MUA

Maratones uniandes

BITMASKS

ISIS 2801

Bitmasks

Bitmasks are defined as **small sets of Booleans.**

```
                       F E D C B A
                       5 4 3 2 1 0
{B,F} = {1,5} = 34 =   1 0 0 0 1 0
```

Bitmask encodings are faster than other structure alternatives

Easy to manipulate with bit-wise operations

Bitmasks

move left the number of times you multiply by 2

```
1 0 0 0 1 0 << 2  = 10001000
```

Bitmasks

move left the number of times you multiply by 2

```
1 0 0 0 1 0 << 2  = 10001000
```

divide by 2

```
1 0 0 0 1 0 >> 1  = 10001
```

Bitmasks

move left the number of times you multiply by 2

```
1 0 0 0 1 0 << 2  = 10001000
```

divide by 2

```
1 0 0 0 1 0 >> 1  = 10001
```

set bit j

```
1 0 0 0 1 0 | 1000  = 101010
```

Bitmasks

move left the number of times you multiply by 2

$$1\ 0\ 0\ 0\ 1\ 0 \ll 2 = 10001000$$

divide by 2

$$1\ 0\ 0\ 0\ 1\ 0 \gg 1 = 10001$$

set bit j

$$1\ 0\ 0\ 0\ 1\ 0 \mid 1000 = 101010$$

check bit j

$$1\ 0\ 0\ 0\ 1\ 0 \ \&\ 1000 = \begin{cases} 0 & \text{if } N[j] == 0 \\ 1 & \text{if } N[j] == 1 \end{cases}$$

Bitmasks

```
1 0 0 0 1 0 & ~(000010) = 101000
```

clear bit j

Bitmasks

```
1 0 0 0 1 0 & ~(000010) = 101000
```

clear bit j

flip the value of bit j

```
1 0 0 0 1 0 ^ 000010  = 100000
```

Bitmasks

```
1 0 0 0 1 0 & ~(000010) = 101000
```

clear bit j

flip the value of bit j

```
1 0 0 0 1 0 ^ 000010  = 100000
```

get least significant bit

```
1 0 0 0 1 0 & ~100011  = 0
```

Bitmasks operations

<<   >> | &   ~   ^

```
0b10101
bin(n)
int(b, 2)
```

>>>

fill with 0 from the left

Odd occurring

Given an integer array, return the only element that occurs an odd number of times

```
{12, 12, 90, 14, 14, 90, 14, 90, 14}
```

Odd occurring

Given an integer array, return the only element that occurs an odd number of times

```
{12, 12, 90, 14, 14, 90, 14, 90, 14}

int findOdd(int arr[]) {
   int res = 0, i;
   int n = sizeof(arr)/sizeof(arr[0]);
   for (i = 0; i < n; i++)
     res ^= arr[i];
   return res;
}
```

Only works for small-constraint problems

Floorboard

You are building a house and are laying the floorboards in one of the rooms. Each floorboard is a rectangle **1 unit wide and can be of any positive integer length**. Floorboards must be laid with their sides parallel to one of the sides of the room and cannot overlap. In addition, the room may contain features such as pillars, which lead to areas of the floor where no floorboards can be laid. The room is rectangular and the features all lie on a unit-square grid within it. You want to know the **minimum** number of floorboards that you need to completely cover the floor.

You are given a `String[] room` containing the layout of the room. Character `j` in element i of `room` represents the grid-square at position `(i, j)` and will be a `'.'` if this square needs to be covered with a floorboard or a `'#'` if the square is a feature where no floorboard can be laid. Return an int containing the minimum number of floorboards you need to completely cover the floor.

Floorboard

The board is **10 x 10**

0)

{"....."
,"....."
,"....."
,"....."
,"....."}

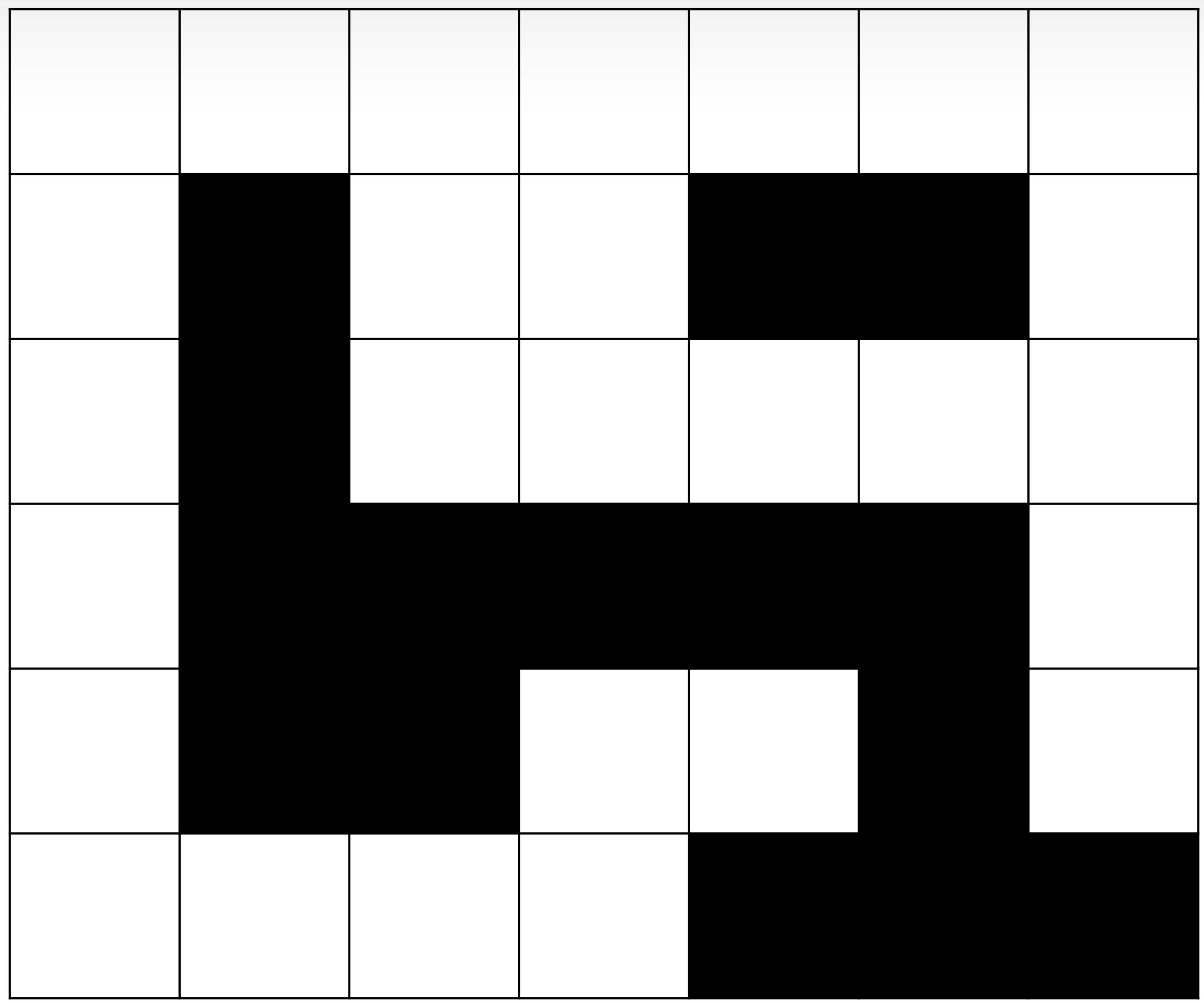
Returns: 5

1)

