misc/cookie

- name: misc/cookie
- author: hellopir2
- solves: 4
- points: 485
- attachments
  - `checker.py`
  - `out.zip`

> What is life if not incomprehensible nonsense? Please untangle this for me. I'm hungry.
>
> Flag is valid ascii.
>
> **Downloads**
>
> - `checker.py`
> - `out.zip`

solution

Giving the solution itself alone is rather complicated and would not provide sufficient context. Refer to § process instead.

process

We will first inspect `checker.py`. It is not very obvious to what it is doing, so you would need some time to read it. But if you do take the time to read it, you should figure out that the following...

`out.txt` describes a 8638×12964 grid (from extracting the very large `out.txt` from `out.zip`), with `^0` describing the input boxes (gray, different variant), `$0` being the active boxes (white), `;3` being the inactive boxes (black), and `g0` for irrelevant boxes (gray).

The goal is that we need to set the input boxes correctly. Then, running the following code on the map:

```
for i in range(l):
    for j in range(w):
        if grid[i][j] == ";3" and neighs(i,j) == 1:
            grid[i][j] = "$0"
```

... which basically says for each box in Z-order, if the box is an inactive box and there is *exactly* 1 active box in its neighbor (corners included), turn the box into an active box.

After running the code, if the input boxes are set correctly, then the input box at `(8633, 12959)` (`grid[-4][-4]`) becomes an active box. If this happens, then the flag is simply the input boxes, interpreted as a bit string, with inactive boxes being `0` and active boxes being `1`.

Now, you might be tempted to brute force the input boxes in reverse. Don't. We wasted a bit of time on this...: `reverse_play_does_not_work`.

Instead, visualize the grid. After using some custom Python code to parse the image as a grid (`view_out_image.py`), we should get the following image:

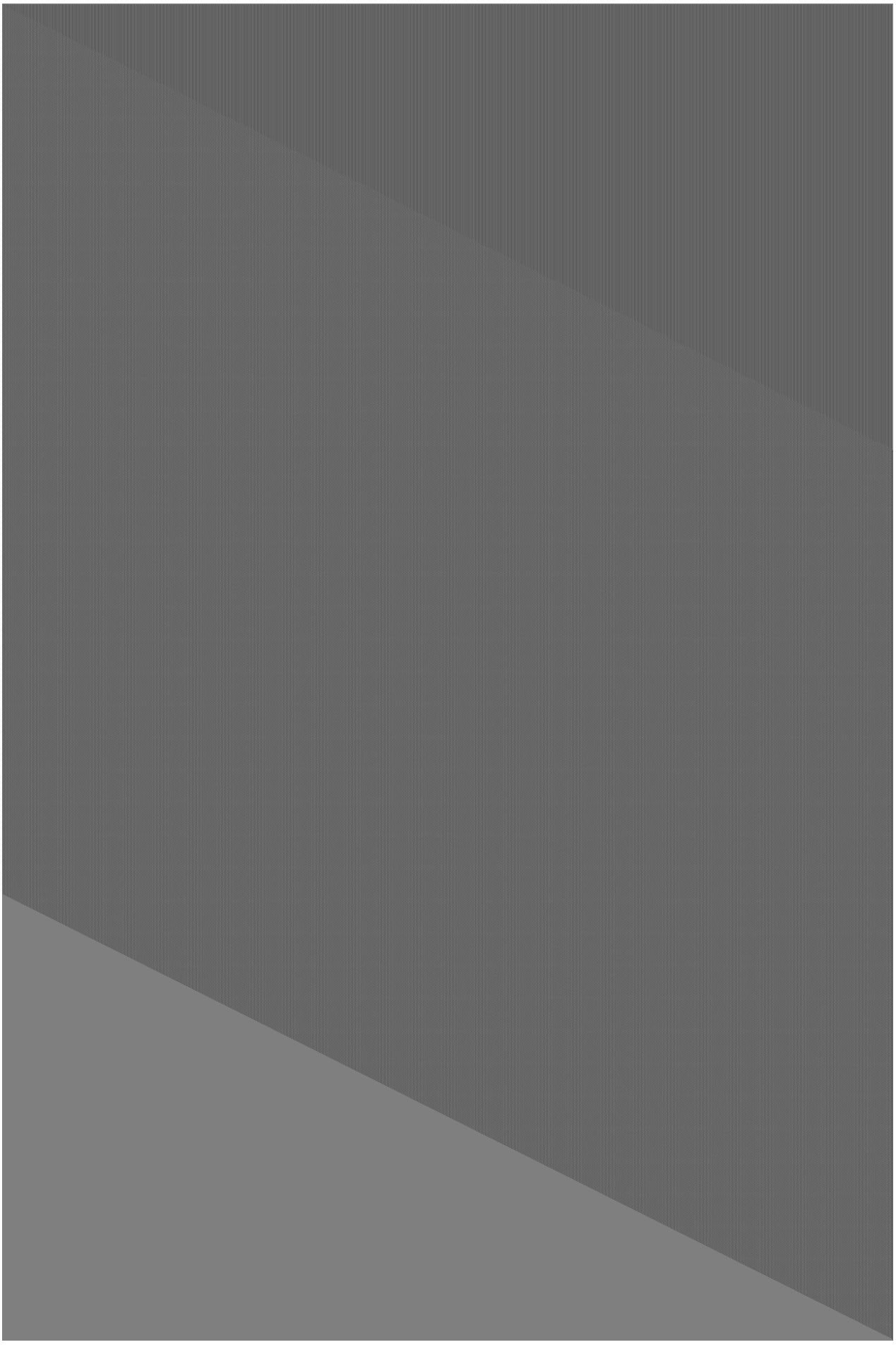
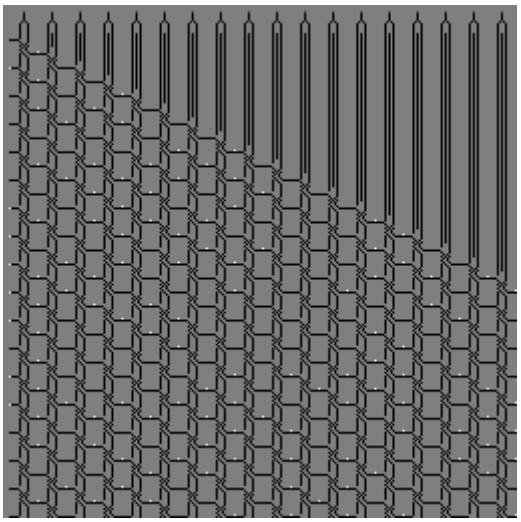


