersects the cone on the first line is (1) and the pixel outside the cone on that line is 0, hence **bit_sequence** should be (0, 1);
  - The sequence of pixels that intersects the cone on the second line is (1, 0, 1) and the pixel outside the cone on that line is 0, hence **bit_sequence** should be (0, 1, 0, 1);
  - The sequence of pixels that intersects the cone on the third line is (1, 0, 0, 0, 1) and the pixel outside the cone on that line is 0, hence **bit_sequence** should be (0, 1, 0, 0, 0, 1);
  - ...
- **nb_of_end_bits**, whose value is a natural number possibly equal to 0, that represents the number of times we want to display the pixel outside the cone, on both sides.

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.

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

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

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

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, **rule_nb**, is meant to be the natural number smaller than 256 that encodes the ECA we want to work with. The second one, **size**, is meant to denote the number of white pixels to display on either side 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 be the sequence of pixels that make up a given line segment, spanning from left to right boundaries of the cone, and preceded with the value v of 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 v at the beginning of **new_line**, and two copies of v at the end, making up **current_line**. So:

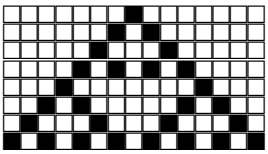
- **current_line[0]**, **current_line[1]** and **current_line[2]** all evaluate to the value of the pixel outside the cone and determine the value of the pixel outside the cone on the next line;
- **current_line[1]**, **current_line[2]** and **current_line[3]** evaluate to the value of the pixel outside the cone for the first two, and the value of the pixel on the left boundary of the cone for the third one, and determine the value of the pixel on the left boundary of the cone on the next line;
- **current_line[-3]**, **current_line[-2]** and **current_line[-1]** evaluate to the value of the pixel outside the cone for the last two, and the value of the pixel on the right boundary of the cone for the first one, and determine the value of the pixel on the right boundary of the cone on the next line.

```
In [17]: def display_ECA(rule_nb, size):
             bit_below = record_rule(rule_nb)
             new_line = [0, 1]
             display_line(new_line, size)
```

```
        for n in range(size):
            current_line = [new_line[0]] * 2 + new_line + [new_line[0]] * 2
            new_line = [bit_below[current_line[i],
                                  current_line[i + 1],
                                  current_line[i + 2]
                                 ] for i in range(len(current_line) - 2)
                        ]
            display_line(new_line, size - n - 1)
```

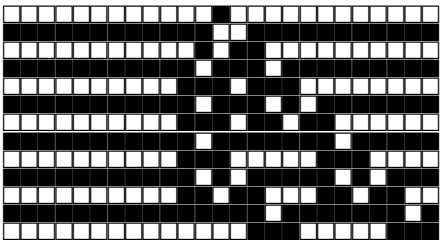
Rule 90 is an example where the outside of the cone consists of nothing but white pixels:

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



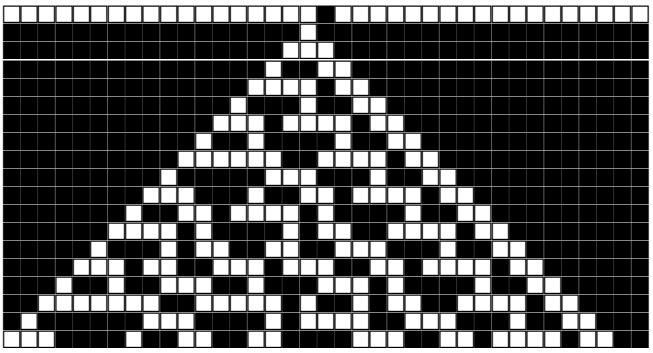
Rule 107 is an example where outside the cone, black and white half-infinite lines alternate:

```
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:

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



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.

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 then generate from its argument **E** the string **f'{E:08b}'**, say **s**, and 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 use other bases:

```
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:

```
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:

```
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 pass as an argument to **key** 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:

```
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}'
```

```
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')
```

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:

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

```
Out[27]: 3
```

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

```
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):
             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:

```
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:

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)
```