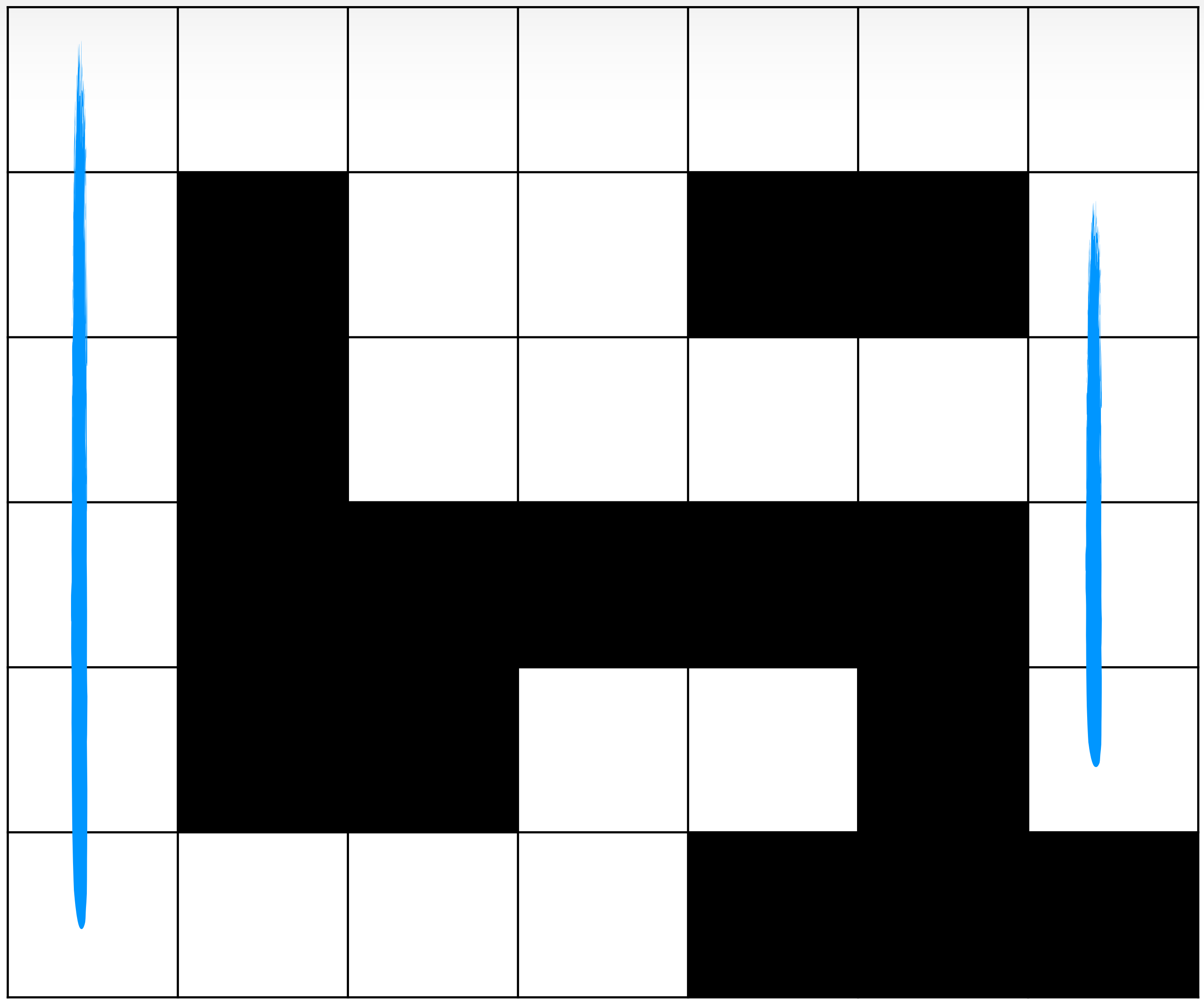
{"......."
,".#..##."
,".#....."
,".#####."
,".##..#."
,"....###"}