image of the grid

A portion of the image so that you can clearly know what is going on:

> 
>
> **a small section of the grid**

Coupled with the knowledge of the code that will run on the entire map, it seems like we are playing with some sort of circuits... It might seem hopeless, but don't give up yet.

First thing is that our inputs are at the top. The output is at the bottom-right corner.

If you look into the pattern, you should be able to identify two kinds of circuits, each with 4 possible inputs:

inputs\circuit	circuit 1	circuit 2
00		
01		
10		
11		

The above images are generated after running the algorithm on the entire circuit image. The left and the top are the 2 inputs. One can observe there are 2 outputs, one to the right, one to the bottom.

Notice that the bottom output is always the top output. So we can think of the bottom output as a passthrough of the top output. This means our inputs passthrough the entire vertical layer of gates.

The right output is the more interesting one. You should be able to identify circuit 1 as a XOR gate and circuit 2 as a passthrough gate.

You should also find some active and inactive boxes in between the vertical layers:

> 
>
> **in between the vertical layers**

And you would realize those with an active box is a NOT gate and those with an inactive box is a passthrough gate. Alternatively, if you interpret the active/inactive box as another input instead of as part of the gate itself, it is a XOR gate.

Finally, for the rightmost vertical layer, is the output layer:

> 
>
> **a section of the output layer**

If you run the output layer, you would realize it is simply an AND gate.

So it looks like we need to manipulate the inputs at the top such that all outputs at the output layer are 1. You might have an idea to solve this by hand, but don't. This is clearly too much. Not knowing what to do, it might still be interested to make the gate into a much more machine-readable format first: `extract_xor.py`.

Then we would get the following output (`xor.txt`):

▶ This is 742 kiB of text here... so maybe do not expand.

> **Legend of the output**
>
> - `0`: Passthrough gate, no user-controllable input
> - `1`: NOT gate, no user-controllable input
> - `>`: Passthrough gate, user-controllable input
> - `X`: XOR gate, user-controllable input

Suddenly, one might have an idea of using mathematics to solve the problem. It turns out we can view the above logic gate system as a system of linear equations mod 2. Also see boolean satisfiability problem § XOR-satisfiability.

Using `sympy`, we can solve the above equation. Let the possible inputs be the unknowns or variables. For each row, count the number of `1`s. If the number of `1`s is even, the linear equation output for that row is `1`, otherwise it is `0`. Also, each `X` corresponds to using an input variable (ensure the correct input variable is used), and all `>` are simply ignored.

A native implementation of solving the above equation might be too slow. Some online searching yields https://github.com/sympy/sympy/issues/19243#issuecomment-1326792289, which mentions that `DomainMatrix` is much faster and provides an example implementation. To make it faster, ignore every (1 + 8n)-th input, because normal ASCII

characters always have the most significant bit as 0. See `solve_xor.py` for a reference implementation.

Now simply solve the equation. Even with the above code and optimizations, it might still take 5 minutes to get the result.

After getting the result, simply interpret the input as a bit string to get the flag text: `5hR13kbu1bS_Ar3_Th3_B3s7_b3c4uS3_th3y_a110W_m3_t0_c0n5trUcT_CircU1ts_68bbf319`. Wrap it in the flag format: `amateursCTF{5hR13kbu1bS_Ar3_Th3_B3s7_b3c4uS3_th3y_a110W_m3_t0_c0n5trUcT_CircU1ts_68bbf319}`. See `flag.py` for a reference implementation and the expected output. Note that the reference implementation assumes the (1 + 8n)-th input is not in the input and is 0.