Returns: 7

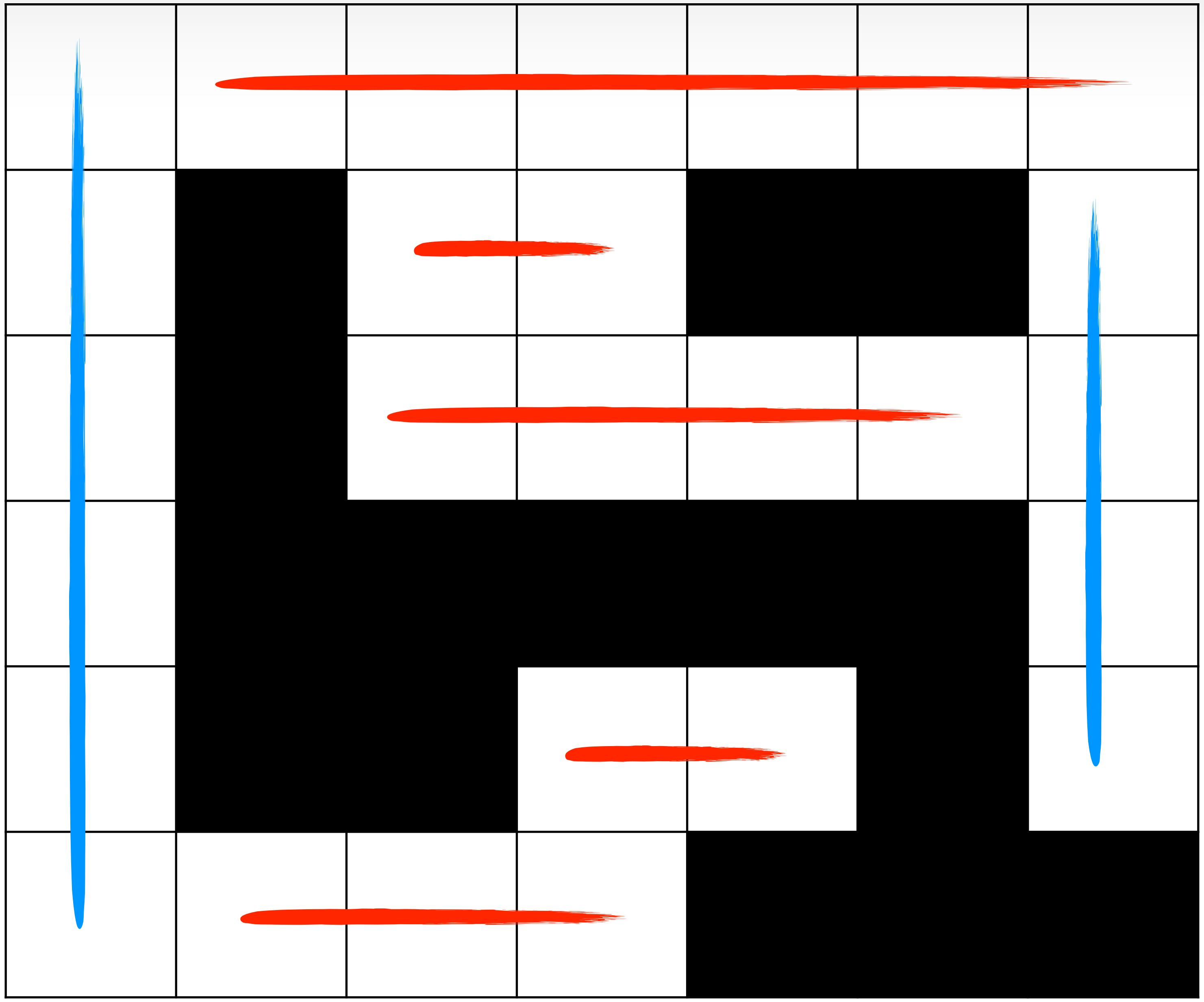
Floorboard



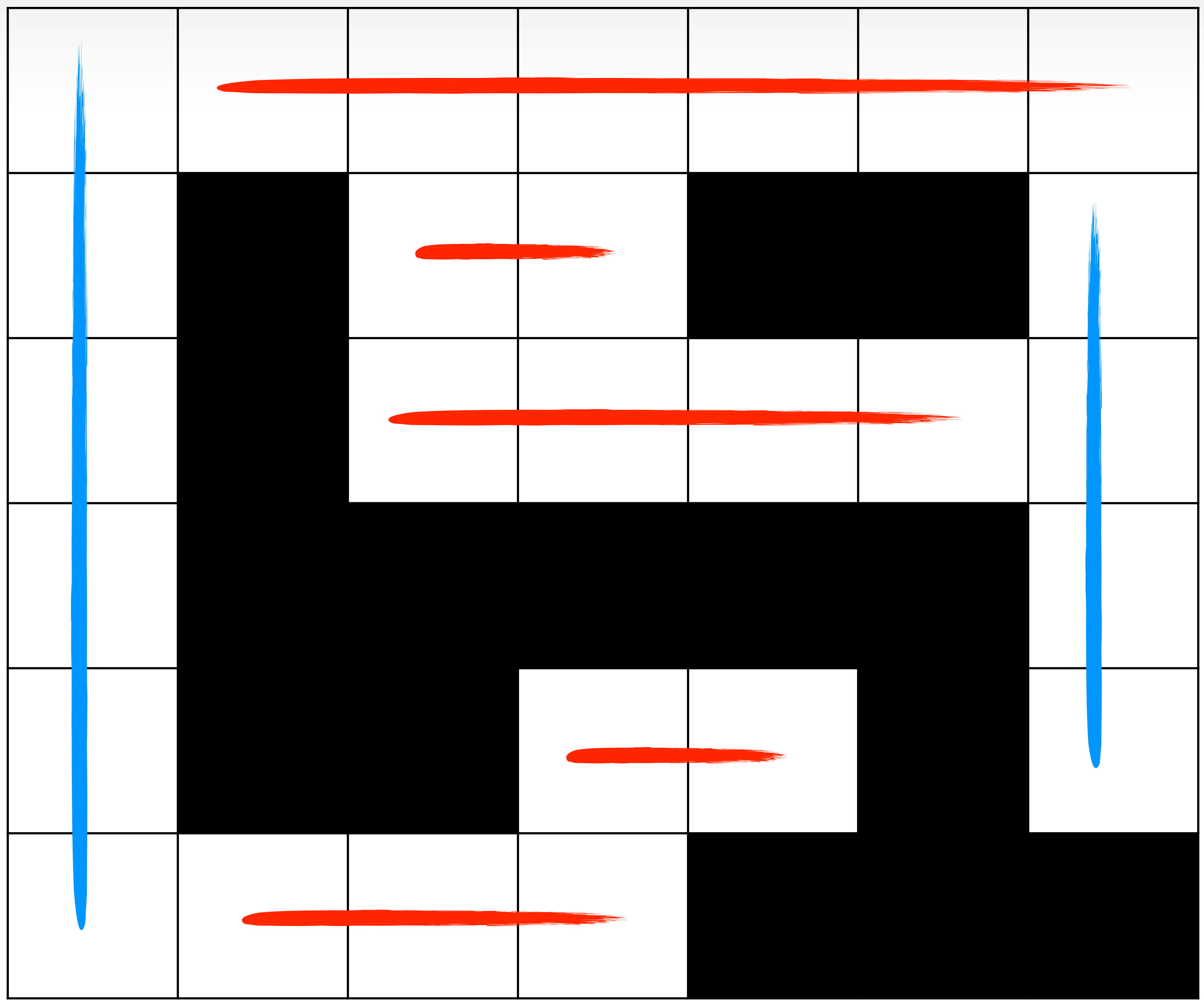
Floorboard



Floorboard



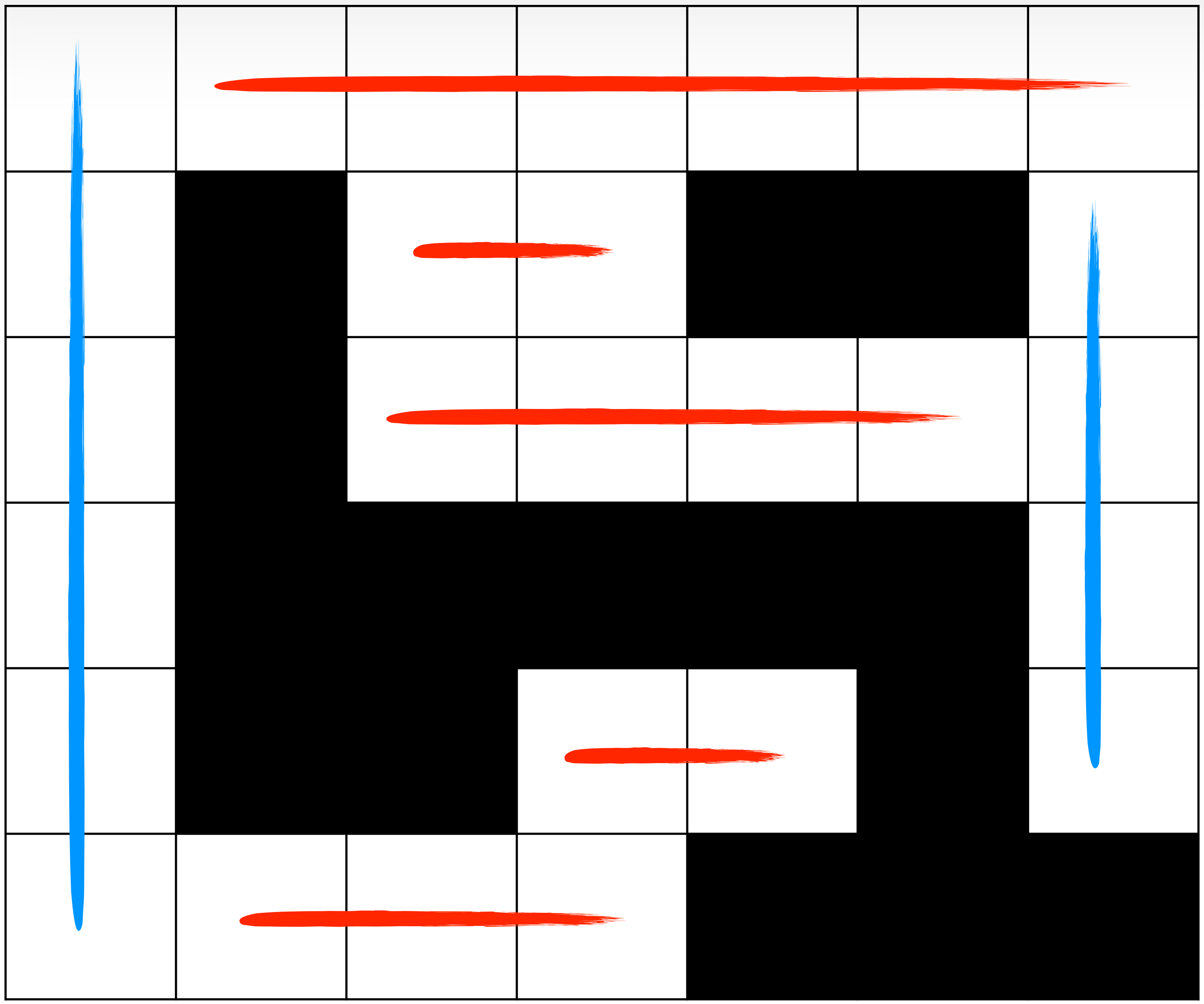
Floorboard



We need 7 boards

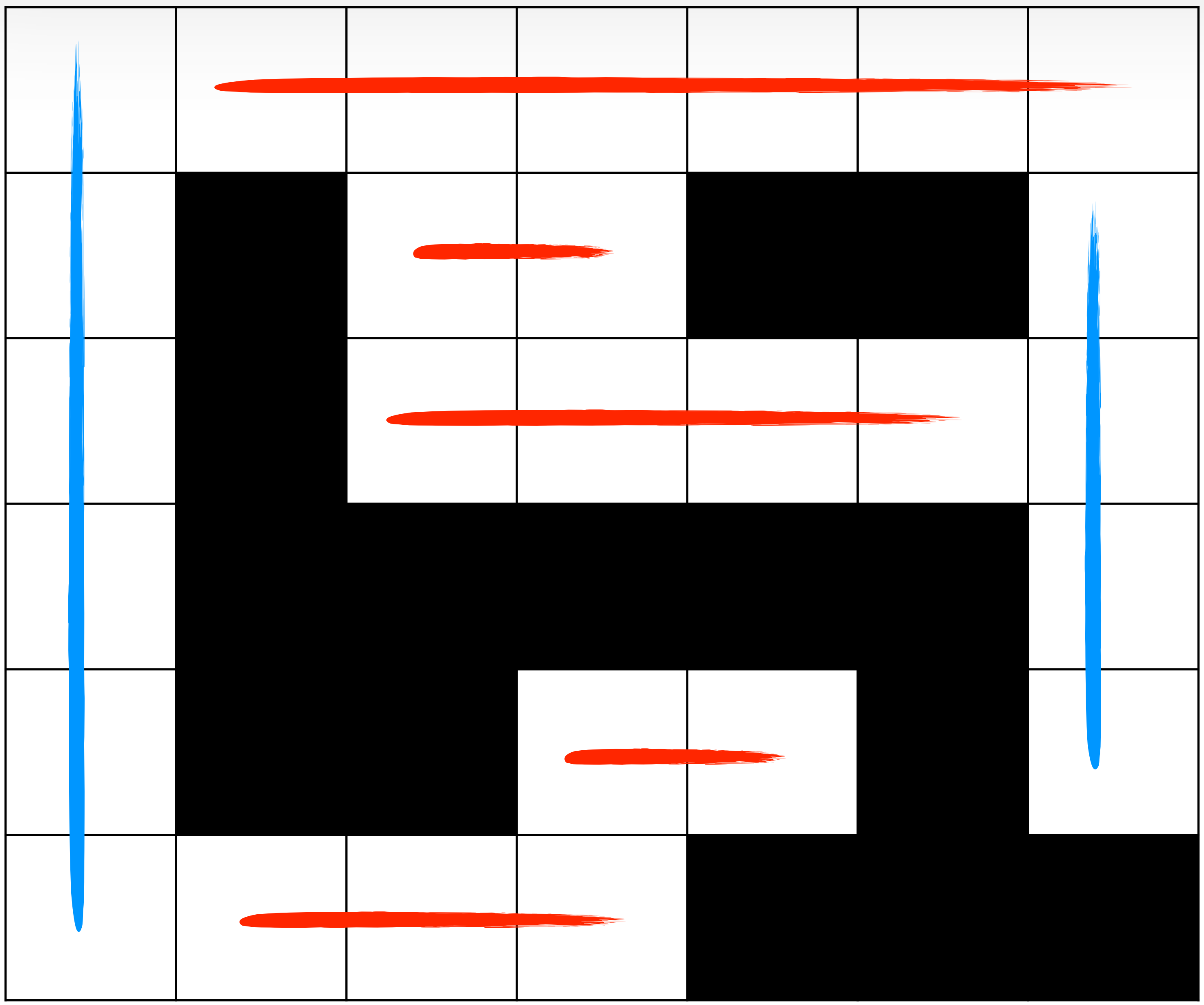
FIRST LOOK

Floorboard



Encode each cell as covered or not covered

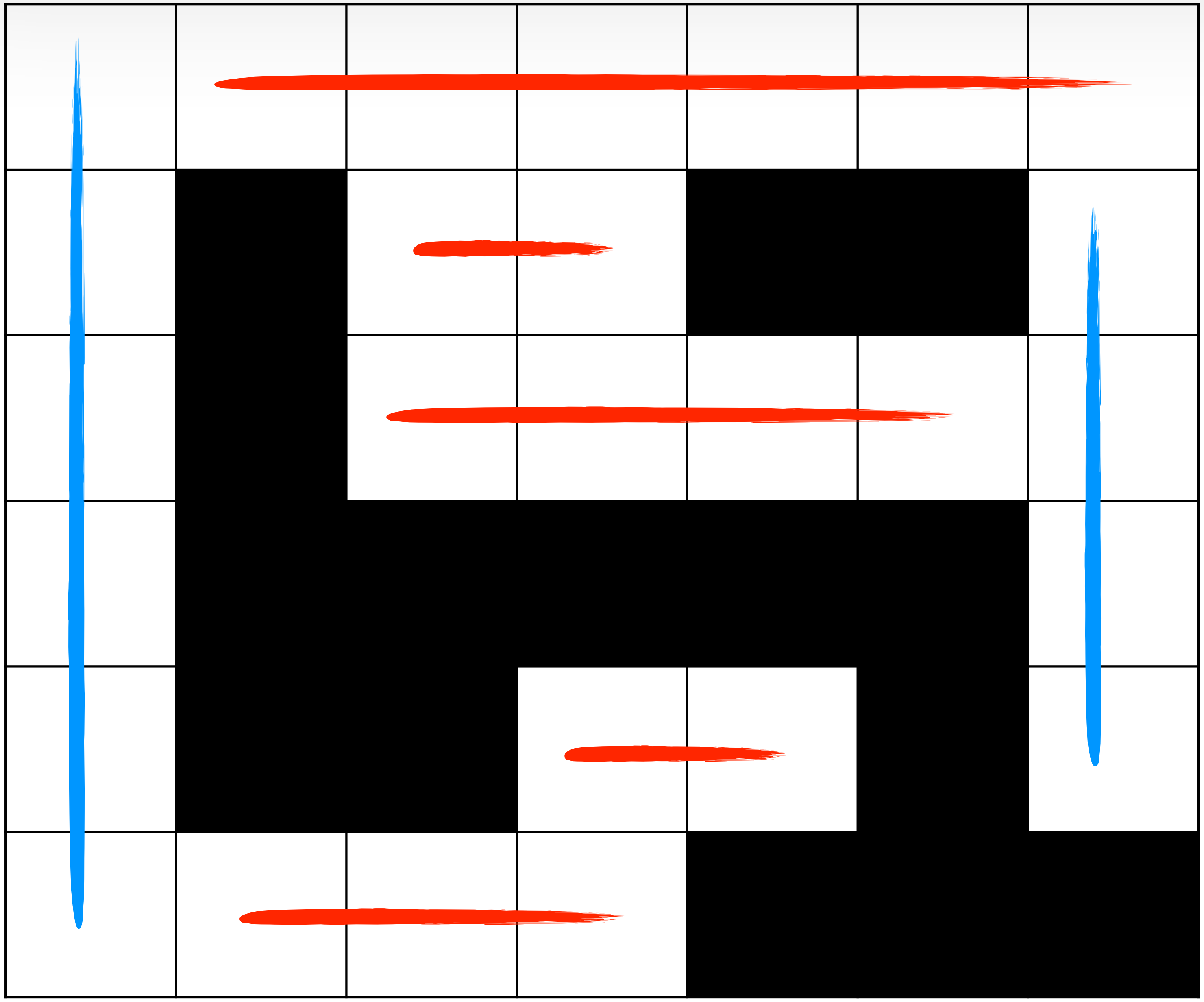
Floorboard



Encode each cell as covered or not covered

using bitmask dp

Floorboard

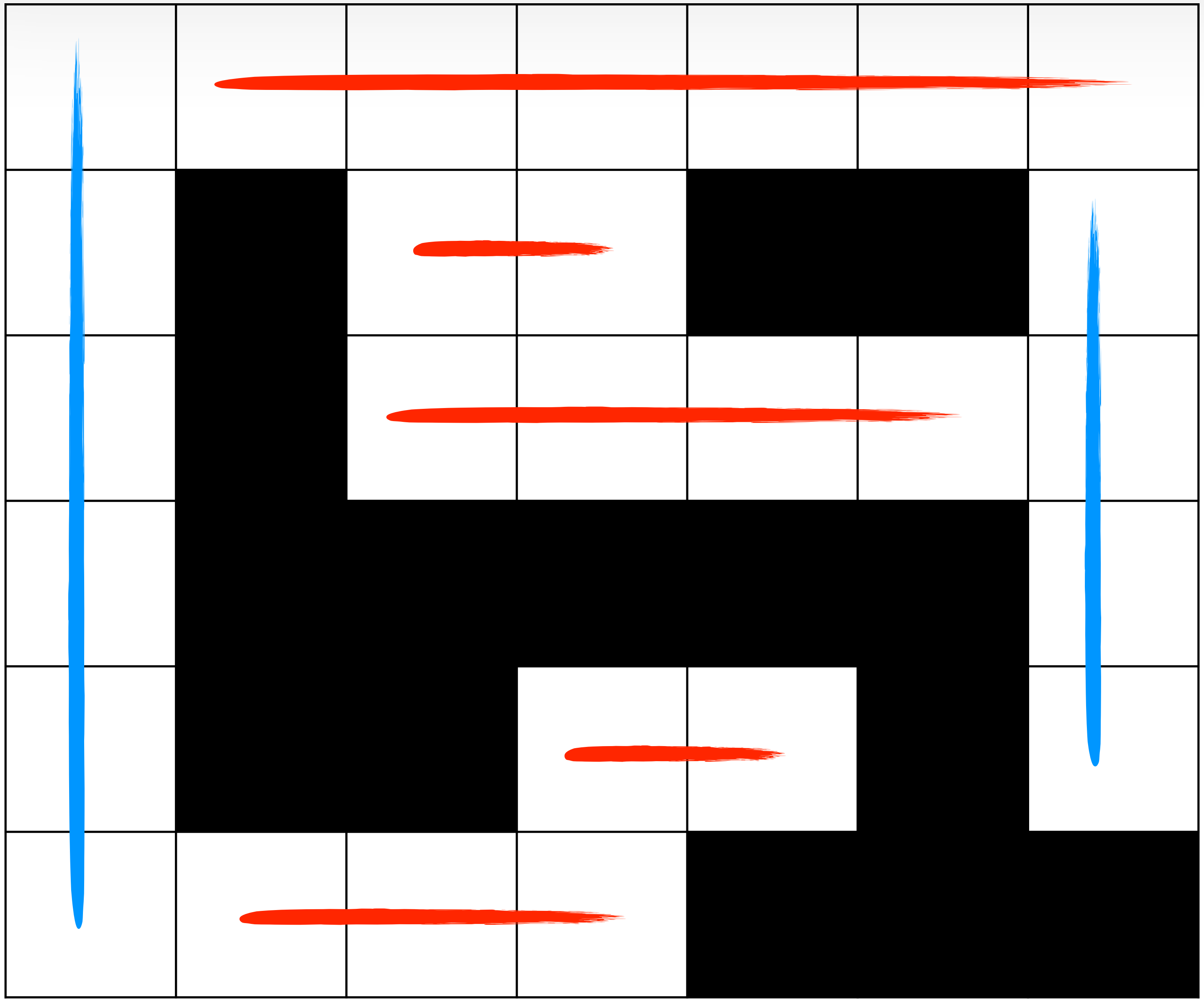


Encode each cell as covered or not covered

using bitmask dp

$$2^{10^2} = 2^{100}$$

Floorboard



Encode each cell as covered or not covered

using bitmask dp

$$2^{10^2} = 2^{100}$$

the complete state is too much information to remember

USE SWEEP SEARCH

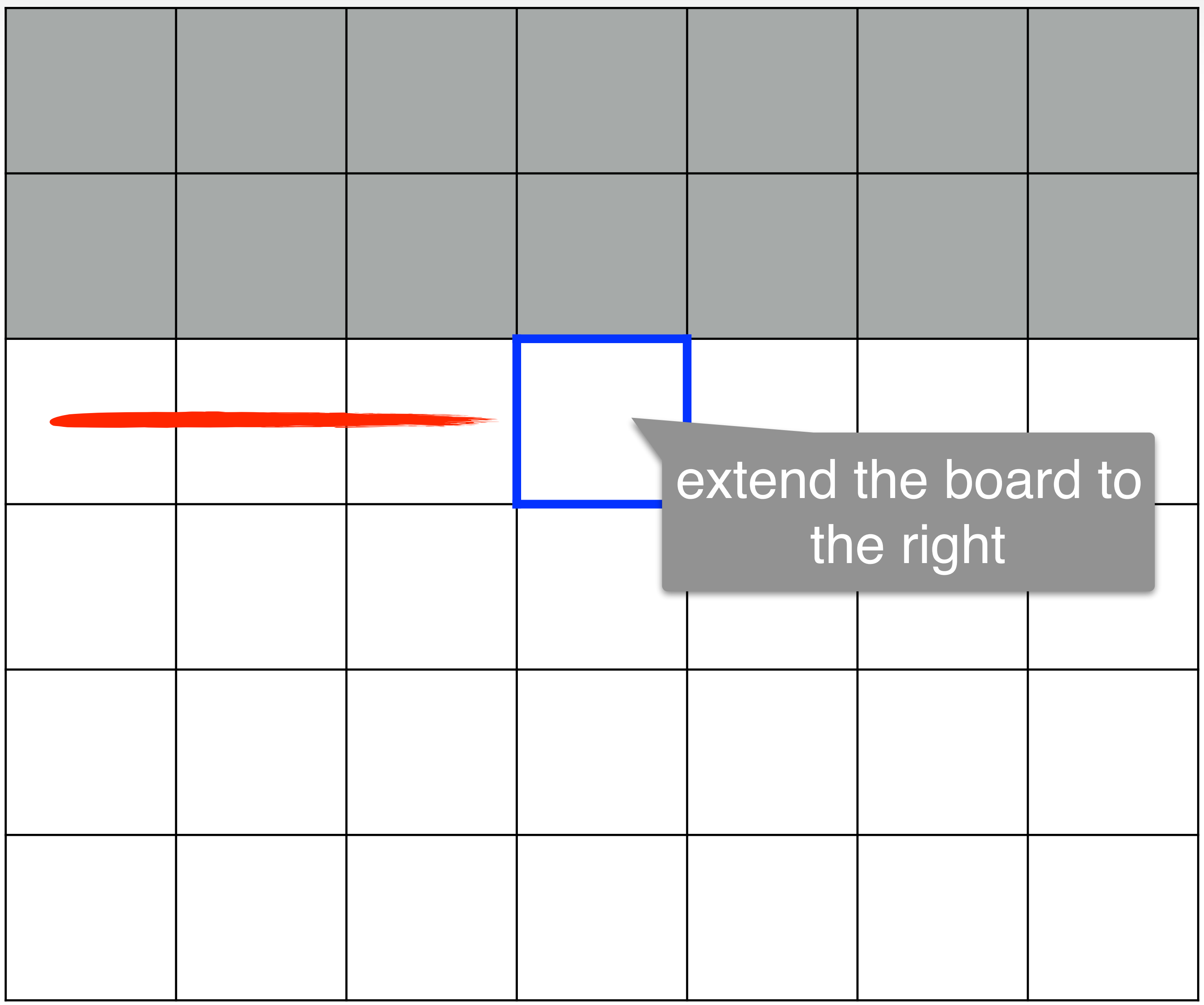
Floorboard

explore cells in row major order

1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31	32	33	34	35
36	37	38	39	40	41	42

1. cover the board
2. use least amount of boards

Floorboard

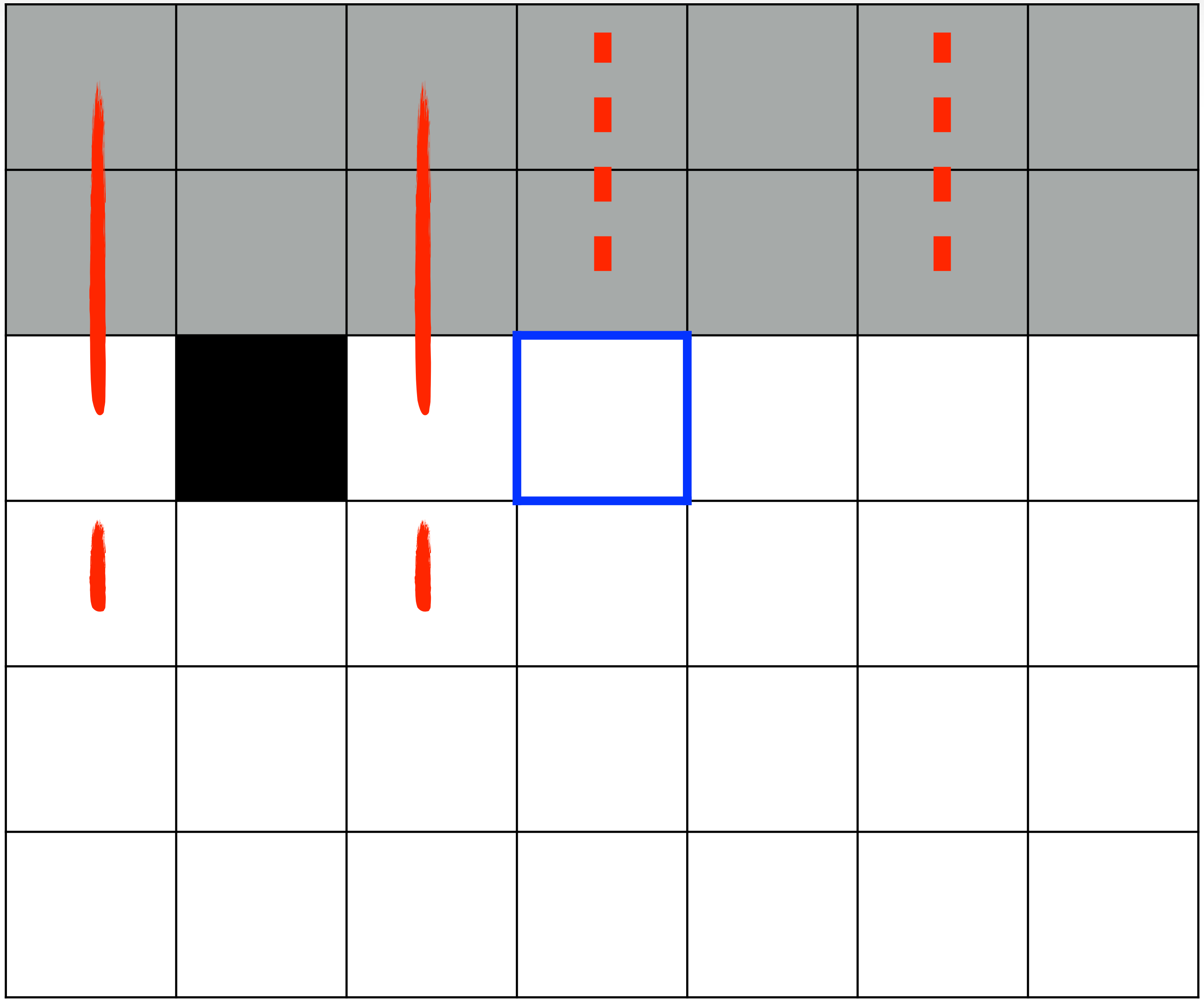


Mask

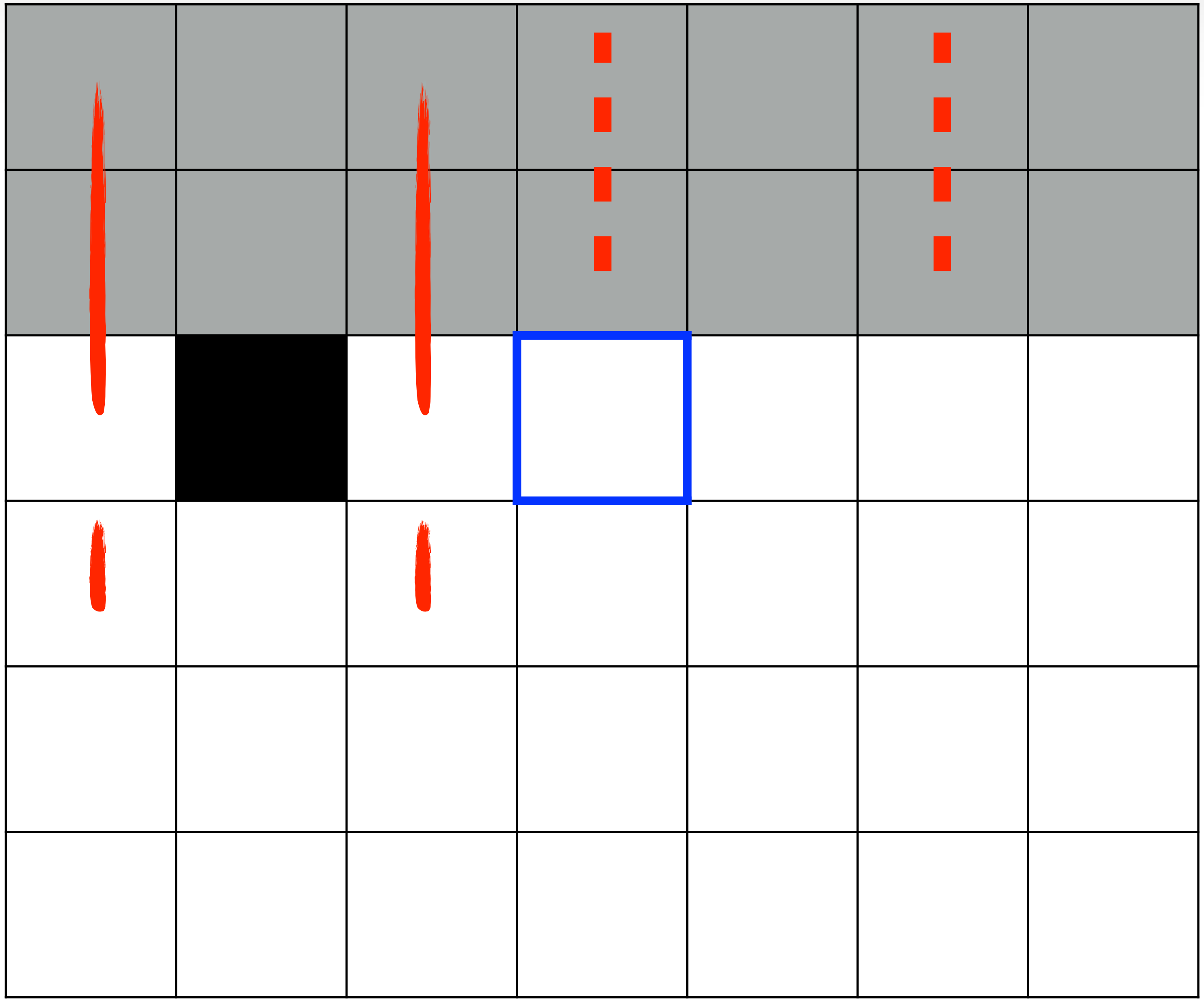
1 := horizontal board
0 := no board

i.e., vertical board behind (have a 0)

Floorboard



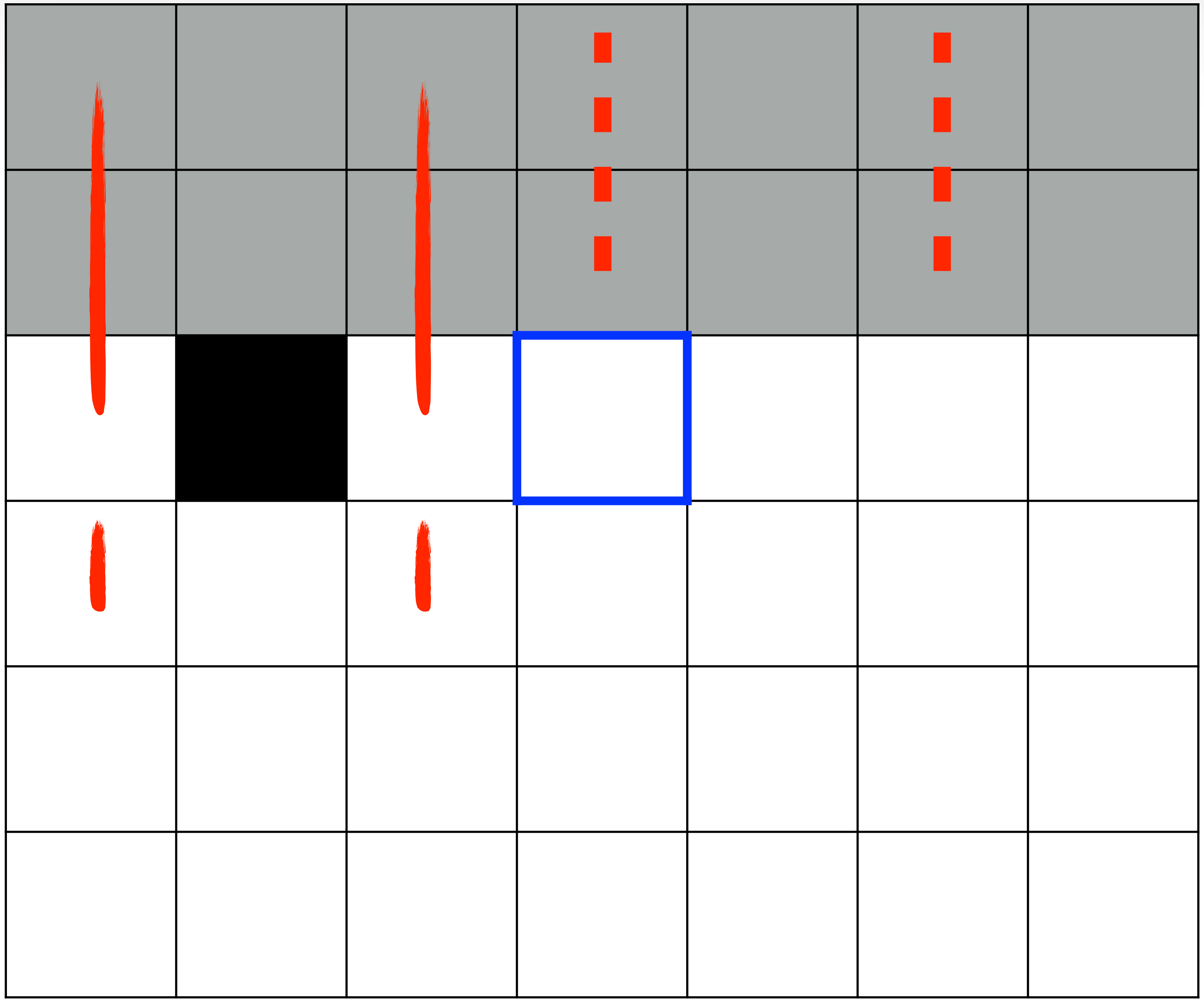
Floorboard



Mask

1 := vertical board
0 := no board

Floorboard

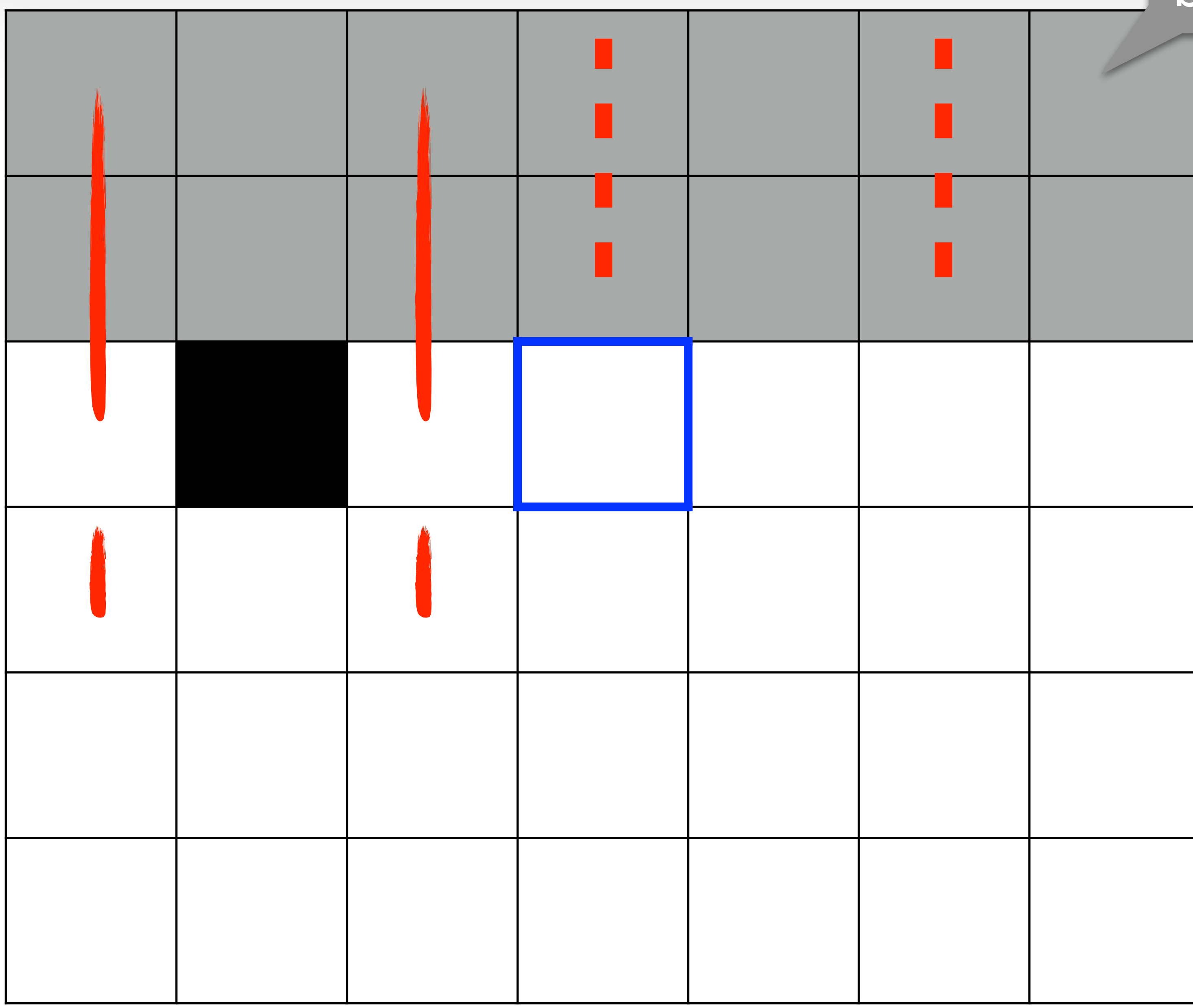


Mask

1 := vertical board
0 := no board

This gives a state space of size 2^c

Floorboard



I want to have a column base analysis of the system

Mask

1 := vertical board
0 := no board

This gives a state space of size 2^c

Floorboard

DP state:

```
2    10   10   2^10
[]   []   []   []
```

board below · row · column · mask of previous row

Choices:

1. start a new board (horizontal or vertical)
2. continue a board (horizontal, vertical)

Floorboard

```
int dp(int board, int i, int j, int mask) {
   if(i == w) {
      return dp(0, 0, j+1, mask);
   }
   if(j == h) {
      return 0;
   }
   if(memoization[board][i][j][mask] != null) {
      return memoization[board][i][j][mask];
   }

   int res = inf;
  //turn off ith bit
   int missingVerticalBoard = ((1 << w) - 1 - (1 << i)) & mask;

   if(blocked[i][j]) {
      res = dp(0,i+1, j, missingVerticalBoard);
   }
```

Floorboard

```
else {
    if(board == 1) { //continue horizontal board
      //eliminate the column behind
      res = Math.min(res,  dp(board, i+1, j, missingVerticalBoard));
    } if((mask & (1 << i)) > 0) { //continue vertical board
        res = Math.min(res, dp(0, i+1, j, mask));
    }       //start  new board
     //new horizontal
    res = Math.min(res,  1 + dp(1, i+1, j, missingVerticalBoard));
     //new vertical;
    res = Math.min(res,  1 + dp(0, i+1, j, missingVerticalBoard | (1 << i)));
    }
    return memoization[board][i][j][mask] = res;
}
```

Reach for the stars

You are given a star shaped stamp +. The black area is covered in ink and the white area is not. When the stamp hits the paper, it leaves a mark for each cell of ink that hits the paper.
For example, ++ can be made with two stampings. Notice the stamp must always remain axis-aligned when hitting the paper. We also require that the stamp be completely contained within the paper. Note a cell of paper stamped once with black ink is indistinguishable from a cell of paper stamped multiple times with black ink. Note also that cells and stamp line up properly, i.e., a cell is either covered completely by the stamp or not covered at all, i.e., the stamp will not cover part of a cell.

Reach for the stars

Given a black and white image, determine the minimum number of times, if possible, you would need to stamp the paper with the star stamp to end up with the design specified.

Input:
The first input line contains a positive integer, n, indicating the number of images to evaluate. Each image starts with a line containing two integers, r and c, ($1 \le r \le 9$, $1 \le c \le 9$), representing the number of rows and columns, respectively. The next r input lines contains c characters each. The characters are either '.', representing a blank cell of the image and '#', representing a cell of the image covered in ink.
Output:
For each image, output "Image #d: v" where v is the minimum number of stampings required to make the image. Replace v with "impossible" (without quotes) if it is not possible to form the image using the star shaped stamp. Leave a blank line after the output for each test case

Reach for the stars

Sample Input:

```
5
1 1
.
1 1
#
3 3
.#.
###
.#.
3 5
.#.#.
#####
.#.#.
4 7
.##.#..
######.
.######
..#..#.
```

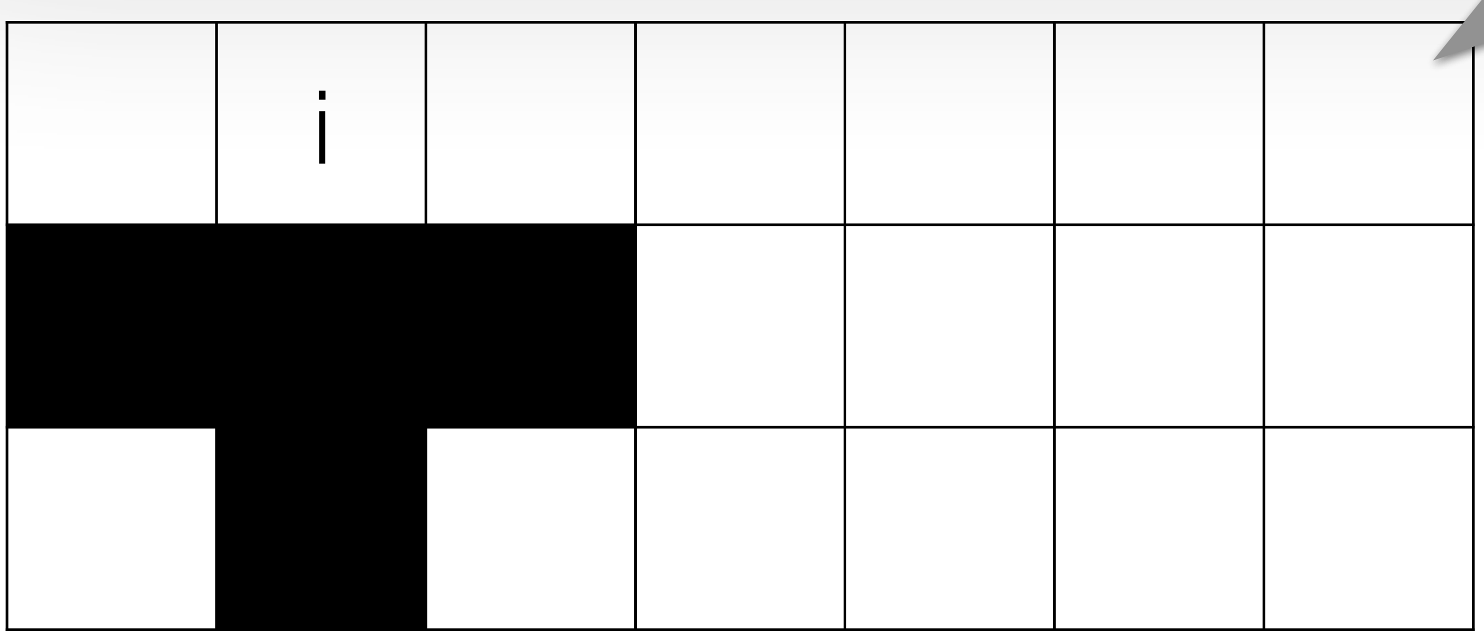
Sample Output:

```
Image #1: 0
Image #2: impossible
Image #3: 1
Image #4: 2
Image #5: 5
```

USE SWEEP SEARCH

Reach for the stars

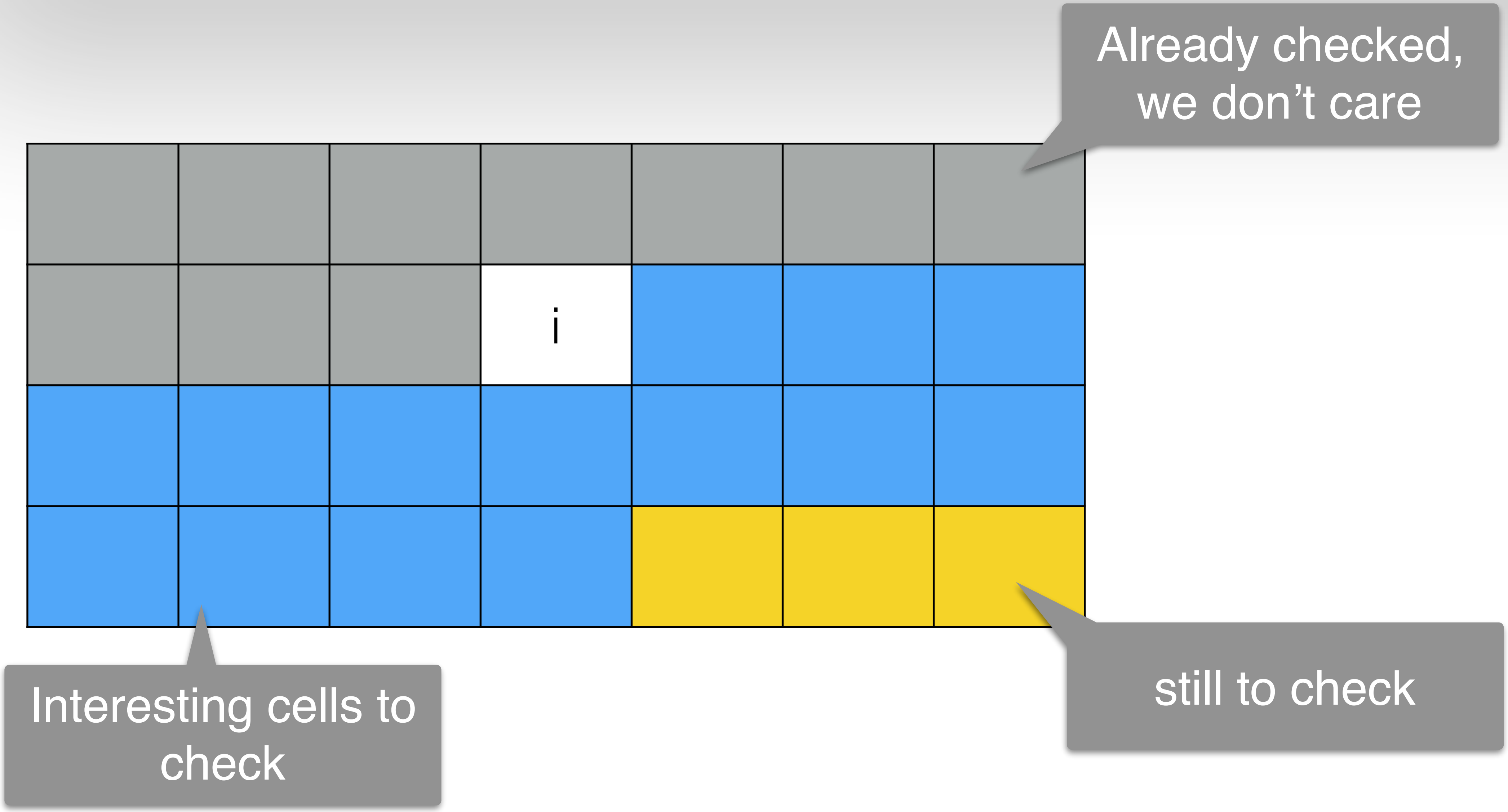
explore cells in row major order



Keep track of 3 rows $\Longrightarrow$ 2^{27} masks

too big, use rolling bitmasks

Reach for the stars



Keep track of 2 rows $\Longrightarrow$ 2^{18} masks

Reach for the stars

Mask

1 := cell is marked
0 := not marked (yet)

moving to the next cell is equivalent to a right shift

		i	1	2	3	4
	c-1	**c**	**c+1**			
		2c				

putting a stamp is just marking the cells $2^0 \mid 2^{c-1} \mid 2^c \mid 2^{c+1} \mid 2^{2c}$

code time